Low Latency for Cloud Data Management

Dissertation with the aim of achieving a doctoral degree at the
Faculty of Mathematics, Informatics, and Natural Sciences

Submitted at the University of Hamburg
by Felix Gessert, 2018

Day of oral defense: December 18th, 2018

The following evaluators recommend the admission of the dissertation:

Prof. Dr. Norbert Ritter

Prof. Dr. Stefan Deßloch

Prof. Dr. Mathias Fischer

There are only two hard things in Computer Science: cache invalidation, naming things, and off-by-one errors.

– *Anonymous*

Acknowledgments

This dissertation would not have been possible without the support and hard work of numerous other people. First and foremost, I would like to thank my advisor Prof. Norbert Ritter for his help and mentoring that enabled this research. Not only has he always given me the freedom and patience to execute my ideas in different directions, but he has formed my perception that academic research should eventually be practically applicable. Therefore, he is one of the key persons that enabled building a startup from this research. I also deeply enjoyed our joint workshops, talks, tutorials, and conference presentations with the personal development these opportunities gave rise to. I am convinced that without his mentoring and pragmatic attitude neither my research nor entrepreneurial efforts would have worked out this well.

I would also like to express my gratitude to my co-advisor Prof. Stefan Deßloch. His insightful questions and feedback on different encounters helped me improve the overall quality of this work.

My sincerest thanks also go to my co-founders Florian Bücklers, Hannes Kuhlmann, and Malte Lauenroth. The extensive discussions with Florian and our relentless efforts to build useful software are the cornerstone this work is built upon. Without this set of technically skilled and personally generous co-founders, this research would never have led to Baqend. I am excitedly looking forward to what more we will be able to achieve together.

I was fortunate to work with exceptional colleagues and co-researchers that made this work not only fruitful, but enjoyable. Wolfram Wingerath provided not only the most critical feedback, but also the most valuable one. At many occasions, our brainstorming sessions sparked pivotal new approaches. Michael Schaarschmidt offered an enthusiastic stream of helpful ideas and new perspectives. Erik Witt contributed numerous improvements to this work as well as inspirations for clarity of thought and writing. Fabian Panse was always available for stimulating conversations and advice. Steffen Friedrich could be counted on to keep the teaching and organizational matters on course. The same is true for Anne Awizen, who never got tired of reminding me about deadlines, forms, and other things that I would surely have missed otherwise.

Many others provided helpful feedback or essential work on Orestes during the course of writing this dissertation, including Konstantin Möllers, Kevin Twesten, Sven Decken, Jörn

Domnik, Julian Tiemann, Julian Schenkemeyer, Nils Gessert, Dirk Bade, Uta Störl, Meike Klettke, and Stefanie Scherzinger.

Most of all, I would like to thank my wife for her encouragement throughout these challenging and thrilling times. Finally, I am also deeply grateful for the support from my family and friends.

Felix Gessert Hamburg, September 24th, 2018

Abstract

English

With the rise of scalable, distributed web applications, latency has become a fundamental challenge for cloud data management. The delays caused by accessing data from cloud services often dominate the performance of web and mobile applications. While modern data management systems address the need for higher scalability and fault tolerance, low latency remains an open issue. How can low-latency queries and reads be enabled without sacrificing central data management abstractions such as consistency levels and transactional isolation?

In this thesis, we investigate caching in cloud data management for dynamic data ranging from database objects to complex query results. In the context of distributed systems, achieving reads that are both fast and consistent is tied to the challenge of maintaining fresh replicated data in proximity to clients. Therefore, we propose the data management platform Orestes that enhances existing NoSQL database systems with low latency. Orestes introduces several new techniques to substantially improve latency in cloud data management. First, it leverages the expiration-based model of web caches available all over the world through a novel cache coherence scheme – *Cache Sketches*. Our approach thus makes caching applicable to highly volatile cloud data while maintaining rigorous consistency levels. Second, transactions are a key concept often sacrificed in state-of-the-art systems for performance reasons. Therefore, we propose an approach for horizontally scalable, low-latency ACID transactions that can be added on top of existing database systems. Third, to enable polyglot persistence, we survey the field of scalable data management and derive a novel classification scheme that relates database implementation techniques to functional and non-functional guarantees. By combining these findings in a unified data management interface, Orestes can provide existing systems as a scalable, low-latency Database-as-a-Service. Fourth, with the design of a polyglot persistence mediator, we argue that the selection of suitable database systems for a given set of requirements can be automated based on service level agreements. Finally, we provide evidence that for typical web applications and database workloads, our approach can improve latency by more than an order of magnitude compared to traditional cloud-hosted backends and database systems.

German

Mit der Verbreitung skalierbarer und verteilter Webanwendungen sind Zugriffslatenzen zu einer grundlegenden Herausforderung für das Cloud Data Management geworden. Die Verzögerungen bei der Abfrage von Daten aus Cloud-Diensten dominieren oft die Performance von Web- und mobilen Anwendungen. Während moderne Datenmanagementsysteme den Bedarf nach höherer Skalierbarkeit und Fehlertoleranz adressieren, bleibt die Latenz eine offene Herausforderung. Wie können Lesezugriffe und Queries mit geringer Latenz beantwortet werden, ohne dabei zentrale Abstraktionen des Datenmanagements wie Konsistenzstufen und transaktionale Isolation aufzugeben?

In dieser Arbeit untersuchen wir Caching im Cloud Data Management für dynamische Daten von Datenbankobjekten bis hin zu komplexen Query-Ergebnissen. Im Kontext verteilter Systeme sind schnelle und korrekte Lesezugriffe mit der Herausforderung verbunden, replizierte Daten konsistent in physischer Nähe zu Usern vorzuhalten. Aus diesem Grund führen wir die Data Management-Plattform Orestes ein, um die Latenzen bestehender NoSQL-Datenbanksysteme zu verringern. Orestes verwendet mehrere neue Techniken, mit denen die Latenzen lesender Operationen im Cloud Data Management erheblich verbessert werden. Erstens nutzt es das expirationsbasierte Modell von Web-Caches, die über ein neues Cachekohärenz-Verfahren namens *Cache Sketches* aktuell gehalten werden. Unser Ansatz macht Caching somit auch für sehr volatile Cloud-Daten anwendbar und stellt dabei konfigurierbare Konsistenzgarantien sicher. Zweitens sind Transaktionen ein Kernkonzept des Datenmanagements, auf das in modernen Systemen oft aus Performancegründen verzichtet wird. Daher schlagen wir einen Ansatz für horizontal skalierbare ACID-Transaktionen mit geringen Latenzen vor, der auf bestehende Datenbanksysteme anwendbar ist. Drittens leiten wir für polyglotte Persistenz durch eine genaue Analyse verfügbarer Ansätze ein Klassifikationsschema ab, das die Implementierungstechniken der Datenbanksysteme mit funktionalen und nicht-funktionalen Garantien in Beziehung setzt. Durch die Anwendung der Systematik auf eine vereinheitlichte Datenmanagement-Schnittstelle kann Orestes bestehende Systeme als skalierbares Database-as-a-Service mit geringer Latenz anbieten. Viertens zeigen wir mit dem Design eines Polyglot Persistence Mediators, dass die Auswahl geeigneter Datenbanksysteme auf Basis von Service Level Agreements automatisiert werden kann. Abschließend belegen wir quantitativ, dass unser Ansatz für typische Webanwendungen und Datenbank-Workloads die Latenz um mehr als eine Größenordnung gegenüber herkömmlichen Backends und Datenbanksystemen verbessert.

Contents

1 Introduction

This thesis examines low latency for web applications and database systems in cloud environments.

Today, web performance is governed by round-trip latencies between end devices and cloud services. Depending on their location, users therefore often experience latency as loading delays when browsing through websites and interacting with content from apps. Latency is responsible for page load times and therefore strongly affects user satisfaction and central business metrics such as customer retention rates or the time spent on a site. In the web, users expect websites to load quickly and respond immediately. However, client devices are always separated from cloud backends by a physical network. The latency for data to travel between devices and cloud servers dominates the perceived performance of an application.

The significance of fast **page load times** has been studied extensively by large web, publishing, and e-commerce companies. Amazon, for example, has found that 100 ms of additional loading time decrease sales revenue by 1% [Lin06]. With Amazon's current revenue, the impact of an additional 10th of a second is over 1 billion USD per year. When users were asked whether they prefer 30 or 10 search results on Google, a majority favored more search results. However, when comparing both variants, Google measured a drop in traffic of 20% [Far06]. The decrease in engagement was caused by 500 ms of additional latency for the search query. This shows that browsing patterns heavily depend on performance, even if users are unaware of their own behavior [Mil68, Nie94, Mye85]. User expectations for performance are increasingly high. According to a survey of 116 companies conducted by the Aberdeen group, the average user satisfaction drops by 16% for every second of load time [Sim08]. 49% of users expect websites to load in 2 seconds or less, according to a survey by Akamai [Tec14]. These expectations are not matched in practice: a median top 500 e-commerce website has a page load time of 9.3 seconds [Eve14].

The wealth of studies [Eve16] shows that many business metrics as well as basic user behavior heavily depend on web performance. At the same time, websites and workloads continuously become more complex while the amount of processed and stored data increases. Additionally, more and more users access websites and services from unreliable mobile networks and different geographical locations. Performance therefore constitutes one of the central challenges of web technology.

To tackle the performance of application backends, cloud computing has emerged. The rise of cloud computing enables applications to leverage storage and compute resources from a large shared pool of infrastructure. The volume and velocity at which data is generated and delivered have led to the creation of NoSQL databases that provide scalability, availability, and performance for data-driven workloads. Combining these two technology trends as **cloud data management**, scalable database systems are now frequently deployed and managed through cloud infrastructures. While cloud data management supports various scalability requirements that have been impossible with deployments on-premises [LS13, ZSLB14], it introduces a performance problem. High latency between application users and cloud services is an inherent characteristic of the distributed nature of cloud computing and the web.

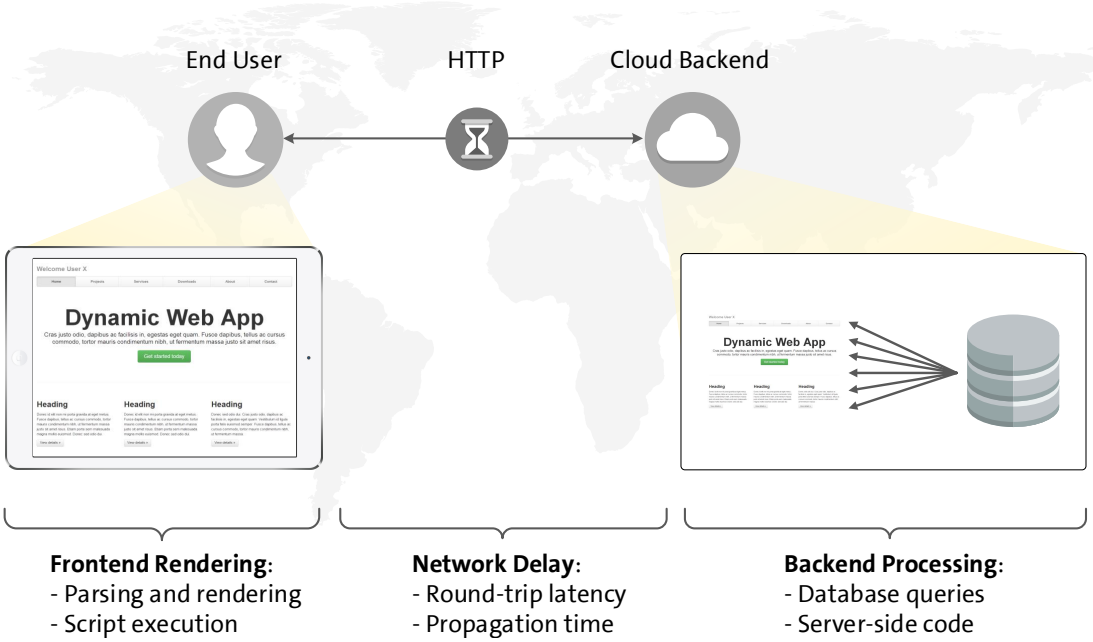


Figure 1.1: The three primary sources of latency and performance problems of web applications: frontend rendering, network delay, and backend processing.

Across the application stack, slow page load times have three sources, as illustrated in Figure 1.1. When a web page is requested, the first source of loading time is the **backend**. It consists of application servers and database systems and assembles the page. The latency of individual OLTP queries and the processing time for rendering HTML slow down the delivery of the site [TvS07].

The **frontend**, i.e., the page displayed and executed in the browser, is the second source of delay. Parsing of HTML, CSS, and JavaScript as well as the execution of JavaScript that can block other parts of the rendering pipeline contribute to the overall waiting time.

As of 2018, loading an average website requires more than 100 HTTP requests [Arc18] that need to be transferred over the **network**. This requires numerous round-trip times that are bounded by physical network latency. This third source of delay typically has the most significant impact on page load time in practice [Gri13].

Any performance problem in web applications can be allocated to these three drivers of latency. When a website is requested by a client, it is generated by the backend, thus causing processing time. The website's HTML is transferred to the browser and all included resources (e.g., scripts, images, stylesheets, data, queries) are requested individually causing additional network latency. Rendering and script execution in the client also contribute to overall latency.

Network bandwidth, client resources, computing power, and database technology have improved significantly in recent years [McK16]. Nonetheless, latency is still restricted by physical network round-trip times as shown in Figure 1.2. When network bandwidth increases, page load time does not improve significantly above 5 MBit/s for typical websites. However, if latency can be reduced, there is a proportional decrease in overall page load time. These results illustrate that cloud-based applications can only be accelerated through latency reduction. As requests cause latency at the network, backend, and database levels, an **end-to-end approach** for minimizing latency is required.

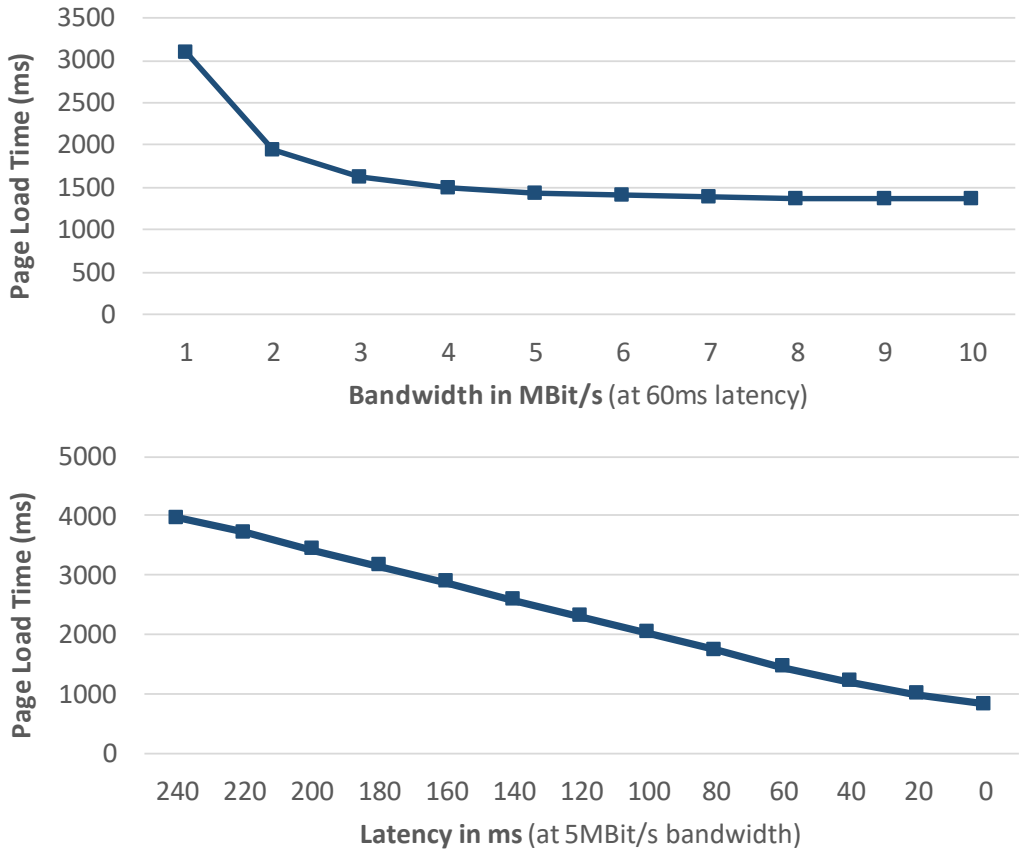


Figure 1.2: The dependency of page load time on bandwidth (data rate) and latency. For typical websites, increased bandwidth has a diminishing return above 5 MBit/s, whereas any decrease in latency leads to a proportional decrease in page load time. The data points were collected by Belshe [Bel10] who used the 25 most accessed websites.

The central goal of this thesis is to provide latency reduction for modern cloud data management to improve performance of applications. We therefore explore how latency can be reduced through caching and replication and study the related trade-offs between performance, scalability, and data freshness.

1.1 Problem Statement

The increasing adoption of cloud computing has led to a growing significance of latency for overall performance. Both users and different application components are now frequently separated by wide-area networks. **Database-as-a-Service** (DBaaS) and **Backend-as-a-Service** (BaaS) models allow storing data in the cloud to substantially simplify application development [CJP+11]. However, their distributed nature makes network latency critical [Coo13]. When clients (e.g., browsers or mobile devices) and application servers request data from a remote DBaaS or BaaS, the application is blocked until results are received from the cloud data center. As web applications usually rely on numerous queries for data, latency can quickly become the central performance bottleneck.

Fueled by the availability of DBaaS and BaaS systems with powerful REST/HTTP APIs for developing websites and mobile apps, the **single-page application architecture** gained popularity. In this two-tier architecture, clients directly consume data from cloud services without intermediate web and application servers as in three-tier architectures. Single-page applications allow more flexible frontends and facilitate the development process. In single-page applications, data is not aggregated and pre-rendered in the application server, but assembled in the client through many individual requests. Consequently, the number of latency-critical data requests is even higher in this architecture [Wag17].

The latency problem has previously been tackled mainly by **replication** [DHJ+07, CDG+08, Hba17, QSD+13, CRS+08, SPAL11, LFKA13, LFKA11] and **caching** techniques [LLXX09, PB03, DFJ+96, ABK+03, LGZ04, LKM+02, BAK+03] to distribute the database system and its data. The central limitation of prior work on replication and caching is a lack of generality: all solutions so far are tied to specific types of data or applications (e.g., static web content), trade read latency against higher write latency, or do not bound data staleness. Furthermore, latency and performance improvements for database systems do not solve the **end-to-end performance problem**. The core problem is that state-of-the-art database systems are not designed to be directly accessed by browsers and mobile devices as they lack the necessary abstractions for access control and business logic. Therefore, servers still need to aggregate data for clients and thus increase latency [FLR+14].

Our goal is to broaden the spectrum of techniques for low latency with an approach that is compatible with any cloud service serving dynamic data. To this end, the major problem to be solved is to efficiently replicate and cache data for low latency while exposing the appropriate tuning knobs to let applications configure consistency and freshness guarantees.

Improving the performance of mostly static data has a long history [GHa+96]. However, latency and consistency are particularly challenging for **dynamic data** that in contrast to static data can be modified arbitrarily and at any point in time. A typical website consists of some mostly static files, for example, scripts, stylesheets, images, and fonts. Web APIs, JSON data, and HTML files, on the other hand, are dynamic and therefore commonly considered uncacheable [LLXX09]. Dynamic data can have various forms depending on the type of the application and the underlying storage [Kle17]. The latency problem hence has to be addressed for both standard file- and record-based access based on a primary key or an identifier (e.g., a URL) as well as query results that offer a dynamic view of the data based on query predicates. As an example, consider an e-commerce website. For the website to load fast, files that make up the application frontend have to be delivered with low latency, e.g., the HTML page for displaying the shop's landing page. Next, data from the database systems also needs to be delivered fast, e.g., the state of the shopping cart or product detail information. And lastly, the performance of queries like retrieving recommended products, filtering the product catalog or displaying search results also heavily depends on latency.

Latency is not only problematic for end users, but it also has a detrimental effect on **transaction processing** [BBC+11, SVS+13, DAEA10, PD10, KPF+13, DFR15a, KKN+08, ZSS+15, DNN+15]. Many applications require the strong guarantees of transactions to preserve application invariants and correct semantics. However, both lock-based and optimistic concurrency control protocols have an abort probability that depends on the overall transaction duration [BN09, Tho98]. If individual operations are subject to high latency, the overall transaction duration is prolonged and consequently, the probability of a deadlock or conflict exhibits a superlinear increase [WV02]. Thus, in environments with high latency, the performance of transaction processing is determined by latency. This is for example the case if an end user is involved in the transaction (e.g., during the checkout in reservation system) or if the server runs the transaction against a remote DBaaS. Thus, to increase the effectiveness of transactions, low latency is required, too.

The complete ecosystem of data management is currently undergoing heavy changes. The unprecedented scale at which data is consumed and generated today has shown a large demand for scalable data management and given rise to non-relational, distributed **NoSQL database systems** [DHJ+07, CDG+08, Hba17, LM10, CD13, SF12, ZS17]. Two central problems triggered this process:

- vast amounts of user-generated content in modern applications and the resulting request loads and data volumes
- the desire of the developer community to employ problem-specific data models for storage and querying

To address these needs, various data stores have been developed by both industry and research, arguing that the era of one-size-fits-all database systems is over [SMA+07]. Therefore, these systems are frequently combined to leverage each system in its respective sweet

spot. **Polyglot persistence** is the concept of using different database systems within a single application domain, addressing different functional and non-functional needs with each system [SF12].

Complex applications need polyglot persistence to deal with a wide range of data management requirements. Until now, the overhead and the necessary know-how to manage multiple database systems prevent many applications from employing efficient polyglot persistence architectures. Instead, developers are often forced to implement one-size-fits-all solutions that do not scale well and cannot be operated efficiently. Even with state-of-the-art DBaaS systems, applications still have to choose one specific database technology [HIM02, CJP+11].

The rise of polyglot persistence [SF12] introduces two specific problems. First, it imposes the constraint that any performance and latency optimization must not be limited to only a single database system. Second, the heterogeneity and sheer amount of these systems make it increasingly difficult to select the most appropriate system for a given application. Previous research and industry initiatives have focused on solving specific problems by introducing new database systems or new approaches within the scope of specific, existing data stores. However, the problem of automatically selecting the most suitable systems and orchestrating their interaction is yet unsolved as is the problem of offering low latency for a polyglot application architecture.

Besides the problem of high network latencies, the applicability of database systems in cloud environments is considerably restricted by the lack of **elastic horizontal scalability** mechanisms and missing abstraction of storage and data models [DAEA13, SHKS15]. In today's cloud data management, most DBaaS systems offer their functionalities through REST APIs. Yet today, there has been no systematic effort on deriving a unified REST interface that takes into account the different data models, schemas, consistency concepts, transactions, access-control mechanisms, and query languages to expose cloud data stores through a common interface without restricting their functionality or scalability. A unified REST interface is a foundation for consolidating multiple storage systems in a scalable polyglot persistence architecture, as it abstracts implementation details of different data stores by working at the level of desired functional and non-functional requirements.

1.2 Challenges

Even with the combination of state-of-the-art work on NoSQL databases, geo-replication, and web technologies, four central challenges remain:

C_1 Latency of Dynamic Data: Web performance is governed by high round-trip latencies from browsers and mobile devices to remote cloud services for fetching dynamic data. Web caching in its prevailing form is incapable of dealing with dynamically changing files, objects, and query results.

C_2 Direct Client Access: Current replication and caching approaches for database systems only marginally improve end-to-end performance, because the abstractions for direct access by clients are missing. This prevents exposing full-fledged Database-as-a-Service systems to browsers and mobile devices and appropriate interfaces for data management, business logic, transactions, authentication, and authorization mechanisms are not available.

C_3 Transaction Abort Rates: As abort rates of transaction processing deteriorate steeply when reads and queries experience high latency, transactions are infeasible for many distributed scenarios. In state-of-the-art approaches, improved transaction performance is often achieved by relaxing transaction guarantees instead of providing better performance for strong isolation levels.

C_4 Polyglot Persistence: Polyglot persistence makes performance optimization and elastic scalability very difficult. Manual polyglot persistence introduces prohibitive management overhead for applications. Furthermore, choosing the most suitable data stores based on functional and non-functional requirements is a cumbersome and error-prone process.

The problem addressed in this thesis is latency reduction for cloud-based applications. In order to achieve low latency in a generic fashion, an end-to-end approach is required to speed up transactions and the delivery of files, database records, and query results from cloud databases and services, while maintaining high scalability and consistency. We therefore pose the following research question:

> **Research Question:** How can the latency of retrieving dynamic data from cloud services be minimized in an application- and database-independent way while maintaining strict consistency guarantees?

To address the above research question, we propose a caching methodology for low latency that caches dynamic data with well-defined consistency levels and is applicable to distributed, polyglot transactions (cf. C_1 and C_3). We devise and implement a Database/Backend-as-a-Service middleware that scales elastically and is capable of exposing database systems for direct, low-latency client access (cf. C_2). To satisfy complex data management requirements, we explore the concept of a Polyglot Persistence Mediator that is capable of orchestrating heterogeneous data stores (cf. C_4).

In the remainder of this chapter, we outline the key contributions of this work and present the structure of this thesis.

1.3 Primary Contributions

We believe that the challenges outlined above can be best solved using a comprehensive caching approach that exploits both existing database systems and wide-spread caching infrastructures. Today, to the best of our knowledge, no other approach is capable of

leveraging the web's expiration-based HTTP caching model and its globally distributed content delivery infrastructure for cloud data management.

This thesis completely relies on standard web caching to provide low-latency data access with rich consistency guarantees to solve the latency problem. Though discussed mainly in the context of Database- and Backend-as-a-Service applications, the method applies to any system serving dynamic data over a REST/HTTP-based interface.

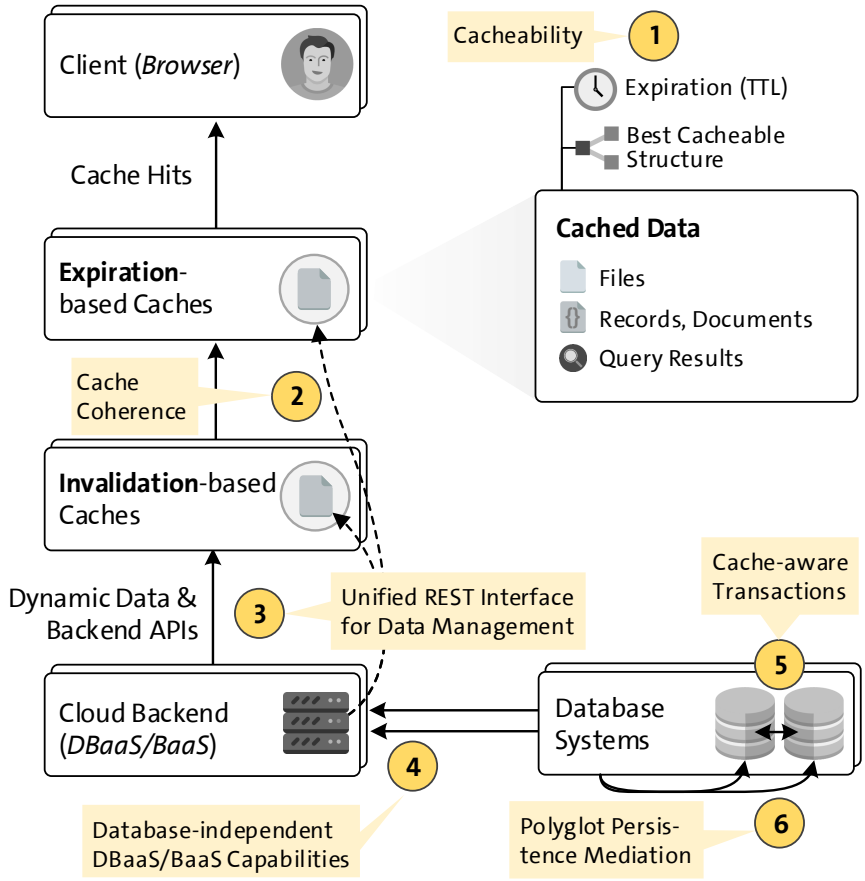


Figure 1.3: High-level contributions of this thesis: (1) and (2) are concerned with Challenge C_1, (3) and (4) with C_2, (5) with C_3, and (6) with C_4.

The primary contributions of this work are summarized in Figure 1.3 and described in more detail in the next three subsections. To cache dynamic content, a runtime decision on **cacheability** is required for each resource, to select data that lends itself to effective caching **(1)**. To utilize the entire web caching infrastructure, a **cache coherence** mechanism for all types of web caches needs to prevent clients from retrieving stale data by accident **(2)**. To make cacheable data directly available to clients for end-to-end latency reduction, a **unified REST interface** exposes abstractions for data management, e.g., queries containing protected data **(3)**. These **DBaaS and BaaS abstractions** are provided in database-independent fashion so that existing database systems are enhanced to work in a multi-tenant environment and with caching **(4)**. Furthermore, database systems that have no support for ACID transactions are provided with **optimistic, cache-aware transactions (5)**. To directly map requirements of applications to a set of databases, we

propose the **Polyglot Persistence Mediator** that routes operations and data to the most suitable system candidates **(6)**.

1.3.1 Web Caching of Records and Query Results with Rich Consistency Guarantees

To improve performance, cloud services need to minimize the delay of accessing data. In this thesis, we propose ORESTES, a comprehensive methodology and service architecture for automatic query, file, and record caching. The expiration-based web caching model gained little attention for data management in the past, as its static expirations (time-to-live) were considered irreconcilable with dynamic data that changes unpredictably. We propose a solution to this apparent contradiction by showing that clients can transparently maintain cache coherence. The main idea is to enable application-independent caching of query results and records with tunable consistency guarantees, in particular, bounded staleness.

We employ two key concepts to incorporate both expiration-based and invalidation-based web caches:

- A **Cache Sketch** data structure to indicate potentially stale data
- Statistically derived cache expiration times to maximize cache hit rates (**TTL estimation**)

The Cache Sketch captures stale data. By piggybacking the data structure at load time, clients can determine which requests can safely be directed to caches or trigger a revalidation if needed. At the same time, stale data is proactively purged from invalidation-based caches (e.g., content delivery networks and reverse proxy caches). To even cache query results, a distributed query invalidation pipeline detects changes to cached query results in realtime. Through an online decision model, the latency-optimal structure for each query result is determined.

The proposed caching algorithms offer a new means for data-centric cloud services to trade latency against staleness bounds, e.g., in a Database-as-a-Service. Besides bounded staleness, several other consistency guarantees like causal consistency or monotonic reads can be chosen at the granularity of operations, while leaving the option for strict consistency at the expense of cache hits. We provide empirical evidence for the scalability and performance of our approach through both simulation and experiments. The results indicate that for read-heavy workloads speed-ups by an order of magnitude can be achieved through our caching approach.

1.3.2 A Database-as-a-Service Middleware for Scalable Web Applications

This work motivates the design of a unified REST API and tackles the challenges of providing it in an extensible, scalable, and highly available fashion through a **middleware**

approach. To this end, we propose an architecture that consists of an independently scalable tier of HTTP servers that map the unified REST API to aggregate-oriented (NoSQL) data stores. The middleware extracts a wide range of DBaaS concerns (e.g., schema management, transactions, and access control) and provides them in a modular, database-independent fashion at the middleware level to support a broad range of application requirements.

Backend-as-a-Service is an extension of the DBaaS model that addresses the need for clients like browsers and mobile devices to query and update cloud databases directly. To allow this, fine-grained access control and integration of protected business logic are required. ORESTES enables any data store and DBaaS to be exposed as a Backend-as-a-Service by supporting these abstractions as a middleware.

We will provide evidence for two major advantages of solving DBaaS and BaaS through a middleware approach:

- **Elastic scalability and performance** can be addressed in a database-independent fashion through our proposed caching approach and workload-based auto-scaling.
- Many central, **functional application requirements** for BaaS and DBaaS can be easily added to existing data stores, including schema management, authentication, access control, real-time queries, ACID transactions, and business logic.

1.3.3 Polyglot Persistence Mediation with Database-Independent Caching and ACID Transactions

In this thesis, we present a novel solution for providing automated polyglot persistence based on service level agreements (SLAs). These SLAs are defined over functional and non-functional requirements of database systems. Therefore, we introduce the concept of the **Polyglot Persistence Mediator** (PPM) that employs runtime decisions on routing data to different backends according to schema-based annotations. The PPM enables applications to either use polyglot persistence right from the beginning or add new systems at any point with minimal overhead. For a typical polyglot persistence scenario, the PPM can improve write throughput by 50-100% while reducing both read and query latency drastically.

The mediation is orthogonal to the other concepts introduced in this thesis and can be combined with the Cache Sketch method. In particular, our cache-ware optimistic transactions support polyglot backends and provide ACID transactions across any set of included data stores that support linearizable updates. We believe that our proposed Polyglot Persistence Mediator is a major step towards controlling and leveraging the heterogeneity in the database landscape.

To determine a meaningful set of functional and non-functional data management requirements, we conduct an in-depth survey of existing data stores. We collect the key findings in the **NoSQL Toolbox** reasoning framework: most data stores are defined through a col-

lection of sharding, replication, storage, and query techniques that define the provided guarantees and functions. This NoSQL toolbox serves as the basis for SLAs in the PPM, where SLAs are attached to hierarchical schemas to allow a ranking of available systems. Furthermore, the toolbox also allows a fine-grained classification of NoSQL databases and serves as a decision guidance for the selection of appropriate system candidates.

1.4 Outline and Previously Published Work

The remainder of this dissertation proceeds as follows. In Chapter 2, we discuss important concepts of cloud data management and the role of caching for web applications. To this end, we examine the backend, network, and frontend with their respective architectures and technologies. For each of the three tiers, we specifically highlight the impact and sources of latency that contribute to end-to-end performance.

In Chapter 3, we outline how low latency can be provided through a cloud data management approach. First, we present a novel database classification scheme that relates functional and non-functional application requirements to database system techniques. We then motivate the ORESTES Database-as-a-Service architecture to solve fundamental data management requirements in a database-independent fashion, while accounting for direct client access.

In Chapter 4, we present the key contribution of this thesis: a generic approach for web caching of records and queries with rich consistency guarantees. We start by introducing a caching scheme for database records and files and show how various levels of consistency can be reached at very low latency. We then extend the approach to caching arbitrary query results. Last, we apply caching to address abort rates of distributed transactions. We provide experimental evidence for the effectiveness of the Cache Sketch approach for each of the scenarios.

In Chapter 5, we introduce the vision of a Polyglot Persistence Mediator that combines the ideas of this work with ongoing research. We begin with an approach for annotating data models with requirements. Next, we explore how the requirements can automatically be mapped to different systems through routing queries and updates to systems at runtime. We illustrate the potential effect of such a Polyglot Persistence Mediator by evaluating a typical application example.

In Chapter 6, we discuss related work. We give a detailed comparison of this thesis to caching and geo-replication approaches from the literature and discuss similarities, differences, and trade-offs. Also, we elucidate how transaction processing, Database-as-a-Service, and polyglot persistence approaches relate to the challenges addressed in this work.

In Chapter 7, we summarize this thesis and its main contributions, discuss opportunities for future work, and conclude.

This dissertation revises material from previous publications in Chapter 2 (cf. [GWFR16, GSW+17, GSW+15]), Chapter 3 (cf. [GWFR16, GBR14, GR15a, GSW+17, GFW+14]), Chapter 4 (cf. [GSW+17, GSW+15]), Chapter 5 (cf. [SGR15]), and Chapter 6 (cf. [GSW+15, GFW+14, GSW+17]).

1.5 List of Own Publications

The work presented in this thesis has produced the following publications:

[SKE+18] Michael Schaarschmidt, Alexander Kuhnle, Ben Ellis, Kai Fricke, Felix Gessert, and Eiko Yoneki. LIFT: Reinforcement Learning in Computer Systems by Learning From Demonstrations. arXiv preprint arXiv:1808.07903 (under submission), 2018.

[WRG18] Wolfram Wingerath, Norbert Ritter, and Felix Gessert. Real-Time & Stream Data Management: Push-Based Data in Research & Practice. Springer, book to be published in late 2018.

[WGW+18] Wolfram Wingerath, Felix Gessert, Erik Witt, Steffen Friedrich, and Norbert Ritter. Real-time Data Management for Big Data. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*. OpenProceedings.org, 2018.

[GSW+17] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Erik Witt, Eiko Yoneki, and Norbert Ritter. Quaestor: Query Web Caching for Database-as-a-Service Providers. *Proceedings of the VLDB Endowment*, 2017.

[GWR17] Felix Gessert, Wolfram Wingerath, and Norbert Ritter. Scalable Data Management: An In-Depth Tutorial on Nosql Data Stores. In *BTW (Workshops)*, volume P-266 of *LNI*, pages 399–402. GI, 2017.

[WGF+17] Wolfram Wingerath, Felix Gessert, Steffen Friedrich, Erik Witt, and Norbert Ritter. The Case for Change Notifications in Pull-Based Databases. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband, 2.-3. März 2017, Stuttgart, Germany*, 2017.

[GR17] Felix Gessert and Norbert Ritter. SCDM 2017 - Vorwort. In *BTW (Workshops)*, volume P-266 of *LNI*, pages 211–213. GI, 2017.

[Ges17] Felix Gessert. Lessons Learned Building a Backend-as-a-Service. *Baqend Tech Blog*, May 2017. (Accessed on 08/11/2017).

[GWFR16] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. NoSQL Database Systems: A Survey and Decision Guidance. *Computer Science - Research and Development*, November 2016.

[GR16] Felix Gessert and Norbert Ritter. Scalable Data Management: NoSQL Data Stores in Research and Practice. In *32nd IEEE International Conference on Data Engineering, ICDE*, 2016.

[SG16] Michael Schaarschmidt and Felix Gessert. Learning Runtime Parameters in Computer Systems with Delayed Experience Injection. In *Deep Reinforcement Learning Workshop, NIPS*, 2016.

[WGFR16] Wolfram Wingerath, Felix Gessert, Steffen Friedrich, and Norbert Ritter. Real-Time Stream Processing for Big Data. *it - Information Technology*, 58(4), January 2016.

[GSW+15] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. The Cache Sketch: Revisiting Expiration-based Caching in the Age of Cloud Data Management. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"*. GI, 2015.

[GR15a] Felix Gessert and Norbert Ritter. Polyglot Persistence. *Datenbank-Spektrum*, 15(3):229–233, November 2015.

[GR15b] Felix Gessert and Norbert Ritter. Skalierbare NoSQL- und Cloud-Datenbanken in Forschung und Praxis. In *Datenbanksysteme für Business, Technologie und Web (BTW 2015) - Workshopband, 2.-3. März 2015, Hamburg, Germany*, pages 271–274, 2015.

[Ges15] Felix Gessert. Low Latency Cloud Data Management through Consistent Caching and Polyglot Persistence. In *Proceedings of the 9th Advanced Summer School on Service Oriented Computing*, 2015.

[SGR15] Michael Schaarschmidt, Felix Gessert, and Norbert Ritter. Towards Automated Polyglot Persistence. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"*, 2015.

[WFGR15] Wolfram Wingerath, Steffen Friedrich, Felix Gessert, and Norbert Ritter. Who Watches the Watchmen? On the Lack of Validation in NoSQL Benchmarking. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"*, 2015.

[GBR14] Felix Gessert, Florian Bücklers, and Norbert Ritter. ORESTES: a Scalable Database-as-a-Service Architecture for Low Latency. In *CloudDB, Data Engineering Workshops (ICDEW)*, pages 215–222. IEEE, 2014.

[GFW+14] Felix Gessert, Steffen Friedrich, Wolfram Wingerath, Michael Schaarschmidt, and Norbert Ritter. Towards a Scalable and Unified REST API for Cloud Data Stores. In Erhard Plödereder, Lars Grunske, Eric Schneider, and Dominik Ull, editors, *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, volume 232 of *LNI*, pages 723–734. GI, 2014.

[FWGR14] Steffen Friedrich, Wolfram Wingerath, Felix Gessert, and Norbert Ritter. NoSQL OLTP Benchmarking: A Survey. In Erhard Plödereder, Lars Grunske,

Eric Schneider, and Dominik Ull, editors, *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, volume 232 of *LNI*, pages 693–704. GI, 2014.

[GB13] Felix Gessert and Florian Bücklers. ORESTES: ein System für horizontal skalierbaren Zugriff auf Cloud-Datenbanken. In *Informatiktage*. GI, March 2013.

2 Background

In this chapter, we describe the technical foundations of scalable, cloud-based web applications and discuss core challenges and requirements for cloud data management. We focus on the performance-relevant areas addressed in this thesis. These can be grouped into the three categories backend, network, and frontend.

Backend performance subsumes data management and scalable server architectures. We give an overview of NoSQL database systems and their use for Database- and Backend-as-a-Service cloud service models. Backend performance is directly related to polyglot persistence and the availability-consistency trade-offs that are associated with the use of sharding and replication techniques.

Network performance of web applications is determined by the design of the HTTP protocol and the constraints of the predominant REST architectural style. As the basic building block of this work, we will review the mechanisms that HTTP provides for web caching and how they relate to the infrastructure of the Internet.

Last, we also give an introduction to **frontend performance**, to show that it is an orthogonal problem to the scope of the performance optimizations introduced by this thesis. We cover sufficient background to follow and motivate the approach of this thesis. For a detailed treatment and differentiation from the state-of-the-art in related work, please refer to Chapter 6.

2.1 End-to-End Latency in Cloud-based Architectures

The continuous shift towards cloud computing has established two primary architectures: two-tier and three-tier applications. Both architectures are susceptible to latency at different levels. The concrete realization can build upon different cloud models, in particular, Database/Backend-as-a-Service, Platform-as-a-Service, and Infrastructure-as-a-Service [YBDS08].

Modern web applications need to fulfill several non-functional requirements:

- **High availability** guarantees that applications remain operational despite failure conditions such as network partitions, server failures, connectivity issues and human error.

- **Elastic scalability** enables applications to handle any growth and decrease in load (e.g., user requests and data volume), by automatically allocating or freeing storage and computing resources in a distributed cluster.
- **Fast page loads** and response times are essential to maximize user satisfaction, traffic, and revenue.
- An **engaging user experience** significantly helps to make users productive and efficient.
- A **fast time-to-market** is the result of the appropriate development, testing and deployment abstractions to quickly release an application to production[1].

The Orestes caching methodology spans several layers of the architecture. Therefore, we discuss the three- and two-tier architecture in the context of the above requirements, before examining the technical foundations of the backend, network, and frontend.

2.1.1 Three-Tier Architectures: Server-Side Rendering

The three-tier architecture is a well-known pattern for structuring client-server applications [TvS07, FLR+14, HW03]. The idea is, to segregate application concerns into three different functional tiers (components). This has the advantage that tiers are loosely coupled, thus facilitating easier development. Furthermore, each tier can be scaled independently based on required resources. The canonical tiers are the **presentation tier**, the **business logic tier** and the **data tier**. In the literature, different definitions of three-tier architectures are used. Tanenbaum and van Steen [TvS07] differentiate between web servers, application servers and database servers as three different tiers of a web application. Fehling et al. [FLR+14] argue that web and application servers are typically just one tier, whereas in a real three-tier application, the presentation tier is completely decoupled from the business logic tier, e.g., by message queues.

We will distinguish between the two-tier and three-tier architecture based on the location of the presentation tier. As shown in Figure 2.1, the classic three-tier architecture includes the presentation layer as part of the backend application. This means that an application or web server executes the presentation and business logic while the data tier serves and stores data using one or more database systems. The client's browser is served the rendered representation, typically in the form of an HTML file and supporting stylesheets (CSS) and JavaScript files (JS). As the client does not execute any significant portion of the presentation and business logic, this architecture is also referred to as a *thin client* architecture. Any user interactions that require business logic (e.g., posting a comment on a social network) are forwarded to the server tiers, which are responsible for performing the desired task. This usually implies the server-rendering of a new HTML view representing

[1] Despite all recent advances in programming languages, tooling, cloud platforms, and frameworks, studies indicate that over 30% of all web projects are delivered late or over-budget, while 21% fail to meet their defined requirements [Kri15].

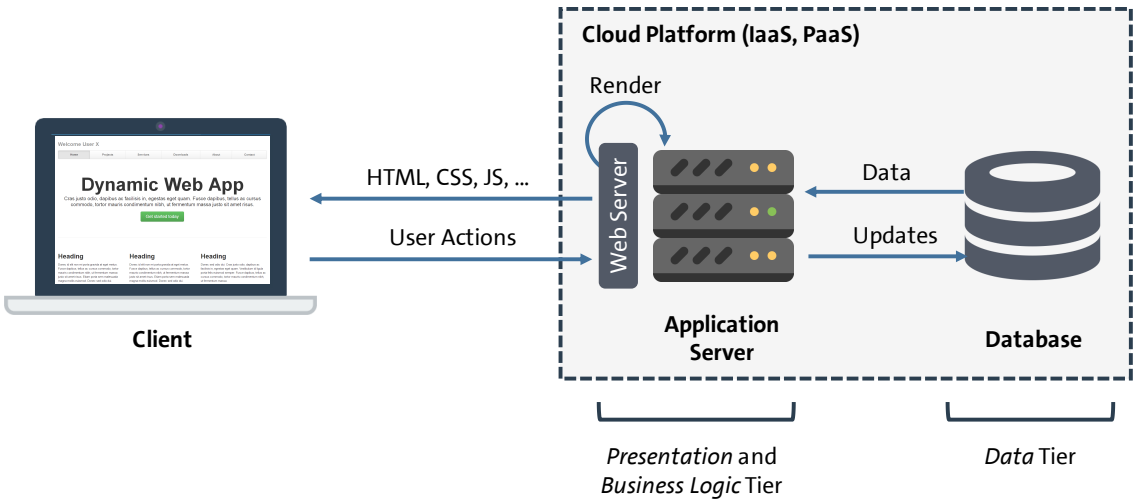


Figure 2.1: The three-tier web application architecture.

a response to the invoked action. An advantage of separating the data tier and business logic tier is that business logic can be stateless and scales efficiently.

Flow

The high-level request flow in a server-rendered three-tier architecture is the following (cf. [FLR+14]):

1. The client requests the website over the HTTP protocol.
2. The web server accepts the request and calls the components for handling the corresponding URL. Usually, the web server is not requested directly, but a load balancer distributes requests over available web servers. The request can be directly executed in the web server (e.g., in PHP) or invoked over the network (e.g., through AJP) or using a queuing system (e.g., RabbitMQ) [Cha15].
3. In the application server, the business logic is executed.
4. Any data required to render the current view is queried from the database and updates are applied to reflect the application state.
5. The response is sent to the client as an HTML document. The web server directly answers subsequent requests for static resources like images and scripts.

Implementation

As a large part of the web uses three-tier architectures, a considerable amount of environments and frameworks for developing and hosting three-tier applications exist. In the context of cloud computing, three-tier architectures can be implemented on **Infrastructure-as-a-Service** (IaaS) and **Platform-as-a-Service** (PaaS) clouds [HDF13, MB16].

PaaS cloud providers such as Windows Azure [Azu17], Google App Engine [App17], and Heroku [Clo17b] offer managed operating systems, application servers, and middleware for running web applications in a scalable fashion. While the provider prescribes the

runtime environment (e.g., supporting Python applications), the application logic can be freely defined. The PaaS abstracts from maintenance and provisioning of operating systems and servers to unburden the application from operational aspects such as scaling, system upgrades, and network configuration. It therefore provides a useful paradigm for the development of three-tier applications. For example, Windows Azure [Azu17] has a built-in notion of the three tiers, as it distinguishes between web roles (the presentation tier), storage services (the data tier) and worker roles (the business logic tier). Web roles and worker roles are scaled independently and decoupled by storage abstractions such as queues, wide-column models, and file systems [CWO+11].

In the IaaS model, full control over virtual machines is left to the tenant. This implies that three-tier architectures can use the same technology stacks as applications in non-cloud environments (**on-premises**). For example, Amazon Web Services (AWS) and [Ama17b] Google Cloud Platform (GCP) [Goo17a] provide the management infrastructure to provision individual virtual machines or containers that can run arbitrary software for each tier in the architectures. Typically a web server (e.g., Apache, IIS, or Nginx [Ree08]), application server (e.g., Tomcat or Wildfly [Wil17]) or reverse proxy (e.g., Varnish [Kam17]) is combined with a web application framework in a particular programming language running the business logic and parts of the presentation tier (e.g., Python with Django, Java with Spring MVC, or Ruby with Sinatra [The17, Wal14]). The business logic tier in turn either employs a database system also hosted on the IaaS provider or connects to Database-as-a-Service offerings to persist and retrieve data.

The **microservice architecture** is a refinement of the three-tier architecture that decomposes the three tiers of the backend [New15, NMMA16]. The central idea of microservices is to decompose the application into functional units that are loosely coupled and interact with each other through REST APIs. Microservices thus offer a light-weight alternative to service-oriented architectures (SOA) and the WebService standards [ACKM04]. In contrast to three-tier applications, microservices do not share state through a data tier. Instead, each microservice is responsible for separately maintaining the data it requires to fulfill its specified functionality. One of the major reasons for the adoption of microservices is that they allow scaling the development of large distributed applications: each team can individually develop, deploy and test microservices as long as the API contracts are kept intact. When combined with server-rendering, i.e., the generation of HTML views for each interaction in a web application, microservices still exhibit the same performance properties as three-tier architectures. Some aspects even increase in complexity, as each microservice is a point of failure and response times for answering a request through aggregation from multiple microservice responses are subject to latency stragglers.

Problems of Server-Rendered Architectures

Three-tier and service architectures with a server-side presentation tier pose different problems with respect to the introduced non-functional requirements (see Section 2.1).

High Availability. As all tiers depend upon the data tier for shared state, the underlying database systems have to be highly available. Any unavailability in the data tier will propagate to the other tiers, thus amplifying potential partial failures into application unavailability.

Elastic Scalability. All tiers need to be independently and elastically scalable, which can induce severe architectural complexity. For instance, if requests passed from the presentation tier to the business logic tier exceed the capacities of the business logic tier, scaling rules have to be triggered without dropping requests. Alternatively, non-trivial backpressure (flow control) mechanisms [Kle17] have to be applied to throttle upstream throughput. In practice, tiers are often decoupled through message queues, which – similar to database systems – have inherent availability-consistency-performance trade-offs.

Fast Page Loads. Server-rendering implies that the delivery of a response is blocked until the slowest service or query returns which hinders fast page loads. Even if each query and service produces a low average or median response time, the aggregate response times are governed by extreme value distributions that have a significantly higher expected value [WJW15, VM14]. While the request is blocked, the client cannot perform any work as the initial HTML document is the starting point for any further processing in the browser and for subsequent requests. Of the potentially hundreds of requests [Arc18], each is furthermore bounded by *network latency* that increases with the distance to the server-side application logic.

Engaging User Experience. As each user interaction (e.g., navigation or submitting a form) produces a new HTML document, the indirection between the user's interactions and observed effects become noticeable. A well-studied result from psychology and usability engineering is that for the user to gain the impression of directly modifying objects in the user interface, response times have to be below 100 ms [Mil68, Nie94, Mye85]. Even if the delivery of static assets is fast, rendering an HTML document, applying updates to the database and performing relevant queries is usually infeasible if any significant network latency is involved. For users, this conveys the feeling of an unnatural, indirect interaction pattern [Nie94].

Fast Time-to-Market. Besides the above performance problems, server-side rendering also induces problems for the software development process. All user interactions need to be executed on the server. In modern web applications, the user interface has to be engaging and responsive. Therefore, parts of the presentation logic are replicated between the server-side presentation tier and the JavaScript logic of the frontend. This duplicates functionality, increasing development complexity and hindering maintainability. Furthermore, by splitting the frontend from the server-side processing, unintended interdependencies arise: frontend developers or teams have to rely on the backend development to proceed, in order to work on the design and structure of the frontend. This hinders agile, iterative development methodolo-

gies such as Scrum [SB02] and Extreme Programming (XP) [Bec00] from being applied to frontend and backend teams separately. As applications shift towards more complex frontends, the coupling of frontend and backend development inevitably increases time-to-market.

2.1.2 Two-Tier Architectures: Client-Side Rendering

To tackle the problems of rigid three-tier architectures, the two-tier architecture evolved [FLR+14]. By two-tier architectures, we will refer to applications that shift the majority of presentation logic into the client. Business logic can be shared or divided between client and server, whereas the data tier resides on the server, to reflect application state across users. The two-tier model is popular for native mobile applications, that are fundamentally based on the user interfaces components offered by the respective mobile operating system (iOS, Windows, Android) and packaged into an installable app bundle [Hil16]. Many web applications also follow this model and are referred to as single-page applications, due to their ability to perform user interactions without loading a new HTML page [MP14]. We will discuss the two-tier architecture in the context of web applications, but most aspects also apply to native mobile apps.

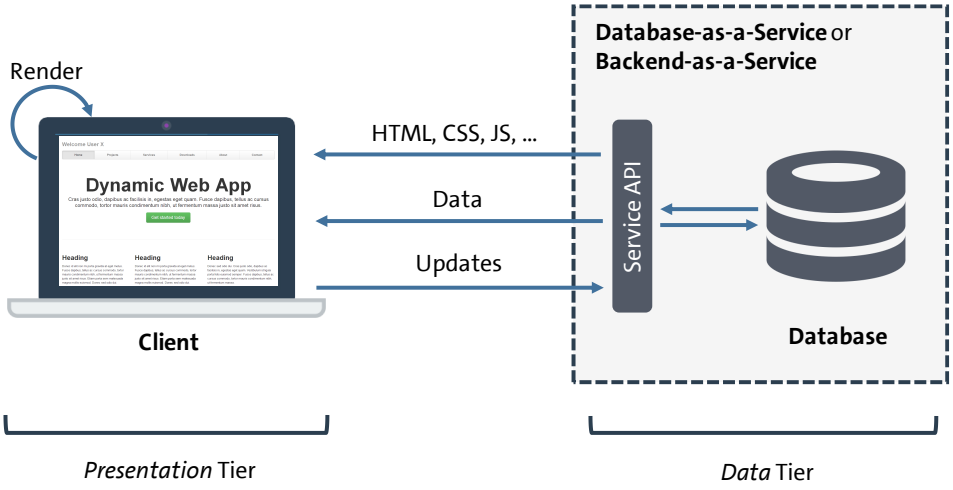


Figure 2.2: The two-tier web application architecture.

The two-tier architecture is illustrated in Figure 2.2. Rendering in the client is performed through the browser's JavaScript runtime engine that consumes structured data directly from the server (e.g., product detail information), usually in the form of JSON[2] [Cro06]. The data tier is therefore responsible for directly serving database objects and queries to clients. The business logic tier is optional and split into unprotected parts directly exe-

[2]The JavaScript Object Notation (JSON) is a self-contained document format, consisting of objects (key-value pairs) and arrays (ordered lists), that can be arbitrarily nested. JSON has gained popularity due to its simpler structure compared to XML. It can be easily processed in JavaScript and thus became the widely-used format for document databases such as MongoDB [CD13], CouchDB [ALS10], Couchbase [LMLM16], and Espresso [QSD+13] to reduce the impedance mismatch [Mai90].

cuted in the client and parts that require confidentiality, security and stricter control and are therefore executed co-located with the data tier. Server-side business logic includes enforcing access control, validating inputs, and performing any protected business logic (e.g., placing an order in an e-commerce shop). Actions carried out by the client can be directly modeled as update operations on the database, with a potential validation and rewriting step enforced by the server.

Request Flow

The request flow in two-tier web application architectures is slightly different from three-tier architectures:

1. With the initial request, the client retrieves the HTML document containing the single-page application logic.
2. The server or cloud service returns the HTML document and the accompanying JavaScript files. In contrast to server-rendered architectures, the frontend's structure is **data-independent** and therefore does not require any database queries or business logic.
3. The client evaluates the HTML and fetches any referenced files, in particular, the JavaScript containing the presentation logic.
4. Via JavaScript, the data required to display the current application view are fetched from the server via a REST/HTTP[3] API either in individual read operations or using a query language (e.g., MongoDB [CD13] or GraphQL [Gra17]).
5. The frontend renders the data using the presentation logic of the JavaScript frontend, typically expressed through a template language.
6. User interactions are sent as individual requests and encode the exact operation performed. The response returns the data necessary to update the frontend accordingly.

Implementation

For the realization of two-tier architectures, the technology choices for three-tier architectures also apply. IaaS and Paas offer low-level abstractions for building REST APIs consumed by single-page applications. Most web application frameworks have support for developing not only server-rendered HTML views, but also for structuring REST APIs. In the Java ecosystem, REST interfaces have been standardized [HS07]. In most other web languages such as (server-side) JavaScript (Node.js), Ruby, Python, and PHP, frameworks employ domain-specific languages or method annotations for minimizing the overhead of defining REST endpoints (e.g., in Ruby on Rails, Django, .NET WCF, Grails, Express, and the Play framework [WP11, The17]). Static files of single-page applications are delivered from a web server, the web application framework, or a content delivery network. The REST APIs are consumed by the frontend that is technologically in-

[3] Besides HTTP, real time-capable protocols like Web Sockets, Server-Sent Events (SSE), or WebRTC can be employed [Gri13].

dependent of the backend and only requires knowledge about the REST resources to implement client-server interactions. One notable exception is the idea of *isomorphic* (also called universal) JavaScript that applies the concept of sharing code (e.g., validation of user inputs) between a frontend and backend that are both implemented in JavaScript [HS16, Dep17, Hoo17, Par17].

Database-as-a-Service (DBaaS) and Backend-as-a-Service (BaaS) models provide high-level abstractions for building and hosting two-tier applications. In the case of a DBaaS, the data tier is directly exposed to clients. As this is insufficient if protected business logic or access control are required, BaaS systems extend the data APIs with common building blocks for business logic in single-page applications. Typical BaaS APIs and functionalities consumed in two-tier applications are:

- Delivery of static files, in particular, the single-page application assets
- DBaaS APIs for access to structured data
- Login and registration of users
- Authorization on protected data
- Execution of server-side business logic and invocation of third-party services
- Sending of push notifications
- Logging and tracking of user data

In Section 2.2.6 we will discuss the characteristics of the DBaaS and BaaS models in detail.

As the frontend becomes more complex and handles the presentation logic and significant parts of the business logic, appropriate tooling and architectures gained relevance. Therefore, numerous JavaScript frameworks for developing and structuring single-page applications have been developed. A large part of these frameworks is based on the Model-View-Controller (MVC) pattern [KP+88] or variants thereof (e.g., Model-View-ViewModel [Gos05]). In client-side MVC architectures, the views generate the document visible to the end user, usually by defining a template language. The model contains the data displayed in the views, so that it embodies both application state and user interface state. A model is filled with data retrieved from the server's data APIs. Controllers handle the interaction between views and models (e.g., events from user inputs) and are responsible for client-server communication. The MVC pattern has been adopted by most widely-used JavaScript frameworks such as Angular [Ang17], Ember [Emb17], Vue [Vue17], and Backbone [Bac17]. Recently, component-based architectures have been proposed as an alternative to MVC frameworks through projects such as Facebook's React [Rea17]. Components represent views, but also encompass event handling and user interface state.

In contrast to two-tier applications, any technological decisions made in the frontend are largely independent of the backend, as a REST API is the only point of coupling. Some frontend frameworks additionally offer server-side tooling to pre-render client views. This can improve the performance of the initial page load and is necessary for crawlers of

search engines that do not evaluate JavaScript for indexing. In native mobile applications, the same principles as for single-page applications apply. A major architectural difference is that the frontend is compiled ahead-of-time so that its business and presentation logic can only be changed with an explicit update of the app. Furthermore, static files are usually not provided by the backend, but packaged into an installable app bundle, which shifts the problem of initial load times to both client-side performance and latency of the consumed server APIs.

Problems of Client-Rendered Architectures

Two-tier architectures can improve on several of the difficulties imposed by three-tier architectures, while other non-functional requirements remain challenging:

High Availability and Elastic Scalability. The task of providing high availability with elastic scaling is shifted to the BaaS or DBaaS backend. As these systems employ a standard architecture shared between all applications built on them, availability and scalability can be tackled in a generic, application-independent fashion. As a DBaaS/BaaS is a managed service, it can furthermore eliminate availability and scalability problems introduced by operational errors such as flawed deployments, inappropriate autoscaling rules, or incompatible versions.

Fast Page Loads. Navigation inside a single-page application is fast, as only the missing data required for the next view is fetched, instead of reloading the complete page. On the other hand, data requests become very latency-critical, as the initial page load depends on data being available for client-side rendering. In two-tier applications, the client can start its processing earlier as there is no initial HTML request blocked in the server by database queries and business logic.

Engaging User Experience. Single-page applications are able to achieve a high degree of interactivity, as much of the business logic can be directly executed on the client. This allows applying updates immediately to remain under the critical threshold of 100 ms for interaction delays.

Fast Time-to-Market. As the frontend and backend are loosely coupled through a REST API and based on different technology stacks, the development process is accelerated. The implementation of the frontend and backend can proceed independently, enabling individual development, testing and deployment cycles for a faster time-to-market.

In summary, many applications are moving towards client-rendered, two-tier architectures, to improve the user experience and development process. This shift reinforces the requirement for low latency, as data transferred from the server to the client is critical for fast navigation and initial page loads.

2.1.3 Latency and Round-Trip Time

Two primary factors influence network performance: latency and bandwidth [Gri13]. **Latency** refers to the time that passes from the moment a packet or signal is sent from a source to the moment it is received by the destination. **Bandwidth** refers to the throughput of data transfer for a network link. We will use the wide-spread term bandwidth (measured in Megabit per second; MBit/s) throughout this thesis, though the formal term **data rate** (or transmission rate) is more precise, as bandwidth in signal theory defines the difference between an upper and lower frequency [Cha15].

Network packets sent from one host to another host travel through several routers and are nested in different network protocols (e.g., Ethernet, IP, TCP, TLS, HTTP). There are different delays at each hop that add up to the end-to-end latency [KR10]:

Processing Delay (d_{proc}). The time for parsing the protocol header information, determining the destination of a packet and calculating checksums determines the processing delay. In modern networks, the processing delay is in the order of microseconds [Cha15].

Queuing Delay (d_{queue}). Before a packet is sent over a physical network link, it is added to a queue. Thus, the number of packets that arrived earlier defines for how long a packet will be queued before transmission over the link. If queues overflow, packets are dropped. This packet loss leads to increased latency as the network protocols have to detect the loss and resend the packet[4].

Transmission Delay (d_{trans}). The transmission delay denotes the time for completely submitting a packet to the network link. Given the size of a packet S and the link's bandwidth, resp. transmission rate R, the transmission delay is S/R. For example, to transfer a packet with $S = 1500\,\mathrm{B}$ over a Gigabit Ethernet with $R = 1\,\mathrm{Gb/s}$ a transmission delay of $d_{trans} = 12\,\mu\mathrm{s}$ is incurred.

Propagation Delay (d_{prop}). The physical medium of the network link, e.g., fiber optics or copper wires, defines how long it takes to transfer the signal encoding the packet to the next hop. Given the propagation speed of the medium in m/s and the distance between two hops, the propagation delay can be calculated.

If a packet has to pass through $N-1$ routers between the sender and receiver, the end-to-end latency L is defined through the average processing, queuing, transmission and propagation delays [KR10]:

$$L = N \cdot (d_{proc} + d_{queue} + d_{trans} + d_{prop}) \tag{2.1}$$

[4] The large buffer sizes can also lead to a problem called *buffer bloat* in which queues are always operating at their maximum capacity. This is often caused by TCP congestion algorithms that increase throughput until package loss occurs. With large queues, many packets can be buffered and delayed before a packet loss occurs, which negatively impacts latency [APB09, Gri13]

Latency (also called *one-way latency*) is unidirectional, as it does not include the time for a packet to travel back. **Round-trip time** (RTT) on the other hand, measures the time from the source sending a request until receiving a response. RTT therefore includes the latency in each direction and the processing time d_{server} required for generating a response:

$$RTT = 2 \cdot L + d_{server} \tag{2.2}$$

In most cases, the propagation delay will play the key role in latency, as networking infrastructure has improved many aspects of queuing, transmission and processing delay significantly. However, propagation delay depends on the constant speed of light and the geographic distance between two hosts. For example, the linear distance between Hamburg and San Francisco is 8 879 km. Given an ideal network without any delays except the propagation at the speed of light (299 792 458 m/s), the minimum achievable latency is $L \approx 29.62\,\text{ms}$ and round-trip time $RTT \approx 59.23\,\text{ms}$. Therefore, to reduce end-to-end latency, distances have to be shortened.

We will discuss the effects of network protocols such as HTTP and TLS on end-to-end latency in Section 2.3. Grigorik [Gri13] gives an in-depth overview of latency and network protocols specifically relevant for the web. Kurose and Ross [KR10], as well as Forouzan [For12] discuss the foundations of computer networking. Van Mieghem [VM14] provides a formal treatment of how networks can be modeled, analyzed and simulated stochastically.

2.1.4 Cloud Computing as a Source of Latency

Besides the two-tier and three-tier architecture, there are numerous other ways to structure applications [FLR+14]. Cloud computing is quickly becoming the major backbone of novel technologies across application fields such as web and mobile applications, Internet of Things (IoT), smart cities, virtual and augmented reality, gaming, streaming, data science, and Big Data analytics. Cloud computing delivers on the idea of **utility computing** introduced by John McCarthy in 1961 that suggests that computing should be a ubiquitous utility similar to electricity and telecommunications [AG17]. In the context of cloud computing, there are several sources of latency across all types of application architectures. In this section, we will summarize the architecture-independent latency bottlenecks that contribute to the overall performance of cloud-based applications.

In the literature, cloud computing has been defined in various different ways [LS13, YBDS08, MB16, FLR+14, BYV+09, MG09, TCB14]. Throughout this thesis, we will use the widely accepted NIST definition [MG09]. It distinguishes between five characteristics of cloud offerings and groups them into three service models and four deployment models. The nature of the service and deployment models motivates, why latency is of utmost relevance in cloud computing.

Characteristics

The characteristics of cloud offerings explain how cloud computing is desirable for both customers and providers. Providers offer *on-demand self-service*, which means that consumers can provision services and resources in a fully automated process. *Broad network access* enables the cloud services to be consumed by any client technology that has Internet access. Cloud providers apply *resource pooling* (multi-tenancy) to share storage, networking, and processing resources across tenants to leverage economies of scale for reduced costs. *Rapid elasticity* demands that resources can be freed and allocated with minimal delay, building the foundation for scalability. The provider exposes a *measured service* that is used for pay-per-use pricing models with fine-grained control, monitoring and reporting of resource usage to the consumer. In practice, the major reasons for companies to adopt cloud computing is the ability to replace capital expenditures (CAPEX) that would have been necessary to acquire hardware and software into operational expenditures (OPEX) incurred by the usage of pay-per-use cloud services. The major incentive for providers is the ability to exploit economies of scale and accommodate new business models.

Service Models

Based on increasing degree of abstraction, three high-level service models can be distinguished:

Infrastructure-as-a-Service (IaaS). In an IaaS cloud, low-level resources such as computing (e.g., containers [Mer14] and virtual machines [BDF+03]), networking (e.g., subnets, load balancers, and firewalls [GJP11]) and storage (e.g., network-attached storage) can be provisioned. This allows deploying arbitrary applications in the cloud while leaving control of the infrastructure to the IaaS provider. In IaaS clouds, latency is particularly relevant for cross-node communication, potentially across different data centers (e.g., between an application server and a replicated database). Example offerings are Amazon Elastic Compute Cloud (EC2) [Ama17b], Softlayer [Sof17], Joyent [Joy17], and Google Compute Engine (GCE) [Goo17a].

Platform-as-a-Service (PaaS). Consumers of PaaS clouds run applications on a technology stack of services, programming languages, and application platforms defined by the provider including explicit support for developing, testing, deploying and hosting the application. In addition to the infrastructure, a PaaS provider also manages operating systems and networks. The role of latency in a PaaS is critical: as there is no control over native computing and storage resources, data management has to be consumed as a service either from the same provider or an external DBaaS. Examples of PaaS vendors are Microsoft Azure [Azu17], Amazon Beanstalk [AWS17], IBM Bluemix [IBM17], Google App Engine [App17], and Heroku [Clo17b].

Software-as-a-Service (SaaS). A SaaS provides a specific cloud-hosted application to users (e.g., email, word processors, spreadsheets, customer relationship management, games, virtual desktops). The provider completely abstracts from the cloud

infrastructure and only allows customization and configuration of the application. Almost all SaaS offerings are consumed as web applications via HTTP, so that client-server latency is crucial for both initial loads and performance of interactions. Examples include Microsoft Office 365 [Off17], Salesforce [Onl17], and Slack [Sla17].

Besides the above three models, other "XaaS" (Everything-as-a-Service) models have been proposed, for example, Storage-as-a-Service, Humans-as-a-Service and Function-as-a-Service amongst many others [KLAR10, DLNW13, TCB14, MB16, HYA+15]. Database-as-a-Service and Backend-as-a-Service as discussed in Section 2.2.6 cut across the three canonical levels of IaaS, PaaS, and SaaS and can be employed in each of the models.

Deployment Models

Deployment models describe different options for delivering and hosting cloud platforms.

Public Cloud. A public cloud is operated by a business, academic, or government organization on its infrastructure and can be used by the general public. Commercial cloud offerings such as Amazon EC2, Google App Engine, and Salesforce fall in this category. In public clouds, latency to users and third-party services is critical for performance.

Private Cloud. A private cloud provides exclusive use for one organization and is hosted on-premises of the consumer. This implies that the hardware resources are mostly static and in order to gain elasticity, public cloud resources may be added on demand, e.g., during load spikes (*cloud bursting* [GSW+12]). Besides commercial solutions such as VMWare vCloud [Clo17c], various open-source platforms for private PaaS and IaaS clouds have been developed, including OpenStack [BWA13], Eucalyptus [NWG+09], and Cloud Foundry [Clo17a]. As private clouds usually cannot exploit a globally distributed set of data centers, tackling wide-area latency to end users is a key challenge.

Hybrid Cloud. In a hybrid cloud (also called multi-cloud deployment), two or more clouds are composed to combine their benefits. There are frameworks for addressing multiple clouds through a common API, e.g., jclouds [Apa17a] and libCloud [Apa17b] as well as commercial providers for multi-cloud deployments, scaling and bursting such as RightScale [Rig17], Scalr [Sca17], and Skytap [Sky17]. Any communication between different cloud platforms is highly latency-sensitive. When offloading critical components like data storage to a different cloud, incurred latency can be prohibitive and outweigh the advantages. On the other hand, if data management makes use of the broader geographic reach of multiple providers through caching or replication [WM13], latency can be reduced substantially as we will show in the next chapters.

The NIST definition [MG09] also defines a community cloud, as a cloud shared between organizations with common concerns. Though the model is not in common use, the same

latency challenges apply: composed backends and remote users are subject to latency bottlenecks.

Latency in Cloud Architectures

In cloud-based applications, latency stems from various sources introduced by the composition of different service and deployment models. We group the latency into three categories:

1. Round-trip times *within a data center* network or LAN are usually in the order of single-digit milliseconds.
2. Latencies between two *co-located data centers* are in the order of 10 ms.
3. For hosts from two *different geographical locations*, latency often reaches 100 ms and more.

Figure 2.3 illustrates the typical latency contributions of several communication links within a distributed web application. In the example, the client is separated from the backend by a high-latency wide area network (WAN) link. The application's business logic is hosted on an IaaS platform and distributed across multiple servers interconnected via local networks. The data tier consists of a database service replicated across different availability zones. For a synchronously replicated database system, the latency between two data centers therefore defines the response time for database updates (for example in the Amazon RDS database service [VGS+17]).

Most complex applications integrate **heterogeneous services** for different functions of the application. For example, an external DBaaS might be consumed from the main application over a high-latency network, since it is shared between two different applications or provides a level of scalability that a database hosted on an IaaS could not provide. Parts of the application might also be developed with a service model that fits the requirements better, for example by offloading user authentication to microservices running on a PaaS. A BaaS could be integrated to handle standard functions such as push notifications. High latency also occurs if third-party services are integrated, for example, a social network in the frontend or a SaaS for payments in the backend. Overall, the more providers and services are involved in the application architecture, the higher the dependency on low latency for performance. As almost all interactions between services evolve around exchanging and loading data, the techniques proposed in this thesis apply to the latency problems in the example.

For further details on cloud models, please refer to Murugesan and Bojanova [MB16], who provide a detailed overview of cloud computing and its foundational concepts and technologies. Bahga and Madisetti [BM13] review the programming models and APIs of different cloud platforms.

In the following, we will provide detailed background on backend and network performance to highlight the different opportunities for tackling latency across the application stack.

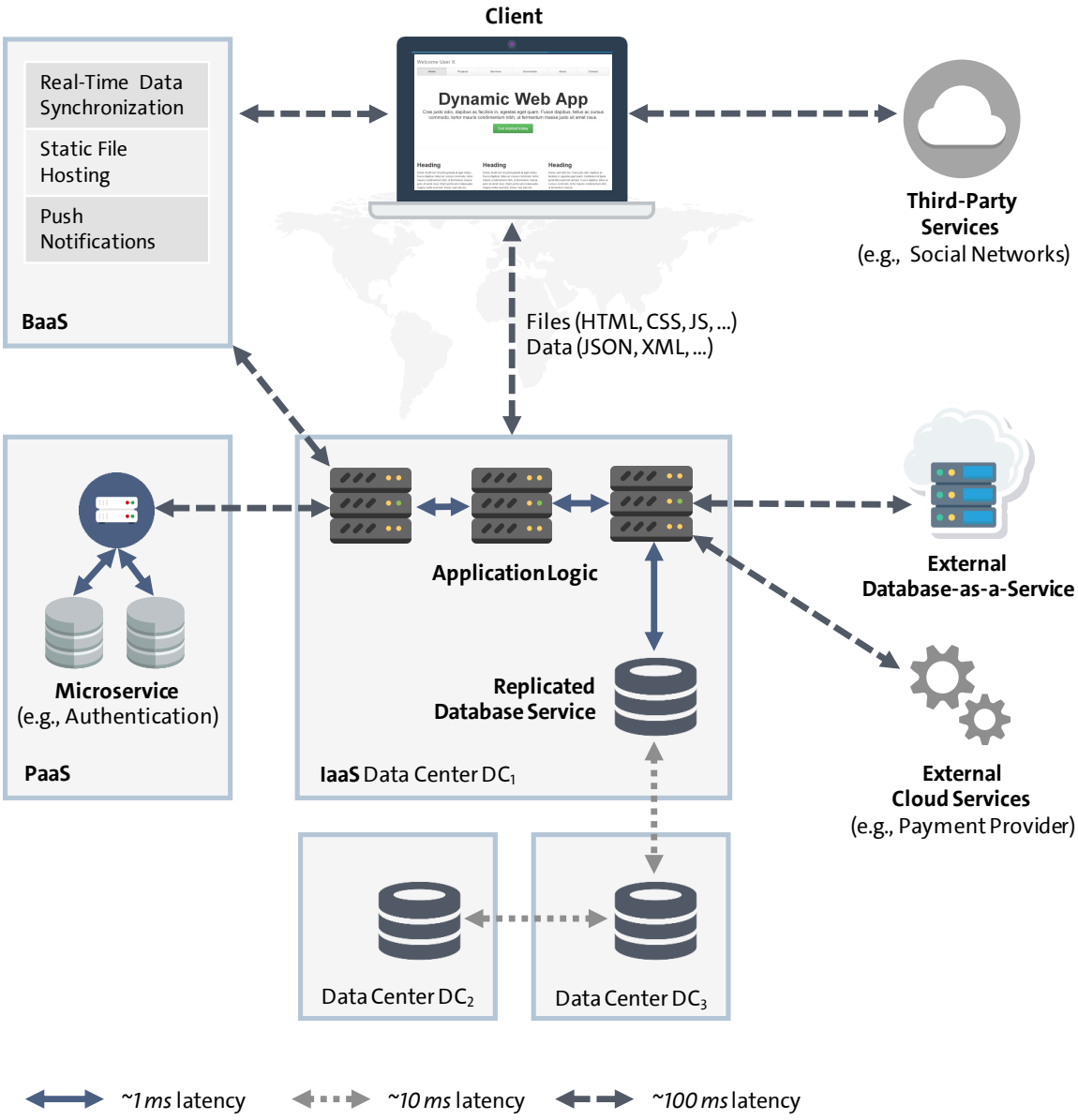


Figure 2.3: Potential sources of latency in distributed, cloud-based applications.

2.2 Backend Performance: Scalable Data Management

Irrespective of the server-side architecture, scalable data management is the primary challenge for high performance. Business and presentation logic can be designed to scale by virtue of stateless processing or by offloading the problem of state to a shared data store. Therefore, the requirements of high availability and elastic scalability depend on database systems.

Today, data is produced and consumed at a rapid pace. This has led to novel approaches for scalable data management subsumed under the term "NoSQL" database systems to handle the ever-increasing data volume and request loads. However, the heterogeneity and diversity of the numerous existing systems impede the well-informed selection of a

data store appropriate for a given application context. In this section, we will provide a high-level overview of the current NoSQL landscape. In Chapter 3 we will furthermore survey commonly used techniques for sharding, replication, storage management, and query processing in these systems, to propose a classification scheme for NoSQL databases. A straightforward and abstract decision model for restricting the choice of appropriate NoSQL systems based on application requirements concludes the survey.

2.2.1 NoSQL Database Systems

Traditional relational database management systems (RDBMSs) provide robust mechanisms to store and query structured data under strong consistency and transaction guarantees and have reached an unmatched level of reliability, stability, and support through decades of development. In recent years, however, the amount of useful data in some application areas has become so vast that it cannot be stored or processed by traditional database solutions. User-generated content in social networks and data retrieved from large sensor networks are only two examples of this phenomenon commonly referred to as **Big Data** [Lan01]. A class of novel data storage systems able to cope with the management of Big Data are subsumed under the term **NoSQL databases**, many of which offer horizontal scalability and higher availability than relational databases by sacrificing querying capabilities and consistency guarantees. These trade-offs are pivotal for service-oriented computing and "as-a-service" models since any stateful service can only be as scalable and fault-tolerant as its underlying data store.

Please note, that throughout this thesis, we address Big Data management, i.e., database and application techniques for dealing with data at high velocity, volume, and variety (coined as the "three Vs" [ZS17]). We only cover Big Data analytics, where it directly concerns the design of our low-latency methodology for data management and refer to our tutorials for further background on systems and approaches for analytics [GR15b, GR16, GWR17].

There are dozens[5] of NoSQL database systems and it is hard for practitioners and researchers to keep track of where they excel, where they fail or even where they differ, as implementation details change quickly and feature sets evolve over time. In this section, we therefore aim to provide an overview of the NoSQL landscape by discussing employed concepts rather than system specificities and explore the requirements typically posed to NoSQL database systems, the techniques used to fulfill these requirements and the trade-offs that have to be made in the process. Our focus lies on key-value, document, and wide-column stores since these NoSQL categories cover the most relevant techniques and design decisions in the space of scalable data management and are well suitable for the context of scalable cloud data management.

In order to abstract from implementation details of individual NoSQL systems, high-level classification criteria can be used to group similar data stores into categories. As shown in

[5] An extensive list of NoSQL database systems can be found at `http://nosql-database.org/`.

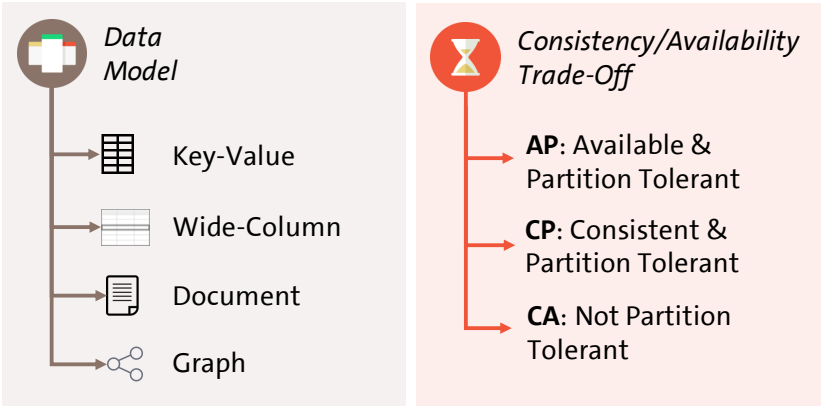


Figure 2.4: The two high-level approaches of categorizing NoSQL systems according to data models and consistency-availability trade-offs.

Figure 2.4, we will describe how NoSQL systems can be categorized by their data model (key-value stores, document stores, and wide-column stores) and the safety-liveness trade-offs in their design (CAP and PACELC).

2.2.2 Different Data Models

The most commonly employed distinction between NoSQL databases is the way they store and allow access to data. Each system covered in this overview can be categorized as either a key-value store, document store, or wide-column store.

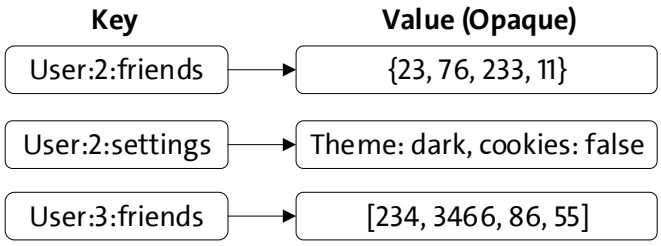


Figure 2.5: Key-value stores offer efficient storage and retrieval of arbitrary values.

Key-Value Stores

A key-value store consists of a set of key-value pairs with unique keys. Due to this simple structure, it only supports get and put operations. As the nature of the stored value is transparent to the database, pure key-value stores do not support operations beyond simple CRUD (Create, Read, Update, Delete). Key-value stores are therefore often referred to as **schemaless** [SF12]: any assumptions about the structure of stored data are implicitly encoded in the application logic (*schema-on-read* [Kle17]) and not explicitly defined through a data definition language (*schema-on-write*).

The obvious advantages of this data model lie in its simplicity. The very simple abstraction makes it easy to partition and query data so that the database system can achieve low latency as well as high throughput. However, if an application demands more com-

plex operations, e.g., range queries, this data model is not powerful enough. Figure 2.5 illustrates how user account data and settings might be stored in a key-value store. Since queries more complex than simple lookups are not supported, data has to be analyzed inefficiently in application code to extract information like whether cookies are supported or not (`cookies: false`).

Document Stores

A document store is a key-value store that restricts values to semi-structured formats such as JSON documents like the one illustrated in 2.6. This restriction in comparison to key-value stores brings great flexibility in accessing the data. It is not only possible to fetch an entire document by its ID, but also to retrieve only parts of a document, e.g., the age of a customer, and to execute queries like aggregations, query-by-example or even full-text search.

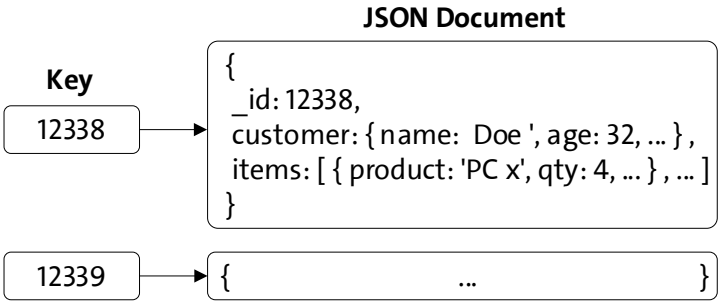


Figure 2.6: Document stores are aware of the internal structure of the stored entity and thus can support queries.

Wide-Column Stores

Wide-column stores inherit their name from the image that is often used to explain the underlying data model: a relational table with many sparse columns. Technically, however, a wide-column store is closer to a distributed multi-level[6] sorted map: the first-level keys identify rows which themselves consist of key-value pairs. The first-level keys are called **row keys**, the second-level keys are called **column keys**. This storage scheme makes tables with arbitrarily many columns feasible because there is no column key without a corresponding value. Hence, null values can be stored without any space overhead. The set of all columns is partitioned into so-called **column families** to co-locate columns on disk that are usually accessed together.

On disk, wide-column stores do not co-locate all data from each row, but instead, values of the same column family *and* from the same row. Hence, an entity (a row) cannot be retrieved by one single lookup as in a document store but has to be joined from the columns of all column families. However, this storage layout usually enables highly efficient data compression and makes retrieving only a portion of an entity fast. All data is stored in

[6] In some systems (e.g., BigTable and HBase), multi-versioning is implemented by adding a timestamp as a third-level key.

lexicographic order of the keys, so that rows that are accessed together are physically co-located, given a careful key design. As all rows are distributed into contiguous ranges (so-called **tablets**) among different **tablet servers**, row scans only involve few servers and thus are very efficient.

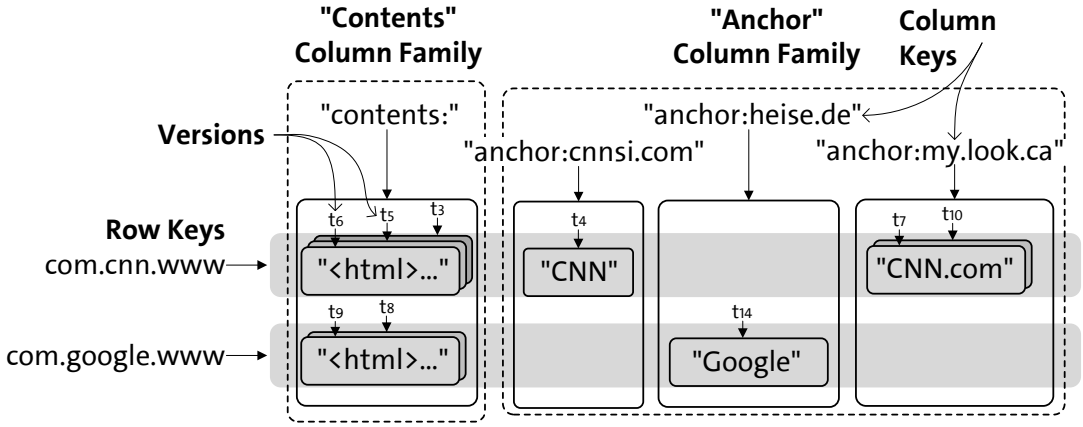


Figure 2.7: Data in a wide-column store.

Bigtable [CDG+08], which pioneered the wide-column model, was specifically developed to store a large collection of web pages as illustrated in Figure 2.7. Every row in the table corresponds to a single web page. The row key is a concatenation of the URL components in reversed order, and every column key is composed of the column family name and a column qualifier, separated by a colon. There are two column families: the "contents" column family with only one column holding the actual web page and the "anchor" column family holding links to each web page, each in a separate column. Every cell in the table (i.e., every value accessible by the combination of row and column key) can be versioned by timestamps or version numbers. It is important to note that much of the information of an entity lies in the keys and not only in the values [CDG+08].

2.2.3 Latency, Consistency, and Availability: Trade-Offs

Another defining property of a database apart from how data is stored and how it can be accessed is the level of consistency that is provided. Some databases are built to guarantee strong consistency and serializability (ACID[7]), while others favor availability (BASE[8]). This trade-off is inherent to every distributed database system and the huge number of different NoSQL systems shows that there is a wide spectrum between the two paradigms. In the following, we explain the two theorems CAP and PACELC according to which database systems can be categorized by their respective positions in this spectrum.

[7] ACID [HR83]: **A**tomicity, **C**onsistency, **I**solation, **D**urability
[8] BASE [Pri08]: **B**asically **A**vailable, **S**oft-state, **E**ventually consistent

CAP

Like the famous FLP Theorem[9] [FLP85], the CAP Theorem, presented by Eric Brewer at PODC 2000 [Bre00] and later proven by Gilbert and Lynch [GL02], is one of the most influential impossibility results in the field of distributed computing. It places an upper bound on what can be accomplished by a distributed system. Specifically, it states that a sequentially consistent read/write register[10] that eventually responds to every request, cannot be realized in an asynchronous system that is prone to network partitions. In other words, the register can guarantee at most two of the following three properties at the same time:

- **Consistency (C)**. Reads and writes are always executed atomically and are strictly consistent (linearizable [HW90]). Put differently, all clients have the same view on the data at all times.
- **Availability (A)**. Every non-failing node in the system can always accept read and write requests from clients and will eventually return with a meaningful response, i.e., not with an error message.
- **Partition-tolerance (P)**. The system upholds the previously displayed consistency guarantees and availability in the presence of message loss between the nodes or partial system failure.

Brewer argues that a system can be both available and consistent in normal operation, but in the presence of a network partition, this is not possible: if the system continues to work in spite of the partition, there is some non-failing node that has lost contact to the other nodes and thus has to decide to either continue processing client requests to preserve availability (AP, **eventually consistent systems**) or to reject client requests in order to uphold consistency guarantees (CP). The first option violates consistency because it might lead to stale reads and conflicting writes, while the second option obviously sacrifices availability. There are also systems that usually are available and consistent but fail completely when there is a partition (CA), for example, single-node systems. It has been shown that the CAP-theorem holds for any consistency property that is at least as strong as causal consistency, which also includes any recency bounds on the permissible staleness of data (Δ-atomicity) [MAD+11]. Serializability as the correctness criterion of transactional isolation does not require strong consistency. However, similar to consistency, serializability cannot be achieved under network partitions either [DGMS85].

The classification of NoSQL systems as either AP, CP or CA vaguely reflects the individual systems' capabilities and hence is widely accepted as a means for high-level comparisons. However, it is important to note that the CAP Theorem actually does not state anything on normal operation; it merely expresses whether a system favors availability or consistency *in the face of a network partition*. In contrast to the FLP-Theorem, the CAP theorem as-

[9] The FLP theorem states, that in a distributed system with asynchronous message delivery, no algorithm can guarantee to reach a *consensus* between participating nodes if one or more of them can fail by stopping.

[10] A read/write register is a data structure with only two operations: setting a specific value (`set`) and returning the latest value that was set (`get`).

sumes a failure model that allows arbitrary messages to be dropped, reordered or delayed indefinitely. Under the weaker assumption of reliable communication channels (i.e., messages always arrive but asynchronously and possibly reordered) a CAP-system is in fact possible using the Attiya, Bar-Noy, Dolev algorithm [ABN+95], as long as a majority of nodes are up[11].

PACELC

The shortcomings of the CAP Theorem were addressed by Abadi [Aba12] who points out that the CAP Theorem fails to capture the trade-off between latency and consistency during *normal* operation, even though it has proven to be much more influential on the design of distributed systems than the availability-consistency trade-off in failure scenarios. He formulates PACELC which unifies both trade-offs and thus portrays the design space of distributed systems more accurately. From PACELC, we learn that in case of a Partition, there is an Availability-Consistency trade-off; Else, i.e., in normal operation, there is a Latency-Consistency trade-off.

This classification offers two possible choices for the partition scenario (A/C) and also two for normal operation (L/C) and thus appears more fine-grained than the CAP classification. However, many systems cannot be assigned exclusively to one single PACELC class and one of the four PACELC classes, namely PC/EL, can hardly be assigned to any system.

Summary

NoSQL database systems support applications in achieving horizontal scalability, high availability and backend performance through differentiated trade-offs in functionality and consistency. For this reason, this thesis fundamentally builds on previous work on these systems and their guarantees, in order to address the requirement for low latency.

2.2.4 Relaxed Consistency Models

CAP and PACELC motivate that there is a broad spectrum of choices regarding consistency guarantees and that the strongest guarantees are irreconcilable with high availability. The goal of our approach is to provide different levels of consistency depending on the requirements of the application. Therefore, we examine the suitable consistency models that fulfill two requirements. First, the models must exhibit sufficient *power* to precisely express latency-consistency trade-offs introduced by caching and replication. Second, the consistency models must have the *simplicity* to allow easy reasoning about application behavior for developers and system architects.

As summarized in Figure 2.8, NoSQL systems exhibit various relaxed consistency guarantees that are usually a consequence of replication and caching. **Eventual consistency** is

[11] Therefore, consensus as used for coordination in many NoSQL systems either natively [BBC+11] or through coordination services like Chubby and Zookeeper [HKJR10] is considered a "harder" problem than strong consistency, as it cannot even be guaranteed in a system with reliable channels [FLP85].

a commonly used term to distinguish between strongly consistent (linearizable) systems and systems with relaxed guarantees. Eventual consistency is slightly stronger than weak consistency, as it demands that in the absence of failures, the system converges to a consistent state. The problem with eventual consistency is that it purely represents a *liveness* guarantee, i.e., it asserts that some property is eventually reached [Lyn96]. However, it lacks a *safety* guarantee: eventual consistency does not prescribe which state the database converges to [Bai15, p. 20]. For example, the database could eventually converge to a null value for every data item and would still be eventually consistent. For this reason, more specific relaxed consistency models provide a framework for reasoning about safety guarantees that are weaker than strong, immediate consistency.

The idea of **relaxing correctness guarantees** is wide-spread in the database world. Even in single-node systems, providing ACID and in particular serializability incurs performance penalties through limited concurrency and contention, especially on multi-core hardware [GLPT76]. As a consequence, weak isolation models relax the permissible transaction schedules by allowing certain concurrency anomalies that are not present under serializability. Bailis et al. [BFG+13] surveyed 18 representative systems claiming to provide ACID or "NewSQL"[12] guarantees. Of these systems, only three provided serializability by default, and eight did not offer serializable isolation at all.

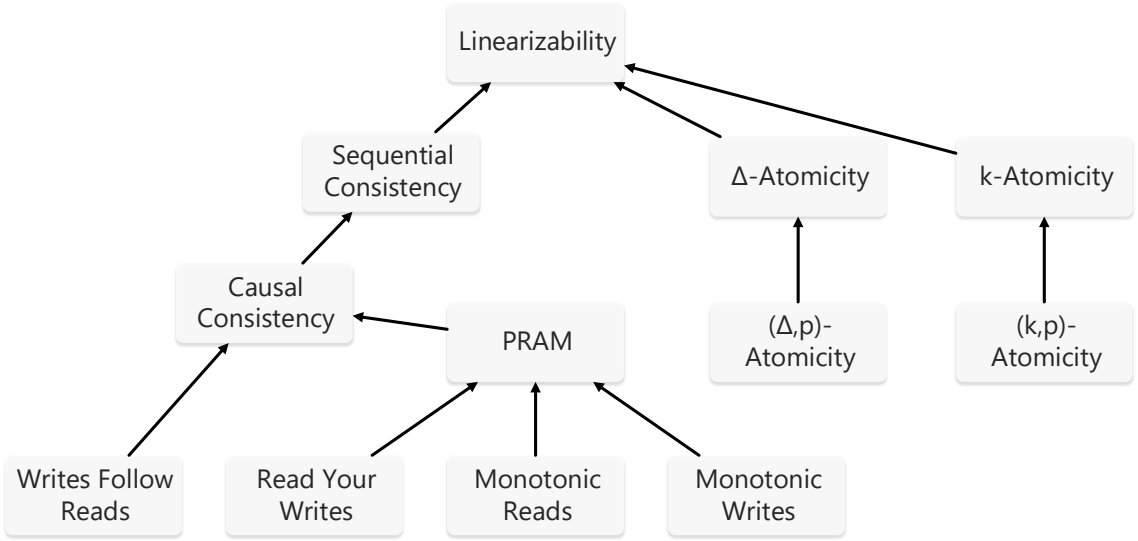


Figure 2.8: An overview of selected consistency models. Arrows indicate which models are subsumed by a stronger model.

Strong Consistency Models

The strongest consistency guarantee in a concurrent system is linearizability (see Definition 2.1) introduced by Herlihy and Wing [HW90]. A linearizable system behaves analo-

[12]The term NewSQL was coined by relational database vendors seeking to provide similar scalability and performance as NoSQL databases while maintaining well-known abstractions such as SQL as a query language and ACID guarantees [GHTC13]. This is achieved by introducing trade-offs that are mostly similar to that of NoSQL databases. Examples are H-Store [KKN+08], VoltDB [SW13], Clustrix [Clu17], NuoDB [Nuo17], and Calvin [TDW+12] that are discussed in Chapter 6.

gously to a single-node system, i.e., each read and write appears to be applied at one defined point in time between invocation and response. While linearizability is the gold standard for correctness, it is not only subject to the CAP theorem, but also hard to implement at scale [LPK+15, ABK+15, BDF+15, DGMS85, KPF+13, TPK+13, BK13, BT11, WFZ+11].

Definition 2.1. *An execution satisfies **linearizability**, if all operations are totally ordered by their arrival time. Any read with an invocation time larger than the response time of a preceding write is able to observe its effects. Concurrent operations must guarantee sequential consistency, i.e., overlapping write operations become visible to all reads in a defined global order.*

Sequential consistency (see Definition 2.2) is a frequently used model in operating system and hardware design that is slightly weaker than linearizability. It does not guarantee any recency constraints, but it ensures that writes become visible for each client in the same order. So in contrast to linearizability, the global ordering of operations is not required to respect real-time ordering, only the local real-time ordering for each client is preserved.

Definition 2.2. *An execution satisfies **sequential consistency**, if there is a global order of read and write operations that is consistent with the local order in which they were submitted by each client.*

Consistency in replicated systems is sometimes confused with consistency in ACID transactions. With respect to ACID, consistency implies that no integrity constraints are violated, e.g., foreign key constraints. In distributed, replicated systems, consistency is an ordering guarantee for reads and writes that are potentially executed concurrently and on different copies of the data. The main correctness criterion for transactional isolation is serializability, which does not require strong consistency. If conflict serializability is combined with strong consistency, it is referred to as strict (or strong) serializability (e.g., in Spanner [Coo13]) or commit order-preserving conflict serializability (COCSR) [WV02]. Just as linearizability, serializability is also provably irreconcilable with high availability [BDF+13].

Staleness-Based Consistency Models

To increase efficiency, staleness-based models allow *stale reads*, i.e., returning outdated data. The two common measures for quantifying staleness are (wall-clock) *time* and object *versions*. In contrast, k-atomicity (see Definition 2.3) [AAB05] bounds staleness by only allowing reads to return a value written by one of the k preceding updates. Thus k-atomicity with $k = 1$ is equivalent to linearizability.

Definition 2.3. *An execution satisfies k-**atomicity**, if any read returns one of the versions written by the k preceding, completed writes that must have a global order that is consistent with real-time order.*

Δ-atomicity (see Definition 2.4) introduced by Golab et al. [GLS11] expresses a time-based recency guarantee. Intuitively, Δ is the upper bound on staleness observed for any read in

the system, i.e., it never happens that the application reads data that has been stale for longer than Δ time units.

Definition 2.4. *An execution satisfies* **Δ-*atomicity***, *if any read returns either the latest preceding write or the value of a write that returned at most* Δ *time units ago.*

Δ-atomicity is a variant of the influential *atomic* semantics definition introduced by Lamport in the context of inter-process communication [Lam86b, Lam86a]. Atomicity and linearizability are equivalent [VV16], i.e., they demand that there is a logical **point of linearization** between invocation and response for each operation at which it appears to be applied instantaneously [HW90]. An execution is Δ-atomic, if by decreasing the start time of each read operation by Δ produces an atomic execution. Lamport also introduced two relaxed properties of *regular* and *safe* semantics that are still often used in the literature. In the absence of a concurrent write, regular and safe reads behave exactly like atomic reads. However, during concurrent writes, safe reads are allowed to return arbitrary values[13]. A read under regular semantics returns either the latest completed write or the result of any concurrent write.

The extension of safety and regularity to Δ-safety, Δ-regularity, k-safety, and k-regularity is straightforward [AAB05, GLS11, BVF+14]. Δ-atomicity will be used throughout this thesis as it is a fundamental guarantee provided by our Cache Sketch approach. Other time-based staleness models from the literature are very similar to Δ-atomicity. Delta consistency by Singla et al. [SRH97], timed consistency by Torres-Rojas et al. [TAR99], and bounded staleness by Mahajan et al. [MSL+11] all express that a write should become visible before a defined maximum delay. Δ-atomicity is hard to measure experimentally due to its dependency on a global time. Golab et al. [GRA+14] proposed Γ-atomicity as a closely related alternative that is easier to capture in benchmarks. The central difference is that the Γ parameter also allows writes to be reordered with a tolerance of Γ time units, whereas Δ-atomicity only considers earlier starting points for reads, while maintaining the order of writes. With NoSQLMark, we proposed an experimental methodology to measure lower and upper staleness bounds [WFGR15].

For illustration of these models, please consider the example execution in Figure 2.9. The result x of the read operation performed by client C_3 depends on the consistency model:

- With *atomicity* (including $k = 1$ and $\Delta = 0$) or *linearizability*, x can be either 2 or 3. x cannot be 4 since the later read of 3 by client C_2 would then violate linearizability.
- Under *sequential consistency* semantics, x can be 0 (the initial value), 1, 2, 3, or 4. As C_3 only performs a read, no local order has to be maintained. It can be serialized to the other clients' operations in any order.
- Given *regular* semantics, x can be either 2, 3, or 4.
- Under *safe* semantics, x can be any value.

[13] The usefulness of this property has been criticized for database systems, as no typical database would return values that have never been written, even under concurrent writes [Ber14].

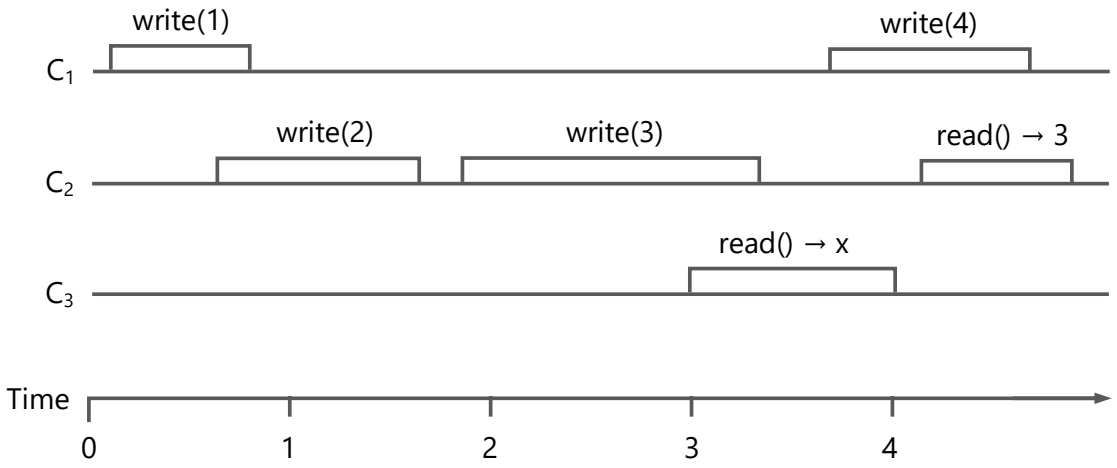


Figure 2.9: An example execution of interleaved reads and writes from three clients that yields different read results depending on the consistency model. Brackets indicate the time between invocation and response of an operation.

- For Δ-*atomicity* with $\Delta = 1$, x can be 2 or 3. With $\Delta = 2$, x can be 1, 2, or 3: if the begin of the read was stretched by $\Delta = 2$ time units to begin at time 1, then 1, 2, and 3 would be reads satisfying atomicity.
- For *k-atomicity* with $k = 2$, x can be 1, 2, or 3: compared to atomicity, a lag of one older object version is allowed.

Δ-atomicity and k-atomicity can be extended to the probabilistic guarantees (Δ,p)-atomicity and (k,p)-atomicity (see Definition 2.5) [BVF+14]. This allows expressing the average time or version-based lag as a distribution. For consistency benchmarks and simulations, these values are preferable, as they express more details than Δ-atomicity and k-atomicity which are just bounded by the maximum encountered values [BWT17, Ber14, BVF+14].

Definition 2.5. *An execution satisfies (Δ,p)-**atomicity**, if reads are Δ-atomic with probability p. Similarly, an execution satisfies (k,p)-**atomicity**, if reads are k-atomic with probability p.*

Session-Based Consistency Models

Data-centric consistency models like linearizability and Δ-atomicity describe consistency from the provider's perspective, i.e., in terms of synchronization schemes to provide certain guarantees. Client-centric or session-based models take the perspective of clients interacting with the database and describe guarantees an application expects within a session.

Monotonic writes consistency (see Definition 2.6) guarantees that updates from a client do not get overwritten or reordered. Systems that lack this guarantee make it hard to reason about how updates behave, as they can be seen by other clients in a different order [Vog09]. For example, in a social network without monotonic write consistency, posts by a user could be observed in a different, potentially nonsensical order by other users.

Definition 2.6. *A session satisfies **monotonic writes consistency**, if the order of all writes from that session is maintained for reads.*

Monotonic reads consistency (see Definition 2.7) guarantees that if a client has read version n of an object, it will later only see versions $\geq n$ [TvS07]. For example on a content website, this would prevent a user from first seeing a revised edition of an article and then upon a later return to the page reading the unrevised article.

Definition 2.7. *A session satisfies **monotonic reads consistency**, if reads return versions in a monotonically increasing order.*

With read your writes consistency (see Definition 2.8) clients are able to observe their own interactions. For example, in a web application with user-generated content, a user could reload the page and still see the update he applied.

Definition 2.8. *A session satisfies **read your writes consistency**, if reads return a version that is equal to or higher than the latest version written in that session.*

Combining the above three session guarantees yields the PRAM consistency level (see Definition 2.9) [LS88a]. It prescribes that all clients observe writes from different processes in their local order, i.e., as if the writes were in a pipeline. However, in contrast to sequential consistency, there is no global order for writes.

Definition 2.9. *If monotonic writes consistency, monotonic reads consistency, and read your writes consistency are guaranteed, **pipelined random access memory (PRAM) consistency** is satisfied.*

With writes follow reads consistency (see Definition 2.10), applications get the guarantee that their writes will always be accompanied by the relevant information that might have influenced the write. For example, writes follow reads (also called session causality) prevents the anomaly of a user responding to a previous post or comment on a website where other users would observe the response without seeing the original post it is based on.

Definition 2.10. *A session satisfies **writes follow reads consistency**, if its writes are ordered after any other writes that were observed by previous reads in the session.*

Causal consistency (see Definition 2.11) [Ady99, BDF+13] combines the previous session guarantees. It is based on the concept of potential causality introduced through Lamport's *happened-before* relation in the context of message passing [Lam78]. An operation a causally depends on an operation b, if [HA90]:

1. a and b were issued by the same client and the database received b before a,
2. a is a read that observed the write b, or
3. a and b are connected transitively through condition 1. and/or 2.

In distributed systems, causality is often tracked using *vector clocks* [Fid87]. Causal consistency can be implemented through a middleware or directly in the client by tracking causal dependencies and only revealing updates when their causal dependencies are visible, too [BGHS13, BKD+13]. Causal consistency is the strongest guarantee that can be achieved with high availability in the CAP theorem's system model of unreliable channels and asynchronous messaging [MAD+11]. The reason for causal consistency being compatible with high availability is that causal consistency does not require **convergence** of replicas and does not imply staleness bounds [GH02]. Replicas can be in completely different states, as long as they only return writes where causal dependencies are met. For this reason, our caching approach in Orestes combines causal consistency with recency guarantees to strengthen the causal consistency model for practical data management use cases that require bounded staleness.

Bailis et al. [BGHS13] argued that **potential causality** leads to a high fan-out of potentially relevant data. Instead, application-defined causality can help to minimize the actual dependencies. In practice however, potential causality can be determined automatically through dependency tracking (e.g., in COPS [LFKA11]), while explicit causality forces application developers to declare dependencies.

Causal consistency can be combined with a timing constraint demanding that the global ordering respects causal consistency with tolerance Δ for each read, yielding a model called *timed causal consistency* [TM05]. This model is weaker than Δ-atomicity: timed causal consistency with $\Delta = 0$ yields causal consistency, while Δ-atomicity with $\Delta = 0$ yields linearizability.

Definition 2.11. *If both PRAM and writes follow reads are guaranteed,* ***causal consistency*** *is satisfied.*

Besides the discussed consistency models, many different deviations have been proposed and implemented in the literature. Viotti and Vukolic [VV16] give a comprehensive survey and formal definitions of consistency models. In particular, they review the overlapping definitions used in different lines of work across the distributed systems, operating systems, and database research community.

We are convinced that consistency trade-offs should be made explicit in data management. While strong guarantees are a sensible default for application developers, there should be an option to relax consistency in order to shift the trade-off towards non-functional availability and performance requirements. The consistency models linearizability, causal consistency, Δ-atomicity, and session guarantees will be used frequently throughout this thesis, as they allow a fine-grained trade-off between latency and consistency.

2.2.5 Polyglot Persistence

As applications become more data-driven and highly distributed, providing low response times to increasingly many users becomes more challenging within the scope of a single

database system. Not only the variety of use cases is increasing, but also the requirements are becoming more heterogeneous: horizontal scalability, schema flexibility, and high availability are primary concerns for modern applications. While RDBMSs cover many of the functional requirements (e.g., ACID transactions and expressive queries), they cannot cover scalability, performance, and fault tolerance in the same way that specialized data stores can. The explosive growth of available systems through the Big Data and NoSQL movement sparked the idea of employing particularly well-suited database systems for subproblems of the overall application.

The architectural style *polyglot persistence* describes the usage of specialized data stores for different requirements. The term was popularized by Fowler in 2011 and builds on the idea of polyglot programming [SF12]. The core idea is that abandoning a "one size fits all" architecture can increase development productivity, resp. time-to-market, as well as performance. Polyglot persistence applies to single applications as well as complete organizations.

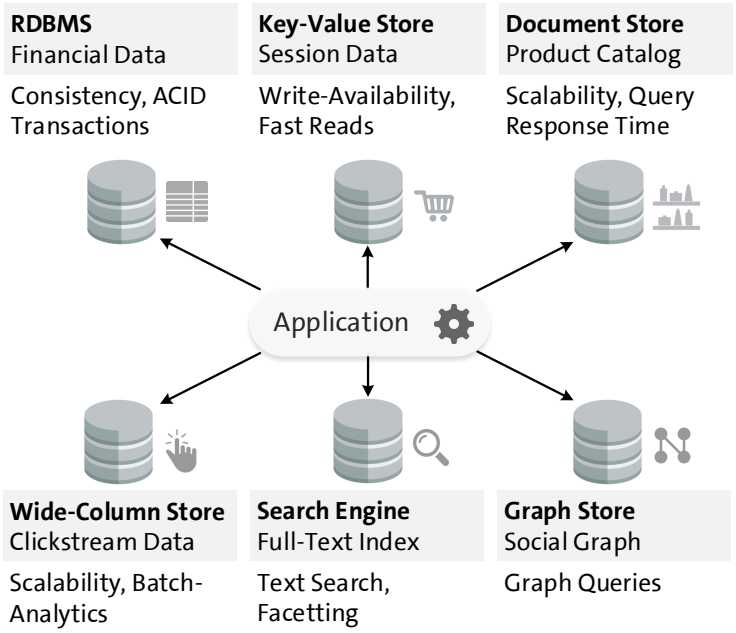


Figure 2.10: Example of a polyglot persistence architecture with database systems for different requirements and types of data in an e-commerce scenario.

Figure 2.10 shows an example of a polyglot persistence architecture for an e-commerce application, as often found in real-world applications [Kle17]. Data is distributed to different database systems according to their associated requirements. For example, financial transactions are processed through a relational database, to guarantee correctness. As product descriptions form a semi-structured aggregate, they are well-suited for storage in a distributed document store that can guarantee scalability of data volume and reads. The log-structured storage management in wide-column stores is optimal for maintaining high write throughput for application-generated event streams. Additionally, they provide interfaces to apply complex data analysis through Big Data platforms such as Hadoop

and Spark [Whi15, ZCF+10]. The example illustrates that in polyglot persistence architectures, there is an inherent trade-off between increased complexity of maintenance and development against improved, problem-tailored storage of application data.

In summary, polyglot persistence adopts the idea of applying the best persistence technology for a given problem. In the following, we will present an overview of different strategies for implementing polyglot persistence and the challenges they entail.

Polyglot Data Management

The requirement for a fast *time-to-market* is supported by avoiding the impedance mismatch [Mai90, Amb12] between the application's data structures and the persistent data model. For example, if a web application using a JSON-based REST API can store native JSON documents in a document store, the development process is considerably simplified compared to systems where the application's data model has to be mapped to a database system's data model.

Performance can be maximized, if the persistence requirements allow for an efficient partitioning and replication of data combined with suitable index structures and storage management. If the application can tolerate relaxed guarantees for consistency or transactional isolation, database systems can leverage this to optimize throughput and latency.

Typical *functional* persistence requirements are:

- ACID transactions with different isolation levels
- Atomic, conditional, or set-oriented updates
- Query types: point lookups, scans, aggregations, selections, projections, joins, subqueries, Map-Reduce, graph queries, batch analyses, searches, real-time queries, dataflow graphs
- Partial or commutative update operations
- Data structures: graphs, lists, sets, maps, trees, documents, etc.
- Structured, semi-structured, or implicit schemas
- Semantic integrity constraints

Among the *non-functional requirements* are:

- Throughput for reads, writes, and queries
- Read and write latency
- High availability
- Scalability of data volume, reads, writes, and queries
- Consistency guarantees
- Durability
- Elastic scale-out and scale-in

The central challenge in polyglot persistence is determining whether a given database system satisfies a set of application-provided requirements and access patterns. While some performance metrics can be quantified with **benchmarks** such as YCSB, TPC, and others [DFNR14, CST+10, CST+10, PPR+11, BZS13, BKD+14, PF00, WFZ+11, FMdA+13, BT14, Ber15, Ber14], many non-functional requirements such as consistency and scalability are currently not covered through benchmarks or even diverge from the documented behavior [WFGR15].

In a polyglot persistence architecture, the boundary of the database form the boundary of transactions, queries, and update operations. Thus, if data is persisted and modified in different databases, this entails consistency challenges. The application therefore has to explicitly control the synchronization of data across systems, e.g., through ETL batch jobs, or has to maintain consistency at the application level, e.g., through commutative data structures. Alternatively, data can be distributed in disjoint partitions which shifts the problem to cross-database queries, a well-studied topic in data integration [Len02]. In contrast to data integration problems, however, there is no autonomy of data sources. Instead, the application explicitly combines and modifies the databases for polyglot persistence [SF12].

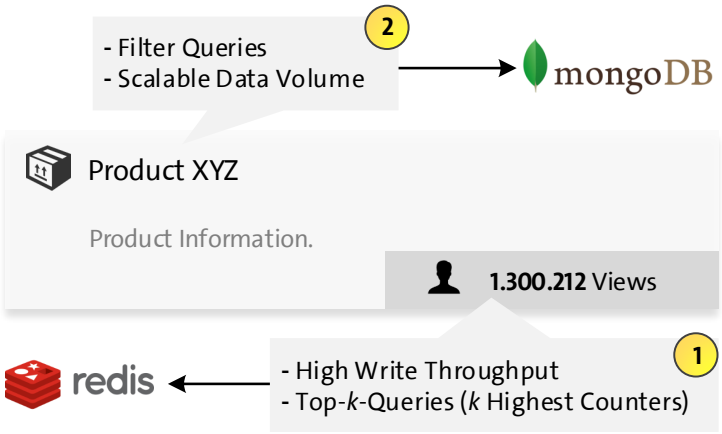


Figure 2.11: Polyglot persistence requirements for a product catalog in an e-commerce application.

Implementation of Polyglot Persistence

To manage the increased complexity introduced by polyglot persistence, different architectures can be applied. We group them into the three architectural patterns *application-coordinated polyglot persistence*, *microservices*, and *polyglot database services*. As an example, consider the product catalog of the introductory e-commerce example (see Figure 2.11). The product catalog should be able to answer simple filter queries (e.g., searching by keyword) as well as returning the top-k products according to access statistics. The functional requirement therefore is that the access statistics have to support a high write throughput (incrementing on each view) and top-k queries **(1)**. The product catalog has to offer filter queries and scalability of data volume **(2)**. These requirements can, for ex-

ample, be fulfilled with the key-value store Redis and the document store MongoDB. With its sorted set data structure, Redis supports a mapping from counters to primary keys of products. Incrementing and performing top-k queries are efficiently supported with logarithmic time complexity in memory. MongoDB supports storing product information in nested documents and allows queries on the attributes of these documents. Using hash partitioning, the documents can efficiently be distributed over many nodes in a cluster to achieve scalability.

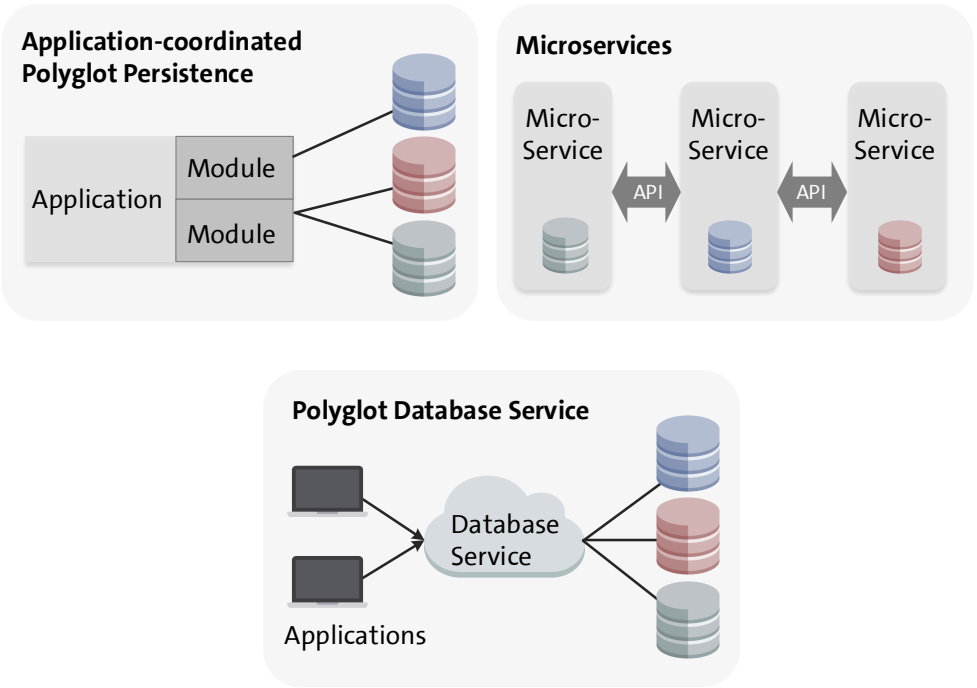


Figure 2.12: Architectural patterns for the implementation of polyglot persistence: application-coordinated polyglot persistence, microservices, and polyglot database services.

With **application-coordinated polyglot persistence** (see Figure 2.12), the application server's data tier programmatically coordinates polyglot persistence. Typically, the mapping of data to databases follows the application's modularization. This pattern simplifies development, as each module is specialized for the use of one particular data store. Also, design decisions in data modeling as well as access patterns are encapsulated in a single module (*loose coupling*). The separation can also be relaxed: for the product catalog, it would not only be possible to model a counter and separate product data. Instead, a product could also be modeled as an entity containing a counter. The dependency between databases has to be considered both at development time and during operation. For example, if the format of the primary key changes, the new key structure has to be implemented for both systems in the code and in the database. Object-NoSQL mappers simplify the implementation of application-coordinated polyglot persistence. However, currently, the functional scope of these mappers is very limited [TGPM17, SHKS15, WBGsS13].

A practical example of application-coordinated polyglot persistence is Twitter's storage of user feeds [Kri13]. For fast read access, the newest tweets for each user are materialized in a Redis cluster. Upon publishing of a new tweet, the social graph is queried from a graph store and distributed among the Redis-based feeds for each relevant user (*Write Fanout*). As a persistent fallback for Redis, MySQL servers are managed and partitioned by the application tier.

To increase encapsulation of persistence decisions, **microservice** architectures are useful [New15] (see Section 2.1.1). Microservices allow narrowing the choice of a database system to one particular service and thus decouple the development and operations of services [DB13]. Technologically, IaaS/PaaS, containers, and cluster management frameworks provide sophisticated tooling for scaling and operating microservice architectures. In the example, the product catalog could be split into two microservices using MongoDB and Redis separately. The Redis-based service would provide an API for querying popular products and incrementing counters, whereas the MongoDB-based microservice would have a similar interface for retrieving product information. The user-facing business logic (e.g., the frontend in a two-tier architecture) simply has to invoke both microservices and combine the result.

In order to make polyglot persistence fully transparent for the application, **polyglot database services** can abstract from implementation details of underlying systems. The key idea is to hide the allocation of data and queries to databases through a generic cloud service API. Some NoSQL databases and services use this approach, for example, to integrate full-text search with structured storage (e.g., in Riak [Ria17] and Cassandra [LM10]), to store metadata consistently (e.g., in HBase [Hba17] and BigTable [CDG+08]), or to cache objects (e.g., Facebook's TAO [BAC+13]). However, these approaches use a defined scheme for the allocation and cannot adapt to varying application requirements. Polyglot database services can also apply static rules for polyglot persistence: if the type of the data is known (for example a user object or a file), a rule-based selection of a storage system can be performed [SGR15].

The currently unsolved challenge is to adapt the choice of a database system to the actual requirements and workloads of the application. A declarative way to achieve this is to have the application express requirements as SLAs to let a polyglot database service determine the optimal mapping. In this thesis, we will explore a first approach for automating polyglot persistence.

In the example, the application could declare the throughput requirements of the counter and the scalability requirement for the product catalog. The database service would then autonomously derive a suitable mapping for queries and data. The core challenge is to base the selection of systems on quantifiable metrics of available databases and applying transparent rewriting of operations. A weaker form than fully-automated polyglot persistence are database services with **semi-automatic polyglot persistence**. In this model, the application can explicitly define which underlying system should be targeted, while

reusing high-level features such as schema modeling, transactions, and business logic across systems through a unified API. Orestes supports this strategy and provides standard interfaces for different database systems to integrate them into a polyglot database middleware exposed through a unified set of APIs.

2.2.6 Cloud Data Management: Database- and Backend-as-a-Service

Cloud data management is the research field tackling the design, implementation, evaluation and application implications of database systems in cloud environments [GR15b, GR16, GWR17, WGW+18]. We group cloud data management systems into two categories: **Database-as-a-Service** (DBaaS) and **Backend-as-a-Service** (BaaS). In the DBaaS model, only data management is covered. Therefore, application logic in a two- and three-tier architecture has to employ an additional IaaS or PaaS cloud. BaaS combines a DBaaS with custom application logic and standard APIs for web and app development. BaaS is a form of *serverless* computing, an architectural approach that describes applications which mostly rely on cloud services for both application logic and storage [Rob16]. Besides the BaaS mode, serverless architectures can also make use of **Function-as-a-Service** (FaaS) providers, that provide scalable and stateless execution of business logic functions in a highly elastic environment (e.g., AWS Lambda, and Google Cloud Functions). BaaS combines the concepts of DBaaS with a FaaS execution layer for business logic.

Database-as-a-Service

Hacigumus et al. [HIM02] introduced DBaaS as an approach to run databases without acquiring hardware or software. As the landscape of DBaaS systems has become similarly heterogeneous as the NoSQL ecosystem, we propose a two-dimensional classification as shown in Figure 2.13. The first dimension is the data model ranging from structured relational systems, over semi-structured or schema-free data to completely unstructured data. The second dimension describes the deployment model.

Cloud-deployed databases use an IaaS or PaaS cloud to provision an operating system and the database software as an opaque application. Cloud providers usually maintain a repository of pre-built machine images containing RDBMSs, NoSQL databases, or analytics platforms that can be deployed as a virtual machine (VM) [BDF+03]. While cloud-deployed systems allow for a high degree of customization, maintenance (e.g., operating system and database updates), as well as operational duties (e.g., scaling in and out) have to be implemented or performed manually.

In **managed** cloud databases, the service provider is responsible for configuration, scaling, provisioning, monitoring, backup, privacy, and access control [CJP+11]. Many commercial DBaaS providers offer standard database systems (e.g., MongoDB, Redis, and MySQL) as a managed service. For example MongoDB Atlas provides a managed NoSQL database [Mon17], Amazon Elastic Map-Reduce [Ama17b] is an Analytics-as-a-Service based on managed Hadoop clusters, and Azure SQL Server offers a managed RDBMS [MTK+11].

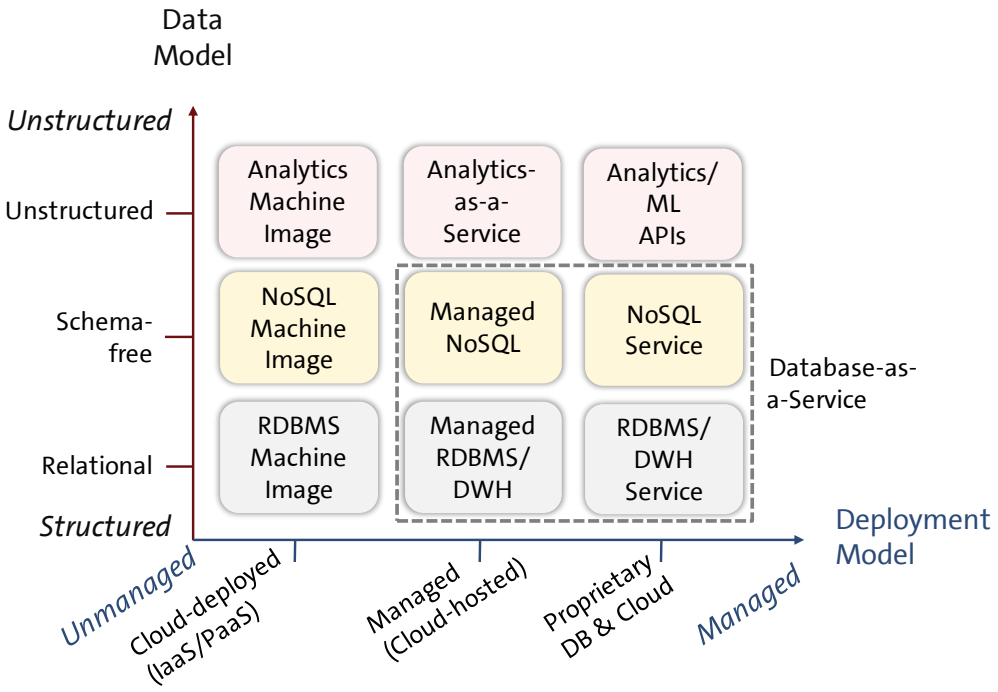


Figure 2.13: Classes of cloud databases and DBaaS systems according to their data model and deployment model.

DBaaS providers can also specifically develop a **proprietary database or cloud infrastructure** to achieve scalability and efficiency goals that are harder to implement with standard database systems. A proprietary architecture enables co-design of the database or analytics system with the underlying cloud infrastructure. For example, Amazon DynamoDB provides a large-scale, multi-tenant NoSQL database loosely based on the Dynamo architecture [Dyn17], and Google provides machine learning (ML) APIs for a variety of classification and clustering tasks [Goo17b].

This thesis will be primarily concerned with the *managed* deployment model, as we are convinced that existing database technology offers a suitable basis for data management functionality but currently lacks the non-functional ability to be provided as a low-latency DBaaS/BaaS. We refer to Lehner and Sattler [LS13] and Zhao et al. [ZSLB14] for a comprehensive overview on DBaaS research.

Backend-as-a-Service

Many data access and application patterns are very similar across different web and mobile applications and can therefore be standardized. This was recognized by the industry and lead to BaaS systems that integrate DBaaS with application logic and predefined building blocks, e.g., for push notifications, user login, static file delivery, etc. BaaS is a rather recent trend and similar to early cloud computing and Big Data processing, progress is currently driven by industry projects, while structured research has yet to be established [Use17, Par17, Dep17].

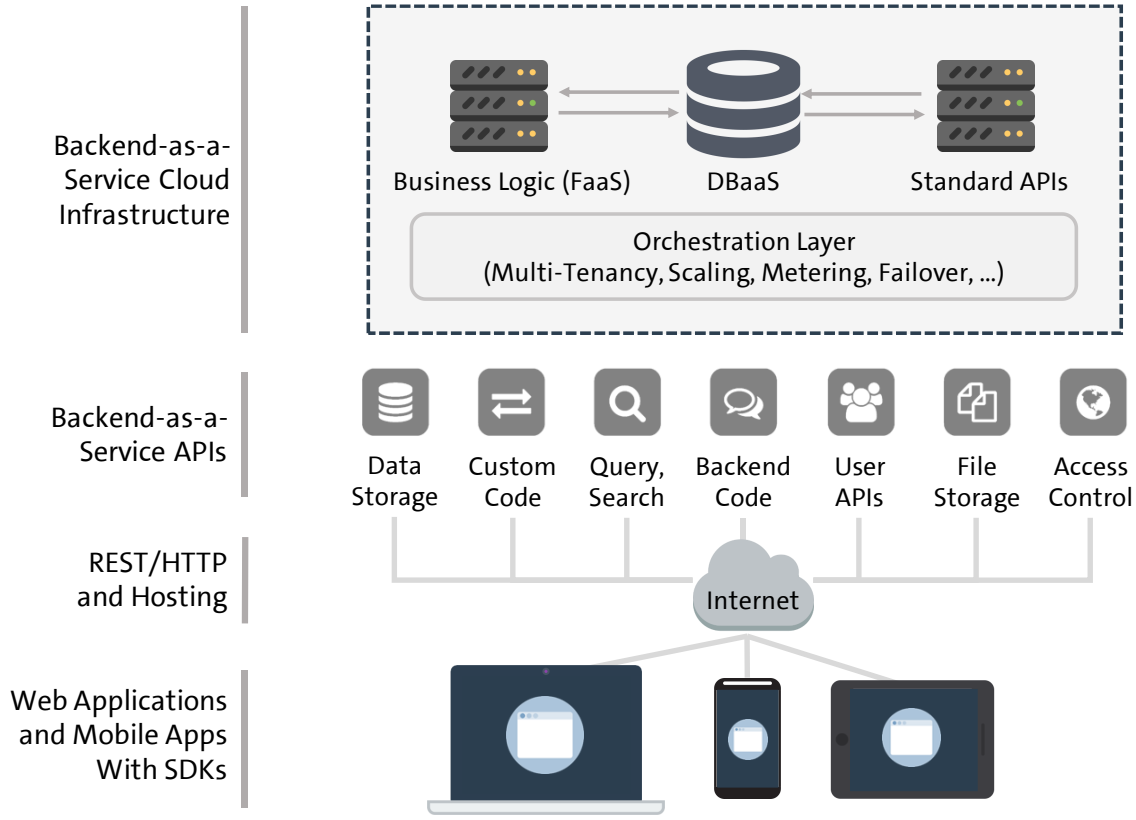


Figure 2.14: Architecture and usage of a Backend-as-a-Service.

Figure 2.14 gives an overview of a generic BaaS architecture as similarly found in commercial services (e.g., Azure Mobile Services, Firebase, and Kinvey [Baq18]) as well as open-source projects (e.g., Meteor [HS16], Deployd [Dep17], Hoodie [Hoo17], Parse Server [Par17], BaasBox [Bas17], and Apache UserGrid [Use17]).

The BaaS cloud infrastructure consists of three central components. The DBaaS component is responsible for data storage and retrieval. Its abstraction level can range from structured relational, over semi-structured JSON to opaque files. The FaaS component is concerned with the execution of server-side business logic, for example, to integrate third-party services and perform data validation. It can either be invoked as an explicit API or be triggered by DBaas operations. The standard API component offers common application functionality in a convention-over-configuration style, i.e., it provides defaults for tasks such as user login, push notifications, and messaging that are exposed for each tenant individually. The cloud infrastructure is orchestrated by the BaaS provider to ensure isolated multi-tenancy, scalability, availability, and monitoring.

The BaaS is accessed through a REST API [Dep17,Hoo17,Par17,Bas17,Use17] (and sometimes WebSockets [HS16]) for use with different client technologies. To handle not only native mobile applications but also websites, BaaS systems usually provide file hosting to deliver website assets like HTML and script files to browsers. The communication with the BaaS is performed through SDKs employed in the frontend. The SDKs provide high-level

abstractions to application developers, for example, to integrate persistence with application data models [TGPM17].

BaaS systems are thus confronted with even stronger latency challenges than a DBaaS: all clients access the system via high-latency WAN network so that latency for retrieving objects, files, and query results determines application performance. Similar to DBaaS systems, BaaS APIs usually provide persistence on top of one single database technology, making it infeasible to achieve all potential functional and non-functional application requirements. The problem is even more severe when all tenants are co-located on a shared database cluster. In that case, one database system configuration (e.g., the replication protocol) prescribes the guarantees for each tenant [ADE12].

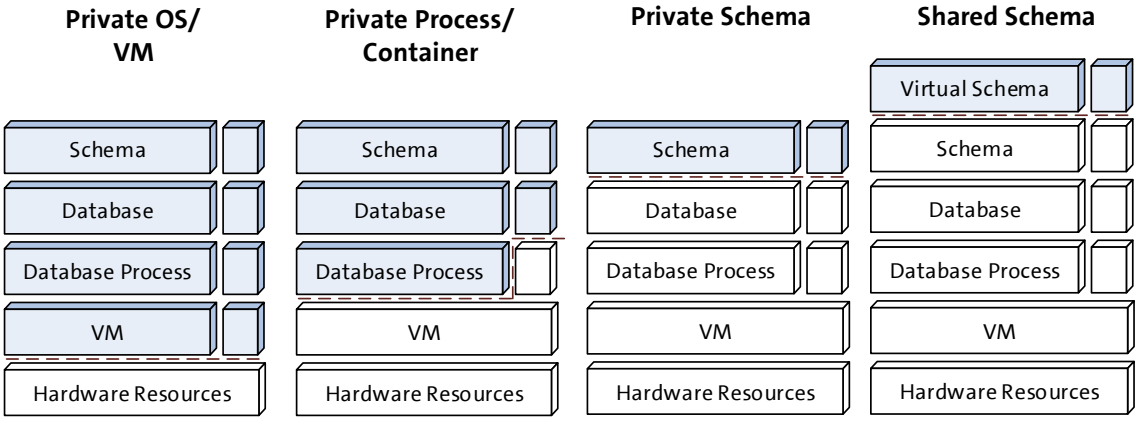


Figure 2.15: Different approaches to multi-tenancy in DBaaS/BaaS systems. The dashed line indicates the boundary between shared and tenant-specific resources.

Multi-Tenancy

The goal of multi-tenancy in DBaaS/BaaS systems is to allow efficient resource pooling across tenants so that only the capacity for the global average resource consumption has to be provisioned and resources can be shared. There is an inherent trade-off between higher isolation of tenants and efficiency of resource sharing [ADE12]. As shown in Figure 2.15, the boundary between tenant-specific resources and shared provider resources can be drawn at different levels of the software stack [MB16, p. 562]:

- With **private operating system (OS)** virtualization, each tenant is assigned to one or multiple VMs that execute the database process. This model achieves a high degree of isolation, similar to IaaS clouds. However, resource reuse is limited as each tenant has the overhead of a full OS and database process.
- By allocating a **private process** to each tenant, the overhead of a private OS can be mitigated. To this end, the provider orchestrates the OS to run multiple isolated database processes. This is usually achieved using container technology such as Docker [Mer14] that isolates processes within a shared OS.

- Efficiency can be further increased if tenants only possess a **private schema** within a shared database process. The database system can thus share various system resources (e.g., the buffer pool) between tenants to increase I/O efficiency.
- The **shared schema** model requires all tenants to use the same application that dictates the common schema. The schema can be adapted to specific tenant requirements by extending it with additional fields or tables [KL11]. A shared schema is frequently used in SaaS applications such as Salesforce [Onl17].

The major open challenge for multi-tenancy of NoSQL systems in cloud environments is database independence and the combination with multi-tenant FaaS code execution. If a generic middleware can expose unmodified data stores as a scalable, multi-tenant DBaaS/BaaS, the problems of database and service architectures are decoupled, and polyglot persistence is enabled. In Chapter 3 we will outline the requirements for a generic, multi-tenant DBaaS/BaaS architecture and present our approach.

Most research efforts in the DBaaS community have been concerned with multi-tenancy and virtualization [ASJK11, AGJ+08, AJKS09, KL11, SKM08, WB09, JA07], database privacy and encryption [KJH15, Gen09, PRZB11, Pop14, KFPC16, PZ13, PSV+14], workload management [CAAS07, ZSLB14, ABC14, Bas12, XCZ+11, TPK+13, LBMAL14, PSZ+07, Sak14], resource allocation [MRSJ15, SLG+09], automatic scaling [KWQH16, LBMAL14], and benchmarking [DFNR14, CST+10, CST+10, PPR+11, BZS13, BKD+14, BT11, BK13, BT14, Ber15, Ber14] (see Section 6.4.4). However, several DBaaS and BaaS challenges have remained unsolved. This thesis is focused on providing the following improvements to DBaaS and BaaS systems:

- **Low latency** access to DBaaS systems, to improve application performance and allow distribution of application logic and data storage
- Unified REST/HTTP access to polyglot data stores with **service level agreements** for functional and non-functional guarantees
- **Elastic scalability** of read and query workloads for arbitrary database systems
- Generic, **database-independent APIs and capabilities** for fundamental data management abstractions such as schema management, FaaS business logic, real-time queries, multi-tenancy, search, transactions, authentication, authorization, user management, and file storage for single databases and across databases.

2.2.7 Latency Problems in Distributed Transaction Processing

Transactions are one of the central concepts in data management, as they solve the problem of keeping data correct and consistent under highly concurrent access. While the adoption of distributed NoSQL databases first lead to a decline in the support of transactions, recently numerous systems have started to support transactions again, often with relaxed guarantees (e.g., Megastore [BBC+11], G-Store [DAEA10], ElasTras [DAEA13], Cloud SQL Server [BCD+11], Spanner [CDE+12], F1 [SVS+13], Percolator [PD10], MDCC

[KPF+13], TAPIR [ZSS+15], CloudTPS [WPC12], Cherry Garcia [DFR15a], FaRMville [DNN+15], Omid [GJK+14], RAMP [BFG+14], Walter [SPAL11], Calvin [TDW+12], H-Store/VoltDB [KKN+08]). The core challenge is that serializability – like strong consistency – enforces a difficult trade-off between high availability and correctness in distributed systems [BDF+13].

The gold standard for transactions is ACID [WV02, HR83]:

Atomicity. A transaction must either commit or abort as a complete unit. Atomicity is implemented through recovery, rollbacks, and atomic commitment protocols.

Consistency. A transaction takes the database from one consistent state to another. Consistency is implemented through constraint checking and requires transactions to be logically consistent in themselves.

Isolation. The concurrent and interleaved execution of operations leaves transactions isolated, so that they do not affect each other. Isolation is implemented through concurrency control algorithms.

Durability. The effects of committed transactions are persistent even in the face of failures. Durability is implemented through logging, recovery, and replication.

A comprehensive overview of centralized and distributed transactions is given by Agrawal et al. [ADE12], Weikum and Vossen [WV02], Öszu and Valduriez [ÖV11], Bernstein and Newcomer [BN09], and Sippu and Soisalon-Soininen [SSS15].

Distributed Transaction Architectures

A *transaction* is a finite sequence of read and write operations. The interleaved operations of a set of transactions is called a *history* and any prefix of it is a *schedule* [WV02]. To provide isolation, concurrency control algorithms only allow schedules that do not violate isolation. The strongest level of isolation is serializability. However, many concurrency control protocols allow certain update anomalies[14] for performance reasons, forming different *isolation levels* of relaxed transaction isolation.

The strongest isolation level of serializability can also be refined into different classes of histories, depending on defined correctness criteria [WV02, p. 109]. In practice and in the context of this thesis, the most relevant class is conflict-serializability (CSR), and its subclass commit order-preserving conflict serializability (COCSR). CSR and COCSR are efficiently decidable and easy to reason about from a developer's perspective.

Figure 2.16 gives an overview of typical distributed transaction architectures as originally described by Gray [GL06] and Liskov [LCSA99] and still used in most systems

[14] Update anomalies describe undesired behavior caused by transaction interleaving [ALO00, Ady99]. A *dirty write* overwrites data from an uncommitted transaction. With a *dirty read*, stale data is exposed. A *lost update* describes a write that does not become visible, due to two transactions reading the same object version for a subsequent write. A *non-repeatable read* occurs if data read by an in-flight transaction was concurrently overwritten. A *phantom read* describes a predicate-based read that becomes invalid due to concurrent transactions writing data that matches the query predicate. *Read and Write Skew* are two anomalies caused by transactions operating on different, isolated database snapshots.

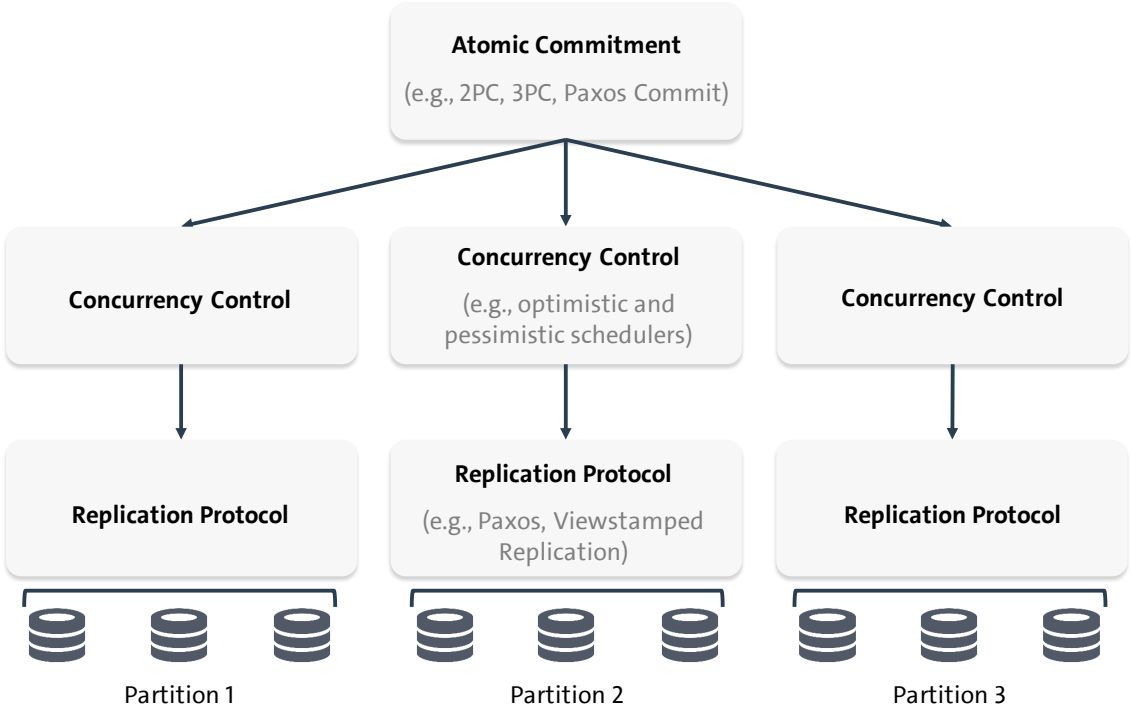


Figure 2.16: Distributed transaction architecture consisting of an atomic commitment protocol, concurrency control and a replication protocol.

[BBC+11, CDE+13, EWS13]. Distributed databases are partitioned into shards, with each shard being replicated for fault tolerance. Therefore, an atomic commitment protocol is required to enforce an all-or-nothing decision across all shards. Common protocols such as two-phase commit (2PC) [Lec09], three-phase commit (3PC) [SS83], and Paxos Commit [GL06] have to make a trade-off between availability and correctness: any correct atomic commitment protocol blocks under some network partitions. The replication protocol is required to keep replicas in sync, so that staleness does not interfere with the concurrency control algorithm. Traditionally, the replication protocol has to ensure linearizability (e.g., through Paxos [Lam98], Virtual Synchrony [BJ87], and Viewstamped Replication [OL88]) but it has been shown that an appropriate concurrency control scheme can potentially tolerate weaker consistency of the underlying replication protocol without compromising isolation [ZSS+15].

There are alternatives to the described transaction processing architecture (e.g., Replicated Commit [MNP+13]). In particular, the atomic commitment protocol is a bottleneck that can be prevented, as we will discuss for our DCAT transaction design in Section 4.8. Furthermore, current transaction architectures assume that the client executing a transaction is a server co-located with the database system. However, in a BaaS scenario, a client can be an (untrusted) end user's mobile device or browser that exhibits very high read, query, and write latency.

Concurrency Control

Concurrency control schemes can be grouped into **pessimistic** and **optimistic** approaches. Pessimistic schemes proactively prevent isolation violations during transaction execution. Optimistic schemes do not interfere with the transaction execution and validate the absence of violations at commit time. The major concurrency control algorithms are:

Lock-based protocols. For operations that would create cyclic conflicts, mutual exclusion can be achieved through locking. According to the two-phase locking (2PL) theorem, any execution of transactions that use 2PL is serializable [EGLT76]. The granularity and types of locks vary in different protocols, as well as the specifics of 2PL [WV02, ÖV11, BN09]. All 2PL-based protocols without preclaiming (acquiring locks at transaction begin) suffer from potential deadlocks or external aborts[15] [GHKO81, GLPT76]. Preclaiming on the other hand is not applicable if accessed objects are unknown in advance but determined through queries, reads, or user interactions.

Non-Locking Pessimistic Protocols. Timestamp Ordering (TO) [Ber99] enforces serializability by ordering conflicting operations by the begin timestamp of transactions. The main downside of TO schedulers is that they produce only a small subset of CSR schedules and therefore cause unnecessary aborts. Serialization Graph Testing (SGT) [Cas81] is another non-locking pessimistic scheme that constructs the conflict graph and prevents it from becoming cyclic. The internal state of SGT can become very large as it is non-trivial to discard old transactions' information.

Multi-version Concurrency Control (MVCC). A straightforward improvement of pessimistic protocols is to decouple concurrent reads by executing them on an immutable snapshot. TO, SGT, and 2PL can easily be extended to incorporate multi-versioning [WV02]. Due to reduced conflict rates, MVCC schedulers such as Serializable Snapshot Isolation [CRF08, FLO$^+$05, PG12] are popular among RDBMSs.

Optimistic Concurrency Control (OCC). Optimistic schedulers operate across three transaction phases. The principle idea is to allow all transactional operations and to apply rollbacks at commit time when serializability would be violated [KR81].

1. **Read Phase.** In the read phase, the transaction performs its operations, including reads, writes, and queries. Writes are not applied to the database but buffered until commit, typically in the client.
2. **Validation Phase.** The validation phase is executed as a critical section and ensures that the transaction can safely commit. The type of validation depends on the optimistic protocol. In Forward-Oriented Optimistic Concurrency Control (**FOCC**) the committing transaction's write set is validated against the read set of all parallel transactions that are still in the read phase [Här84]. In Backward-

[15] Follwing the terminology of Bailis et al. [BDF$^+$13] we refer to *external aborts* as transaction rollbacks caused by a system's implementation (e.g., for deadlock prevention) whereas *internal aborts* are triggered by the transaction itself (e.g., as a rollback operation).

Oriented Optimistic Concurrency (**BOCC**), the committing transaction's read set is validated against the write set of all transactions that completed while the committing transaction was in the read phase. To resolve a conflict, two strategies are possible:

- **Kill/Broadcast-OCC**: transactions that are running and preventing the committing transaction from completing are aborted.
- **Die**: the committing transaction aborts.

In BOCC, only the *Die* strategy is applicable, as conflicting transactions are already committed. FOCC permits both resolution strategies. However, FOCC has two important drawbacks. First, it needs to consider reads of active transactions, which prevents serving them from caches or replicas. Second, the FOCC validation has to block concurrent reads and thus strongly limits concurrency and performance.

3. **Write Phase.** If validation was successful, the transaction's changes are persisted to the database and made visible. Usually, this also includes writing recovery information into logs to ensure durability.

The problem of pessimistic concurrency control is that preventing violations of serializability requires transactional reads and writes to be forwarded to the scheduler. In replicated or cached systems, this defeats the purpose of **data distribution**. This also applies to MVCC as it requires local tracking of transaction-specific versions which cannot be offloaded to replicas or caches without including them in the concurrency control algorithm. Therefore, in highly distributed systems, optimistic transactions are advantageous, as they allow to combine client-local processing of reads and writes with a global commit decision [BBC+11, DAEA10, DAEA13, CDE+12, SVS+13, DFR15a, DNN+15]. Stonebraker et al. [SMA+07] identify "locking-based concurrency control mechanisms" as a substantial performance bottleneck and one of the relics of System R that hinder the progress of database systems.

Impact of Latency On Transaction Success

Compared to pessimistic mechanisms, optimistic concurrency control offers the advantage of never blocking running transactions due to lock conflicts. The downside of optimistic transactions is that they can lead to transaction aborts since this is the only way of handling cyclic read/write conflicts [KR81].

Locking strategies suffer from deadlocks. Let A be a random variable that describes the outcome of a transaction. Gray et al. [GHKO81] showed that the probability of aborts $P(A = 1)$ increases with the second power of the number T of parallel transactions and with the fourth power of transaction duration D [BN09]:

$$A(w) = \begin{cases} 0 & \text{if w = commit} \\ 1 & \text{if w = abort} \end{cases} \tag{2.3}$$

$$P(A=1) \sim D^4 \text{ and } P(A=1) \sim T^2 \tag{2.4}$$

Deadlocks are resolved by rollbacks. Thus, the more high-latency reads are involved in a pessimistic transaction, the higher the abort probability. In general, optimistic transactions are superior for read-intensive workloads while pessimistic transactions are more appropriate for write-intensive workloads [WV02].

In a simplified model, Franaszek et al. [FRT92] showed the **quadratic effect** of optimistic transactions that states that the abort probability is k^2/N, where k is the number of objects accessed in transactions and N the size of the database [Tho98]. This model assumes preclaiming, an even access probability across all objects, and that every read object is also written. In that case, if the first transaction accesses n objects and the second m, the probability of accessing at least one object in both transactions is:

$$P(n,m) = 1 - \frac{\binom{N-n}{m}}{\binom{N}{m}} \approx 1 - (1 - \frac{n}{N})^m \approx \frac{nm}{N} \tag{2.5}$$

Thus, if all transactions read and write k objects, the abort probability for two concurrent transactions is $P(k,k) = \frac{k^2}{N}$, the quadratic effect. However, this model has many limitations, most importantly the assumption of preclaiming, the missing distinction between reads and writes, and the discarded influence of latency. We will therefore derive a more realistic model in Section 4.8 to signify the impact of latency.

Example of High-Latency Transactions

To illustrate the role of latency in transaction processing, we briefly discuss an example application use case. In the web, high latency is ubiquitous, especially for applications employing the DBaaS and BaaS model. Transactions requiring client-server round-trips are therefore usually avoided through heuristics, compensations, and other non-transactional workarounds.

As an example consider a checkout process in a booking system, e.g., for an airline or a theatre. A transaction would proceed in two steps:

1. The available seats are read from the database and shipped over a high-latency network to the end user.
2. The end user performs a selection of seats in the frontend and sends a booking or reservation request (i.e., a write) to the system, back over the high-latency network.

This use case is difficult to implement with lock-based concurrency control, as applying read locks in step 1 would cause very high deadlock probabilities and block resources in the database system. In practice, this use case is solved by decoupling step 1 and step 2 into two unrelated transactions [SF12]. If step 2 cannot be applied due to a violation of isolation (i.e., seats were concurrently booked) the transaction is rolled back, and the user is presented with an error. This solution is effectively an optimistic transaction implemented in the application layer. Even a database system with native optimistic concur-

rency control could not prevent these errors. Furthermore, for security reasons, a database transaction API cannot be exposed to end users, but only to the server-side business logic tier. For modern web applications, it would therefore be preferable to provide the transaction logic in an application-independent, client-accessible way while minimizing aborts through shorter transaction durations.

Challenges

In summary, the major challenges of distributed transaction processing tackled in this thesis are:

- **High-latency environments** have a detrimental effect on transaction aborts in both pessimistic and optimistic concurrency control algorithms.
- Current concurrency control schemes prevent leveraging **geo-distributed caching**, as they cannot prevent or bound staleness.
- **Polyglot persistence transactions** are usually impossible or require explicit support from each system. Enhancing non-transactional database systems with cross-database ACID transactions is an open research challenge.
- Transaction APIs are traditionally designed for three-tier applications and do not support end users directly executing transactions. This type of access is required for **BaaS architectures** and simplifies the development of data-driven web applications.

2.2.8 Low-Latency Backends through Replication, Caching, and Edge Computing

There are three primary backend-focused technologies that are concerned with lowering latency. Replication, caching, and edge computing follow the idea of distributing data storage and processing for better scalability and reduced latency towards dispersed clients. However, in their current form they do not solve the end-to-end latency problem for reads, queries, and transactions, cannot be combined in polyglot persistence architectures, and do not allow dynamic trade-offs between consistency and performance requirements.

Eager and Lazy Geo-Replication

To improve scalability and latency of reads, **geo-replication** distributes copies of the primary database over different geographical sites. *Eager geo-replication* (e.g., in Google's Megastore [BBC+11], Spanner [CDE+13, CDE+12], F1 [SVS+13], MDCC [KPF+13], and Mencius [MJM08]) has the goal of achieving strong consistency combined with geo-redundancy for failover. However, it comes at the cost of higher write latencies that are usually between 100 ms [CDE+12] and 600 ms [BBC+11]. The second problem of eager geo-replication is that it requires extensive, database-specific infrastructure which introduces system-specific trade-offs that cannot be adapted at runtime. For example, it is not possible to relax consistency on a per-operation basis, as the guarantee is tied to

the system-wide replication protocol (typically variants of Paxos [Lam01]). Also, while some eagerly geo-replicated systems support transactions, these suffer from high abort rates, as cross-site latency in commit protocols increases the probability of deadlocks and conflicts [SVS+13].

Lazy geo-replication (e.g., in Dynamo [DHJ+07], BigTable/HBase [CDG+08, Hba17], Cassandra [LM10], MongoDB [CD13], CouchDB [ALS10], Couchbase [LMLM16], Espresso [QSD+13], PNUTS [CRS+08], Walter [SPAL11], Eiger [LFKA13], and COPS [LFKA11]) on the other hand aims for high availability and low latency at the expense of consistency. Typically, replicas are only allowed to serve reads, in order to simplify the processing of concurrent updates. The problem of lazy geo-replication is that consistency guarantees are lowered to a minimum (eventual consistency) or cause a prohibitive overhead (e.g., causal consistency [LFKA11, LFKA13]). Similar to eager geo-replication, system-specific infrastructure is required to scale the database and lower latency. Therefore, providing low end-to-end latency for web applications through a network of different replica sites is often both financially and technically infeasible. Furthermore, geo-replication requires the application tier to be co-located with each replica to make use of the distribution for latency reduction. Geo-replication is nonetheless an indispensable technique for providing resilience against disaster scenarios.

Caching

Caching has been studied in various fields for many years (see Section 6.1). It can be applied at different *locations* (e.g., clients, proxies, servers, databases), *granularities* (e.g., files, records, pages, query results) and with different *update strategies* (e.g., expirations, leases, invalidations). Client-side caching approaches are usually designed for application servers and therefore not compatible with REST/HTTP, browsers and mobile devices [ÖV11, FCL97, WN90, KK94, ÖV11, CALM97, Ora17, TGPM17]. Mid-tier (proxy) caches provide weak guarantees in order not to create synchronous dependencies on server-side queries and updates or only cache very specific types of data [KW97, TWJN01, KW98, PB08, FFM04, Fre10, Vak06, YADL99, YADL98, BDK+02]. The various approaches for server-side caching have the primary goal of minimizing query latency by offloading the database for repeated queries [AAO+12, CLL+01b, KLM97, Kam17, GMA+08, BAC+13, APTP03a, BAM+04, LGZ04, LR01a, BBJ+10, LLXX09].

Combining expiration-based and invalidation-based cache maintenance is an open problem, as both mechanisms provide different consistency guarantees and therefore would degrade to the weaker model when combined. In practice, most caching approaches rely on the application to maintain cache coherence instead of using declarative models that map consistency requirements to cache coherence protocols [Rus03a, ABMM07, CSH+16, SHKS15, Ama16, Fit04, NFG+13, XFJP14]. Very few caching approaches tackle end-to-end latency for the web at all or consider the distributed nature of cloud services. Caching and replication approaches bear many similarities, as caching is a form of lazy, on-demand

replication [RS03]. In this work, we will consolidate previous work in both fields into a combined mechanism.

Edge Computing

A **cloudlet** is a "data center in a box" [AG17, p. 7] that can be deployed in proximity to mobile devices for reduced latency. The idea is to enhance the computing capacities of mobile devices by offloading computationally expensive operations to cloudlets [SBCD09]. Typical applications for the concept of cloudlets are virtual and augmented reality that require powerful resources for rendering and low latency for interactivity. For data management, cloudlets are less useful as they would have to replicate or cache data from the main data center and would therefore have to act as a geo-replica.

Fog computing takes the idea of highly distributed cloud resources further and suggests provisioning storage, compute, and network resources for Internet of Things (IoT) applications in a large amount of interconnected "fog nodes" [BMZA12]. By deploying fog nodes close to end users and IoT devices, better quality of service for latency and bandwidth can potentially be achieved. Fog computing targets applications such as smart grids, sensor networks, and autonomous driving and is therefore orthogonal to web and mobile applications [SW14].

Edge computing refers to services and computations provided at the network edge. Edge computing in CDNs has already been practiced for years through reverse proxy caches that support restricted processing of incoming and outgoing requests [Kam17, PB08]. *Mobile edge computing* enhances 3G, 4G, and 5G base stations to provide services close to mobile devices (e.g., video transcoding) [AR17].

The problem of cloudlets, fog computing, and edge computing regarding low latency for web applications is that they do not provide integration into data management and shared application data but instead expose independent resources. Therefore, data shipping is required to execute business logic on the edge which shifts the latency problem to the communication path between edge nodes and cloud data storage. We will, however, show that edge computing can support end-to-end latency reduction by performing data management operations on cached data, in particular, authentication and authorization.

Challenges

For a detailed treatment comparing replication and caching approaches from the literature, please refer to Chapter 6. In summary, the open challenges of replication, caching, and edge computing for low latency cloud data management are:

- Eager geo-replication introduces high **write and commit latency**, while lazy geo-replication does not allow fine-grained **consistency choices**.
- Replication requires extensive, database-specific infrastructure and cannot be employed for **polyglot persistence**.

- Geo-replicated database systems assume the co-distribution of **application logic** and do not have the abstractions and interfaces for direct DBaaS/BaaS access by clients.
- Common caching approaches only improve **backend performance** instead of end-to-end latency or suffer from the same limitations as geo-replication.
- **Expiration-based caching** is considered irreconcilable with non-trivial consistency requirements.
- Edge computing does not solve the **distribution of data** and hence does not improve latency for stateful computations and business logic.

2.3 Network Performance: HTTP and Web Caching

For any distributed application, the network plays a significant role for performance. In the web, the central protocol is HTTP (Hypertext Transfer Protocol) [FGM+99] that determines how browsers communicate with web servers and that is used as the basis for REST APIs (*Representational State Transfer*). For cloud services across different deployment and service models, REST APIs are the default interface for providing access to storage and compute resources, as well as high-level services. Most DBaaS, BaaS, and NoSQL systems provide native REST APIs to achieve a high degree of interoperability and to allow access from heterogeneous environments. This section reviews relevant foundations of HTTP and networking with respect to performance and latency, as well as their role in cloud data management. In particular, we will highlight which challenges the standardized behavior of the web caching infrastructure imposes for data-centric services.

2.3.1 HTTP and the REST Architectural Style

The REST architectural style was proposed by Fielding as an a-posteriori explanation for the success of the web [Fie00]. REST is a set of constraints that – when imposed on a protocol design – yield the beneficial system properties of scalability and simplicity the designers of the HTTP standard developed for the web [FGM+99]. Most services in cloud computing environments are exposed as REST/HTTP[16] services, as they are simple to understand and consume in any programming language and environment [DFR15b]. Another advantage of HTTP is its support by mature and well-researched web infrastructure. REST and HTTP are not only the default for web and mobile applications but also an alternative to backend-side RPC-based (Remote Procedure Call) approaches (e.g., XML RPC or Java RMI [Dow98]), binary wire protocols (e.g., PostgreSQL protocol [Pos17]) and web services (the SOAP and WS-* standards family [ACKM04]).

HTTP is an application-layer protocol on top of the Transmission Control Protocol (TCP) [Pos81] and lies the foundation of the web. With REST, the key abstractions of interactions

[16] In principle, the REST architectural style is independent of its underlying protocol. However, as HTTP dominates in practical implementations, we will refer to REST as its combination with HTTP [WP11].

are represented by HTTP *resources* identified by URLs. In a DBaaS API, these resources could for example be queries, transactions, objects, schemas, and settings. Clients interact with these resources through the *uniform interface* of the HTTP methods `GET`, `PUT`, `POST` and `DELETE`. Any interface is thus represented as a set of resources that can be accessed through HTTP methods. Methods have different semantics: `GET` requests are called *safe*, as they are free of side-effects (nullipotent). `PUT` and `DELETE` requests are idempotent, while `POST` requests may have side-effects that are non-idempotent. The actual data (e.g., database objects) can take the form of any standard *content type* which can dynamically be negotiated between the client and server through HTTP (*content negotiation*). Many REST APIs have default representations in JSON, but other formats (e.g., XML, text, images) are possible, too. This extensibility of REST APIs allows services to present responses in a format that is appropriate for the respective use case [RAR13].

The integration and connection of resources is achieved through *hypermedia*, i.e., the mutual referencing of resources [Amu17]. These references are similar to links on web pages. A resource for a query result could for instance have references to the objects matching the query predicate. Hypermedia can render a REST interface self-descriptive. In that case, an initial URL to a root resource is sufficient to explore the complete interface by following references and interpreting self-describing standard media types. HTTP is a request-response protocol, which means that the client has to pose a request to receive a response. For the server to proactively push data, other protocols are required.

The *constraints* of REST describe common patterns to achieve scalability [Fie00]. In the context of cloud services, these constraints are:

Client-Server. There is a clear distinction between a client (e.g., a browser or mobile device) and the server (e.g., a cloud service or web server) that communicate with each other using a client-initiated request-response pattern [FGM+99].

Statelessness. If servers are stateless, requests can be load-balanced, and servers may be replicated for horizontal scalability.

Caching. Using caching, responses can be reused for future requests by serving them from intermediate web caches.

Uniform Interface. All interactions are performed using four basic HTTP methods to create, read, update, and delete (*CRUD*) resources.

Layered System. Involving intermediaries (e.g., web caches, load balancers, firewalls, and proxies) in the communication path results in a layered system.

In practice, many REST APIs do not adhere to all constraints and are often referred to as web APIs, i.e., custom programming interfaces using HTTP as a transport protocol [RAR13]. For example, the Parse BaaS uses `POST` methods to perform idempotent operations and `GET` requests for operations with side-effects [Par17]. As a consequence, such web APIs are potentially unscalable and may be treated incorrectly by intermediaries. Unlike web services, REST does not require interface descriptions and service discovery.

However, the OpenAPI initiative is an attempt to standardize the description of REST APIs and allowing code generation for programming languages [Ope17]. Richardson et al. [RAR13], Allamaraju [All10], Amundsen [Amu17], and Webber et al. [WPR10] provide a comprehensive treatment of REST and HTTP.

The challenge for data management is to devise a REST API that leverages HTTP for scalability through statelessness and caching and that is generic enough to be applicable to a broad spectrum of database systems. To this end, a resource structure for different functional capabilities is required (e.g., queries and transactions) as well as system-independent mechanisms for stateless request processing and caching of reads and queries.

2.3.2 Latency on the Web

For interoperability reasons, REST APIs are the predominant type of interface in cloud data management. HTTP on the other hand has to be used by any website. The performance and latency of HTTP communication are determined by the protocols that are involved during each HTTP request.

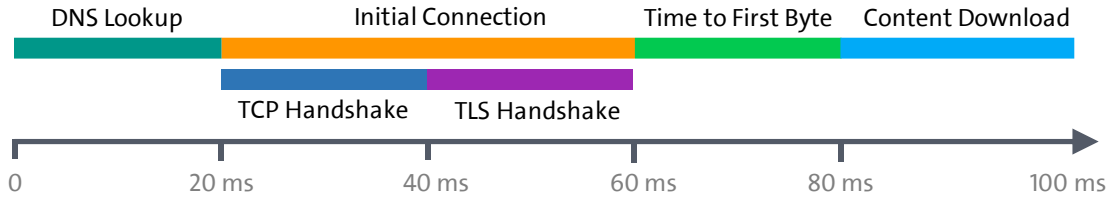


Figure 2.17: Latency components across network protocols of an HTTP request against a TLS-secured URL.

Figure 2.17 shows the latency components of a single HTTP request illustrated with exemplary delays:

1. First, the URL's domain (e.g., `example.com`) is resolved to an IP address using a UDP-based **DNS lookup**. To this end, the client contacts a configured DNS resolver. If the DNS entry is uncached, the resolver will contact a root DNS server that redirects to a DNS server responsible for the top-level domain (e.g., for `.com`). That name server will in turn redirect to the authoritative name server registered by the owner of the domain. This name server then returns one or multiple IP addresses for the requested host name. Depending on the location of the (potentially geo-redundant) DNS servers and the state of their caches, a typical DNS query will return in 10-100 ms. Like in HTTP, DNS caching is based on TTLs with its associated staleness problems [TW11].
2. Next, a **TCP connection** between the client and the server is established using a three-way handshake. In the first round-trip, connection parameters are negotiated (`SYN`, `SYN-ACK` packets). In the second round-trip, the client can send the first portion

of the payload. There is ongoing research on *TCP fast open* [CCRJ14], a mechanism to avoid one round-trip by sending data in the first `SYN` packet.

3. If the server supports and requires end-to-end encryption through HTTPS, respectively TLS (Transport Layer Security), a **TLS handshake** needs to be performed [Gri13]. This requires two additional round-trips during which the server's certificate is checked, session keys are exchanged, and a cipher suite for encryption and signatures is negotiated. TLS protocol extensions have been specified to allow data transmission during half-open TLS connections to reduce TLS overhead to one round-trip (*TLS false start*). Alternatively, clients can reuse previous session parameters negotiated with the same server, to abbreviate the handshake (*TLS session resumption*)[17].
4. When the connection is established, the client sends an HTTP request that consists of an HTTP method, a URL, the protocol version as well as HTTP headers encoding additional information like the desired content type and supported compression algorithms. The server processes the requests and either fully assembles the response or starts transmitting it, as soon as data is available (*chunked encoding*). The delay to the moment where the client receives the first response bytes is referred to as **time-to-first-byte** (TTFB).
5. Even though the connection is fully established, the response cannot necessarily be transmitted in a single round-trip, but requires multiple iterations for the **content download**. TCP employs a *slow-start algorithm* that continuously increases the transmission rate until the full aggregate capacity of all involved hops is saturated without packet loss and congestion [MSMO97]. Numerous congestion control algorithms have been proposed, most of which rely on packet loss as an indicator of network congestion [KHR02, WDM01]. For large responses, multiple round-trips are therefore required to transfer data over a newly opened connection, until TCP's congestion window is sufficiently sized[18]. Increasing the initial TCP congestion window from 4 to 10 segments is ongoing work [CCDM13] and allows for typically $10 \cdot 1500\,\text{B} = 15\,\text{KB}$ of data transmitted with a single round-trip, given the maximum transmission unit of 1500 B of an Ethernet network.

In the best case and with all optimizations applied, an HTTP request over a new connection can hence be performed with one DNS round-trip and two server round-trips. DNS requests are aggressively cached, as IPs for DNS names are considered stable. The DNS overhead is therefore often minimal and can additionally be tackled by geo-replicated DNS servers that serve requests to nearby users (*DNS anycast*). To minimize the impact of TCP

[17] Furthermore, the QUIC (*Quick UDP Internet Connections*) protocol has been proposed as UDP-based alternative to HTTP that has no connection handshake overhead [Gri13]. A new TLS protocol version with no additional handshakes has also been proposed [Res17].

[18] The relationship between latency and potential data rate is called the *bandwidth-delay product* [Gri13]. For a given round-trip latency (delay), the effective data rate (bandwidth) is computed as the maximum amount of data that can be transferred (product) divided by the delay. For example, if the current TCP congestion window is 16 KB and the latency 100 ms, the maximum data rate is 1.31 MBit/s.

and TLS handshakes, clients keep connections open for reuse in future requests, which is an indispensable optimization, in particular for request-heavy websites.

The current protocol version 2 of HTTP [IET15] maintains the semantics of the original HTTP standard [KR01] but improves many networking inefficiencies. Some optimizations are inherent, while others require active support by clouds services:

- **Multiplexing** all requests over one TCP connection avoids the overhead of multiple connection handshakes and circumvents head-of-line blocking[19].
- **Header Compression** applies compression to HTTP metadata to minimize the impact of repetitive patterns (e.g., always requesting JSON as a format).
- If a server implements **Server Push**, resources can be sent to the client proactively whenever the server assumes that they will be requested. This requires explicit support by cloud services, as the semantics and usage patterns define, which content should be pushed to reduce round-trips. However, inadequate use of pushed resources hurts performance, as the browser cache is rendered useless.
- By defining dependencies between resources, the server can actively **prioritize** important requests.

As of 2017, still less than 20% of websites and APIs employ HTTP/2 [Usa17]. The techniques developed in this thesis apply to both HTTP/1.1 and HTTP/2, but profit from the improvements of HTTP/2. When all above protocols are in optimal use, the remaining latency bottleneck is the round-trip latency between API and browser clients and the server answering HTTP requests.

In **mobile networks**, the impact of HTTP request latency is even more severe. Additional latency is caused by the mobile network infrastructure. With the older 2G and 3G mobile network standards, latencies between 100 ms (HSPA) and 750 ms (GPRS) are common [Gri13, Ch. 7]. With modern 4G LTE-Advanced (*Long Term Evolution*) networks, the standards prescribe strict latency bounds for better user experience. As mobile devices share radio frequencies for data transmission, access has to be mediated and multiplexed. This process is performed by a radio resource controller (RRC) located in the radio towers of the LTE cells that together comprise the radio access network (RAN). At the physical level, several latency-critical steps are involved in a request by a mobile device connected via a 4G network:

1. When a mobile device sends or receives data and was previously idle, it negotiates physical transmission parameters with the RRC. The standard prescribes that this **control-plane latency** must not exceed 100 ms [DPS13].
2. Any packet transferred from the mobile device to the radio tower must have a **user-plane latency** of below 5 ms.

[19] Head-of-line blocking occurs when a request is scheduled, but no open connection can be used, as responses have not yet been received.

3. Next, the carrier transfers the packet from the radio tower to a packet gateway connected to the public Internet. This **core network latency** is not bounded.
4. Starting from the packet gateway, normal **Internet routing** with variable latency is performed.

Thus, in modern mobile networks, one-way latency will be at least 5-105 ms higher than in conventional networks. The additional latency is incurred for each HTTP request and each TCP/TLS connection handshake, making latency particularly critical for mobile websites and apps.

In summary, to achieve low latency for REST/ and HTTP, many network parameters have to be explicitly optimized at the level of protocol parameters, operating systems, network stacks, and servers [Gri13]. In-depth engineering details of TCP/IP, DNS, HTTP, TLS, and mobile networking are provided by Grigorik [Gri13], Kurose and Ross [KR10], and Tanenbaum [TW11]. However, with all techniques and best practices applied, physical latency from the client to the server remains the main bottleneck, as well as the time-to-first-byte caused by processing in the backend. Both latency contributions can be addressed through caching.

2.3.3 Web Caching

HTTP allows resources to be declared cacheable. They are considered fresh for a statically assigned lifetime called time-to-live (TTL). Any cache in the request/response chain between client and server will serve a cached object without contacting the origin server. The HTTP caching model's update strategy is purely *expiration-based*: once a TTL has been delivered, the respective resource cannot be invalidated before the TTL has expired. In the literature, expiration-based caching is also known as the *lease model* [HKM+88, Mog94, Vak06] and has been proposed by Gray et al. [GC89] long before HTTP. In contrast, *invalidation-based* caches use out-of-band protocols to receive notifications about URLs that should be purged from the cache (e.g., non-standardized HTTP methods or separate purging protocols). This model is in wide use for many non-HTTP caches, too [Car13, ERR11, Lwe10, BBJ+10, LLXX09]. As the literature is lacking a survey of web caching in the light of data management, we give a concise overview of web cache types, scalability mechanisms, and consistency aspects of expiration-based and invalidation-based HTTP caching.

Types of Web Caches

The closer a web cache is to the network edge, the more the network latency decreases. We distinguish between six types of web caches, based on their network *location* as shown in Figure 2.18 [LLXX09, Nag04]:

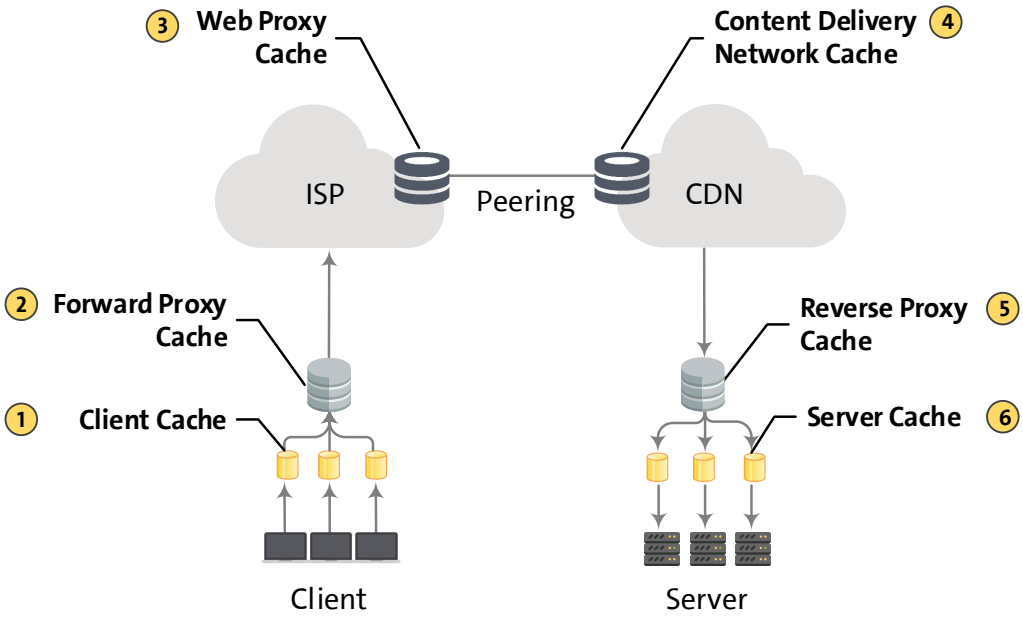


Figure 2.18: Different types of web caches distinguished by their location. Caches 1-3 are expiration-based, while caches 4-6 are invalidation-based.

Client Cache. A cache can be directly embedded in the application as part of the browser, mobile app, or an HTTP library [FGM+99]. Client caches have the lowest latency, but are not shared between clients and rather limited in size.

Forward Proxy Cache. Forward proxy caches are placed in networks as shared web caches for all clients in that network. Being very close to the application, they achieve a substantial decrease in network latency. Forward proxy caches can either be configured as explicit proxies by providing configuration information to clients through protocols such as PAC and WPAD [GTS+02] or by transparently intercepting outgoing, unencrypted TCP connections.

Web Proxy Cache. Internet Service Providers (ISPs) deploy web proxy caches in their networks. Besides accelerating HTTP traffic for end users, this also reduces transit fees at Internet exchange points. Like client and forward proxy caches, web proxy caches are purely expiration-based.

Content Delivery Network (CDN) Cache. CDNs provide a distributed network of web caches that can be controlled by the backend [PB07]. CDN caches are designed to be scalable and multi-tenant and can store massive amounts of cached data. Like reverse proxy caches and server caches, CDN caches are usually invalidation-based.

Reverse Proxy Cache. Reverse proxy caches are placed in the server's network and accept incoming connections as a surrogate for the server [Kam17]. They can be extended to perform application-specific logic, for example, to check authentication information and to perform load balancing over backend servers.

Server Cache. Server caches offload the server and its database system by caching intermediate data, query results, and shared data structures [Fit04, NFG+13, XFJP14, CLL+01b, GMA+08, BAC+13]. Server caches are not based on HTTP, but explicitly orchestrated by the database system (e.g., DBCache [BAM+04]) or the application tier (e.g., Memcache [Fit04]).

The defining characteristic of all web caches is that they transparently interpret HTTP caching metadata as *read-through* caches. This means that when a request causes a cache miss, the request is forwarded to the next cache or the origin server and then the response is cached according to the provided TTL. Web caches always forward write requests, as these come in the form of opaque `POST`, `PUT`, and `DELETE` requests whose semantics are implicit properties of a REST/HTTP API. The effectiveness of web caching is measured by a *cache hit ratio* that captures the percentage of all requests that were served from a cache and the *byte hit ratio* that expresses the corresponding data volume.

Scalability of Web Caching

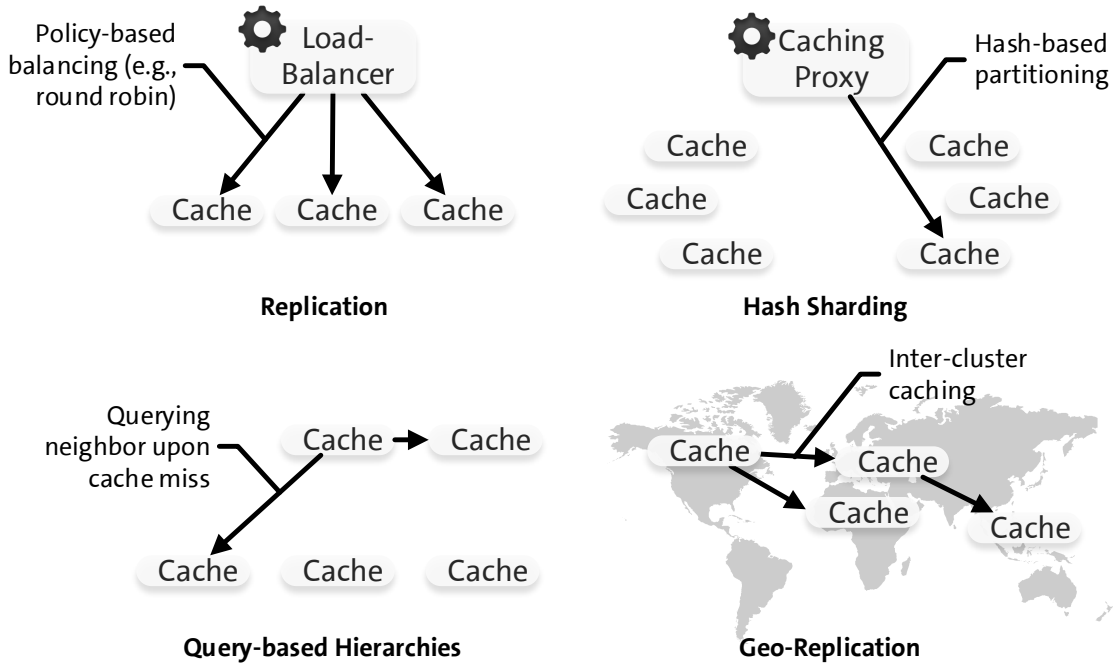


Figure 2.19: Scalability mechanisms of web caches: replication, sharding, query-based hierarchies, and geo-replication.

To employ web caches for cloud data management, they have to support scalability. It is widely unknown in the database community that web caches scale through the same primary mechanisms as most NoSQL databases: replication and hash sharding. Figure 2.19 gives an overview of these techniques in the context of web caches. Load balancers that can work on different levels of the protocol stack forward HTTP requests to web caches using a policy like round-robin or a uniform distribution [GJP11]. In contrast to database systems, no replication protocols are required, as each replica fetches missing

resources on demand. Partitioning the space of cached objects for a cluster of caches is achieved by hash sharding the space of URLs. Requests can then be forwarded to URL partitions through the Cache Array Routing Protocol (CARP) [Wan99]. Hierarchies of communicating web caches (*cache peering* [KR01]) build on query-based protocols like the Inter Cache Protocol (ICP) [Wes97], the Hypertext Caching Protocol (HTCP) [VW99], or Cache Digests [FCAB00]. The underlying idea of query-based protocols is that checking another cache replica's entries is more efficient than forwarding a request to the origin server. Finally, global meshes of web caches in CDNs can rely on inter-cluster exchanges for geo-replication [PB08]. In practice, CDN providers exploit the fact that a cache lookup of a URL maps well to a key-value interface. This allows scaling cache clusters by deploying web caches as a proxy on top of a distributed key-value store [Spa17].

Web caching increases read scalability and fault tolerance, as objects can still be retrieved from web caches if the backend is temporarily unavailable [RS03]. As web caches only fetch content lazily, elasticity is easy to achieve: web cache replicas can be added at any time to scale reads.

Expiration-Based Web Caching

HTTP defines a `Cache-Control` header that both clients and servers leverage to control caching behavior. The server uses it to specify *expiration*, whereas the client employs it for *validation*.

Expirations are provided as TTLs at the granularity of seconds in order to be independent from clock synchronization. Additionally an `Age` header indicates how much time has passed since the original request, to preserve correct expirations when caches communicate with each other. The actual expiration time t_{exp} is then computed using the local clock's timestamp at the moment the response was received $now_{res}()$, giving $t_{exp} = now_{res}() + TTL - Age$. The server can set separate expirations for shared web caches (`s-max-age`) and client caches (`max-age`). Furthermore, it can restrict that responses should not be cached at all (`no-cache` and `must-revalidate`), should only be cached in client caches (`private`), or should not be persisted (`no-store`). By default, the *cache key* that uniquely identifies a cached response consists of the URL and the host. The `Vary` header allows to extend the cache key through specified request headers, e.g., `Accept-Language`, in order to cache the same resource in various representations.

Clients and web caches can **revalidate** objects by asking the origin server for potential modifications of a resource based on a version number (called `ETag` in HTTP) or a `Last-Modified` date (*cache validators*). The client thus has a means to explicitly request a fresh object and to save transfer time, if resources have not changed. Revalidations are performed through conditional requests based on `If-Modified-Since` and `If-None-Match` headers. If the timestamp or version does not match for the latest resource (e.g., a database object), the server returns a full response. Otherwise, an empty response with a `304 Not Modified` status code is returned.

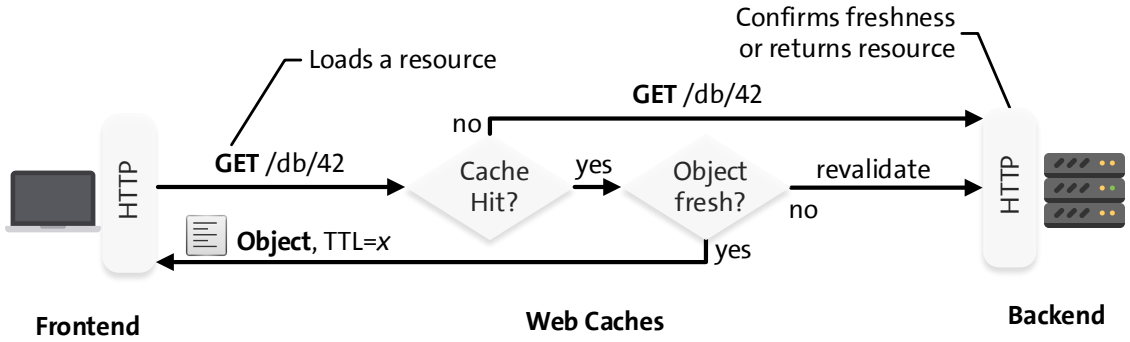


Figure 2.20: Validation of resource freshness in expiration-based HTTP caching.

Figure 2.20 illustrates the steps a web cache performs when handling a request: if the object of the requested URL was not previously cached, the web cache forwards the request to the backend. If a cache hit occurs, the cache determines whether the local copy of the resource is still fresh by checking $now() > t_{exp}$. If the object is still fresh, it is returned to the client without any communication to the backend. If $now() > t_{exp}$ and the cached resource has a cache validator, the web cache revalidates the resource, otherwise, the request is forwarded. This logic is performed for any cache in the chain from the client cache to reverse proxy caches. In a revalidation, clients can furthermore bound the age of a response (`max-age` and `min-fresh`), allow expired responses (`max-stale`) or explicitly load cached versions (`only-if-cached`). CDNs and reverse proxies typically ignore revalidation requests and simply serve the latest cached copy, in order to secure the origin against revalidation attacks [PB08].

The consistency model of expiration-based caching is Δ-atomicity. The problem is that Δ is a high, fixed TTL in the order of hours to weeks [RS03], as accurate TTLs for dynamic data are impossible to determine. This makes the native caching model of HTTP unsuitable for data management and is the reason why REST APIs of DBaaS, BaaS, and NoSQL systems explicitly circumvent HTTP caching [Dep17,Hoo17,Par17,ALS10,Ama17a,Dyn17, CWO+11, BGH+15, Dat17].

Invalidation-Based Web Caching

CDNs and reverse proxy caches are *invalidation-based* HTTP caches. They extend the expiration-based caching model and additionally expose (non-standardized) interfaces for asynchronous cache invalidation. The backend has to explicitly send an invalidation to every relevant invalidation-based cache. While CDN APIs forward invalidations internally with efficient broadcasting protocols (e.g., bimodal multicast [Spa17]), employing many reverse proxies can lead to a scalability problem, if many invalidations occur. In general, an invalidation is required if a resource was updated or deleted and invalidation-based caches have observed an expiration time greater than the current time: $\exists t_{exp} : now() < t_{exp}$. For DBaaS/BaaS systems this condition is non-trivial to detect, since updates may affect otherwise unrelated query results and objects.

Besides their invalidation interfaces, CDNs (e.g., Akamai and Fastly [BPV08, Spa17]) and reverse proxies (e.g., Varnish, Squid, Nginx, Apache Traffic Server [Kam17,Ree08,Wes04]) often also provide further extensions to HTTP caching:

- Limited **application logic** can be executed in the cache. For example, the Varnish Control Language (VCL) allows to manipulate requests and responses, perform health checks and validate headers [Kam17].
- **Prefetching** mechanisms proactively populate the cache with resources that are likely to be requested in the near future.
- Edge-side **templating** languages like ESI [TWJN01] allow to assemble responses from cached data and backend requests.
- By assigning tags to cacheable responses, efficient bulk invalidations of related resources can be performed (**tag-based invalidation**).
- Distributed **Denial of Service** (DDoS) attacks can automatically be mitigated and detected before the backend is compromised [PB08].
- Updated resources can be proactively pushed (**prewarming**).
- Real-time **access logs** may be used by the application for analytics and accounting.
- **Stale resources** can be served while the backend is offline (`stale-on-error`) or during revalidations (`stale-while-revalidate`) [IET15].

For latency, an important characteristic of invalidation-based caches is their ability to maintain long-lived backend connections that incoming requests can be multiplexed over. This significantly reduces the overhead of connection handshakes as they only have to be performed over low-latency links between clients and CDN edge nodes. In many cases, cloud services have end-to-end encryption as a requirement for authenticity, privacy, data integrity, and confidentiality. To this end, TLS certificates are deployed to CDNs and reverse proxies to terminate TLS connections on the network edge and to establish different connections to the backend. Thus, for encrypted REST APIs and websites, only client, CDN, and reverse proxy caches apply for HTTP caching, whereas forward and web proxy caches only observe encrypted traffic.

Previous research on web caching as discussed in Chapter 6 has focused on cache replacement strategies [PB03, BCF+99, LLXX09], CDN architectures [PB08, FFM04, Fre10], cache cooperation [KR01, Wes97, VW99, FCAB00], proxy and client extensions [RXDK03, TWJN01, BR02], and changes to the caching model itself [KW97, Wor94, KW98, BDK+02]. Further treatments of expiration-based and invalidation-based web caching are provided by Rabinovich and Spatscheck [RS03], Labrindis et al. [LLXX09], Nagaraj [Nag04], Buyya et al. [BPV08], and Grigorik [Gri13].

2.3.4 Challenges of Web Caching for Data Management

Both expiration-based and invalidation-based caching are challenging for data management, as they interfere with the consistency mechanisms of database systems. Figure 2.21 gives an example of how web caching affects consistency.

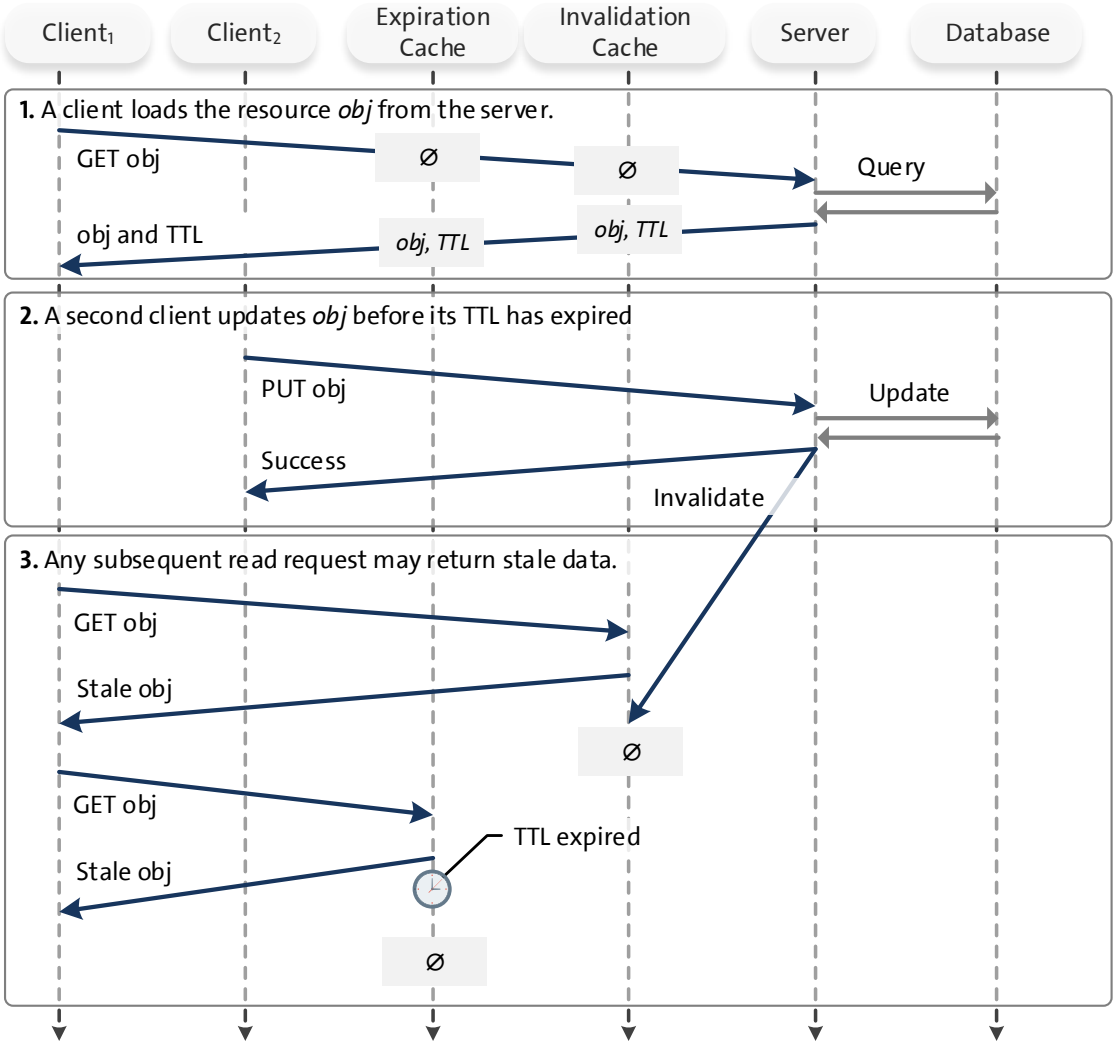


Figure 2.21: Cache coherence problems of web caches for data management caused by access of two different clients.

1. On the first request, the server has to set a TTL for the response. If the TTL is too low, caching has no effect. If it is too high, clients will experience many stale reads. Due to the dynamic nature of query results and objects in data management, TTLs are not known in advance.
2. When a second client updates the previously read object before its TTL expired, caches are in an inconsistent state. Even if the server could issue an invalidation (which is usually impossible for query results), the invalidation is asynchronous and only takes effect at some later point in time.

3. Reads that happen between the completed update and the initially provided expiration time will cause stale reads at expiration-based caches.

In conclusion, web caching for data management is considerably restricted because of several challenges:

- **Expiration-based caching** either degrades consistency (high TTLs) or causes very high cache miss rates (low TTLs).
- **Cache coherence** for DBaaS and BaaS REST APIs is currently achieved by marking all types of dynamic data as uncacheable.
- Currently, **TTL estimation** is a manual and error-prone process leading to low caching efficiency as TTLs do not adapt to changing workloads and differences between individual query responses.
- **Cache invalidation** requires detecting changes to files, objects, and query results in real-time based on the updates performed against the data management API.
- Fetching **dynamic data** (e.g., query results) via REST/HTTP requires contacting a remote server, which involves the full end-to-end latency from the client to the server.
- With standard HTTP caching, clients cannot control **consistency requirements** on a per-user, per-session, or per-operation basis, as the server provides the HTTP caching metadata used by intermediate caches.

2.4 Frontend Performance

Frontend performance is concerned with how fast data can be rendered and computations be performed at the client side. In principle, the frontend is out of the scope of a data management solution as proposed in this thesis. However, as the SDK and API layer of a DBaaS/BaaS reach into the environment of the mobile device and utilize its networking and caching capabilities, some aspects of browsers are highly relevant for end-to-end performance. We will specifically examine frontend performance for browsers. In native mobile apps, most principles apply too, but applications can choose from different storage options like the files system and embedded relational databases. However, due to the absence of a browser cache, the task of maintaining cache consistency with remote storage has to be handled by the application.

As of 2018, an average website downloads 107 different HTTP resources with a total size of over 3 MB of data to be transferred [Arc18]. The web has evolved through three major forms of websites. **Hypertext documents** are simple text-based documents interconnected through links and formatted through basic markup for the content's structure. **Web pages** enrich hypertext documents through support for rich media types such as images, audio, and video, as well as complex layout and styling of the document's appearance. Finally, **web applications** add behavior to websites through JavaScript logic and the ability to programmatically request REST/HTTP APIs (Ajax). Web applications

are usually implemented with single-page application frameworks that help to structure the application through architectural patterns and templating for rendering data into UI elements [Ang17, Emb17, Vue17, Rea17]. With the growing prevalence and complexity of web applications, the impact of latency increases.

2.4.1 Client-Side Rendering and Processing

The **critical rendering path** (CRP) describes the process that a browser performs in order to render a website from HTML, JavaScript, and CSS resources [Fir16, Gri13]. The dependency graph between these critical resources, i.e., files required for the initial paint, determines the length, size, and weight of the CRP. The *length* of the CRP is the minimum number of network round-trips required to render the web page. The *size* of the CRP is the number of critical resources that are loaded. The *weight* (also called "critical bytes") of the CRP is the combined size of all critical resources measured in bytes.

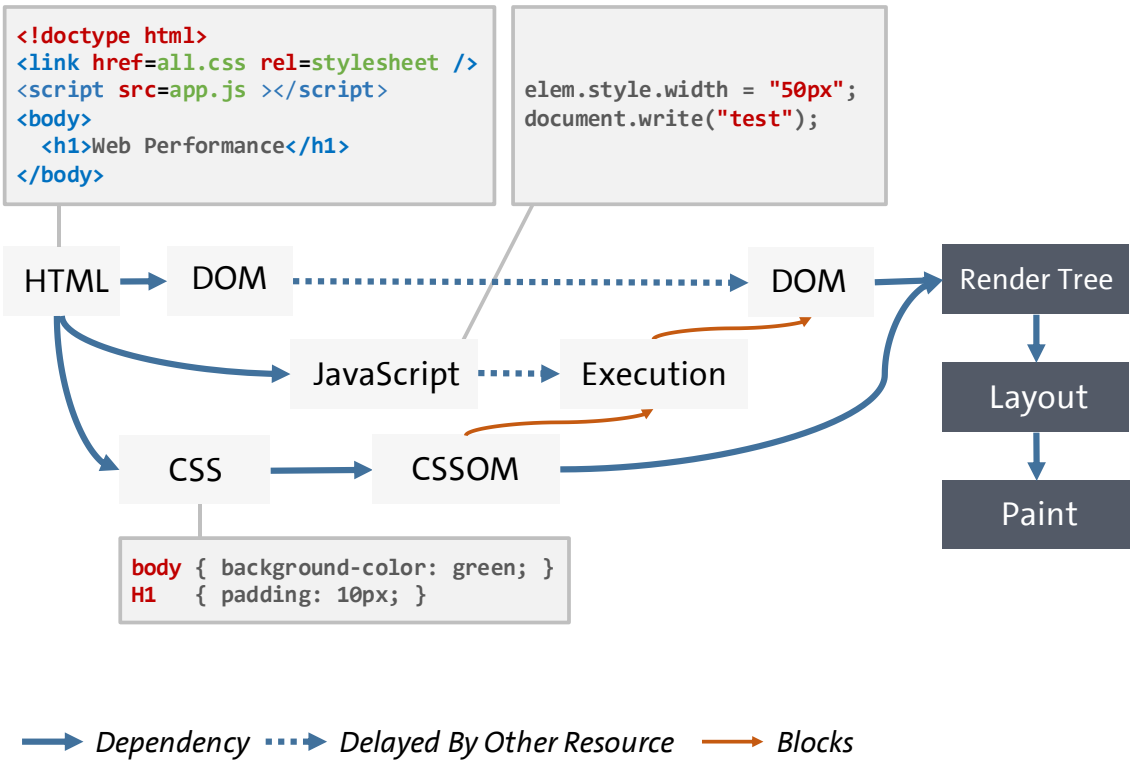


Figure 2.22: The critical rendering path as a model for frontend performance.

The execution of the CRP is illustrated in Figure 2.22. After receiving the HTML from the network, the browser starts parsing it into a Document Object Model (DOM). If the HTML references CSS and JavaScript resources, the parser (respectively its look-ahead heuristics) will trigger their background download as soon as they are discovered. The CSS stylesheet is parsed into a CSS object model (CSSOM). CSS is *render-blocking*, as rendering can only proceed when the CSSOM is fully constructed and thus all styling information available. JavaScript can modify and read from both the DOM and CSSOM. It is *parser-blocking* as the HTML parser blocks until the discovered JavaScript is executed.

Furthermore, JavaScript execution blocks until the CSSOM is available causing a chain of interdependencies. Only when the DOM and the CSSOM are constructed, and JavaScript is executed, the browser starts to combine styling and layout information into a render tree, computes a layout, and paints the page on the screen.

The process of frontend performance optimization involves reducing the size, length, and weight of the CRP. Typical steps are loading JavaScript asynchronously, deferring its parsing, preconnecting and preloading critical resources, inlining critical CSS, applying compression, minification, and concatenation, optimizing JavaScript execution and CSS selector efficiency, loading "responsive" images based on screen size [Wag17]. HTTP/2 eliminates the necessity for many common performance workarounds that negatively impact cacheability, for example, concatenation of resources [IET15].

End-user performance can be measured using different web performance metrics:

- Browsers implement events that indicate the completeness of the rendering process. The **DomContentLoaded** event is fired once the DOM has been constructed and no stylesheets block JavaScript execution.
- The **first paint** occurs when the browser renders the page for the first time. Depending on the structure of the CRP this can, for example, be a blank page with a background color or a visually complete page. The first paint metric can be refined to the *first meaningful paint* [Sak17] which is defined through the paint that produces the largest change in the visual layout.
- Once all resources of the website (in particular images, JavaScript and stylesheets) have been downloaded and processed, the **load** event is fired. The event indicates the completion of loading from an end user's perspective. However, any asynchronous requests triggered through JavaScript are not captured in the load event. Therefore, the DomContentLoaded and load event can be decreased by loading resources through code without actually improving user-perceived performance.
- As all above metrics do not capture the rendering process itself, the **speed index** metric was proposed as a means of quantifying visual completeness over time [Mee12]. It is defined as $\int_0^\infty 1 - VC(t)\,\mathrm{d}t$, where $VC(t) \in [0,1]$ is the visual completeness as a function of time. Experimentally, the speed index is usually calculated through video analysis of a browser's loading process. In contrast to other metrics, the speed index also accounts for API requests performed by web applications.

Latency remains the major factor for frontend performance, once all common frontend optimizations (e.g., inlined critical CSS) and network optimizations (e.g., gzip compression) have been applied. The length of the CRP determines how many round-trips occur before the user is presented with the first rendered result. In the ideal case, the length of the CRP can be reduced to one single round-trip by only including asynchronous JavaScript and inlining CSS. In practice, however, the length and size of the critical rendering path is usually much longer [Wag17, Fir16]. The increasing predominance of web applications based on rich client-side JavaScript frameworks that consume data via API requests ex-

tends the impact of latency beyond the CRP. During navigation and rendering, the latency of asynchronously fetched resources is crucial to display data quickly and to apply user interactions without perceptible delay.

2.4.2 Client-Side Caching and Storage

In recent years, it became evident that moving more application logic into the client also requires **persistence** options to maintain application state within and across user sessions. Several client-side storage and caching APIs have been standardized and implemented in browsers. A comprehensive overview of client-side storage mechanisms is provided by Camden [Cam16]. In the following, we provide an overview of storage technologies relevant for this thesis:

HTTP Browser Cache. The browser cache [IET15,FGM+99] works similar to other HTTP caches, except that it is exclusive to one user. Its main advantage is that it transparently operates on any HTTP resource. On the other hand, however, it cannot be programmatically controlled by the JavaScript application and operates purely expiration-based. Also, cached data can be evicted at any time making it impossible to build application logic on the presence of cached client-side data.

Cookies. Through the HTTP `Cookie` header, the server can store strings in the client. Cookie values are automatically attached to each client request [IET15]. Cookies are very limited in control, size, and flexibility and therefore mainly used for session state management and user tracking. Cookies frequently cause performance problems as they can only be accessed synchronously and have to be transferred with each request.

Web SQL. The goal of the WebSQL specification is to provide SQL-based access to an embedded relational database (e.g., SQLite) [Cam16]. However, as browser support is lacking, the development of WebSQL has mostly ceased in favor of the IndexedDB API.

IndexedDB. The IndexedDB specification [AA17] describes a low-level database API that offers key-value storage, cursors over indices, and transactions. Despite its lack of a declarative query language, it can be used to implement an embedded database system in the client. In contrast to the browser cache, storage is persistent and controlled via an API. However, this implies that custom cache coherence or replication is required if IndexedDB is used to store a subset of the backend database.

Service Worker Cache. Service Workers are background processes that can intercept, modify, and process HTTP requests and responses of a website [Ama16]. This allows implementing advanced network behavior such as an offline mode that continues serving responses even though the user lacks a mobile network connection. The Service Worker cache is a persistent, asynchronous map storing pairs of HTTP requests and responses. The default cache coherence mechanism is to store data

indefinitely. However, the JavaScript code of the Service Worker can modify this behavior and implement custom cache maintenance strategies.

Local and Session Storage. The DOM storage APIs [Cam16] allow persisting key-value pairs locally for a single session (`SessionStorage`) or across sessions (`LocalStorage`). The API only allows blocking `get` and `set` operations on keys and values. Due to its synchronous nature, the API is not accessible in background JavaScript processes (e.g., Service Workers).

The central problem of client-side storage and caching abstractions is that they have to be **manually controlled** by the application. Besides first attempts, there is furthermore no coupling between query languages and persistence APIs employed in the client and the DBaaS/BaaS [ALS10, GAAU15, LMLM16]. This forces application developers to duplicate data-centric business logic and maintain cache consistency manually. The error-prone and complex task of manual cache maintenance prevents many applications from incorporating client-side storage into the application's data management. Thus, a solution is required that transparently consolidates client-side storage into data management while preserving invariants such as consistency guarantees and correctness of query results. Client-side caching and storage standards potentially enable serving web applications in the absence of network connectivity (offline mode). However, this also requires new mechanisms for cache coherence of reads and query results as well as synchronization and concurrency control for updates made while being offline.

2.5 Summary

In this chapter, we highlighted core performance challenges across the web application stack. Performance depends on data storage and business logic in the backend, networking and caching infrastructures, as well as frontend rendering. First, we discussed the requirements of web applications that include high availability, elastic scalability, quick page loads, engaging user experience, and a fast time-to-market. We showed that these requirements are difficult to achieve in cloud-based two- and three-tier architectures. In particular, latency poses a central challenge in heterogeneous cloud environments. The conflict between latency and correctness becomes evident in NoSQL database systems and their various levels of relaxed consistency guarantees. Furthermore, the combination of data storage systems in polyglot persistence architectures complicates data management. When transactions are involved in distributed data management, latency furthermore becomes a problem for transaction abort rates. Often, database systems cannot be efficiently employed in web applications as they lack Database- and Backend-as-a-Service interfaces for access from other cloud services and client devices.

The network and the protocols involved in communication with cloud services are the fundamental cause of high latency. We discussed how most aspects of networking can be optimized, leaving end-to-end latency resulting from physical distance as the major re-

maining performance challenge. Even though REST and HTTP are widely used for DBaaS, BaaS, and NoSQL systems, their caching model does not fit the requirements of data management. Expiration-based caches interfere with the consistency guarantees of database systems, whereas invalidation-based caching requires non-trivial change detection for dynamic data. Therefore, latency reduction through HTTP caching is an open problem for cloud data management.

Frontend performance is defined by the highly latency-dependent critical rendering path. Modern browsers potentially allow latency reduction for data-centric API requests through storage abstractions. However, cache coherence needs to be solved in order to avoid sacrificing consistency for reduced latency.

In the remainder of this dissertation, we will address the latency, scalability, and consistency challenges across the data management stack to achieve better performance for a wide spectrum of web and mobile applications.

3 Providing Low Latency for Cloud Data Management

In this chapter, we analyze the data management requirements that complement the performance requirements outlined in the previous chapter. Our goal is to derive a cloud data management platform that solves the end-to-end latency problem in a database-independent, scalable fashion and which is applicable to many web applications and database systems. Thus, we introduce **Orestes** as a middleware to solve direct client access (Challenge C_2, see p. 6), to provide unified access to polyglot persistence (Challenge C_4), and to lay the foundation for tackling latency of dynamic data (Challenge C_1). To this end, a comprehensive collection of data management capabilities and requirements needs to be supported without restricting the applicability of latency reduction techniques.

To derive a set of requirements for the platform we introduce, this chapter proposes a top-down view of scalable data management: instead of contrasting the implementation specifics of individual representatives, we introduce a comparative classification model that relates functional and non-functional requirements to techniques and algorithms employed in NoSQL databases. This framework for grouping and comparing implementation techniques encompasses the current state in scalable data management and allows us to contrast trade-offs and derive an appropriate architecture for polyglot data management.

Based on the survey of data management requirements, we propose Orestes as a scalable Database- and Backend-as-a-Service middleware for low latency. Its unified REST API is designed to cater to heterogeneous data management requirements, while universally applying web caching for latency reduction. Orestes forms the basis for low-latency polyglot persistence by unifying access to different database systems and providing them with generic abstractions for schema management, access control, multi-tenancy, and serverless functions. The goal is to cache and maintain data and enable secure access in two-tier cloud application architectures. We will analyze which functional requirements can be provided in a database-independent fashion and discuss their scalable implementation.

3.1 A Classification Scheme for NoSQL Database Systems

In this section, we highlight the design space of distributed database systems, concentrating on sharding, replication, storage management, and query processing. The goal is to provide a comprehensive set of data management requirements that have to be considered for designing a flexible DBaaS/BaaS middleware. Therefore, we survey the implementation techniques of systems and discuss how they are related to different functional and non-functional properties (goals) of data management systems.

Every significantly successful database is designed for a particular class of applications, or to achieve a specific combination of desirable system properties. The simple reason why there are so many different database systems is that it is not possible for any system to achieve all desirable properties at once. Traditional relational databases such as PostgreSQL have been built to provide the full functional package: a very flexible data model, sophisticated querying capabilities including joins, global integrity constraints, and transactional guarantees. On the other end of the design spectrum, there are key-value stores like Dynamo that scale with data and request volume and offer high read and write throughput as well as low latency, but barely any functionality apart from simple lookups.

In order to illustrate which techniques are suitable to achieve specific system properties, we provide the **NoSQL Toolbox** (Figure 3.1) that connects each technique to the functional and non-functional properties it enables (positive edges only). In the following, we will review each of the four major categories of techniques in scalable data management: sharding, replication, storage management, and query processing.

3.1.1 Sharding

Several distributed relational database systems such as Oracle RAC or IBM DB2 pureScale rely on a **shared-disk architecture** where all database nodes access the same central data repository (e.g., a NAS or SAN). Thus, these systems provide consistent data at all times, but are also inherently difficult to scale. In contrast, the (NoSQL) database systems in the focus of this dissertation are built upon a **shared-nothing architecture**, meaning each system consists of many servers with private memory and private disks that are connected through a network. Thus, high scalability in throughput and data volume is achieved by **sharding** (partitioning) data across different nodes (**shards**) in the system.

There are three basic distribution techniques: range partitioning, hash partitioning, and entity-group sharding.

Range Partitioning

To make efficient scans possible, data can be partitioned into ordered and contiguous value ranges by **range-sharding**. However, this approach requires some coordination through a

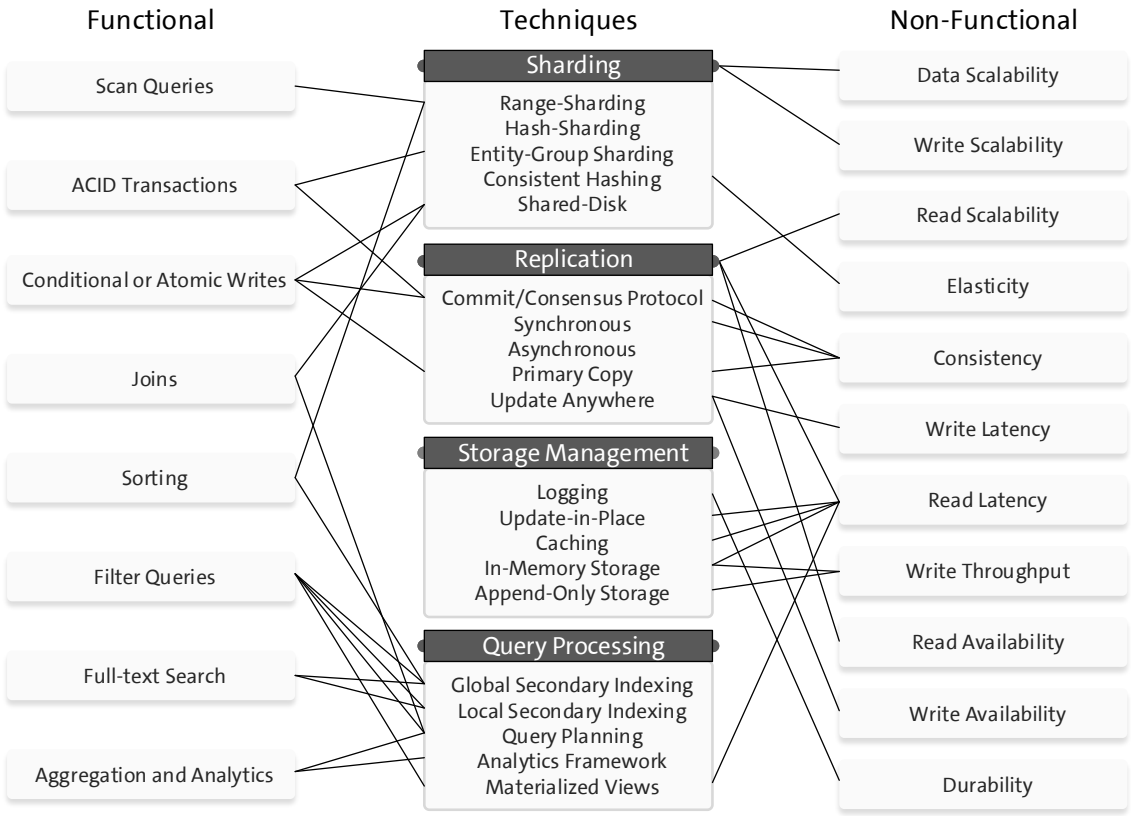


Figure 3.1: The NoSQL Toolbox: It connects the techniques of NoSQL databases with the desired functional and non-functional system properties they support.

master that manages assignments. To ensure elasticity, the system has to be able to detect and resolve hotspots automatically by further splitting an overburdened shard.

Range sharding is supported by wide-column stores like BigTable, HBase or Hypertable [Wie15] and document stores, e.g., MongoDB, RethinkDB, Espresso [QSD+13] and DocumentDB [STR+15].

Hash Partitioning

Another way to partition data over several machines is **hash-sharding** where every data item is assigned to a shard server according to some hash value built from the primary key. This approach does not require a coordinator and also guarantees data to be evenly distributed across the shards, as long as the used hash function produces an even distribution. The obvious disadvantage, though, is that it only allows lookups and makes scans impossible. Hash sharding is used in key-value stores and is also available in some wide-column stores like Cassandra [LM10] or Azure Tables [CWO+11].

The shard server that is responsible for a record can be determined as $server_{id} = hash(id)$ *mod servers*, for example. However, this hashing scheme requires all records to be reassigned every time a new server joins or leaves, because it changes with the number of shard servers (*servers*). Consequently, it is infeasible to use in elastic systems like Dynamo, Riak, or Cassandra, which allow additional resources to be added on-demand and again

be removed when dispensable. For increased flexibility, elastic systems typically use **consistent hashing** [KLL+97] where records are not directly assigned to servers, but instead to logical partitions which are then distributed across all shard servers. Thus, only a fraction of data has to be reassigned upon changes in the system topology. For example, an elastic system can be downsized by offloading all logical partitions residing on a particular server to other servers and then shutting down the now idle machine. For details on how consistent hashing is used in NoSQL systems, please refer to DeCandia et al. [DHJ+07].

Entity-Group Sharding

A data partitioning scheme with the goal of enabling single-partition transactions on co-located data is **entity-group sharding**. Partitions are called entity-groups and either explicitly declared by the application (e.g., in G-Store [DAEA10] and MegaStore [BBC+11]) or derived from transactions' access patterns (e.g., in Relational Cloud [CJP+11] and Cloud SQL Server [BCD+11]). If a transaction accesses data that spans more than one group, data ownership can be transferred between entity-groups or the transaction manager has to fall back to more expensive multi-node transaction protocols.

3.1.2 Replication

In terms of CAP (cf. Section 2.2.3), conventional RDBMSs are often CA systems run in single-server mode: the entire system becomes unavailable on machine failure. System operators therefore secure data integrity and availability through expensive, but reliable high-end hardware. In contrast, NoSQL systems like Dynamo, BigTable, or Cassandra are designed for data and request volumes that cannot possibly be handled by one single machine, and therefore run on clusters consisting of potentially thousands of servers[1]. Since failures are inevitable and will occur frequently in any large-scale, distributed system, the software has to cope with them on a daily basis [Ham07]. In 2009, Dean [Dea09] stated that a typical new cluster at Google encounters thousands of hard drive failures, 1 000 single-machine failures, 20 rack failures and several network partitions due to expected and unexpected circumstances in its first year alone. Many more recent cases of network partitions and outages in large cloud data centers have been reported [BK14]. Replication allows the system to maintain availability and durability in the face of such errors. But storing the same records on different machines (**replica servers**) in the cluster introduces the problem of synchronization between them and thus a trade-off between consistency on the one hand and latency and availability on the other.

Gray et al. [GHa+96] propose a two-tier classification of different replication strategies according to *when* updates are propagated to replicas and *where* updates are accepted. There are two possible choices on tier one ("when"): **eager** (*synchronous*) replication propagates incoming changes synchronously to all replicas before a commit can be returned to the client, whereas **lazy** (*asynchronous*) replication applies changes only at the

[1] Low-end hardware is used, because it is substantially more cost-efficient than high-end hardware [HB09, Section 3.1].

receiving replica and passes them on asynchronously. The great advantage of *eager* replication is consistency among replicas, but it comes at the cost of higher write latency and impaired availability due to the need to wait for other replicas [GHa+96]. *Lazy* replication is faster, because it allows replicas to diverge. As a consequence, though, stale data might be served. On the second tier ("where"), again, two different approaches are possible: either a **master-slave** (*primary copy*) scheme is pursued where changes can only be accepted by one replica (the master) or, in a **update anywhere** (*multi-master*) approach, every replica can accept writes. In *master-slave* protocols, concurrency control is not more complex than in a distributed system without replicas, but the entire replica set becomes unavailable, as soon as the master fails. Multi-master protocols require complex mechanisms for prevention or detection and reconciliation of conflicting changes. Techniques typically used for these purposes are versioning, vector clocks, gossiping, and read repair (e.g., in Dynamo [DHJ+07]), and convergent or commutative data types [SPBZ11] (e.g., in Riak).

All four combinations of the two-tier classification are possible. Distributed relational systems usually perform *eager master-slave* replication to maintain strong consistency. *Eager update anywhere* replication as for example featured in Google's Megastore [BBC+11] suffers from a heavy communication overhead generated by synchronization and can cause distributed deadlocks which are expensive to detect. NoSQL database systems typically rely on *lazy* replication, either in combination with the master-slave approach (CP systems, e.g., HBase and MongoDB) or the update anywhere approach (AP systems, e.g., Dynamo and Cassandra). Many NoSQL systems leave the choice between latency and consistency to the client, i.e., for every request, the client decides whether to wait for a response from any replica to achieve minimal latency or for a certainly consistent response (by a majority of the replicas or the master) to prevent stale data. In this work, an alternative technique for latency reduction is proposed, where data is cached close to applications using web caching and cache coherence protocols.

An aspect of replication that is not covered by the two-tier scheme is the distance between replicas. The obvious advantage of placing replicas near one another is low latency, but close proximity of replicas might also reduce the positive effects on availability; for example, if two replicas of the same data item are placed in the same rack, the data item is not available on rack failure in spite of replication. But more than the possibility of mere temporary unavailability, placing replicas nearby also bears the peril of losing all copies at once in a disaster scenario.

Geo-replication can protect the system against unavailability and data loss and potentially improves read latency for distributed access from clients. *Eager* geo-replication, as implemented in Google's Megastore [BBC+11], Spanner [CDE+13], MDCC [KPF+13], and Mencius [MJM08] allows for higher write latency to achieve linearizability or other strong consistency models. In contrast, *lazy* geo-replication as in Dynamo [DHJ+07], PNUTS [CRS+08], Walter [SPAL11], COPS [LFKA11], Cassandra [LM10], and BigTable [CDG+08] relaxes consistency in favor of availability and latency. Charron-Bost et al. [CBPS10, Chap-

ter 12] and Öszu and Valduriez [ÖV11, Chapter 13] provide a comprehensive discussion of database replication.

3.1.3 Storage Management

For best performance, database systems need to be optimized for the storage media they employ to serve and persist data. These are typically main memory (RAM), solid-state drives (SSDs), and spinning disk drives (HDDs) that can be used in any combination. Unlike RDBMSs in enterprise setups, distributed NoSQL databases avoid specialized shared-disk architectures in favor of shared-nothing clusters that are based on commodity servers (employing commodity storage media).

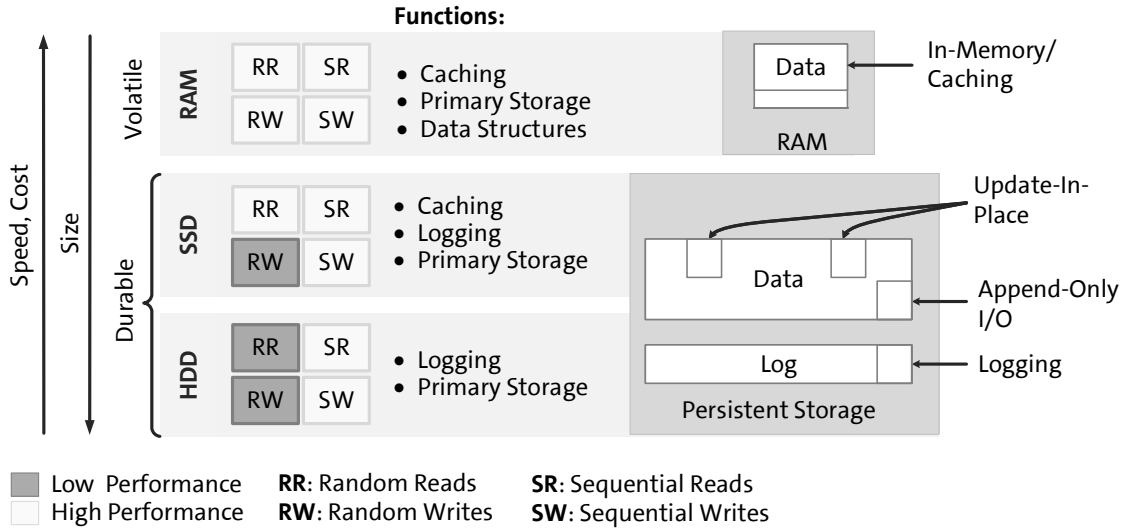


Figure 3.2: The storage pyramid and its role in NoSQL systems.

Storage devices are typically visualized as a "storage pyramid" (see Figure 3.2) [Hel07]. The huge variety of cost and performance characteristics of RAM, SSD, and HDD storage and the different strategies to leverage their strengths (storage management) is one reason for the diversity of NoSQL databases. Storage management has a spatial dimension (where to store data) and a temporal dimension (when to store data). Update-in-place and append-only I/O are two complementary spatial techniques of organizing data; in-memory prescribes RAM as the location of data, whereas logging is a temporal technique that decouples main memory and persistent storage and thus provides control over when data is actually persisted. Besides the major storage media, there is also a set of transparent caches (e.g., L1-L3 CPU caches and disk buffers, not shown in the figure), that are only implicitly leveraged through well-engineered database algorithms that promote data locality.

Stonebraker et al. [SMA+07] have found that in typical RDBMSs, only 6.8% of the execution time is spent on "useful work", while the rest is spent on:

- buffer management (34.6%), i.e., caching to mitigate slower disk access

- latching (14.2%), to protect shared data structures from race conditions caused by multi-threading
- locking (16.3%), to guarantee logical isolation of transactions
- logging (11.9%), to ensure durability in the face of failures
- hand-coded optimizations (16.2%)

This motivates that large performance improvements can be expected if RAM is used as primary storage (cf. **in-memory** databases [ZCO+15]). The downside are high storage costs and lack of durability – a small power outage can destroy the database state. This can be solved in two ways: the state can be replicated over n in-memory server nodes protecting against $n-1$ single-node failures (e.g., HStore, VoltDB [KKN+08, SW13]) or by **logging** to durable storage (e.g., Redis or SAP Hana [Car13, Pla13]). Through logging, a random write access pattern can be transformed to a sequential one comprised of received operations and their associated properties (e.g., redo information). In most NoSQL systems, the commit rule for logging is respected, which demands every write operation that is confirmed as successful to be logged and the log to be flushed to persistent storage. In order to avoid the rotational latency of HDDs incurred by logging each operation individually, log flushes can be batched together (group commit) which slightly increases the latency of individual writes, but drastically improves overall throughput.

SSDs and more generally all storage devices based on NAND flash memory differ substantially from HDDs in various aspects: "(1) asymmetric speed of read and write operations, (2) no in-place overwrite – the whole block must be erased before overwriting any page in that block, and (3) limited program/erase cycles" [MKC+12]. Thus, a database system's storage management must not treat SSDs and HDDs as slightly slower, persistent RAM, since random writes to an SSD are roughly an order of magnitude slower than sequential writes. Random reads, on the other hand, can be performed without any performance penalties. There are some database systems (e.g., Oracle Exadata, Aerospike) that are explicitly engineered for these performance characteristics of SSDs. In HDDs, both random reads and writes are 10-100 times slower than sequential access. Logging hence suits the strengths of SSDs and HDDs which both offer a significantly higher throughput for sequential writes.

For in-memory databases, an **update-in-place** access pattern is ideal: it simplifies the implementation and random writes to RAM are essentially equally fast as sequential ones, with small differences being hidden by pipelining and the CPU-cache hierarchy. However, RDBMSs and many NoSQL systems (e.g., MongoDB) employ an update-in-place update pattern for persistent storage, too. To mitigate the slow random access to persistent storage, main memory is usually used as a cache and complemented by logging to guarantee durability. In RDBMSs, this is achieved through a complex buffer pool which not only employs cache-replace algorithms appropriate for typical SQL-based access patterns, but also ensures ACID semantics. NoSQL databases have simpler buffer pools that profit from simpler queries and the lack of ACID transactions. The alternative to the buffer pool model is

to leave caching to the OS through virtual memory (e.g., employed in MongoDB's MMAP storage engine). This simplifies the database architecture, but has the downside of giving less control over which data items or pages reside in memory and when they get evicted. Also read-ahead (speculative reads) and write-behind (write buffering) transparently performed by the operating system lack sophistication as they are based on file system logics instead of database queries.

Append-only storage (also referred to as log-structuring) tries to maximize throughput by writing sequentially. Although log-structured file systems have a long research history, append-only I/O has only recently been popularized for databases by BigTable's use of Log-Structured Merge (LSM) trees [CDG+08] consisting of an in-memory cache, a persistent log, and immutable, periodically written storage files. LSM trees and variants like Sorted Array Merge Trees (SAMT) and Cache-Oblivious Look-ahead Arrays (COLA) have been applied in many NoSQL systems (e.g., Cassandra, CouchDB, LevelDB, Bitcask, RethinkDB, WiredTiger, RocksDB, InfluxDB, TokuDB) [Kle17]. Designing a database to achieve maximum write performance by always writing to a log is rather simple, the difficulty lies in providing fast random and sequential reads. This requires an appropriate index structure that is either actively maintained as a copy-on-write (COW) data structure (e.g., CouchDB's COW B-trees) or only periodically persisted as an immutable data structure (e.g., in BigTable-style systems). An issue of all log-structured storage approaches is costly garbage collection (compaction) to reclaim space of updated or deleted items.

In virtualized environments like Infrastructure-as-a-Service clouds, many of the discussed characteristics of the underlying storage layer are hidden. In the future, the availability of storage class memory combining speed of main memory with persistence will also require novel approaches for storage management [NSWW16].

3.1.4 Query Processing

The querying capabilities of a NoSQL database mainly follow from its distribution model, consistency guarantees, and data model. **Primary key lookup**, i.e., retrieving data items by a unique ID, is supported by every NoSQL system, since it is compatible to range- as well as hash-partitioning. **Filter queries** return all items (or projections) that meet a predicate specified over the properties of data items from a single table. In their simplest form, they can be performed as *filtered full-table scans*. For hash-partitioned databases, this implies a *scatter-gather* pattern where each partition performs the predicated scan and results are merged. For range-partitioned systems, any conditions on the range attribute can be exploited to select partitions.

To circumvent the inefficiencies of $O(n)$ scans, secondary indexes can be employed. These can either be **local secondary indexes** that are managed in each partition or **global secondary indexes** that index data over all partitions [BBC+11]. As the global index itself has to be distributed over partitions, consistent secondary index maintenance would necessitate slow and potentially unavailable commit protocols. Therefore, in practice, most

systems only offer eventual consistency for these indexes (e.g., Megastore, Google AppEngine Datastore, DynamoDB) or do not support them at all (e.g., HBase, Azure Tables). When executing global queries over local secondary indexes, the query can only be targeted to a subset of partitions, if the query predicate and the partitioning rules intersect. Otherwise, results have to be assembled through scatter-gather. For example, a user table with range-partitioning over an age field can service queries that have an equality condition on age from one partition, whereas queries over names need to be evaluated at each partition. A special case of global secondary indexing is full-text search, where selected fields or complete data items are fed into either a database-internal inverted index (e.g., MongoDB) or to an external search platform such as ElasticSearch or Solr (Riak Search, DataStax Cassandra).

Query planning is the task of optimizing a query plan to minimize execution costs [Hel07]. For aggregations and joins, query planning is essential as these queries are very inefficient and hard to implement in application code. The wealth of literature and results on relational query processing is largely disregarded in current NoSQL systems for two reasons. First, the key-value and wide-column model are centered around CRUD and scan operations on primary keys which leave little room for query optimization. Second, most work on distributed query processing focuses on OLAP (online analytical processing) workloads that favor throughput over latency whereas single-node query optimization is not easily applicable for partitioned and replicated databases [Kos00, ESW78, ÖV11]. However, it remains an open research challenge to generalize the large body of applicable query optimization techniques, especially in the context of document databases[2].

In-database analytics can be performed either natively (e.g., in MongoDB, Riak, CouchDB) or through external analytics platforms such as Hadoop, Spark and Flink (e.g., in Cassandra and HBase). The prevalent native batch analytics abstraction exposed by NoSQL systems is MapReduce[3] [DG04]. Due to I/O, communication overhead, and limited execution plan optimization, these batch- and micro-batch-oriented approaches have high response times. **Materialized views** are an alternative with lower query response times. They are declared at design time and continuously updated on change operations (e.g., in CouchDB and Cassandra). However, similar to global secondary indexing, view consistency is usually relaxed in favor of fast, highly-available writes, when the system is distributed [LLXX09]. As only few database systems come with built-in support for ingesting and querying unbounded streams of data, **near-real-time analytics** pipelines commonly implement either the **Lambda Architecture** [MW15] or the **Kappa Architecture** [Kre14]: the former complements a batch processing framework like Hadoop MapRe-

[2] Currently only RethinkDB can perform general θ-joins. MongoDB's aggregation framework has support for left-outer equi-joins in its aggregation framework and CouchDB allows joins for pre-declared MapReduce views.

[3] An alternative to MapReduce are generalized **data processing pipelines**, where the database tries to optimize the flow of data and locality of computation based on a more declarative query language (e.g., MongoDB's aggregation framework [Mon17]).

	Funct. Req.								Non-Funct. Req.										
	Scan Queries	ACID Transactions	Conditional Writes	Joins	Sorting	Filter Queries	Full-Text Search	Analytics	Data Scalability	Write Scalability	Read Scalability	Elasticity	Consistency	Write Latency	Read Latency	Write Throughput	Read Availability	Write Availability	Durability
MongoDB	x		x		x	x	x	x	x	x	x		x	x	x	x	x		x
Redis	x	x	x								x		x	x	x	x	x		
HBase	x		x		x			x	x	x	x	x	x	x		x			
Riak							x	x	x	x	x	x		x	x	x	x	x	
Cassandra	x		x		x		x	x	x	x	x	x		x		x	x	x	
MySQL	x	x	x	x	x	x	x	x			x		x						

	Techniques																		
	Range-Sharding	Hash-Sharding	Entity-Group Sharding	Consistent Hashing	Shared-Disk	Transaction Protocol	Sync. Replication	Async. Replication	Primary Copy	Update Anywhere	Logging	Update-in-Place	Caching	In-Memory Storage	Append-Only Storage	Global Indexing	Local Indexing	Query Planning	Analytics Framework	Materialized Views
MongoDB	x	x					x	x		x			x	x			x	x	x	
Redis								x	x		x		x							
HBase	x						x		x		x		x		x					
Riak		x		x				x		x	x	x	x			x	x		x	
Cassandra		x		x				x		x	x		x		x	x	x			x
MySQL					x	x		x	x		x	x	x				x	x		

Figure 3.3: A direct comparison of functional requirements, non-functional requirements and techniques among MongoDB, Redis, HBase, Riak, Cassandra, and MySQL according to the proposed NoSQL Toolbox.

duce with a stream processor such as Storm [BROL14] and the latter exclusively relies on stream processing and forgoes batch processing altogether.

3.2 System Case Studies

In this section, we provide a qualitative comparison of some of the most prominent key-value, document, and wide-column stores. We present the results in strongly condensed comparisons and refer to the documentation of the individual systems and our tutorials [GR15b, GR16, GWR17, WGW+18] for in-detail information. The proposed **NoSQL Toolbox** (see Figure 3.1, p. 81) is a means of abstraction that can be used to classify database systems along three dimensions: functional requirements, non-functional requirements, and the techniques used to implement them. We argue that this classification characterizes many database systems well and thus can be used to meaningfully contrast different database systems: Table 3.3 shows a direct comparison of MongoDB, Redis,

HBase, Riak, Cassandra, and MySQL in their respective default configurations. A more verbose comparison of central system properties is presented in Table 3.1 (see p. 91).

The methodology used to identify the specific system properties consists of an in-depth analysis of publicly available documentation and literature on the systems [Mon17, CD13, Car13, San17, Hba17, Ria17, CH16, LM10, MyS17]. Furthermore, some properties had to be evaluated by researching the open-source code bases, personal communication with the developers, as well as a meta-analysis of reports and benchmarks by practitioners.

The comparison elucidates how SQL and NoSQL databases are designed to fulfill very different needs: RDBMSs provide a broad set of functionalities whereas NoSQL databases excel on the non-functional side through scalability, availability, low latency, and high throughput. However, there are also large differences among the NoSQL databases. Riak and Cassandra, for example, can be configured to fulfill many non-functional requirements, but are only eventually consistent and do not feature many functional capabilities apart from data analytics and, in case of Cassandra, conditional updates. MongoDB and HBase, on the other hand, offer stronger consistency and more sophisticated functional capabilities such as scan queries and – only in MongoDB – filter queries, but do not maintain read and write availability during partitions and tend to display higher read latencies. As the only non-partitioned system in this comparison apart from MySQL, Redis shows a special set of trade-offs centered around the ability to maintain extremely high throughput at low latency using in-memory data structures and asynchronous master-slave replication.

This diversity illustrates that for enabling low latency cloud data management, no single database technology can cover all use cases. Therefore, latency reductions have to operate across different database systems and requirements.

3.3 System Decision Tree

Choosing a database system always means to choose one set of desirable properties over another. To break down the complexity of this choice, we present a binary decision tree in Figure 3.4 that maps trade-off decisions to example applications and potentially suitable database systems. The leaf nodes cover applications ranging from simple caching (left) to Big Data analytics (right). Naturally, this view on the problem space is not complete, but it vaguely points towards a solution for a particular data management problem.

The first split in the tree is along the access pattern of applications: they either rely on fast lookups only (left half) or require more complex querying capabilities (right half). The fast lookup applications can be distinguished further by the data volume they process: if the main memory of one single machine can hold all the data, a single-node system like Redis or Memcache probably is the best choice, depending on whether functionality (Redis) or simplicity (Memcache) is favored. If the data volume is or might grow beyond RAM capacity or is even unbounded, a multi-node system that scales horizontally might be more appropriate. The most important decision in this case is whether to favor availability (AP)

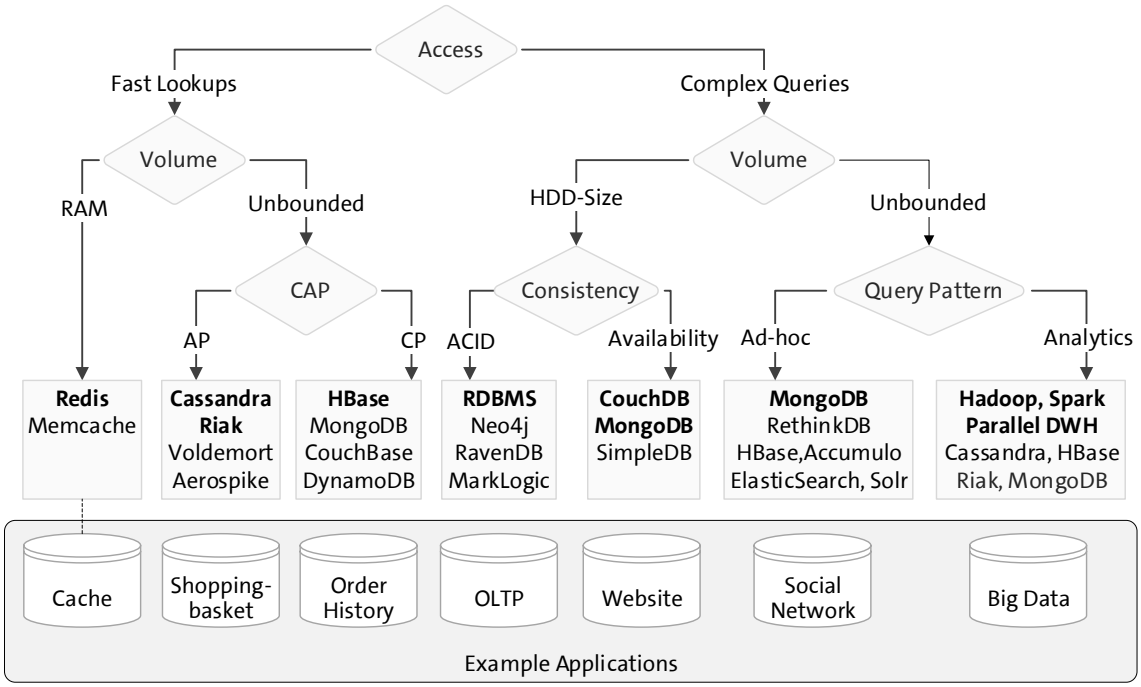


Figure 3.4: A decision tree for mapping requirements to (NoSQL) database system candidates.

or consistency (CP) as described by the CAP theorem. Systems like Cassandra and Riak can deliver an always-on experience, while systems like HBase, MongoDB, and DynamoDB deliver strong consistency.

The right half of the tree covers applications requiring more complex queries than simple lookups. Here, too, we first distinguish the systems by the data volume they have to handle according to whether single-node systems are feasible (HDD-size) or distribution is required (unbounded volume). For common OLTP (online transaction processing) workloads on moderately large data volumes, traditional RDBMSs or graph databases like Neo4J are optimal, because they offer ACID semantics. If, however, availability is essential, distributed systems like MongoDB, CouchDB or DocumentDB, are preferable.

If data volume exceeds the limits of a single machine, the choice depends on the prevalent query pattern: when complex queries have to be optimized for latency, as for example in social networking applications, MongoDB is very attractive, because it facilitates expressive ad-hoc queries. HBase and Cassandra are also useful in such a scenario, but excel at throughput-optimized Big Data analytics, when combined with Hadoop.

In summary, we are convinced that the proposed top-down model is an effective decision support to filter the vast amount of NoSQL database systems based on central requirements. The **NoSQL Toolbox** furthermore provides a mapping from functional and non-functional requirements to common implementation techniques in order to categorize the constantly evolving NoSQL space. In the following, we will conceive a DBaaS/BaaS middleware architecture that is designed to cover an as large subset of the decision tree as possible within a coherent REST/HTTP API.

Dimension	MongoDB	HBase	Cassandra	Riak	Redis
Model	Document	Wide-Column	Wide-Column	Key-Value	Key-Value
CAP	CP	CP	AP	AP	CP
Scan Performance	High (with appropriate shard key)	High (only on row key)	High (using compound index)	N/A	High (depends on data structure)
Disk Latency per Get by Row Key	∼ Several disk seeks	∼ Several disk seeks	∼ Several disk seeks	∼ One disk seek	In-Memory
Write Performance	High (append-only I/O)	High (append-only I/O)	High (append-only I/O)	High (append-only I/O)	Very high, in-memory
Network Latency	Configurable: nearest slave, master (*read preference*)	Designated region server	Configurable: R replicas contacted	Configurable: R replicas contacted	Designated master
Durability	Configurable: none, WAL, replicated (*write concern*)	WAL, row-level versioning	WAL, W replicas written	Configurable: writes, durable writes, W replicas written	Configurable: none, periodic logging, WAL
Replication	Master-slave, synchronicity configurable	File-system-level (HDFS)	Consistent hashing	Consistent hashing	Asynchronous master-slave
Sharding	Hash- or range-based on attribute(s)	Range-based (row key)	Consistent hashing	Consistent hashing	Only in Redis Cluster: hashing
Consistency	linearizable (master writes with quorum reads) or eventual (else)	Linearizable	Eventual, optional linearizable updates (*lightweight transactions*)	Eventual, client-side conflict resolution	Master reads: linearizable, slave reads: eventual
Atomicity	Single document	Single row, or explicit locking	Single column (multi-column updates may cause dirty writes)	Single key/value pair	Optimistic multi-key transactions, atomic Lua scripts
Conditional Updates	Yes (mastered)	Yes (mastered)	Yes (Paxos-coordinated)	No	Yes (mastered)
Interface	Binary TCP	Thrift	Thrift or TCP/CQL	REST or TCP/Protobuf	TCP/Plain-Text
Special Data Types	Objects, arrays, sets, counters, files	Counters	Counters	CRDTs for counters, flags, registers, maps	Sets, hashes, counters, sorted Sets, lists, HyperLogLogs, bit vectors
Queries	Query by example (filter, sort, project), range queries, MapReduce, aggregation, limited joins	Get by row key, scans over row key ranges, project CFs/columns	Get by Partition Key and filter/sort over cluster key, FT-search	Get by ID or local secondary index, materialized views, MapReduce, FT-search	Data Structure Operations
Secondary Indexing	Hash, B-Tree, geospatial indexes	None	Local sorted index, global secondary hash index, search index (Solr)	Local secondary indexes, search index (Solr)	Not explicit
License	GPL 3.0	Apache 2	Apache 2	Apache 2	BSD

Table 3.1: A qualitative comparison of MongoDB, HBase, Cassandra, Riak, and Redis.

3.4 Requirements for Low Latency Cloud Data Management

The NoSQL Toolbox motivates that cloud data management has to consolidate heterogeneous functional and non-functional requirements. The focus of this work is to complement any given data management requirements with low latency for better application performance. To this end, our low-latency architecture has to meet four high-level requirements.

Database Independence. To maximize performance for many use cases, the underlying mechanism has to be applicable to arbitrary legacy database systems, without requiring changes to the data store itself.

Database- and Backend-as-a-Service Functionality. In order to improve end-to-end latency, DBaaS and BaaS abstractions have to be combined in an architecture with direct access by web and mobile clients.

Scalability, Availability, and Multi-Tenancy. The primary non-functional requirements are (a) scalability to a large number of users and concurrent operations, (b) availability in the face of machine failures, and (c) the ability to consolidate multiple, isolated tenants in the cloud data management environment.

Low Latency with Tunable Consistency. Due to the inherent trade-off between consistency and latency, application developers should be able to flexibly choose between stronger guarantees and better performance.

3.5 Orestes: A Data Management Middleware for Low Latency

To address the above requirements, we introduce Orestes (**O**bjects **REST**fully **E**ncapsulated in **S**tandard Formats)[4]. Its high-level architecture is depicted in Figure 3.5. The Orestes architecture directly maps the desired properties to architectural structures:

- By defining a REST API as a superset of NoSQL database system capabilities, Orestes can expose various backend databases. To this end, Orestes defines a set of data management interfaces that can be implemented by specific database bindings. By allowing bindings to implement any combination of the data management interfaces, **database independence** is achieved.
- Instead of relying on the backend databases to fulfill every functional requirement, Orestes enhances the systems with **Database- and Backend-as-a-Service functionality**. These generic capabilities include schema management, authentication, access control, real-time query processing, push notifications, file hosting, data validation, and FaaS code execution.
- The Orestes servers are designed to be stateless containers in order to achieve horizontal **scalability, availability, and multi-tenancy**. When additional resources are

[4] Orestes is used as the technical basis for the commercial Backend-as-a-Service offering Baqend (see `www.baqend.com`).

required for a workload, new Orestes servers can be provisioned and de-provisioned at any time enabling elasticity. The server is designed to be run in container clusters, so multi-tenancy is achieved in a private container model. Load balancers can dismiss unavailable Orestes servers, so that availability is not degraded.

- By applying caching in the generic parts of the system, Orestes achieves latency reduction across database systems. The Cache Sketch approach (cf. Section 4.1) for **low latency with tunable consistency** levels only requires a CRUD or a query interface to be implemented by a specific database system and can handle the caching mechanisms in a database-agnostic fashion.

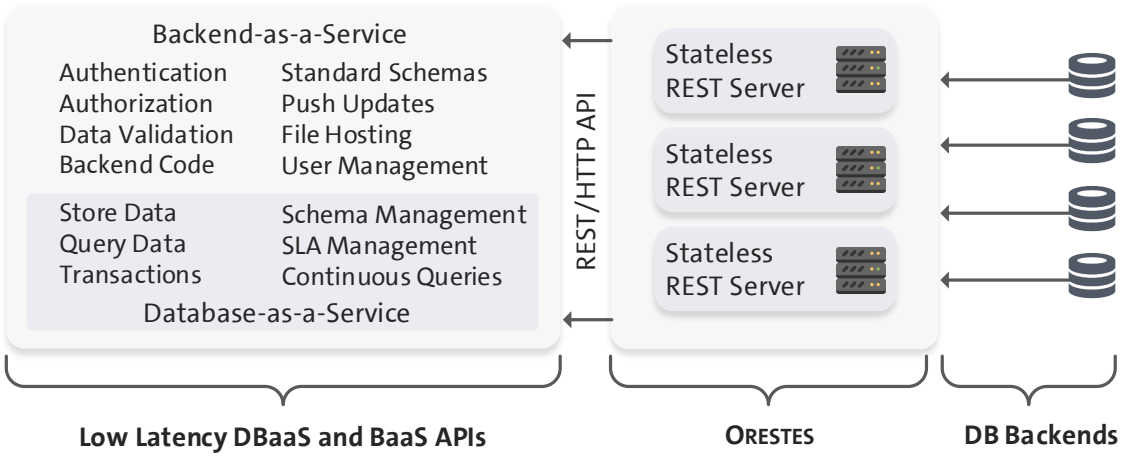


Figure 3.5: The High-Level Architecture of the Orestes middleware.

In the following, we will discuss how Orestes achieves a foundation for low latency cloud data management by covering the REST API, a polyglot persistence architecture, scaling and multi-tenancy mechanisms as well as the database-independent data management functions applied on top of the supporting data stores.

3.5.1 Architecture

To expose existing data stores as a BaaS without prior modification, the Orestes middleware and its **unified REST API** have to be powerful enough to expose the possible capabilities of the underlying database system (e.g., conditional updates) without compromising its non-functional properties (e.g., scalability of data volume or linearizable consistency). To this end, the Orestes architecture is comprised of a superset of database system capabilities spanning from client-side persistence APIs to the server's REST interface.

Figure 3.6 shows the Orestes middleware architecture designed to meet the requirements for database independence, Database- and Backend-as-a-Service functions, scalability, availability, and multi-tenancy as well as low latency with tunable consistency. The architecture encompasses the complete path from the client to the server and can therefore be split into the three parts: server, network, and client.

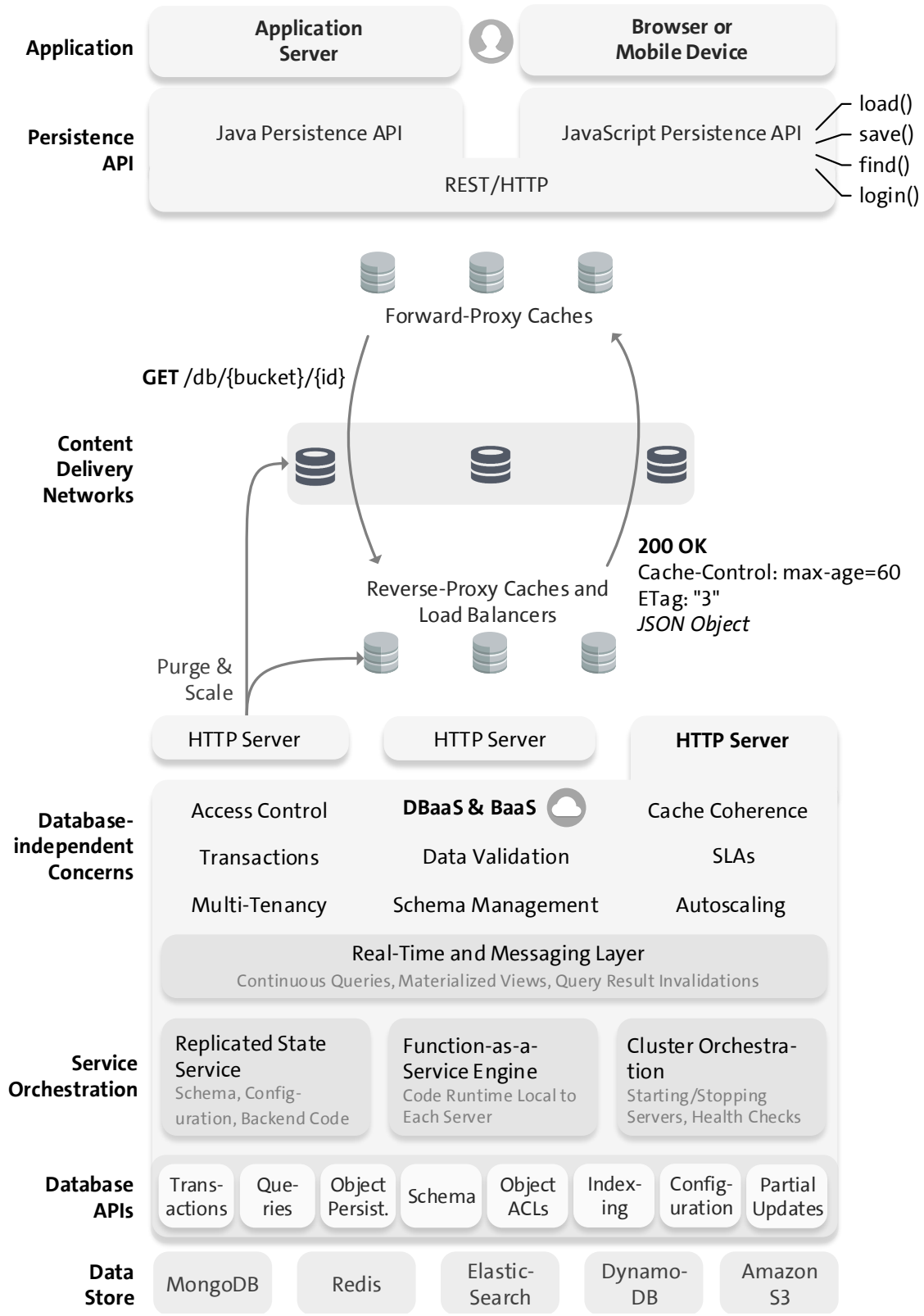


Figure 3.6: The Orestes middleware architecture with an exemplary request for loading a database object.

Server

The DBaaS/BaaS layer consists of a variable number of Orestes servers. The Orestes servers expose the REST/HTTP API and map it to the underlying database systems. The server tier and the database tier can be scaled and deployed independently. Through the stateless design of the Orestes servers, latency and throughput are only bounded by the saturated database system as the middleware can scale horizontally.

We distinguish between three types of **modules** in the server middleware. *Data Modules* express the mapping of data operations in the REST API to the underlying database (e.g., CRUD operations, queries, indexing, system configuration). Data Modules have to be implemented for each database that is to be exposed through the unified REST API. *Default Modules* on the other hand implement database-independent concerns that can be provided by default on top of a database system through a combination of data modules and middleware services (e.g., authentication, authorization, data validation, backend code, push notifications, transactions, schema management, SLA management, elastic scaling). Default modules can be overwritten to leverage existing native capabilities (e.g., table-level ACLs for authorization). *Core Modules* contribute the technical foundations of the system and are orthogonal to the underlying database (e.g., web caching, load balancing, HTTP networking, logging, TLS encryption).

One of the central default modules in Orestes is the **real-time and messaging layer InvaliDB**. It provides continuous query processing on top of any underlying database system by operating purely at the level of generic Orestes objects. A scalable messaging layer connects Orestes servers to the InvaliDB stream processing cluster for the exchange of after-images (the state of objects after an update) as well as query registrations and match notifications. The ability to match updates against queries is necessary to enable caching of query results. The details of the real-time query matching as well as cache coherence for query results are described in depth in Chapter 4.

Service orchestration involves all processes concerned with tying together the Orestes servers as one coherent unit, exchanging shared state, managing tenants and providing the FaaS environment for backend code execution. The replicated state service holds the application schema, configuration parameters and the uninterpreted backend code functions. It is partitioned in order to scale with increasing numbers of tenants and stored metadata. The FaaS engine is co-located with each Orestes server and executes application-defined procedures (implemented in JavaScript/Node.js [TV10] in the Orestes prototype). Backend code can have the form of *handlers* that are executed in event-based fashion upon CRUD operations. Alternatively, they can be established as explicit microservices that can be invoked by applications directly. This approach towards server-side code execution is employed in our BaaS architecture, because business logic and validation rules should oftentimes not be disclosed to clients.

The cluster orchestration is handled by a service that starts and stops new servers automatically based on workloads and machine or network failures. At the physical level, the

cluster is based on containerization (Docker Swarm [Swa17] in the Orestes prototype). This architecture allows assigning many logical Orestes servers to physical servers in the form of individual containers [Mer14]. Pairs of Orestes server containers and FaaS containers form an isolated network that does not interact with internal services or other tenants, but allows free communication to the Internet and between paired containers. Multi-tenancy is therefore straightforward, as each tenant runs in a different pair of containers that can be deployed on the cluster with little overhead and scaled across physical machines.

Module interfaces with different data management and BaaS capabilities decouple the access to underlying database systems. The major interfaces are:

CRUD. The CRUD interface contains the abstractions to create, read, update, and delete objects. Depending on the database, these objects can be key-value pairs, documents, records, rows, or nodes in a graph. Orestes assumes version numbers in order to allow caching and concurrency control. The CRUD interface has to be implemented for each database system.

Schema. The schema API allows to create and evolve the application's data model. Typically, this interface is provided as a default module by Orestes and mapped to the replicated state service. However, schemaful database systems like RDBMSs can implement this interface.

Orestes. The Orestes interface bundles all service-related information and actions. This includes configuration, service discovery, system health, caching metadata, and information about rate-limited users. The Orestes interfaces are not implemented for specific systems, but provided in a generic fashion.

Query. The query interface allows executing database-specific queries. Orestes makes no assumptions on the structures of the query, but requires a list of objects to be returned as a result and that pagination, respectively cursor operations are supported. The implementation of this interface is beneficial for performance. Without it, Orestes cannot exploit database-internal optimizations and therefore has to fall back to full-table scans to filter data.

Prepared Query. Through the prepared query interface, queries can be registered in the database. On the one hand, this allows the database to pre-optimize and parse the query. On the other hand, it allows to only expose certain parameterizations of queries to users.

Index. The index API is closely tied to queries and enables the definition of explicit secondary indexes for various data types, ranging from primitive types such as strings to full-text and geospatial data.

File. The file interface allows storing and retrieving blobs, as well as serving them over the web as assets for websites. Some database systems explicitly support file storage

(e.g., GridFS in MongoDB [CD13]) and can provide this capability. For systems that do not, Orestes falls back to the object store (S3 [Ama17a] in the prototype).

User. The user interface is concerned with the registration, management, and login of users via different protocols (e.g., OAuth [Har14]). As this interface can map to CRUD operations, its implementation is optional.

Code. The code API allows updating and retrieving server-side handlers and methods. The implementation in the prototype realizes a scalable Node.js tier as an FaaS.

Device. Installations of mobile apps are tracked through the device interfaces. This allows sending push notifications to specific groups of devices or individual users. The implementation in the prototype supports the two wide-spread push protocols by Google and Apple [YAD14].

Partial Update. The partial update API permits modifying database objects in place, to circumvent read-modify-write cycles that are prone to contention. Without explicit partial update support, Orestes will use optimistic concurrency control based on version numbers of objects.

Transaction. The transaction API supports ACID transactions over arbitrary objects. The default implementation provides scalable cache-aware transactions as described in Chapter 4. For transactional database systems, the interface can be overridden.

Asset. Using the asset API, clients can use Orestes as a CDN: Orestes will fetch provided URLs from a given origin and apply the same caching techniques as for any data directly stored in the service. The interface thus allows easy integration and acceleration of legacy systems.

Event Sockets. As the only non-HTTP interface, the event sockets API exposes real-time queries and event notifications. Clients can pose queries via WebSocket connections and receive updates to the respective query results in real time. The default implementation using InvaliDB provides horizontal scalability and abstracts entirely from the underlying data store.

Integration of new database systems into the polyglot architecture of Orestes is easy, because many features are available out-of-the-box through default modules. The Orestes server prototype is implemented in Java 8 and uses the Jetty framework [Jet] as a central HTTP networking component. The REST API uses a declarative framework that we implemented and made available as open-source[5]. An overview of the internal server architecture is given in Section 3.5.9.

Network

Orestes relies on **HTTP** as the core protocol and uses web caches through their standardized expiration-based caching model. The Orestes servers can be exposed through load balancers or reverse proxy caches, as their statelessness allows handling arbitrary client

[5] `https://github.com/Baqend/restful-jetty`

requests without sticky sessions. Orestes has an extensible purging interface to support a broad set of invalidation-based caches with non-standard invalidation APIs. Optimizing network performance (see Section 2.3) is handled by Orestes and does not require specific support from the database systems. Optimizations include using fast TCP and TLS handshakes for lower connection establishment latency, as well as HTTP/2 and its new performance mechanisms.

Orestes can be combined with any type of **Content Delivery Network (CDN)** for invalidation-based caching. In the general case, it will be treated by Orestes similar to any other cache. However, if the CDN supports the Varnish Configuration Language (VCL) [Kam17], the Orestes prototype enhances the cache with the ability to validate access tokens used for authentication, perform authorization on cached resources, and apply preventive measures against distributed denial of service (DDoS) attacks, in particular by handling rate-limited requesters from the cache only. This form of edge computing allows minimizing latency for protected cached data without compromising latency.

Client

Clients can either be pure **DBaaS clients** (application servers) in three-tier architectures or **BaaS clients** (browsers and mobile devices) in two-tier architectures. While Orestes can be adopted in any programming language and platform that supports HTTP communication, persistence APIs provide additional benefits. In particular, they can obtain a seamless integration in the application data model by exposing objects as native classes in the respective language. Additionally, developers get easy-to-use APIs for working with the objects and executing queries, while the persistence API takes care of managing the object lifecycles and ensuring that identical objects share an identity in the scope of a persistence session [TGPM17]. In the following we describe the concrete persistence APIs for the Orestes prototype – similar APIs could be developed for other programming languages as well.

The primary language used for clients in the Orestes prototype is **JavaScript**. JavaScript allows the persistence API to be applied for any type of websites and web app as well as mobile application frameworks based on web views (e.g., Ionic [Hil16]) or JavaScript engines (e.g., React Native, Titanium, NativeScript [Rea17]). The Orestes JavaScript SDK is based on concepts of the Java Persistence API [DeM09] from which it inherits the model of entity managers, dirty checking, persistence by reachability, and object lifecycles. However, it extends it in many data management aspects like explicit support for semi-structured schemas and continuous queries. The BaaS functions are also deeply integrated into the persistence API. For example, the client and server can share data validation code and high-level APIs support login and registration of users.

Listing 3.1 shows an example of the JavaScript API. It makes use of language features introduced with the ECMAScript 2015 standard [Int17]. All asynchronous operations in the SDK are based on the `Promise` (also known as futures) concept, to avoid unstructured callback code. In the example, first, a new `Message` object is created and inserted (lines

```
1  //Create and insert a new message object
2  const msg = new DB.Message();
3  msg.name = 'Felix';
4  msg.message = 'Hello World.';
5  msg.insert().then(() => console.log('insert completed.'));
6  //Perform a query
7  DB.Message.find()
8      .matches('name', /^Fel/)
9      .descending('createdAt')
10     .limit(30)
11     .resultList()
12     .then((result) => console.log('result received.'));
13 //Register a user
14 DB.User.register(new DB.User({'username':
       'test@example.com'}), 'password');
15 //Load a user and perform a partial, commutative update
16 DB.User.load(userId).then((user) => {
17     const update = user.partialUpdate()
18       .set('nickname', 'Alice') //sets 'nickname' to 'Alice'
19       .inc('age', 1);           //increments 'age' property
20     return update.execute();
21 });
```

Listing 3.1: Example of using the Orestes JavaScript SDK.

1 to 5). The `Message` class is generated by the SDK based on the schema defined at the server and inherits methods for CRUD operations, as well as automatic dirty checking, to detect when changes need to be persisted to the server. All object instances are managed by an entity manager that guarantees referential integrity and is also responsible for cache coherence. Next, a query is executed using a builder pattern (lines 5 to 12). After that, a registration for a user is performed as a typical BaaS operation (lines 13 to 14). Afterwards, a user is fetched by its ID, and a commutative partial update is performed as an example of a more advanced data management operation (lines 15 to 21).

Besides JavaScript, the Orestes prototype also supports TypeScript [Mic16], a statically typed language that is a superset of the JavaScript standard [Int17] which it is transpiled to. The advantage of TypeScript is that Orestes can generate so-called *typings* from the schema. Typings make the data model available for compile-time type checking and code completion, thus preventing potential bugs. The SDK is complemented by a Command Line Interface (CLI) to facilitate the development and deployment process. The CLI allows to start and stop tenants as well as deploying local folders, schemas and backend code to staging and production environments. Additionally, the CLI supports cloning boilerplate projects for different frontend environments and frameworks (e.g., React, React Native, Angular, Bootstrap, Ionic, and Vue).

The Java API of the Orestes prototype is a low-level API with the primary purpose of providing a foundation for integration and performance tests. It replaces a previous im-

plementation of the Java Data Objects (JDO) standard [Rus03b] that does not match the requirements of BaaS systems well. A different Java API is used for Android, that offers SDK abstractions geared particularly towards the requirements of mobile devices with limited computing and power capacities [Dom18]. A similar iOS SDK for Orestes is based on Swift [Sch17]. As the Orestes interface is additionally specified in the OpenAPI standard [Ope17], client SDKs for roughly 30 programming languages can be generated automatically. These do not provide advanced persistence features, but offer easy access to the REST interface through a typed API.

3.5.2 Unified REST API

To make Orestes a universal data management platform for different types of applications, a **unified REST API** is pivotal. To expose operations under flexible requirements, it needs to be clearly structured by database capabilities (cf. Section 3.1), similar to the Orestes interfaces. Therefore, the unified REST API is also composed of different modules, as shown in Figure 3.7. They are defined through a declarative REST specification language that we introduce in order to describe the effects of HTTP methods on resources identified by URI patterns. The specification documents are similar to routers in MVC web frameworks (e.g., Play), but enhanced to capture descriptions, types of parameters, and return values. The REST specification is loaded by the Orestes HTTP servers for validation, conversion (e.g., from JSON to a schema object), and to generate an interactive REST API documentation on the server's dashboard. From the REST specification, the Orestes servers generate OpenAPI documents [Ope17] to make the REST API discoverable and interoperable for different programming languages.

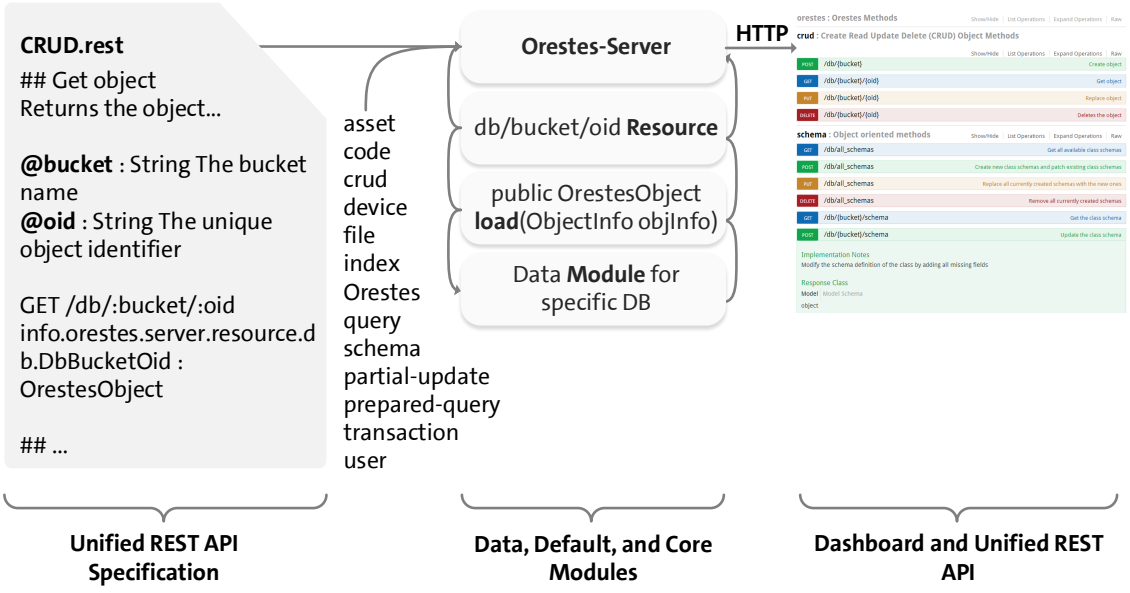


Figure 3.7: Composition of the unified REST API through resource specifications.

Operation	Request	Response
Create object	POST /db/:bucket	201 New object is stored and returned 202 Request accepted for transaction 404 Permission denied 409 Conflict, the object already exists 461 Bucket not found 462 Invalid object ACLs 466 Permission denied
Get object	GET /db/:bucket/:id	200 Object exists and is returned 304 If-None-Match header version matches 404 Object not found or permission denied 412 The If-Match header did not match 461 Bucket not found
Replace object	PUT /db/:bucket/:id	200 Updated object is stored and returned 202 Request accepted for transaction 409 Unique constraint violated 412 Object out of date 462 Invalid object ACLs 466 Permission denied
Delete object	DELETE /db/:bucket/:id	202 Request accepted for transaction 204 Object successfully deleted 412 Object out of date 462 Invalid object ACLs 466 Permission denied
List all bucket names	GET /db	200 List of all buckets
List objects in bucket	GET /db/:bucket/ids?start=0&count=-1	200 List of all objects 466 Permission denied 461 Bucket not found
Export objects of bucket	GET /db/:bucket	200, 466, 461 (see above)
Import objects to bucket	PUT /db/:bucket	200, 466, 461 (see above)
Delete objects in bucket	DELETE /db/:bucket	200, 466, 461 (see above)

Table 3.2: REST resources for CRUD with parameterized requests and potential responses.

The operations declared in each REST specification correspond to one or more methods in the data and default modules in the Orestes server. These communicate with the underlying database via its specific drivers and protocols. For an end-to-end example, consider a web application in which a user loads her profile. The call is performed through the JavaScript persistence API: `DB.profile.find(id)`. It employs the unified REST API by posing an HTTP request: `GET db/profiles/id`. For illustration, suppose the Orestes middleware wraps a MongoDB cluster. After parsing and checking the request, the server will call the CRUD data module's `load` method for MongoDB. The data module will then issue a query to MongoDB that returns the document (`db.find()`). Based on the schema defined for `Profile`, the server converts and returns the requested object. The fully-typed profile object returned by the SDK can eventually be displayed, for instance by feeding it into a template of a single-page application framework like Angular or React.

Table 3.2 shows exemplary **REST resources** and interactions for the CRUD module. This subset of the data management API includes operations for creating, retrieving, deleting, and overwriting objects, as well as discovery of buckets and contained objects. Orestes abstracts from data models by referring to aggregates as *objects* (e.g., a document, record, or key-value pair) and collections of objects as *buckets* (e.g., tables, folders, or namespaces). The HTTP methods are followed by URL templates with path placeholders prefixed by ":" and URL parameters assigned with their respective default value. The REST API makes use of HTTP's extensibility of status codes. For example, when an object is loaded (`GET /db/:bucket/:id`) the canonical `404` status code indicates that the object was not found, while the non-standardized status code `461` which is also in the class of `4xx` client errors, expresses that the bucket for the requested object does not exist. This refines the semantics of the REST API and simplifies the development of persistence SDKs.

The central requirement dominating the design of the unified REST API is that any side-effects impacting data management capabilities (see Section 3.1) have to be eliminated. By strictly incorporating the *semantics* of HTTP [IET15], Orestes enables cacheability of data and correct behavior of middleboxes like load balancers, firewalls, and web caches. In case of CRUD operations, new objects are created using the `POST` method which is non-idempotent: multiple invocations create multiple new objects. Similar to `PUT` and `DELETE` requests, any standard web cache will perform an *invalidation by side-effect* for `POST` requests, assuming that the requested resource has changed through the request. Replacing and overwriting objects are idempotent operations and therefore use the `PUT` and `DELETE` HTTP verbs. Since reads of objects are free or side-effects, they are performed with the `GET` method against a unique URL. Conditional request headers can be attached for revalidation using an `ETag` version number as the cache validator.

In practice, REST APIs for cloud data management can lead to performance problems. Two conditions we examined in detail are *falsely non-persistent HTTP methods* and *temporary TCP deadlocks*. Falsely non-persistent HTTP methods cause non-idempotent requests to open a new TCP connection, provoking unnecessary round-trips and negatively impacting latency. Temporary TCP deadlocks of 200-500 ms can occur by an interference of the *Nagle algorithm* and the *delayed ACK algorithm* defined in the TCP protocol standard to increase the effectiveness of TCP buffer management [APB09]. Both problems which we described in more detail in [GFW+14] are reasons for the difficulty of consolidating performance requirements with powerful abstractions in data management REST APIs.

A recently proposed alternative to REST APIs for data-centric services is GraphQL [Gra17]. Though GraphQL is utilized by large companies (e.g., Facebook), it exhibits the above problem of failing in performance requirements due to the inadequate use of HTTP. The problem arises as both queries and update operations are posed as `POST` requests with a JSON body as payload. While this simplifies the implementation of GraphQL interfaces, it is not suitable for low-latency cloud data management: queries and reads are inherently uncacheable, as HTTP caching requires the use of `GET` requests and the associated caching headers [FR14]. Further, GraphQL operation cannot be retried in case of connection loss

(e.g., by a proxy or a browser), as HTTP assumes non-idempotency. Our proposed unified REST API avoids this problem by aligning the semantics of HTTP with the requirements of cloud data management.

3.5.3 Polyglot Data Modeling and Schema Management

To accommodate the demand for both schemaless and schemaful data modeling, Orestes has to support hybrid schemas that allow strict typing, but do not enforce it. To achieve database independence, data modeling and schema management have to be decoupled from the database system. This enables polyglot persistence, as data models are checked and maintained at the database-independent middleware and mapped to the underlying database systems that can be either schemaless or schemaful.

To this end, Orestes integrates **object-oriented schemas** as a default module. Schema-free databases (e.g., Redis [Car13] and MongoDB [CD13]) thus get “bolt-on" rich schemas, whereas schema-aware databases (e.g., PostgreSQL [Pos17], Versant [Ver17]) can expose their own schemas. In most cases, explicit schemas are an advantage as they enable type-checking and validation that can prevent data corruption [Kle17]. The key idea in Orestes is to allow the co-existence of schemaless fields with typed fields for a single bucket. An Orestes schema is a mapping of field names to types for a particular bucket. Whenever an entity is saved, field values are checked against the types defined by the schema. Several field types are therefore supported:

Primitive Types. The primitive types *String*, *Boolean*, *Integer*, *Double*, *Date*, *DateTime*, *Time*, *GeoPoint* are scalar fields. If a database system does not implement a particular type, it will be stored as the next more general type in the hierarchy (e.g., a *String* instead of a *DateTime*).

References. To enable navigational access patterns and normalized data models, Orestes supports *references* as native types. Internally, references are represented as Strings (e.g., `/db/Todo/84b9...`). A *file reference* type allows mixing structured and unstructured data, by offloading the file contents to a specialized data store.

Collections. The three collection types *Set*, *Map*, *List* enable simple modeling of composition and 1:n relationships. The type of a collection can be any of the other types, including collections. A list stores a resizable array of elements, a set automatically eliminates duplicates, and a map consists of key-value pairs.

Subtypes. Nested data models are an efficient way to structure certain 1:1 and 1:n relationships. *Embedded objects* are as powerful as normal Orestes objects, but only exist in the context of their containing object and do not have an individual identity. Embedded objects, collections, and references can be combined arbitrarily.

Schema-free Fields. By exposing *JSON objects* and *JSON arrays*, schema-free parts of a data model can be stored as documents. If the underlying database system supports queries on schema-free data, the content of JSON fields can be queried similar to

other fields. As the JSON types grant all freedoms of schema-free data stores, the application has to manage their schema evolution explicitly.

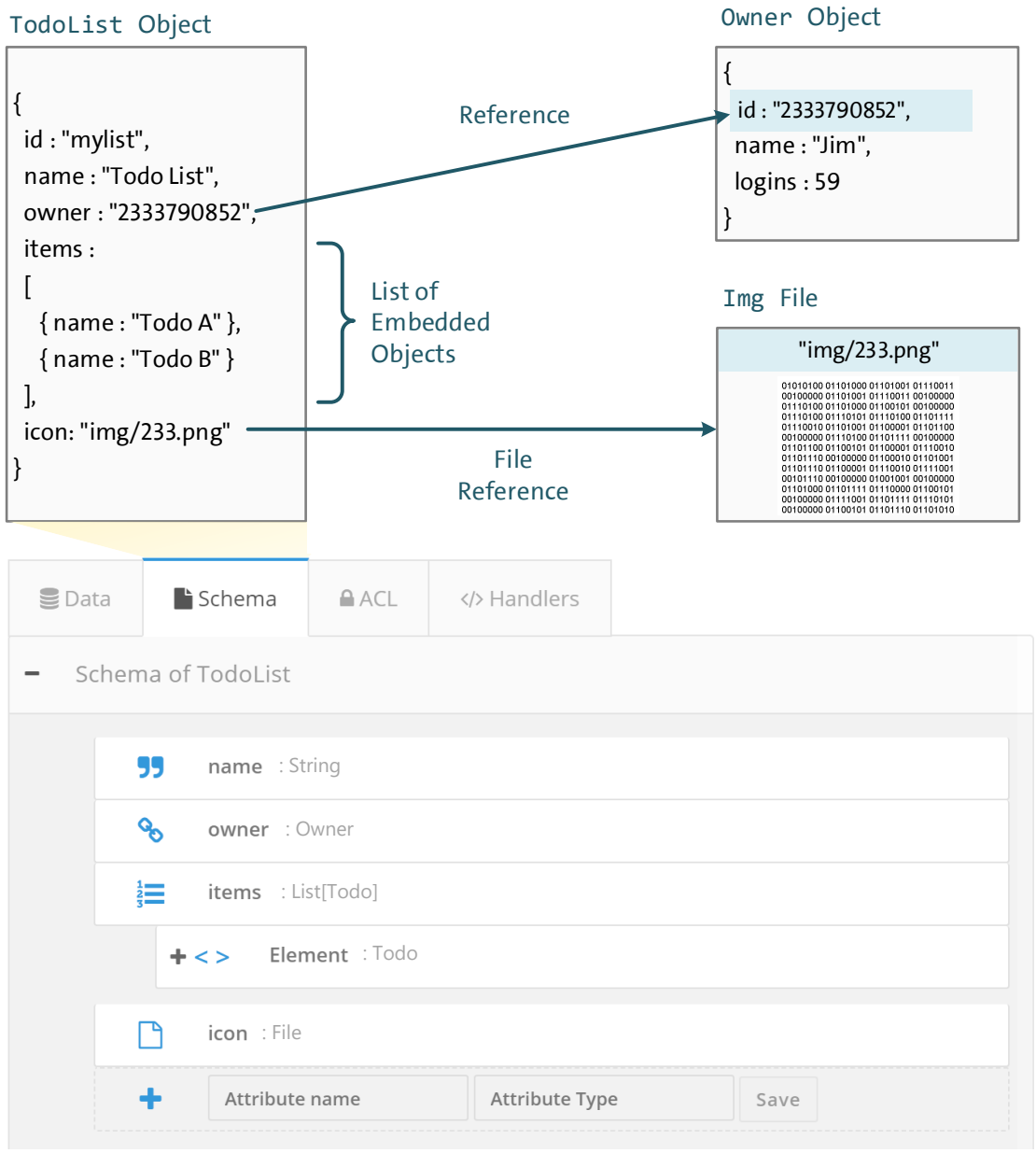


Figure 3.8: Example data model combining schemaless and schemaful elements, also showing the graphical schema editor.

The flexibility of the proposed data model is illustrated in Figure 3.8. As a use case, consider the data model of a simple to-do list application. The `TodoList` schema consists of primitive fields such as the `name` of the list and the `ID`. Since a user can create multiple lists (1:n relationship), every list has a reference to the respective owner object. A list has a composition relationship with its contained items that do not exist outside the context of a list and are therefore modeled as a list of embedded objects. As a simple example of polyglot persistence, the illustrated list holds an icon image as a first-class abstraction that

is transparently persisted to an object store and referenced by the respective `TodoList` object (e.g., Amazon S3 [Ama17a]). The lower part of Figure 3.8 shows the data modeling UI for designing schemas in the Orestes server's dashboard. Besides the graphical interface, the schema API exposes methods to evolve and create schemas as well as creating indexes and constraints on fields. The schema has a simple JSON representation that can be used to manage and version it as part of a source code repository and to deploy it via the CLI. In addition, a UML-based graphical meta-modeling editor can be used to manage the complete schema of an application [Mö17].

To allow reuse of defined models, the schema supports **generalization** hierarchies. When a bucket extends another bucket, it automatically inherits all of its parent's fields. All objects extend the `OrestesObject` which prescribes the existence of a primary key (`ID`). By default, the `ID` is generated as a UUID to ensure an even distribution for range sharding over the identifier. Furthermore, each object has a `createdAt` and `updatedAt` field that the Orestes server automatically maintains for convenience. The `ACL` property contains object-level access rules as described in 3.5.4. A `version` field is updated automatically and lays the foundation for cache revalidations and concurrency control.

Schema management is provided by the Orestes server, unless the underlying database system explicitly opts into implementing the schema API. Schema changes can be grouped into two categories based on their potential effects on data and the required coordination:

Safe Changes. Adding fields, adding new buckets, and changing field types to a parent type are commutative, associative, and idempotent and thus can be safely applied asynchronously. As safe changes are non-destructive towards existing data, they can be applied without side-effects at a later point in time [RRS+13].

Unsafe Changes. Deleting and renaming fields, dropping buckets, reordering fields, and changing field types to a non-parent type have side-effects and therefore require coordination. Unsafe changes can potentially corrupt existing data and negatively interact with concurrent schema changes. Therefore, unsafe changes have to be applied atomically across all Orestes servers in a synchronous fashion. Unsafe changes can be mitigated by using schemaless JSON types, thus offloading the responsibility of schema migration to the application.

Every Orestes server holds the complete schema in memory. To ensure global consistency of the data model, unsafe changes are published across all Orestes servers using the two-phase commit protocol [Lec09]. This ensures that all unsafe updates are applied atomically. Changes requiring data migration are executed by only one of the Orestes servers during this process. Safe changes, on the other hand, can be propagated asynchronously without blocking other schema updates.

3.5.4 Authentication and Access Control

To extend a DBaaS to a BaaS, user and access control are required as clients cannot be trusted to execute or see protected application logic. Ideally, even database systems without fine-grained access control should be enabled for bucket- and object-level access control. In Orestes, this is achieved by **role-based access control** (RBAC) and default schemas for common BaaS use cases: users, roles, and installations. These BaaS schemas have special semantics. *User* instances are automatically created when a user registers[6]. All user logins are checked against stored user objects. Registration and login are thus provided as database-independent default modules. *Roles* consist of a set of users that are granted that role and are stored in a roles bucket. *Installations* track mobile apps that are installed through app stores and are used to send push notifications to specific devices or groups of devices.

The core requirement for authentication and access control is their compatibility with caching for low latency and horizontal scalability. Therefore, Orestes uses a novel type of **access token** that can be interpreted in caches that support scripting languages without requiring coordination or session state lookups. In the Orestes prototype, VCL [Kam17] is used for the implementation of token handling in proxy caches and CDNs, but the overall scheme is generic. Upon successful login, an access token is generated. To enable stateless authentication and authorization, the token has to encode permissions and the user's identity in a secure way. To this end, we propose a variation of a token scheme proposed by Fu et al. [FSSF01] used for each user:

$$expires =< t_1 > \& renew =< t_2 > \& data =< u > \& digest =< HMAC_k(token) > \quad (3.1)$$

For the token, & indicates concatenation, $u =< ID(user) > \& < \{ID(role) \,|\, user \in role\} >$ is a binary encoding of a user's roles, and $k =< tenant\_ID \,, \& < server\_secret >$ is the secret for securely signing the token. The token thus contains the user's ID and his roles and a signature guaranteeing integrity (an HMAC for performance reasons). The same server secret used for signatures is also employed to hash passwords (each with an additional salt [Sch96]), so that a copy of the user database cannot be attacked unless an Orestes server is compromised, too. The two timestamps t_1 and t_2 indicate for how long the token is valid. Until t_1 expires, the token can be renewed providing the given token as authorization. t_1 thus bounds the maximum duration for which a token stolen by an attacker can be exploited. t_2 is the refresh interval of the token that is prescribed to guarantee the freshness of permissions encoded in the token as well as timely revocations. t_2 is typically in the order of minutes to hours, whereas t_1 can be in the order of months to years.

Based on this token, the Orestes servers check Access Control Lists (ACLs) on **schema level** and on **object level**. An ACL grants or denies access to users and roles for certain

[6] Besides a simple registration without identity validation, the Orestes prototype supports email-based registration and the OAuth protocol for providers such as Facebook, Google, GitHub, and LinkedIn.

operations: `read` and `write` on object level and `read`, `query`, `create`, `delete`, `extend schema`, `subclass schema` on schema level. Typically, schema-level ACLs are assigned at design time, when data models are defined. As more complex application rights cannot be purely defined declaratively in schema ACLs, an object ACL API exposes individual permissions for each object. As an example, consider the previous to-do list application: a typical configuration would grant schema-level insert rights for every user. However, modifications of a list would be limited to the respective creator. To express this, the predefined ACL property of a `TodoList` instance would be modified programmatically: `todo.acl.allowWriteAccess(DB.User.me)`.

To maximize the flexibility of the permission system, rights can be **granted** but also be **denied**. All of the above operations, e.g., `read` and `write` for objects, are represented by two lists (allowed and denied) containing user and role references. The decision procedure for determining whether access can be granted is based on the idea that deny rules win over allow rules. For a certain operation and user, access is first checked against schema-level rules and upon success also validated against object-level rules:

1. If a user has `admin` rights[7], access is always granted. This enables the development and administration of the application.
2. If no rules are defined, (public) access is granted.
3. If the user or one of her roles are contained in the deny list, access is denied.
4. If rules exist, but the user or one of her roles are not explicitly allowed, access is denied. Otherwise, the user or one of her roles are contained in the allow list, so access is granted.

Any objects that are publicly visible are eligible to caching in arbitrary web caches. For many websites and web applications, this is the most substantial part of the data. However, if read and query access is restricted by permissions, cacheability and therefore latency of the respective objects, query results, and files would be impacted negatively. Therefore, in our proposed scheme, even reverse proxy caches and CDNs can perform the ACL checks for read operations, if they support sufficiently expressive configuration languages (e.g., VCL). This scheme is illustrated in Figure 3.9. In order to enable authorization per object, a merged view of schema- and object-level ACLs is attached to protected responses. This metadata is used by supported caches for validation of access rights on cached data and stripped before forwarding the response to clients. The formats of schema-level and object-level ACLs are compatible, so that access can be checked based on bitwise operations and substring matching. Thus, even configuration languages that are not Turing-complete can validate tokens.

In contrast to schema-level ACLs, object-level ACLs need a specific data module for a particular database system. In MongoDB, for instance, object-level ACLs are enforced by

[7] In Orestes, there are three predefined roles. `admin` users have full access to everything. Users that are logged in automatically have the `LoggedIn` role. If an operation is triggered by backend code, it receives the `node` role. This allows to easily adapt permissions to the user's context.

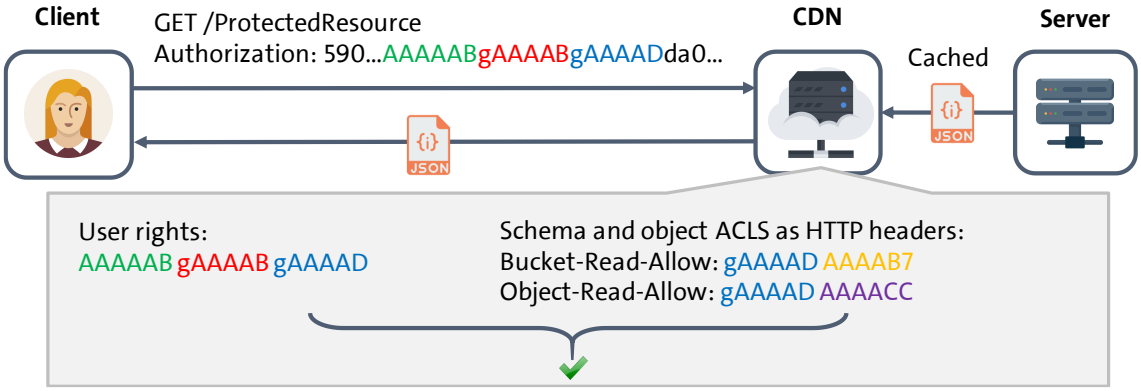


Figure 3.9: Example request validated against a protected, cached resource in the CDN.

conditioning data operations over a per-object, indexed ACL field containing allowed and denied users and groups. Thus, the task of ACL validation is pushed down to the query engines of database systems whenever possible. As the access token is self-contained, no shared server state is required. The same permission system is applied to files: folders can define schema-level ACLs, and the metadata of objects can contain read and write permissions. Schema-level ACLs and ACLs of cached objects and files are handled by Orestes independently of the database systems and object stores. The rich permission system thus enables the use of any supported data store in the context of serverless, client-driven application architectures.

3.5.5 Function-as-a-Service

To complement data storage with scalable business logic, Orestes incorporates a database-independent Function-as-a-Service layer (**backend code**). Server-side scripts can be registered for execution of critical business logic. In order to facilitate application development, server-side code is executed using the same JavaScript SDK that clients use (*isomorphic JavaScript*). The two primary requirements that the FaaS layer addresses are the need for application-specific backend APIs as well as the extensibility of data operations with custom behavior.

To this end, Orestes introduces **code modules** and **handlers**. They serve the purpose of running and structuring backend code closely integrated with stored data. Typical application use cases are tasks that require considerable resources or confidentiality, e.g., a payment process, training machine learning models, image processing, or validation of user-uploaded files. To make use of the large JavaScript ecosystem, the package repository *npm* is integrated into the FaaS engine of our prototype, allowing arbitrary libraries and extensions to be installed and reused. Orestes categorizes backend code as follows:

Code Modules. Code can be stored as a module under a certain path. On the one hand, this allows calling the module as a REST API, if it exposes the respective handlers for

HTTP methods. On the other hand, the module can be imported in other backend code to enable code reuse and a clear application architecture.

Scheduled Code. Code modules can be scheduled for repeated execution using cron patterns [SS94, SGGS12] that define the intervals of automatic invocation. Orestes takes care of ensuring that scheduled code is called at most once per interval and that jobs are distributed evenly across Orestes servers.

Insert Handlers. Each bucket can have associated **before** and **after** handlers that are called before or after successful execution of the respective operation on the underlying data store. The insert handler can, for example, modify or abort the operation, invoke third-party services, check special permissions, or maintain materialized views.

Update Handlers. Update handlers are similar to insert handlers and are executed for changes to existing objects. Update operations in Orestes are object-based and not set-based, to enable efficient caching through invalidations by side-effect (cf. Section 2.3).

Delete Handlers. In contrast to update and insert handlers, delete handlers do not have access to a provided object, but instead need to base their behavior on the identity of the object being deleted.

Validation Handlers. Before inserts and updates, any registered validation handlers check for violations of custom rules that cannot be defined declaratively in the schema (e.g., checking whether a `name` field starts with "X"). Like the schema, the validation code is made available in the client to combine preliminary client-side checks (e.g., in forms) with secure server-side validation.

Noticeably, *load* and *query* handlers are not provided, as their execution cannot be guaranteed in the presence of caches. Handlers and code modules do not affect caching, as they always have to be handled by the server. Backend code in Orestes enables the application to make a trade-off between data shipping and function shipping [ÖV11] in cases where both client-side and server-side processing would be possible.

The FaaS abstractions provided by Orestes are independent of the implementation of the code execution and can therefore rely on available FaaS platforms (e.g., AWS Lambda [Rob16]). However, to minimize latency, the execution engine should be co-located with the REST API server to prevent unnecessary network round-trips. In the Orestes prototype, tenants are therefore deployed in pairs of co-located Node.js and Orestes server containers (cf. p. 95). The Java-based Orestes servers can thus efficiently communicate with the Node.js FaaS engine through low-latency inter-process communication. The trade-off is that lower latency is chosen over independent scaling of the FaaS and BaaS tier that can only be provisioned in pairs. Since idle or underutilized Orestes or Node.js containers consume negligible resources, the latency benefit outweighs the stronger coupling. The advantage of containerizing the FaaS component is that new instances can be scaled within seconds with very little memory, CPU, and space overhead [Mer14, Swa17].

3.5.6 Query Processing

Queries are the primary means of data access for most database systems and therefore should not be restricted in their expressiveness by Orestes. Purely key-based lookups are handled without queries and are often the default for object-oriented data models that rely heavily on navigational access. Unlike queries, however, key-based access can be easily abstracted for all potential data stores. Queries, on the other hand, vary substantially depending on the category of database system that is exposed through the middleware (cf. Section 2.2). To allow different data models, Orestes therefore does not enforce a common query language and does not interpret or parse queries. Instead, queries are submitted by clients as opaque query predicates accompanied by general parameters for defining the maximum number of returned objects, an offset, sorting instructions, and potential projections of the result. The query module therefore only has to forward queries appropriately to the database system, preferably applying projections and ACL checks directly on the database to skip post-processing in Orestes. The database system can also implement the index module, enabling application developers to explicitly set secondary, unique, spatial, and full-text indexes based on the schema editor.

The primary goal for any read operations in Orestes is to enable low latency access through caching. This is achieved for queries through learning and invalidation mechanisms described in detail in Chapter 4. Besides these classic **pull-based** queries where clients explicitly have to pose a request in order to receive a result, Orestes also supports **push-based** queries. A pushed-based query is formulated using the exact same semantics as for a pull-based query. However, instead of receiving a single query result, a *self-maintaining query* is exposed as an abstraction for real-time changes.

Listing 3.2 shows examples for both types of queries in the JavaScript SDK. The query builder in the client constructs filter queries that are supported for both push- and pull-based queries (lines 1 to 5). The regular query (`resultList`) is executed against the

```
1  //Query with filtering conditions
2  var query = DB.Todo.find()
3    .matches('name', /^NoSQL/)
4    .containsAny('tags', 'lecture', 'video', 'book')
5    .equal('done', false);
6
7  //Regular pull-based query
8  query.ascending('name')
9    .resultList(...); //Called once
10
11 //Push-based query
12 query.descending('createdAt')
13   .limit(30)
14   .resultStream(...); //Called every time the result changes
```

Listing 3.2: Comparison of pull-based and push-based queries.

REST API and forwarded to the database – for example MongoDB for the Orestes prototype (lines 7 to 9). Push-based queries, in contrast, use WebSockets to enable real-time communication[8] (lines 11 to 14). The query is also sent to the Orestes server and executed in the database while also being forwarded to the distributed real-time query processing engine **InvaliDB**. As soon as InvaliDB detects that an update changes the previous result of a pull-based query, it sends a notification to subscribed clients. The architecture and properties of InvaliDB are described in Section 4.4, since the task of sending change notifications is mostly similar to that of query result cache invalidations.

From a software engineering perspective, continuous queries provided as self-maintaining results have the advantage that semantics are easy to reason about and do not require knowledge of query languages for real-time and streaming data [WGFR16, GÖ10, YG16, CGH+17]. As real-time query matching is decoupled from the database system, continuous queries can be posed in any query language that is supported by InvaliDB (MongoDB queries in the prototype). For example, if Orestes is configured as a middleware for the key-value store Redis, continuous queries can still be posed as more powerful MongoDB queries, as the real-time processing is purely based on the polyglot abstractions of Orestes. In contrast to database systems with built-in real-time interfaces (e.g., RethinkDB, Firebase, Meteor, and Parse) Orestes does not prescribe one particular query language and can enhance any database system with a push-based query interface for real-time queries.

In summary, Orestes makes as few assumptions about query capabilities of underlying database systems as possible. This allows cacheable pull-based and scalable push-based queries across different database systems.

3.5.7 Concurrency Control

In two-tier architectures, **concurrent updates** must be expected, since clients are directly involved in data modifications. Therefore, synchronization mechanisms for protecting against update anomalies are required for both single-object updates and multi-object transactions. Our concurrency strategy is coupled with the requirement that updates should be executed without overhead on latency and scalability, which precludes expensive mutual exclusion schemes.

To allow comparison of object versions, all Orestes objects have to expose a **version** field. The only requirement is that two different versions of an object have non-equal version numbers. Thus the implementation can be different depending on the built-in versioning schemes of the database system. In case of MongoDB, for example, documents are not versioned by the data store itself. Therefore, the version field is implemented by initializing objects with version numbers of 1 and incrementing them in each update, implicitly leveraging MongoDB's `$inc` update operator. Version numbers are exposed in Orestes as *ETag*

[8] Push notification services for mobile platforms (e.g., Apple Push Notification Service and Google Cloud Messaging) are not suitable for real-time query maintenance, as they have unpredictable latency [YAD14].

(entity tag) HTTP headers. For REST clients, this allows the use of the conditional header `If-Match` for specifying the update behavior for POST, PUT, and DELETE operations.

Based on the versioning scheme, Orestes offers three types of concurrency control with different performance trade-offs and guarantees:

Conditional Updates. A common concurrency anomaly in BaaS systems is a lost update [WV02]: two clients read the same object, both perform an update, and one modification gets overwritten. To prevent this anomaly, conditional updates apply a server-side validation to ensure that the version which the update is based on is still up-to-date. At REST level, the condition is expressed through an `If-Match` and the latest ETag of the object known to the client. At the database level, this type of update requires strong consistency and conditional updates at a per-operation level. When a conditional update fails, Orestes returns the newest object version. As the update cannot be retried automatically, the JavaScript SDK offers a convenient way of retrying updates through an `optimisticSave` method that calls an update method with the latest object version, potentially multiple times.

Partial Updates. As an alternative to the read-modify-write cycle of conditional updates, partial updates directly apply updates in the database. Orestes expresses partial updates as a collection of atomic update operations on field level. Atomic updates include setting a value, arithmetic operations (e.g., increment), adding, removing, and replacing elements in collections and bitwise operations. Under high concurrency, partial updates are preferable to conditional updates as they always succeed. Often, the application can exploit commutativity of partial updates to ensure application invariants. As an example, consider a stock counter in an e-commerce application. Using conditional updates, the entity containing the stock value has to be read, updated, and written back. Since subtractions are commutative, though, partial updates can be used to perform concurrent stock updates from different clients in single round-trips. At the database level, partial updates rely on the capability to perform updates without reading objects first, which is possible in most document and wide-column stores (e.g., in MongoDB [Mon17], DocumentsDB [STR+15], and HBase [Hba17]). In contrast to conditional updates, this does not require strong consistency and allows the database system to apply partial updates through convergent or commutative replicated data types [LPS09].

ACID Transactions. Conditional updates and partial updates only prevent anomalies for isolated write operations. To ensure correctness of multiple operations, multi-key ACID transactions are required. Orestes achieves them in a database-independent fashion, allowing any database system with linearizable updates to be exposed as a transactional data store. The scheme is compatible with caching and described in detail in Section 4.8.

By providing the three above concurrency schemes, applications can trade performance against ease of development and the required degree of correctness. While commercial

BaaS systems and many NoSQL databases fall short in their support of strong concurrency semantics, we are convinced that the ability to opt into more sophisticated concurrency schemes is strictly necessary for complex or critical applications. In polyglot persistence architectures, they should therefore be provided on top of existing data stores whenever possible. Conditional updates offer the simplest programming model, but drastically reduce throughput under high concurrency. Partial updates are powerful for single-object updates, but require the application developer to reason about the commutativity of updates. ACID transactions are the most powerful mechanism and the only option to ensure correct serialization of concurrent updates spanning multiple objects.

3.5.8 Scalability and Multi-Tenancy

The crucial requirement of the Orestes scalability model is to avoid introducing any bottleneck to elasticity and scalability that is not present in the underlying database systems. At the same time, read scalability should be increased by offloading data stores from reads, queries, and file downloads. This goal is achieved by scaling each tenant independently from other tenants and the database system. The approach is shown in Figure 3.10.

A **management and orchestration service** is responsible for deploying new tenants, maintaining the cluster and scaling the infrastructure. Orestes is designed to be provided on top of IaaS clouds, so that new VMs can be provisioned on demand and added to the cluster. To avoid the substantial resource and time overhead of providing a VM for each new tenant and tenants that need to scale, Orestes servers run in lightweight containers for process-level isolation. For example, in the Orestes prototype, VMs and containers are orchestrated through the cluster framework Docker Swarm [Swa17] that coordinates with a master server co-located with the Orestes management server. Like the Apache Storm master (Nimbus), the stream processing system that the prototype of the real-time layer is based on, Swarm achieves resilience to machine failures by storing state in a Zookeeper (ZK) instance under control of the Orestes management service [HKJR10].

Each **virtual machine** in the cluster comprises an TLS terminations proxy that accepts connections from CDNs and forwards requests to a reverse proxy cache. Besides caching responses in the reverse proxy, this setup allows zero-downtime configuration updates, as a new mapping of container endpoints can be updated in the reverse proxy without dropping active TCP connections. For each tenant, at least one **container pair** of Orestes servers and the FaaS runtime is active. The container pair forms a private network where the FaaS component can contact Internet services and the Orestes server, but no internal services. Usually, the underlying database systems are addressed through a proxy co-located with each VM that forwards operations and data to the independently scaled database clusters.

The cluster is maintained as a set of N VMs, each of which contains M container pairs. Thus, Orestes can provide the following scalability, multi-tenancy, and availability characteristics:

Scaling cluster capacity. If machines in the cluster become overloaded according to metrics like CPU, memory, and network usage [LBMAL14], new VMs are provisioned and added to the cluster. Afterwards, new or migrated containers can leverage the increased resource pool. To scale down the cluster capacity, containers are migrated away from a VM, and the host machine is stopped.

Provisioning tenants. To add a new tenant, a pair of containers is provisioned and scheduled to the VM with the most remaining capacity. The Orestes container is registered in the reverse proxy and exposed through a combination of a hostname and TCP port that gets registered in CDNs.

Updating tenants. In order to update a tenant without downtime (e.g., with a new Orestes server), a new container pair is provisioned first. Afterwards, the connections to the old container pair are drained by redirecting requests only to the new containers. Eventually, the old container can be safely shut down without affecting any users.

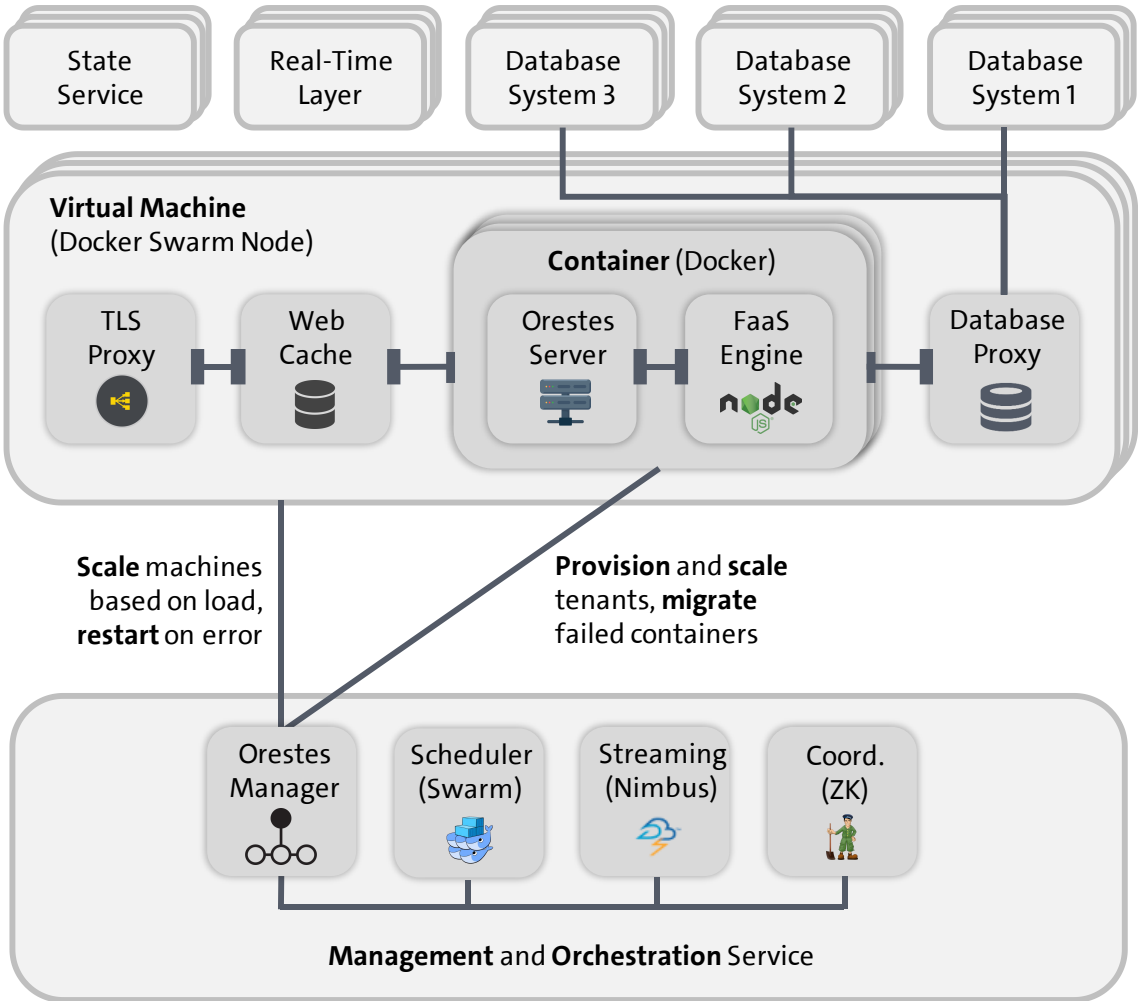


Figure 3.10: Scalability and multi-tenancy model of Orestes and its prototype implementation.

Scaling out and in. To scale an individual tenant, its resources can be monitored similar to VMs, since they are reported by the container runtime. Under high load, a new pair of containers can be added at any time and take over a fraction of the overall load.

Reacting to failures. When a tenant fails, CDNs detect failed requests and report the error to the management service. The failed containers can then be restarted on any available machine. Similarly, when a VM becomes unavailable, all containers on the machine are restarted over the cluster.

Scaling the database system. In the typical setup, all tenants are distributed over a single database cluster and use sharding to generate uniform load of data and operations. As many NoSQL systems support sharding [GWFR16], Orestes can proactively scale in and out the database system to match the aggregate workload of all tenants.

Like any other system, Orestes is subject to the **CAP/PACELC trade-off**. The design goal of Orestes is to keep any guarantees provided by the database system. In addition, Orestes offers consistent operations, schema management, and transactions. These are not available under certain network partitions, e.g., when the Orestes server is separated from the state service while a schema update has to be applied. Therefore, Orestes only guarantees availability for coordination-free operations that only require the database system to be available. Nonetheless, the CAP/PACELC trade-offs of specific database systems are maintained and only forwarded through Orestes. As Orestes supports a superset of potential non-functional properties, operations that are subject to unavailability during network partitions will become unavailable in Orestes, too. The only notable exception are cached objects, files, and query results. When the database system is unavailable, Orestes can continue to serve (potentially stale) data from caches. In contrast to availability, scalability is always enhanced by Orestes, as solely non-cached read operations and updates reach the database systems.

3.5.9 Server Implementation

The architecture of Orestes is geared towards providing low latency and high throughput for the REST API. For performance reasons, the Orestes prototype is written in Java. The Java Virtual Machine was chosen for the compromise between little overhead over native code, maintainability, and the availability of rich server frameworks for providing an HTTP server that is able to scale vertically with cores and CPUs [Oak14]. Also, the language had to be common enough to have good support for database system drivers that are employed in the data modules. Orestes is based on asynchronous, non-blocking I/O for high network throughput. The internal processing is structured in stages, similar to the staged event-driven architecture (SEDA [WCB01]) commonly found in web server implementations like Nginx.

The internal architecture of the Orestes server prototype with respect to the processing of an incoming request is shown in Figure 3.11. The request travels through several

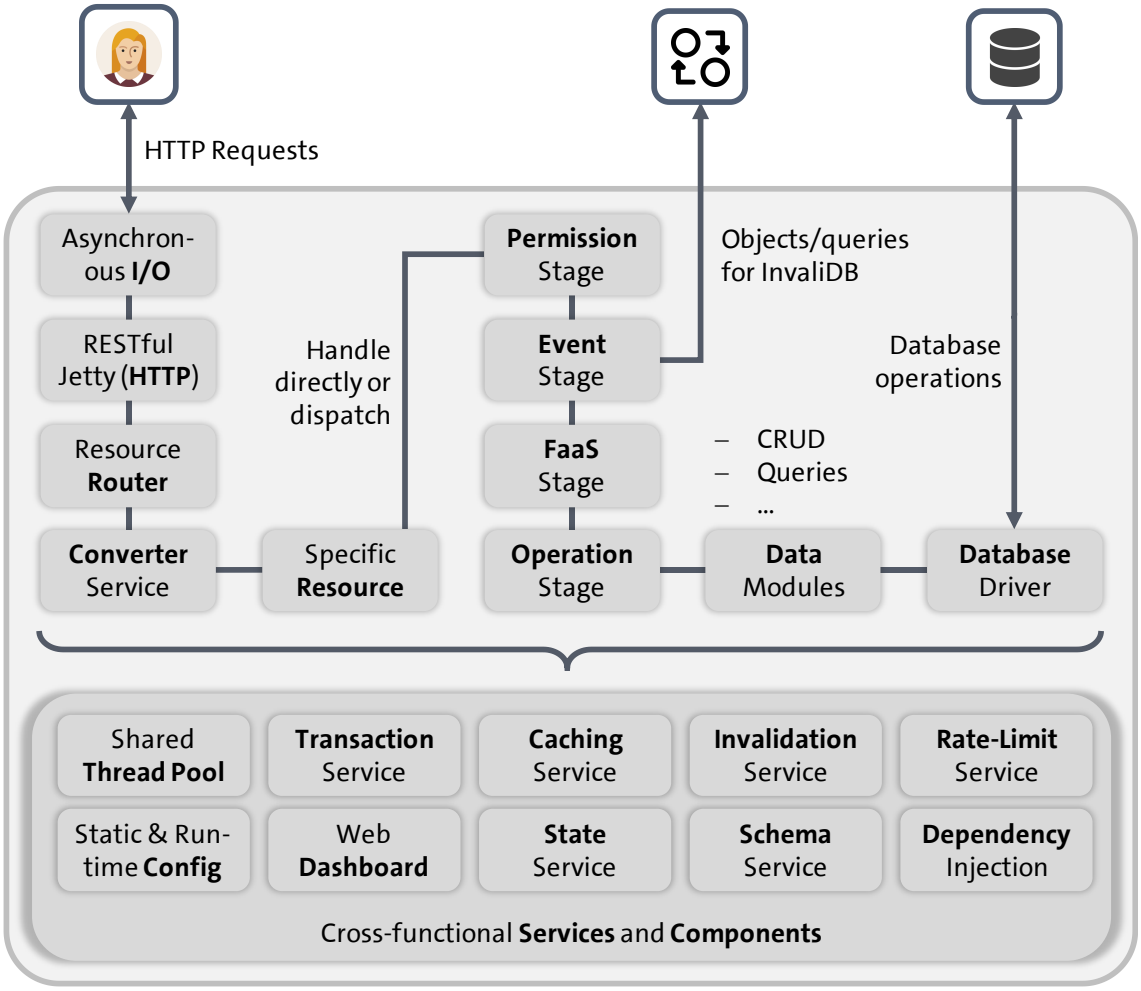


Figure 3.11: Prototype architecture of the Orestes server with respect to processing of incoming requests.

components in the direction of the database systems before being returned via the same path in return:

1. An incoming **HTTP request** is handled by an asynchronous non-blocking connector in order to minimize the memory overhead and latency of network communication.
2. The HTTP request is **parsed** by RESTful Jetty, our open-source enhancement of the Jetty HTTP server framework.
3. Based on the requested URL and passed headers, a registered `Resource` class (e.g., for `/db/bucket/id`) is selected by the **router** to handle the request.
4. If the request has a body, its content is **converted** to appropriate data structures (e.g., an `OrestesObject` containing fields and values).
5. The **resource** implementation decides how to handle a request. For data operations like CRUD and queries, the request is dispatched to a staged pipeline. Otherwise, the request may be directly answered (e.g., to return the server's current version).

a) The first stage checks whether the provided token represents sufficient **permissions** to perform the request. Each stage can perform logic either on the way of the request to the database or when the response is traveling back through the server.

b) The **event** stage provides the real-time layer InvaliDB with object snapshots and queries for both continuous queries and query result cache invalidation.

c) The **FaaS** stage invokes handlers, code modules, and scheduled code. Afterwards, it returns the results of the invocation.

d) The **operation** stage passes the request to the appropriate data module implemented for specific database systems. This step can either be based on a static configuration or our concept of the Polyglot Persistence Mediator.

6. In the **data** module, the request is mapped to the specific database driver. Depending on the operation, this step can be very simple or require additional logic (e.g., for checking object-level ACLs).
7. When the database returns the result, it is passed through the same components and stages in **reverse order** to apply any checks or modifications for the response. Typically the communication to the database systems is synchronous, as even modern NoSQL databases mostly use blocking, thread-based I/O [GWFR16].
8. Cross-cutting concerns are structured in **services**. For example, the state service offers a generalized interface for storing and retrieving state across Orestes servers such as registered FaaS code.

In summary, Orestes supports both horizontal and vertical scalability. Horizontal scalability relies on load balancing over multiple Orestes server instances. Vertical scalability is enabled by a staged server architecture that relies on request-parallel, asynchronous HTTP communication for low latency and high throughput.

3.6 Discussion

The purpose of Orestes is to enable low latency with tunable consistency for a broad set of database systems. This is achieved by a middleware design that promotes database independence of central data management capabilities such as schema management and caching. To exploit the benefits in a broad set of application contexts, Orestes exposes data stores as a DBaaS and BaaS by enhancing them with the necessary functional extensions (e.g., backend code) and security mechanisms (e.g., object-level access control). By making the middleware tier scale independently from database systems, multi-tenant cloud data management is achieved without compromising the heterogeneous trade-offs found in NoSQL database systems.

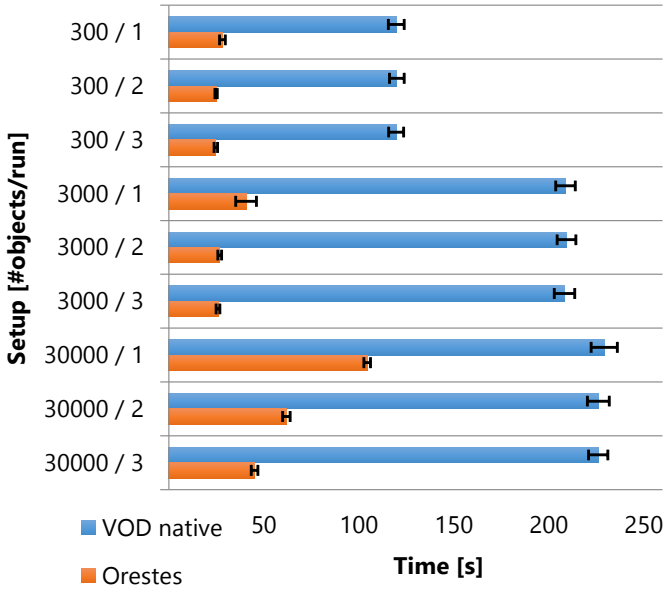
To illustrate the usefulness of Orestes with respect to requirements-driven development of data-centric applications, consider a typical development process for a scalable two-tier application:

1. First, an Orestes **data model** is defined that leverages the hybrid approach of mixing well-known data types with more flexible schemaless elements. The schema design uses a visual editor in a web dashboard to allow easy definition and sharing of the core application data model.
2. One or more **database technologies** are chosen to provide the DBaaS/BaaS used by the application. The step can either happen manually based on the NoSQL toolbox proposed in the beginning of this chapter, or through an automized process as described in Chapter 5.
3. If the application is based on an existing corpus of data, it is imported and transformed to the specified data model.
4. The application's web or app **frontend** is developed without requiring any previously developed server-side application logic (e.g., HTML template rendering). Orestes exposes a database REST API that is consistent across projects and therefore easy to apply.
5. Any critical or confidential parts of the business logic are developed as **backend code** modules, handlers, or scheduled code using the same APIs as the frontend to perform queries, load and save objects, and manage users.
6. Access to data is **secured** by defining appropriate schema ACLs during the modeling phase, and setting object-level ACLs that protect access according to any application-level notion of permissions.
7. The application artifacts like the schema, backend code, and frontend assets are **versioned and shared** across teams through an arbitrary version control system. Based on the source code repository, the application is deployed to testing, staging, and production environments using a CLI either from development machines or continuous integration servers throughout the different phases of development.
8. In production, Orestes handles the database and middleware infrastructure to deliver data consumed and displayed in the frontend with minimal latency while transparently scaling with data volume and request load.
9. If new applications based on the same data are created, they can share and reuse the backend side using the same REST interface without compromising performance.

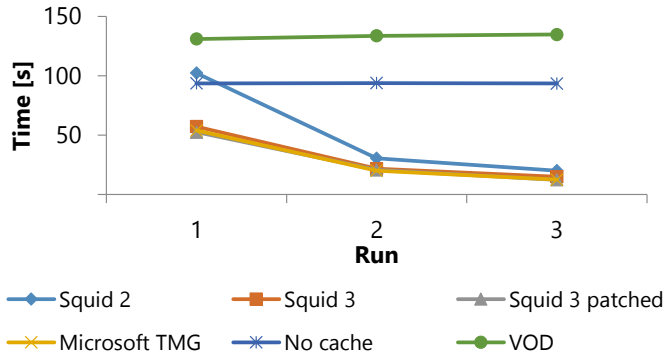
While any application could also be developed through the explicit use of disparate database and cloud storage technologies, we believe that a common API and middleware for database systems is a promising way to solve critical non-functional and functional problems across many systems. Thus, development efforts can be focused on the application-specific problems, while generic components are already available.

HTTP Performance Evaluation

In Chapter 4, we will provide in-depth evaluations of the performance and latency improvements obtained through Orestes. To give a first quantitative impression on the potential effects and the overhead of Orestes, we present results for experiments on the REST/HTTP layer in the following. Figure 3.12 compares the performance of Orestes to that of a database-specific access protocol[9].



(a) Performance of Orestes compared to VOD for 300/3 000/30 000 total database objects over 3 runs.



(b) Performance for different web caches and Orestes with and without caching.

Figure 3.12: Evaluation of the Orestes REST/HTTP layer in a micro-benchmark compared to native database access.

In the experiments, 50 client machines connected to an Orestes server with a Versant Object Database (VOD) are separated by a network latency of 165 ms ± 2 ms using the Amazon EC2 regions Ireland and California. Web Caching is performed through a forward-proxy cache co-located with the clients. The workload that is concurrently executed by all clients

[9]The detailed results are published in [GBR14].

is a micro-benchmark modeled after a social networking scenario (500 operations per second, 90% reads, 10% writes, navigational access with sporadic queries and transactions). There were three consecutive runs for different sizes of the database comparing Orestes exposing VOD against VOD with its binary TCP protocol.

Figure 3.12a shows that the average execution time of the test is heavily reduced with Orestes. The performance improvements are not only a consequence of the applied web caching. The ability of the Orestes unified interface to combine different operations in a single round-trip, makes network communication significantly more effective in high-latency environments. For example, by batching a transaction commit into a single HTTP request, multiple round-trips can be saved.

Figure 3.12b reports the results of the same micro-benchmark for a setup of hardware machines using a single client, different caches (both in Hamburg) accessing Orestes/VOD (California). The results show that the large performance advantage is consistent across different web caches. This is a consequence of Orestes relying on standard HTTP caching. Furthermore, the experiments also show that Orestes without any caching is competitive to custom database protocols. For detailed results in this setting, we refer to [GBR14] as well as the evaluations discussed in Chapter 4.

Due to the low overhead and efficient communication of updates in Orestes, even without any caching, the overall performance of Orestes is roughly 30% better compared to VOD's native protocol. Thus, the overall performance of REST/HTTP as the only interface of Orestes, does not impose a substantial overhead. Any latency optimization achieved by Orestes therefore directly improves performance over native database protocols.

3.7 Summary

In this chapter, we derived an architecture that enables low latency across different database systems. The set of data management requirements that should be supported was gathered through an extensive survey of scalable NoSQL database systems. We introduced a classification scheme that captures the dependencies between database implementation techniques and the related functional and non-functional properties. This **NoSQL toolbox** was used to deduct a simple **decision tree** to narrow down the possible system choices to candidates that support them.

The surveyed set of capabilities allowed us to propose **Orestes** as a DBaaS/BaaS middleware architecture that maintains the different systems' trade-offs and enhances them with low-latency access, BaaS capabilities for two-tier architectures, and a scalable multi-tenancy model. We then discussed how these extensions can be provided in an efficient and expressive fashion and proposed a scalable and database-independent implementation. Orestes employs a unified REST API as a superset of data management capabilities that can be consumed directly from any, even untrusted, clients. To this end, a data-centric access control model is proposed, as well as an FaaS execution environment. Orestes tack-

les query processing and concurrency control in a database-agnostic middleware. Thus, these features are available irrespective of the individual databases' capabilities. In particular, real-time queries enable users to subscribe to specific queries and optimistic ACID transactions introduce strong guarantees for interleaved multi-object modifications.

The Orestes architecture offers a universal DBaaS/BaaS environment that we utilize to obtain our core contribution of latency reduction in cloud data management as described in the following chapter.

4 Web Caching for Cloud Data Management

This chapter introduces a cache coherence scheme for dynamic data to significantly reduce latency on the web. The approach leverages traditional web caching technology and enhances it with the capability to keep data up-to-date in real time on a global scale. Freshness is particularly challenging in cloud data management, as data tends to be volatile and may change at any time. For this reason, we introduce means to expose fine-grained trade-offs between the desired latency and the guaranteed consistency levels for applications. This approach enables data-driven web applications and mobile apps that rely on cloud-hosted data and are still able to achieve imperceptible response times.

First, we will introduce the **Cache Sketch** data structure that allows Orestes to transfer the task of maintaining cache coherence from the server to the client. Using the Cache Sketch, we enable end devices to keep expiration-based caches up-to-date with very little overhead. The strategy is complemented by proactive updates to invalidation-based caches executed by the DBaaS/BaaS. The Cache Sketch enables several tunable consistency levels for caching objects and files. The key guarantee is Δ-atomicity that bounds staleness and exposes Δ as a parameter that can be chosen per request, session, or application.

Based on the construction of the Cache Sketch, we extend the caching scheme to support dynamic query results. To this end, changes to query results have to be detected in real time. To enable high cache hit rates and low space overhead, we propose **TTL Estimation** as a mechanism to determine the approximate time until a cached query result becomes stale. To minimize the number of round-trips to assemble a query result, we introduce an analytic model that determines the best encoding of results for a given access pattern.

To solve the problem of high abort rates for optimistic transactions, we propose **Distributed, Cache-Aware Transactions (DCAT)**. DCAT exploits our caching scheme to lower the overall transaction execution time and significantly reduces aborts. This allows applications to fall back to the strong guarantees of ACID transactions when the guarantees of the Cache Sketch alone are insufficient.

We provide evidence for the performance improvements through both Monte Carlo simulation and experimental evaluation. For typical workloads on the web, the Cache Sketch

approach can improve performance by an order of magnitude and therefore is a major step towards a web without loading times.

4.1 Cache Sketches: Bounding Staleness through Expiring Bloom Filters

In this section, we introduce the **Cache Sketch methodology** that employs automatic caching of database objects and files requested through a REST/HTTP API. As motivated in Chapter 2, latency for cloud services is an open challenge that can currently not be addressed through web caching, as expiration-based models offer insufficient consistency guarantees for data management. Subsequently in Section 4.4, we extend this approach from object and file caching to query result caching.

We achieve **cache coherence** through a dual approach: *expiration-based* web caches (browser/device caches, forward proxy caches, ISP caches) are kept consistent through client-side revalidations enabled by the Cache Sketch data structure, whereas *invalidation-based* web caches are updated by purge requests issued by the database service. The proposed caching methodology is applicable to any data-serving cloud service, but particularly well-suited for DBaaS and BaaS systems. The Cache Sketch is a Bloom filter of potentially stale objects maintained in the database service. To determine whether an object can safely be fetched from caches, clients query the Cache Sketch before reads. If the object's ID is contained in the Cache Sketch, a revalidation request is sent, as intermediate caches might hold a stale copy. The issued HTTP revalidation request instructs caches to check whether the database object has a different version than the locally cached copy. If a false positive occurs, a harmless revalidation on a non-stale object is performed, which has performance implications comparable to those of a cache miss.

Clients leverage the Cache Sketch for three different goals: fast application and session starts (*cached initialization*), cached reads with consistency guarantees (*bounded staleness*), and low-latency transactions (*distributed cache-aware transactions*). For *cached initialization*, clients transparently store every fetched object in the client cache (usually the browser cache). At the begin of a new session or page visit, the Cache Sketch is transferred, so clients can check which cached copies from the last session are still up-to-date. The number of necessary requests is thus reduced to the cache miss ratio of intermediate caches. To maintain Δ-*bounded staleness*, the Cache Sketch is refreshed in intervals of Δ. The interval constitutes a controllable upper bound on the staleness of loaded objects. Similarly, *distributed cache-aware transactions* load the Cache Sketch at transaction begin. Subsequent transactional reads exploit cached objects, reducing the overall duration and associated abort probability of the transaction.

By optimistically caching all objects and employing the Cache Sketch to only revalidate stale objects, almost the same cache hit ratio is achieved as if the time to the next write was known in advance. An object can only be removed from the Cache Sketch once

it is known to have been expired in all caches. Thus, precise estimations of expiration times impact cache hit ratios after writes as well as the number of necessary invalidations. To tune the inherent trade-off between cache hits, stale reads, invalidations, and false positives towards a given preference, we present the *Constrained Adaptive TTL Estimator* (CATE) that complements the Cache Sketch by adjusting cache expirations to optimize the trade-off.

This section covers three central contributions:

- We propose the **Cache Sketch** as a data structure that enables the use of expiration- and invalidation-based web caching for cloud data management systems to combine the latency benefits of caching with rich consistency guarantees.
- We describe the Constrained Adaptive TTL Estimation (**CATE**) algorithm that computes object expiration dates to minimize stale read probabilities and invalidations while maximizing cache hits.
- We present the Monte Carlo caching simulation Framework YCSB wrapper for Monte Carlo simulation (**YMCA**) that allows to analyze and optimize caching strategies and Cache Sketch parameters for pluggable network, database, and caching topologies.

In the following, we first present the Cache Sketch with its properties and effects. Next, we outline the TTL estimation problem and a possible solution. Afterwards, we introduce the YMCA simulation framework and present simulated and empirical results for the proposed combination of web caching and cloud data management.

4.1.1 The Cache Sketch Scheme

The expiration-based caching model of HTTP was deliberately designed for scalability and simplicity. It therefore lacks cache coherence protocols and assumes a static TTL (time to live) which indicates the time span for which a resource is valid, allowing every cache to keep a copy. This model works well for immutable content, for example a particular version of a JavaScript library. With the rise of REST APIs for cloud services, however, this model fails in its naive form – TTLs of dynamic content, in particular database objects and query results are not known in advance. This has led to database interfaces that forbid caching in the first place as staleness would be uncontrollable otherwise (cf. Section 2.3.3).

Figure 4.1 shows an architectural overview of how our Cache Sketch approach addresses this problem. Every cache in the request path serves cached database objects requested by their respective keys to the client, which can either be an end user's browser, a mobile application, or an application server. The Bloom filter of the client Cache Sketch is queried to send a request either as normal request (object not contained) or a revalidation (object contained). The revalidation forces caches to update their copy using an HTTP request conditioned over the object's version (*ETag*).

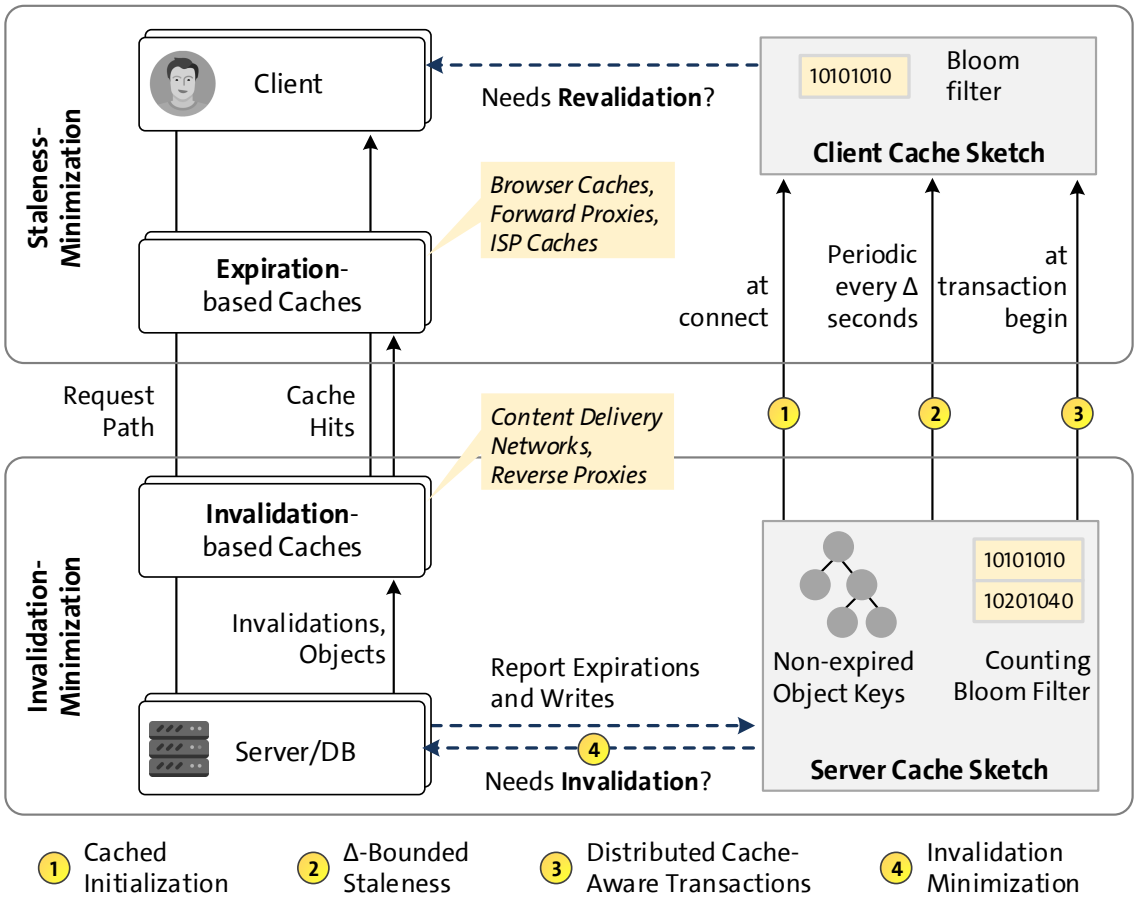


Figure 4.1: Architectural overview of the client and server Cache Sketch.

The database service tracks the highest TTLs per resource provided at a cache miss. On a subsequent write, the object is added to a Counting Bloom filter [BM03] of the server Cache Sketch and removed when the object's TTL is expired. The database service is furthermore responsible for purging objects from invalidation-based caches (CDNs and reverse proxy caches), which allows them to answer revalidations. To minimize invalidation broadcasts, purges are only sent, if the server Cache Sketch reports an object as non-expired. It is important to note, that this scheme does not require any modifications of the HTTP protocol or web cache behavior. The proposed Cache Sketch approach is further not specific to a particular database service architecture and can be realized either directly in the nodes of database system or as a tier of stateless REST servers exposing the database. We chose the latter approach by building on Orestes as described in detail in Chapter 3.

There are several advantages of caching database objects close to clients:

- Cache hits have **lower latency** and higher throughput than uncached requests, as TCP throughput is inversely proportional to round-trip time [Gri13].
- The database service is under **lighter load**, as it only has to handle write requests and cache misses.

- Clients profit from requests of other clients, as all caches except the browser/device cache are shared and thus increase **read scalability**.
- **Flash crowds**, i.e., load spikes caused by unexpected and sudden popularity, are mitigated by caching and do not bring down the database service [FFM04].
- **Temporary unavailability** of the database service can be compensated for reads by letting CDNs and reverse proxies serve cached objects while the service is unreachable.

4.1.2 The Client Cache Sketch

For each potentially non-expired, cached object x, the client Cache Sketch has to contain its key $key(x)$. For now, we will only consider key/ID lookups – the most common access pattern in key-value, document, and wide-column stores – and discuss how the scheme can be extended to query results in the following section. The client Cache Sketch is based on Bloom filters, as they satisfy several properties that are central to our requirement for efficient detection of potentially stale objects:

Additions and Removals in Server, Lookups in Client. The server needs the ability to add and remove potentially stale objects from the Cache Sketch, whereas the client needs to check for containment. The Bloom filter with its extension to counting offers these operations for the client and server, respectively [FCAB00].

Fast Client-Side Lookups. The Bloom filter supports $O(1)$ lookups and thus is very efficient for clients to be queried as an in-memory data structure for any read.

Small Size. As the client Cache Sketch needs to be transferred often, its size should be as small as possible. Bloom filters are within a factor of 1.44 of the theoretical lower bound of required space for a given false positive rate [BM03].

No False Negatives. Any false negative for a Cache Sketch lookup would entail an inconsistent read with unbounded staleness. Therefore, false negatives must be prevented, which is the case for Bloom filters. Occasional false positives, on the other hand, are permissible, as they only impact latency (cache misses) and not correctness.

Generation in Constant Time. To scale with the number of clients, fetching the Cache Sketch should not impose any computation overhead in the server, but instead correspond to a constant-time in-memory dump of a data structure. This can be achieved with Bloom filters, as described in the following section.

According to Broder et al. [BM03] the concept of using Bloom filters for sets, when space requirements are of high importance and false positives can be tolerated, is known as the *Bloom filter Principle* (cf. [BM03]). A Bloom filter is a space-efficient encoding of a set and is based on bit vectors and hash functions. Bloom filters were developed in 1970 by Burton Bloom [Blo70] as a memory-efficient data structure for hyphenation programs. The idea is to query a word to determine whether it is suitable for rule-based hyphenation,

like 90% of all English words, or whether it needs to be looked up in a dictionary on hard disk. Memory efficiency is critical to maintaining the data structure in main memory, while the negative side-effect of a false-positive is negligible (the lookup of a word from disk in Bloom's original setting). Bloom filters have since been used for a wide variety of database problems [BM03, FCAB00]. In the last 10 years, their usefulness has been discovered for many other areas such as collaboration, routing, and data analysis [TRL12].

We only cover aspects of Bloom filters and probabilistic data structures that are relevant to our approach and refer to the comprehensive treatment of their mathematical properties by Broder et al. [BM03] for details. A discussion and comparison to alternative data structures for the Cache Sketch is provided in Section 6.1.4.

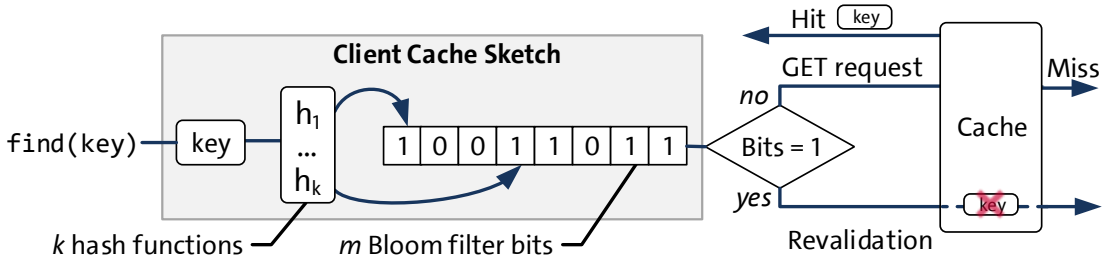


Figure 4.2: Database read using the Cache Sketch.

As shown in Figure 4.2, a read on a key is performed by querying the Bloom filter b_t of the client Cache Sketch c_t that was generated at time t. A Bloom filter represents a set $S = key_1, key_2, ..., key_n$ with n elements through a bit array of length m. The key is hashed using k independent uniformly distributed hash functions that map from the key domain to $[1,m]$, where m is the bit array size of b_t. If all bits $h_1(key), \ldots, h_k(key)$ equal 1, the object is contained and has to be considered stale. Thus, false positives can occur: if all positions of a key have been set to 1 by inserting other keys, it is wrongly recognized as being contained. Orestes abstracts from this by transparently performing the lookup for each client-side load and query operation and automatically refreshes the Cache Sketch according to the desired consistency level.

4.1.3 Proof of Δ-Atomicity

Theorem 4.1 deducts the central guarantee offered by the Cache Sketch using the time-based consistency property Δ-atomicity [GLS11] (cf. Chapter 2). Δ-atomic semantics assert that every value becomes visible during the first Δ time units after the acknowledgment of its write.

Theorem 4.1. *Let c_{t_3} be the client Cache Sketch generated at time t_3, containing the key $key(x)$ of every object x that was written before it expired in all caches, i.e., every x for which holds that $\exists r(x,t_1,TTL), w(x,t_2) : t_1 + TTL > t_3 > t_2 > t_1$. The read $r(x,t_1,TTL)$ is a cache miss on x at time t_1, where TTL is the time to live provided for that read and $w(x,t_2)$ is a write on x at time t_2 happening after the read.*

A read on object x performed at time t_4 using c_{t_3} satisfies Δ-atomicity with $\Delta = t_4 - t_3$, i.e., the read is guaranteed to see only objects that are at most Δ time units stale.

Proof. Consider there was a read issued at time t_4 which used the latest Cache Sketch c_{t_3} and that returned an object x that was stale for $\Delta > t_4 - t_3$ which would violate Δ-atomicity. This implies that x must have been written at a time $t_2 < t_3$ as otherwise $\Delta < t_4 - t_3$ (Δ-atomicity no violated). For x to be stale, there must have been a previous read $r(x,t_1,TTL)$ with $t_1 + TTL > t_4 > t_2$ so that x was still cached in at the time of the stale read. However, by the construction of c_{t_3} the object's key is contained in c_{t_3} until $t_1 + TTL$. Therefore, the read at $t_4 < t_1 + TTL$ using c_{t_3} could not have been stale (proof by contradiction). □

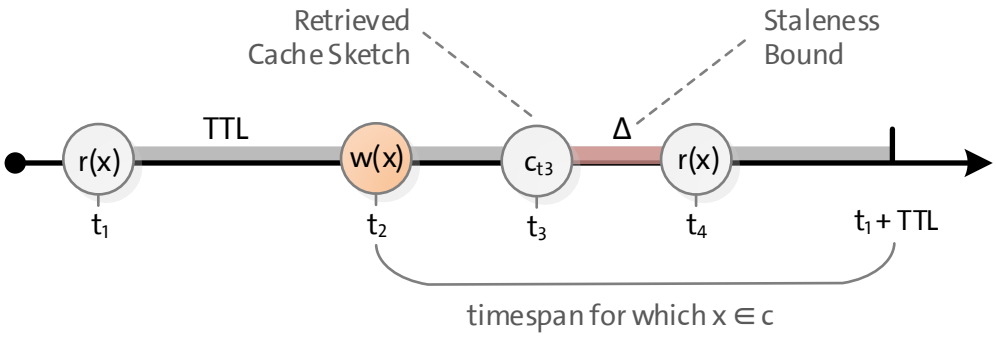


Figure 4.3: Illustration of the proof of Δ-atomicity for the Cache Sketch.

Theorem 4.1 states that clients can tune the desired degree of consistency by controlling the age Δ of the Cache Sketch: the age directly defines Δ-atomicity. The proof with its used variables is illustrated in Figure 4.3 that shows the different operations and timestamps. The guarantee relies on the linearizability of the underlying database system, i.e., writes are assumed to be directly visible to uncached reads. If the database is only eventually consistent with Δ_{db}-atomicity, the guarantee is weakened to $(\Delta + \Delta_{db})$-atomicity[1]. Similarly, if the invalidation-based caches only support asynchronous invalidations (which is typical for real-world CDNs [PB08]) with Δ_c-atomicity, the consistency guarantee becomes $(\Delta + \Delta_c)$-atomicity[2].

If Δ_c is either unbounded or an undesired source of uncertainty, Δ-atomicity can be established in two ways. First, invalidation-based caches can be treated as pure expiration-based caches by not letting them answer revalidation requests. The trade-off is that this increases read latency and the load on the database service. Second, invalidations can be performed synchronously. This is a good option for reverse proxy caches located in the net-

[1] Bailis et al. [BVF+12] have extensively studied the staleness of Dynamo-style systems. They found that with high probability Δ_{db} is very low (in the order of single-digit milliseconds) and for many configurations not perceivable at all.

[2] We are not aware of any scientific studies on CDN purge latencies. Anecdotally, the Fastly CDN used in our evaluations employs the bimodal multicast protocol for invalidations with measured latencies typically much lower than 200ms [Spa17].

work of the database service. Here, the trade-off is that cache misses have higher latency and can be blocked by the unavailability of a cache node (no partition tolerance).

4.1.4 Controlling Consistency

Definition 4.1 introduces three client-driven age-control techniques for the Cache Sketch. *Cached initialization* builds on the insight that initially $\Delta = 0$ for a Cache Sketch piggybacked upon connection. This implies that every cached object can be used without degrading consistency, i.e., loading the Cache Sketch is at least as fresh as loading all initially required objects in bulk, which may also include all static resources (images, scripts, etc.) of the application or website.

Definition 4.1. *A client follows* ***cached initialization****, if all initial reads are performed using a freshly loaded Cache Sketch c_t. A read at t_{now} follows* ***Δ-bounded staleness****, if it only uses c_t if $t_{now} < t + \Delta$. A* ***distributed cache-aware transaction*** *started at t_s uses c_{t_s} for transactional reads.*

Δ-bounded staleness guarantees Δ-atomicity by not letting the age of the Cache Sketch exceed Δ. Updates may be performed *eagerly* or *lazily*. With eager updates, the client updates c_t in intervals of Δ. As this may incur updates despite the absence of an actual workload, lazy updates only fetch a new Cache Sketch on demand. To this end, if a read request is issued at $t_{now} > t + \Delta$, the request is turned into a revalidation instructing the service to append the Cache Sketch to the result. Hence, at the mild cost of a cache miss at most every Δ time units, c_t is updated without additional requests.

Similar to cached initialization, a *distributed cache-aware transaction* (DCAT) is started by loading the Cache Sketch[3]. The caching model is only compatible with optimistic transactions as reads are performed in caches which cannot participate in a lock-based concurrency control scheme. By having clients collect the read sets of their transactions consisting of object IDs and version numbers, the database service can realize the transaction validation using BOCC+, as described in Section 4.8. The important alteration that DCAT brings to this scheme is that cached reads can drastically reduce the duration T of the transaction, while the Cache Sketch limits staleness during transaction execution. Since the abort probability of optimistic transactions quickly grows with T [Tho98], lowering T through cache hits can greatly reduce abort rates.

Figure 4.4 shows an end-to-end example of Cache Sketch usage, in which a client reads two different objects x_3 and x_2 first and then updates a third object x_1. First, the client fetches the Cache Sketch (step 1). Then, x_3 is loaded which is not contained in the Cache Sketch and therefore requested normally, resulting in a cache hit (step 2). Next, the client reads x_2 which is contained and hence a revalidation is sent, causing the expiration-based cache to evict its cached copy (step 3). The server returns x_2 with a new TTL/expiration

[3]Transactions could potentially start by reusing an already available Cache Sketch, however this increases stale reads that always lead to an abort.

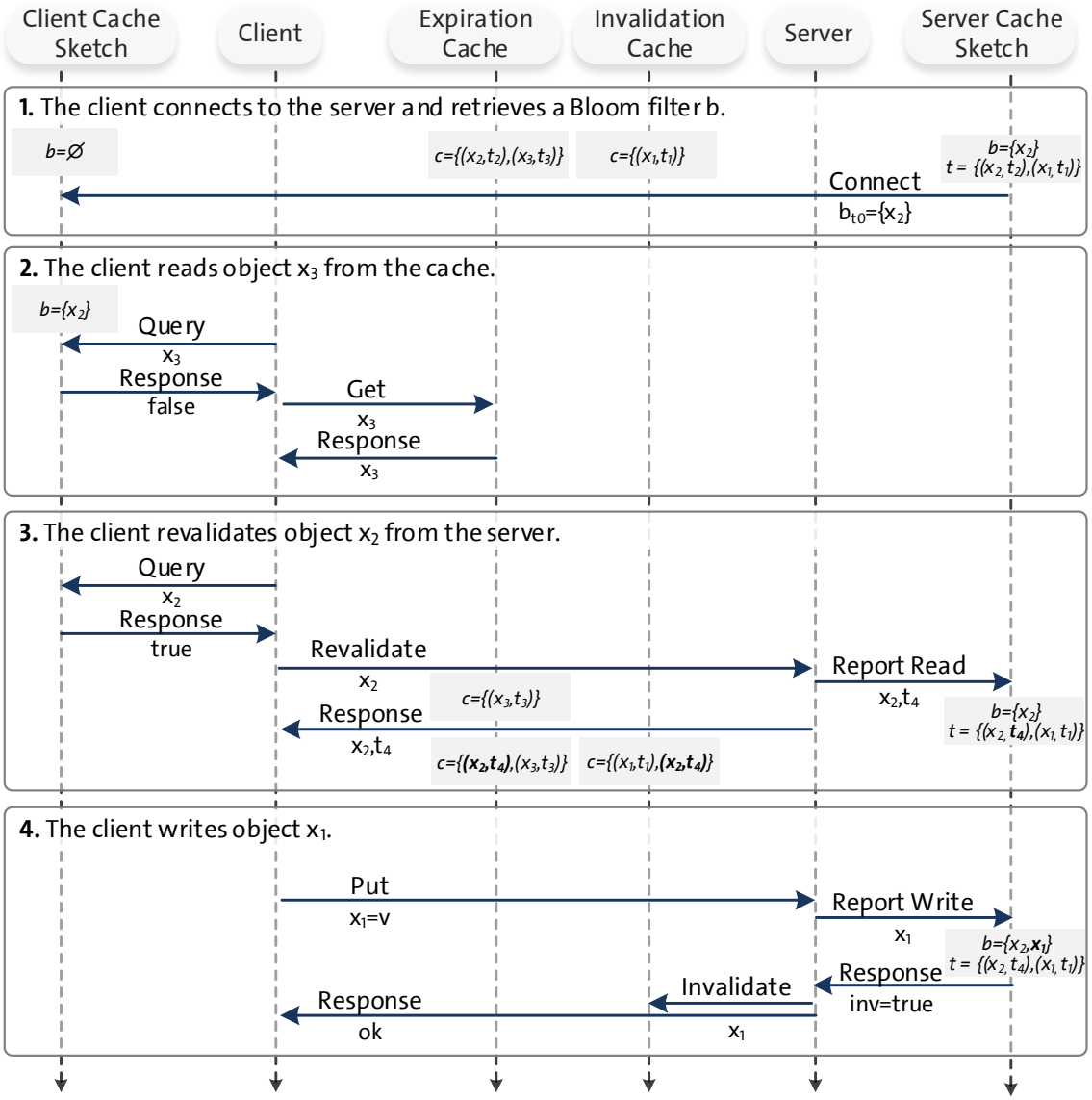


Figure 4.4: An end-to-end example of the proposed Cache Sketch methodology.

date t_4, which is saved in both invalidation-based and expiration-based caches. Additionally, the new expiration date is also reported to the server Cache Sketch, where expiration state is tracked. On the subsequent write on x_1, it is added to Counting Bloom filter of the server Cache Sketch, since its expiration date t_1 has not yet passed (step 3). This also tells the server to invalidate the object in invalidation-based caches. Any later readers are therefore able to revalidate x_1 from an invalidation-based cache.

As discrepancies between actual and estimated TTLs can cause extended periods for which objects are contained in the Cache Sketch and considered stale, clients perform a *differential whitelisting*: every object that has been revalidated since the last Cache Sketch update is added to a whitelist and considered fresh until the next Cache Sketch renewal[4]. While

[4]This assumes that all requests by a client pass through the same set of caches, which is true for most networks and TLS-encrypted connections.

this may incur stale reads, staleness of whitelisted objects is bounded to Δ and thus performance is improved without violating the overall consistency guarantee.

The achieved consistency levels are similar to those provided by related work (e.g., Pileus [TPK+13]), but we employ widespread web caches instead of custom replication sites and support both key-based access and queries. A detailed analysis of consistency guarantees provided by the Cache Sketch is given in Section 4.5, including both object and query caching.

4.1.5 The Server Cache Sketch

The purpose of the server Cache Sketch c_t^s specified in Definition 4.2 is the efficient and correct generation of a client Cache Sketch as defined in Theorem 4.1. This requires two important capabilities the client Cache Sketch lacks. First, the server Cache Sketch must support *removal* of keys in order to evict expired items. Second, it must support *invalidation queries* which report, whether a write has to be propagated as an invalidation.

Definition 4.2. *The server Cache Sketch c_t^s consists of a Counting Bloom filter cb_t containing all elements of c_t and a mapping of keys to their maximum expiration date $e = \{(k_i,t_i)|max_{t_i=t_r+TTL}(r(x,t_r,TTL) \wedge t_i > t_{now})\}$. When x is updated or deleted, k_x is added to cb_t iff $\exists t > t_{now} : (k_x,t) \in e$. Similarly, an invalidation is only necessary, if $\exists t > t_{now} : (k_x,t) \in e$.*

The employed Counting Bloom filter [FCAB00] is an extension of the Bloom filter that allows removals and can be implemented to materialize the corresponding normal Bloom filter, so retrievals of c_t do not require any computation. This is achieved by coupling the increment and decrement operations on counters to setting the respective bit in the materialized Bloom filter, if the counter is greater than 0. To make the retrieval of the Cache Sketch efficient, the size m of the Bloom filter must be chosen carefully. The false positive rate p is determined by the size m of the bit vector, the number of inserted elements n, and the number of hash functions k: $p \approx (1 - exp(-kn/m))^k$. The optimal number of hash functions is $k = \lceil ln(2) \cdot (n/m) \rceil$, giving the size as $m = -n \cdot ln(p)/ln(2)^2$.

The Orestes prototype supports multiple implementations of Bloom filters. Depending on whether the Cache Sketch is shared between Orestes servers, either an in-memory implementation or a distributed, persistent implementation based on the key-value store Redis is used (for details see Section 4.3.5)

4.1.6 Optimizing Cache Sketch Size

A simple model is to choose m such that transferring c_t only requires a single round-trip, even at connection startup. This is achieved, if the message size of m bits (and some HTTP metadata) measured in TCP segments of 1 460 bytes does not exceed the initial TCP congestion window size 10, i.e., $m \approx 10 \cdot 1460B = 11680B$ (cf. Chapter 2). For a false positive rate $p \leq 0.05$, the filter could hence contain up to $n \approx 18732$ distinct objects. If n

increased to 50 000, p would grow logarithmically to $p = 0.326$. If the Bloom filter is only transferred over an already established connection (e.g., after loading an HTML page), it can be significantly larger without incurring an additional round-trip[5]. Furthermore, HTTP responses are compressed with Gzip which reduces the size for sparse Bloom filters with many 0 bits, so that despite the static size of m, the transferred size is proportional to the number of stale objects.

The server Cache Sketch represents shared state between all server nodes and therefore has to scale with reads and writes alike. It is part of the critical request path as read, update, and delete operations require modifying it. **Read scalability** is achieved by replicating the complete Cache Sketch at Redis-level and balancing loads of the Bloom filter over the replicas[6]. **Write scalability** is reached through *partitioning*. Previously, we assumed a single c_t^s of every tenant's database. As a generalization, c_t^s can be partitioned and replicated based on buckets (resp. tables, collections, classes) by maintaining a separate c_t^s for each bucket. This solves two problems. First, updates to the Cache Sketch scale horizontally, mitigating potential write bottlenecks. Second, if an aggregate Cache Sketch for all tables is too large, clients can opt to fetch the Cache Sketch only for the required tables. To expose the aggregate Cache Sketch, the database service assembles the Cache Sketch by performing a union over the respective Bloom filters, which is a simple bitwise OR operation over their respective bit vectors [BM03]. Clients can also exploit the table-specific Cache Sketches to decrease the total false positive rate at the expense of loading more individual Cache Sketches.

4.1.7 Quantifying (Δ,p)-Atomicity for the Web Caching Model

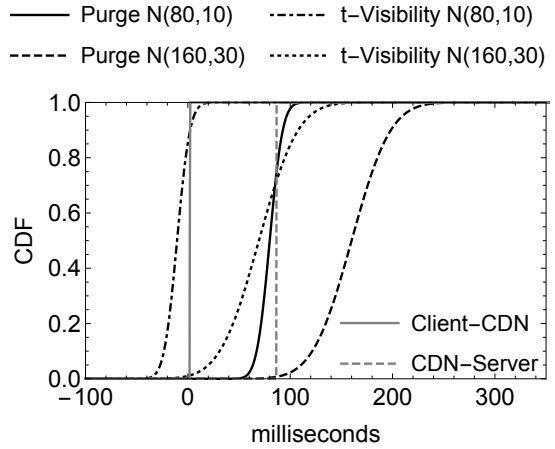
For consistent database systems, the Cache Sketch guarantees $(\Delta + \Delta_c)$-atomicity, where Δ_c is the upper bound for the staleness of objects read from invalidation-based caches. Δ_c largely overestimates staleness, since access is often local to geographic regions and seldom governed by worst-case delays. We therefore refine Δ_c to (Δ_c,p)-atomicity[7], which a read satisfies, if it is Δ_c-atomic with probability p [BVF+12]. The probability p for (Δ_c,p)-atomic semantics can be expressed through the round-trip latencies T_{cc} (client-cache), T_{sc} (server-cache) and T_i (invalidation). A revalidation or cache miss hitting an invalidation-based cache is Δ_c-atomic, if the time for the corresponding write acknowledgment to travel back to the client issuing the write plus the time for the read to reach the cache subtracted from the invalidation latency is smaller than Δ_c:

$$p = Pr[T_i - (T_{sc}/2 + T_{cc}/2 + T_{cc}/2) \leq \Delta_c] \tag{4.1}$$

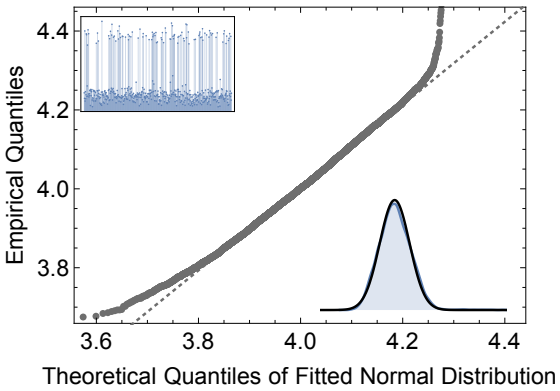
[5] This is an effect of the TCP slow-start algorithm which continuously increases the congestion window [WDM01].

[6] While it would be possible to fully replicate the Cache Sketch between Orestes servers through reliable broadcast or consensus protocols such as Raft or Paxos, there are two major downsides. First, horizontal scalability would be sacrificed – every server would have to process each read and write operation as a local Cache Sketch modification. Second, write latency would increase, since at least two round-trips between Orestes servers would be necessary for every Cache Sketch update.

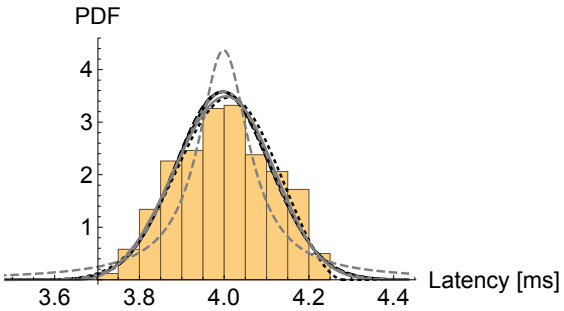
[7] (Δ,p)-atomicity is also referred to as t-Visibility [BVF+14].



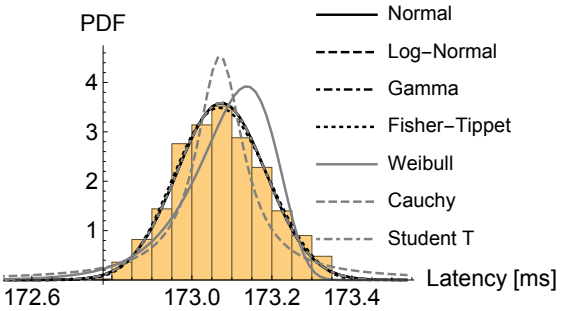
(a) (Δ_c,p)-atomicity in the web caching model: cumulative probability distribution for reading a value that was stale for at most Δ ms after a write acknowledgment.



(c) QQ-Plot for the normal fit of T_{cc}: except for outliers, latencies can be described by a normal distribution.



(b) Distribution of client-cache latency T_{cc}.



(d) Distribution of server-cache latency T_{sc}

Figure 4.5: Analysis of exemplary latencies and their effect on (Δ_c,p)-atomicity.

We gathered real-world latency traces to quantify (Δ_c,p)-atomicity and to feed our later simulations with realistic assumptions. Figure 4.5 summarizes our findings. The setup consisted of a client located in the Amazon EC2 California region, a server in EC2 Ireland, and the Fastly CDN as an example of an invalidation-based web caching system. We derived maximum-likelihood distribution fits for T_{cc} and T_{sc} for different continuous distribution families as shown in Figure 4.5b and Figure 4.5d, after applying the Tukey-outlier criterion [Tuk77] to account for measurement noise, such as the noisy-neighbor problem [GSHA11].

Though there is consensus in the networking literature that in the general case, network delays cannot be modeled using a single distribution [VM14], the normal and Gamma

	Normal	Log-Normal	Gamma	Fisher-Tippet	Weibull	Student T
Client-CDN	0.21	0.21	0.21	0.20	0.32	0.21
CDN-Server	0.68	0.68	0.68	0.94	0	0.68
Client-Server	0.94	0.94	0.94	0.52	0	0.94

Table 4.1: Cramér-von Mises p-values for maximum-likelihood fits of different latency distributions.

distribution yield good fits for the described setup as shown in Table 4.1 (goodness-of-fit p-values 0.21 and 0.68 with the Cramér-von Mises test [VdV00]). This is illustrated in the QQ-plot in Figure 4.5c, which shows that apart from the tails of the raw data (with outliers), the normal distribution describes T_{cc} accurately.

Based on this data, (Δ_c, p)-atomicity can be computed according to Equation 4.1 which is shown in Figure 4.5a with $T_{cc}/2 \sim N(2.00, 0.06)$ and $T_{sc}/2 \sim N(86.54, 0.06)$ for two T_i distributions. For $T_i \sim N(80, 10)$, which we found to be a good upper bound in our experiments, the probability of reading a fresh value starts high at 90% and quickly converges[8]. For caches located nearer to the server, the probability would converge even faster because of the lower physical latency.

In conclusion, with asynchronous invalidations that exhibit (Δ_c, p)-atomicity, the Cache Sketch guarantees $(\Delta + \Delta_c, p)$-atomicity. This allows precise reasoning about the latency-consistency trade-off for a given scenario of latency distributions and eases the decision on whether invalidations should be allowed to be asynchronous.

4.2 Cacheability Estimation: Whether and How Long to Cache

The TTLs for which caches are allowed to store objects significantly affect cache hits, stale reads, invalidations and false positives in the Cache Sketch. For instance, objects that experience a write-only workload but are cached with large TTLs will hurt performance, as each write will entail an unnecessary invalidation and fill up the Cache Sketch. Likewise, read-heavy objects will suffer from small cache hit ratios, when assigned TTLs are too small. The usefulness of the Cache Sketch depends on its false positive rate. Therefore, we introduce the concept of **TTL estimators** which try to minimize costs.

Definition 4.3. *A TTL estimator $E(id, \lambda_m^{id}, \lambda_w^{id}) \rightarrow TTL_{id}$ maps an object's historic cache miss rate λ_m^{id} and write rate λ_w^{id} to a TTL that minimizes the cost function:*

$$cost = w_1 \cdot \frac{\#cachemisses}{\#ops} + w_2 \cdot \frac{\#invalidations}{\#ops} + w_3 \cdot \frac{\#stalereads}{\#ops} + w_4 \cdot p \qquad (4.2)$$

The cost function is parameterized by weights w_i that express the relative severity of each condition: the number of caches misses, invalidations, stale reads, and the false positive rate of the Cache Sketch. For example, in a setup with a slow single server, many invalidation-based caches, and an application with low consistency requirements, w_1 and w_2 would be large to protect the server, while w_3 and w_4 would be smaller. We defined the miss and write rates as the default input of the estimator, though implementations could potentially require different input parameters, e.g., the global distribution of writes over objects in a bucket or the exact timestamps of past cache misses. TTL estimation subsumes the question of whether to cache at all: a TTL of 0 returned by the estimator indicates that the given object should not be cached (e.g., for objects that are mostly written).

[8] In the plot the negative millisecond range for t-visibility is shown, which indicates the probability of reading a new value before the respective write acknowledgment was received [BVF+12].

The estimator is invoked for cache misses to decide on the next TTL. As a baseline, we propose the *Static Estimator* $E_{static}(id) = TTL_{max}$ that always minimizes cache miss costs through a high static TTL. The trade-off is that every write on object x happening t seconds before the expiration causes an invalidation, opens the possibility of a stale read caused by the asynchronous invalidation and forces the cache sketch to contain x for the remaining t seconds, increasing its false positive rate. This implies that the static estimator should only be employed, if the Cache Sketch is large enough to provide a reasonable false positive rate for holding all objects that might be updated in a time window of TTL_{max}.

A straightforward improvement is thus obvious: instead of always estimating very large TTLs, TTLs should rather be correlated to the expected time to the next write on an object. Furthermore, TTLs should also be lower, when the workload is write-dominant. Similarly when reads dominate writes, the TTLs should be increased to prevent cache misses at the expense of some additional invalidations, stale reads, and objects in the Cache Sketch. This is the central intuition behind the improved algorithms proposed next.

4.2.1 Stochastic Model

To make the improved TTL estimation feasible, some assumptions have to be made. First, we assume, that the per-object workloads are readily available to estimators in the form of cache-miss rates λ_m^{id} and write rates λ_w^{id}. Second, to estimate the probability of writes in certain time intervals, a continuous-time stochastic process of writes $\{W(t), t \in T\}$ is assumed where the random variables $W(t)$ model the amount of writes seen until time t. Intuitively, given that exactly one write happened in the interval $[0,t]$, the time of occurrence should be uniformly distributed over $[0,t]$. This requirement is met by the *Poisson process*, which is the most commonly used stochastic process for modeling arrival processes [VM14]. It is characterized by increments that follow a Poisson distribution:

$$Pr[W(t+s) - W(s) = k] = (\lambda_w t)^k / k! e^{-\lambda_w t} \tag{4.3}$$

The write rate is λ_w, i.e., the expected amount of writes in a time interval of length t is $E[W(t)] = \lambda_w t$.

A central property for our TTL estimation problem is that inter-arrival times between writes T_i are **exponentially distributed** with mean $1/\lambda_w$, i.e., $Pr[T_i < TTL] = 1 - e^{-\lambda_w TTL}$. This implies that knowing an object's write rate is sufficient information to derive the expected time of the next write $E[T_i] = 1/\lambda_w$ and the quantiles $Q(p,\lambda_w) = -ln(1-p)/\lambda_w$. As the stochastic process of reads is unobservable (hidden through caches), we specifically do neither require knowledge about the *workload mix* (i.e., object-specific read-write ratios) nor the *popularity distribution* (i.e., the underlying distribution of object access frequencies). Instead, the TTL estimator implicitly adapts to these conditions.

In principle, the TTL estimator can be employed in two different modes. In general, the TTL estimator can be called for each cache miss in order to estimate a new TTL. This

has the advantage that the newest read rate is always considered and it accounts for the memorylessness [MU05] of the inter-arrival times distribution. However, it requires frequent recomputations. Instead, TTLs can be computed once for the first read after an invalidation and afterwards be read from the expiration mapping of the server Cache Sketch where they are stored anyway.

4.2.2 Constrained Adaptive TTL Estimation

The goal of the **Constrained Adaptive TTL Estimator (CATE)** is to minimize the cost function (see Equation 4.3), while constraining the size of the Cache Sketch to meet a good false positive rate. To this end, CATE adjusts TTLs to the cache miss rate λ_r and write rate λ_w instead of merely estimating the time to the next write. The estimation approach is illustrated in Figure 4.6a: write and cache miss metrics are aggregated in the server and fed into the estimator for each cache miss to retrieve a new TTL. The algorithm is based on four design choices:

1. *Read-only* objects yield TTL_{max} and *write-only* objects are not cached.
2. If the miss rate λ_m approximately equals the write rate λ_w, the object should be cached for its expected lifetime expressed by the *interarrival time median* of writes $Q(0.5,\lambda_w)$, i.e., the TTL is chosen so that the probability of a write before expiration is 50%.
3. A *ratio function* $f\colon \mathbb{R} \to [0,1]$ expresses how the miss-write ratio impacts the estimated TTLs. It maps the imbalance between misses and writes to p_{target} which gives the TTL as the quantile $Q(p_{target},\lambda_w)$. If for instance misses dominate writes, $p = 0.9$ would allow a 90% chance of a write before expiration, in order to increase cache hits. Using quantiles over TTLs for the ratio function has two advantages. First, the probability of a write happening before the expiration is easier to interpret than an abstract TTL. Second, the quantile scales with the write rate. The ratio function and its parameters can be tuned to reflect the weights in the cost function.
4. *Constraints* on the false positive rate of the Cache Sketch and the number of invalidations per time period are satisfied by lowering TTLs.

Algorithm 1 describes CATE. The ESTIMATE procedure is invoked for each cache miss. It requires three constants: the maximum TTL TTL_{max}, the ratio function f, and the $slope$ which defines how strongly f translates the imbalance between misses and writes into smaller or greater TTLs.

First, the miss-write *imbalance* is calculated (line 4). We define it to be 0 if $\lambda_m = \lambda_w$, x if λ_m is x times greater than λ_w and $-x$ if λ_w is x times greater than λ_m (line 5). Next, the ratio function maps the imbalance to the allowed probability p_{target} of a write (and invalidation) before the expiration date. p_{target} is capped at $p_{max} = Pr[T_i < TTL_{max}]$, so that the estimated TTL never gets larger than TTL_{max}. We consider three types of ratio functions shown in

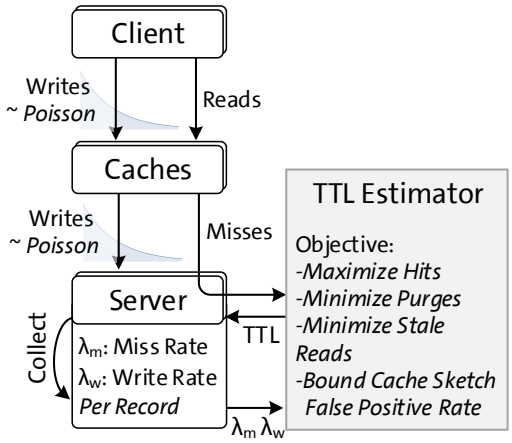
Algorithm 1 Constrained Adaptive TTL Estimation (CATE)

1: **procedure** ESTIMATE(λ_m : miss rate, λ_w : write rate)$\rightarrow TTL$
2: **constants:** $TTL_{max}, slope$, f : ratio function
3: **if** $\lambda_w = NIL$ **then return** TTL_{max}
4: $$imbalance = \begin{cases} \lambda_m/\lambda_w - 1 & \text{if } \lambda_m \geq \lambda_w \\ -(\lambda_r/\lambda_w - 1) & \text{else} \end{cases}$$
5: $p_{max} \leftarrow Pr[T_i < TTL_{max}] = (1 - e^{-\lambda_w TTL_{max}})$
6: **if** f **is** linear **then** $p_{target} \leftarrow 0.5 + slope \cdot imbalance$
7: **else if** f **is** logistic **then** $p_{target} \leftarrow p_{max}/(2p_{max} \cdot e^{-slope \cdot imbalance})$
8: **else if** f **is** unweighted **then** $p_{target} \leftarrow \lambda_m/(\lambda_m + \lambda_w)$
9: **if** Cache Sketch capacity exceeded **then**
10: Decrease p_{target} by a penalty proportional to false positive rate
11: **if** Invalidation budget exceeded **then**
12: Decrease p_{target}
13: $$TTL = \begin{cases} 0 & \text{if } p_{target} \leq 0 \\ TTL_{max} & \text{if } p_{target} \geq p_{max} \\ Q(p_{target}, \lambda_w) & \text{else} \end{cases}$$
14: **return** TTL

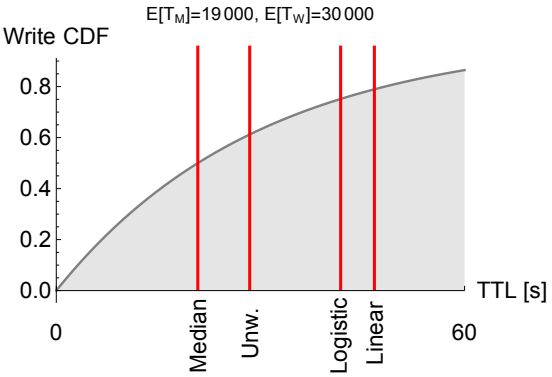
Figure 4.6c: a *linear* and a *logistic* function of the imbalance, as well as the *unweighted* fraction of misses in all operations (lines 6 to 8).

In order not to overfill the Cache Sketch, its current false positive rate is considered. If it exceeds a defined threshold, p_{target} is decreased to trade invalidations on non-expired objects against revalidations on expired objects (lines 9 to 10). By lowering the probability of writes on non-expired objects, Cache Sketch additions decrease, too. Invalidations are treated similarly: if the budget of allowed invalidations is exceeded, p_{target} is decreased (lines 11 to 12). In this way, Cache Sketch additions and invalidations are effectively rate-limited. The optimal amount to decrement depends on the severity a of a violation and can be computed as $p_{target} = p_{target} \cdot (1 - f)^a$, where f is the degree of violation, for example the difference between the allowed and actual false positive rate. Last, the TTL derived as the quantile $Q(p, \lambda_w)$ is returned (lines 13 to 14).

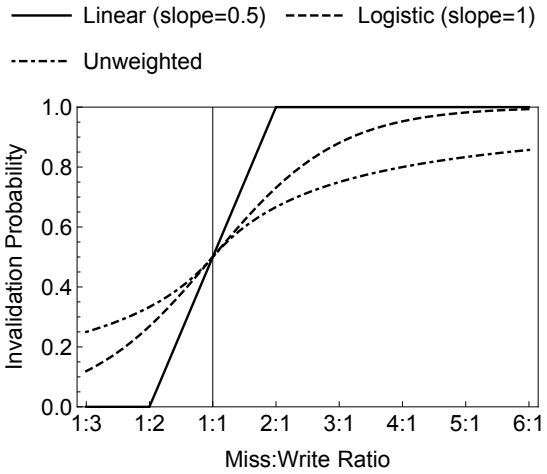
Figure 4.6b gives an example of estimated TTLs for a read-heavy scenario, as well as the corresponding probability $Pr[T_i < TTL]$ of a write before expiration. By construction, all three ratio functions yield a TTL that is higher than the median time between two writes in order to drive cache misses down. The magnitude of this TTL correction is determined by the ratio function and its slope. This makes it obvious that minimizing the cost function requires tuning of the ratio function in order to meet the relative weights between misses, invalidations, stale reads, and false positives. As finding the right TTL_{max} and $slope$ in a running system is a cumbersome, manual, and error-prone process, we introduce a framework in Section 4 that chooses parameters using Monte Carlo simulations to find the best solution under a given workload and error function.



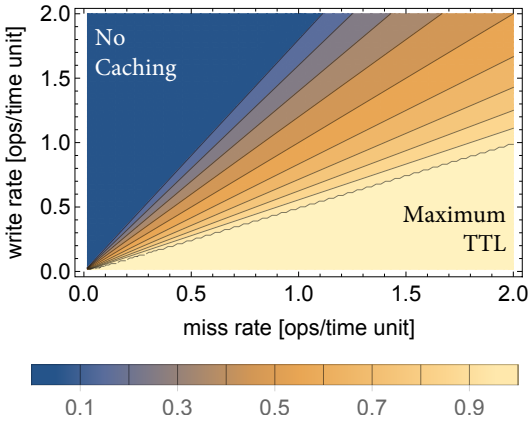
(a) The TTL estimation process.



(b) TTL estimations for an example workload.



(c) Ratio functions.



(d) p_{target} contour plot for linear ratio function.

Figure 4.6: Constrained Adaptive TTL Estimation.

Figure 4.6d shows the effect of different miss and write rates as a contour plot of the linear ratio function. In the upper left area, writes clearly dominate misses, so the estimator opts to not cache the object at all – frequent invalidations would clearly outweigh seldom cache hits. In the bottom right area, on the other hand, misses dominate writes, so the object is cached for TTL_{max}. The area in between gradually shifts to higher TTLs (values of p_{target}), with the steepness of the ascent varying with the slope.

As explained above, estimating TTLs requires each Orestes server to have approximations of **write** and **miss rates** for each object. To this end, inter-arrival times are monitored and averaged over a time window using a simple moving average (SMA) or exponentially-weighted moving average (EWMA). The space requirements of the SMA are high, as the latest arrival times for each object have to be tracked, whereas the EWMA only requires a single value. Similarly, a cumulative moving average (CMA) requires little space, but weighs older inter-arrival times as heavily as newer ones. While this assumption is optimal for Poisson processes, it fails for non-stationary workloads, e.g., when the popularity of objects decreases over time. To address the overall space requirements, sampling can be applied. More specifically, exponentially-biased reservoir sampling is an appropriate

stream sampling method that prefers newly observed values over older ones [Agg06]. The reservoir is a fixed-size stream sample, i.e., a map of object IDs to their write and miss moving averages. In the Orestes approach of load-balanced middleware service nodes, every server already sees an unbiased sample of operations, whereas in the case that Cache Sketch maintenance is co-located with each partitioned database node, only local objects would have to be tracked, lowering the space requirements.

4.2.3 TTL Estimation for Fluctuating Workloads

The CATE algorithm is based on the assumption of a pure Poisson process. In particular it relies on static inter-arrival times, i.e., a stationary stochastic process. However, many real-world workloads exhibit **seasonality**, e.g., shopping sites experiencing heavier traffic before holidays. Furthermore, the slope of the read/write imbalance and the quantile are two hyperparameters that are difficult to tune.

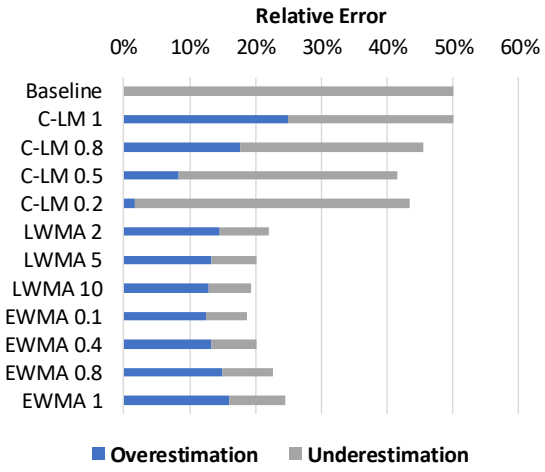
Therefore, we propose a few simple alternative TTL estimators for objects and files that only rely on an object's *CreatedAt* (*C*) and *LastModified* (*LM*) timestamps[9], as well as the n latest write timestamps $t_m, t_{m-1}, \ldots, t_{m-n}$:

C-LM Model. We introduce a modification of the Alex protocol [GS96] for object and file TTL estimation. The idea is to predict a TTL as the weighted time between the point in time of the estimation and the last modification or the creation: $\alpha \cdot (t_{now} - max(C_{id}, LM_{id}))$. Like the other estimations, the result is normalized to the range $[0, TTL_{max}]$ as $E_{norm}(id) = max(0, min(TTL_{max}, E(id)))$. The intuition of the C-LM model is that inter-arrival times vary heavily over time and therefore the best indicator for the write frequency of an object is its latest modification. The C-LM model assumes a distribution that is not memoryless, as – in contrast to the Poisson model of CATE – estimates increase with elapsed time.

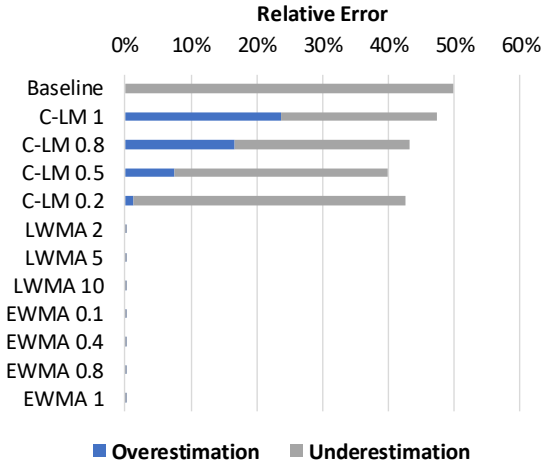
LWMA Estimator. To capture the near past of an object in its TTL estimate, the linearly-weighted moving average (LWMA) estimator considers the last n writes with linearly decreasing weights: $E(id) = \frac{2}{n(n+1)} \sum_{i=1}^{n} i \cdot t_{m-n+i}$. For a small n, e.g., 1, 3, or 5, the space overhead is negligible and only the most recent changes of inter-arrival times are reflected. For the first invocations, a default TTL has to be assumed until sufficient writes have been observed. Without the assumption of a memoryless inter-arrival time distribution, the estimate has to be shifted to $E_{shifted} = max(0, E(id) - (t_{now} - t_m))$ to account for the passed time since the last write.

EWMA Estimator. The exponentially-weighted moving average (EWMA) estimator is similar to the LWMA, but requires less space as only the last estimate and the last inter-arrival time are used: $E_m(id) = \alpha \cdot (t_m - t_{m-1}) + (1 - \alpha) \cdot E_{m-1}(id)$. The relevance of past inter-arrival times decreases linearly and, like for the C-LM model, an appropriate $\alpha \in [0,1]$ parameter has to be set.

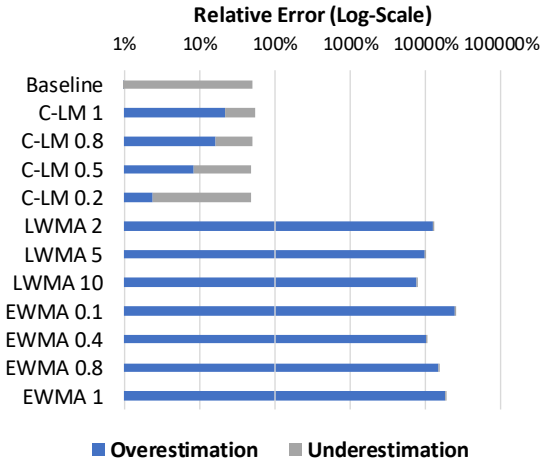
[9] Both fields are automatically maintained by Orestes for each object and file.



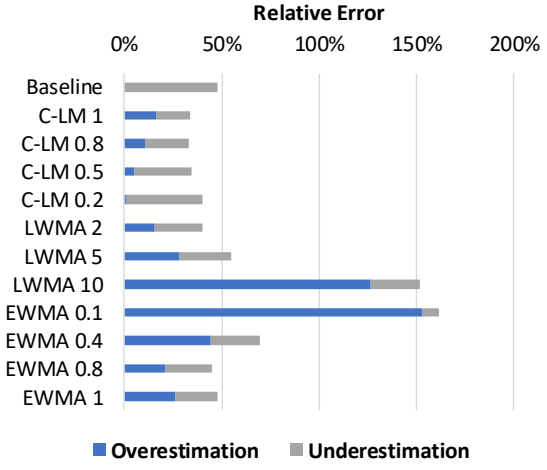
(a) Poisson process.



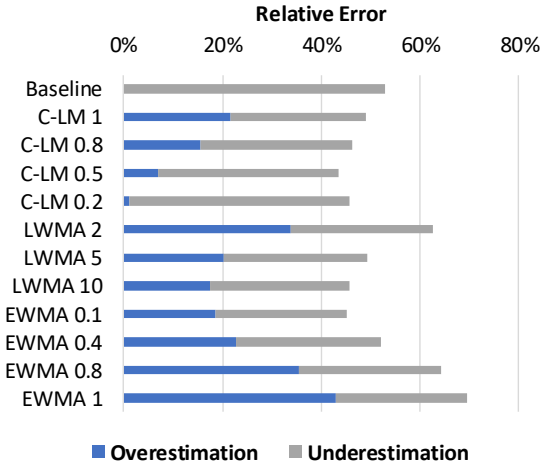
(d) Static inter-arrival times (fixed-rate writes).



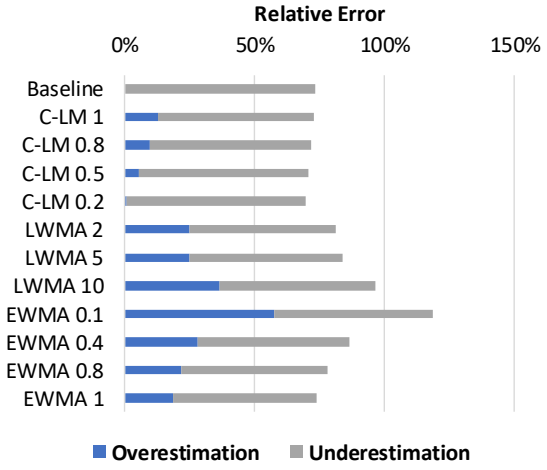
(b) Writes of Orestes production application.



(e) Low-attention Wikipedia article (“BaaS”).



(c) Medium-attention article (“Hamburg”).



(f) High-attention article (“Donald Trump”).

Figure 4.7: Prediction errors of TTL estimators for different workloads.

We evaluated the above estimators for several workloads to compare their feasibility given real-world inter-arrival times. To this end, we compared the estimators to a static baseline estimator that always predicts a TTL of 0 and measures the relative overestimation and underestimation for the actual TTL (the normalized difference between the estimated expiration and the time of the next write) for the respective workload. The evaluated workloads are a Poisson process, fixed inter-arrival times, an object-access workload from an Orestes production application, and the inter-arrival times of changes for three different Wikipedia articles of different attention levels ("Backend-as-a-Service", "Hamburg", and "Donald Trump").

The results are shown in Figure 4.7. The estimators were evaluated in different parameterizations as indicated by the number after the name. The C-LM model was tested for $\alpha \in \{0.2,0.5,0.8,1\}$, the LWMA estimator for $n \in \{2,5,10\}$ and the EWMA estimator for $\alpha \in \{0.1,0.4,0.8,1\}$.

As expected, the EMWA and LWMA estimators that both converge to average inter-arrival times perform best for the Poisson process (see Figure 4.7a) especially when past measurements outweigh newer ones (lower α, resp. higher n). The C-LM model, however, is biased towards underestimation for the Poisson process. Therefore, if the cost function favors fewer invalidations and a low false positive rate over a high cache hit rate, the C-LM model is preferable, otherwise the LWMA and EWMA estimators are. For a fixed write rate, the LWMA and EWMA estimators quickly adopt the correct inter-arrival time and yield the best result (see Figure 4.7d). In case of the Orestes production application, writes follow a mixed distribution with long periods of inactivity mixed with sudden activity. This leads to massive overestimation by the LWMA and EWMA estimators, whereas the distribution-agnostic C-LM model yields good results (see Figure 4.7b). The same is true for the three Wikipedia articles (see Figure 4.7e,4.7c,4.7f): as past write rates are apparently not a good indicator for the next editorial content updates, the C-LM model achieves the lowest overestimation and total error.

In summary, if no single inter-arrival time distribution can be assumed, the C-LM model achieves very good results, while allowing to tune the degree of overestimation against underestimation through the α parameter. It also requires no additional space, as required information is attached to each object. Another advantage of the C-LM model is that it continues to work well in multi-server deployments where individual servers estimate TTLs independently, as it does not require to observe each write operation. As another distinction to the other estimators, the C-LM model can be used without prior workload analysis and is therefore the default choice in Orestes for unknown workloads. The C-LM model can also be combined with the false positive rate and invalidation budgeting of CATE to constrain the number of false positives and invalidations.

4.3 Evaluation of the Cache Sketch for Object Caching

In this section we will evaluate the Cache Sketch through both simulation and cloud-based experiments. This combination is necessary to study the hidden system parameters (e.g., cache hit rates), to tune hyperparameters, as well as to measure real-world performance of all involved components.

4.3.1 YMCA: An Extensible Simulation Framework for Staleness Analysis

We have implemented a Yahoo! Cloud Serving Benchmark **(YCSB) wrapper for Monte Carlo simulation (*YMCA*)** with arbitrary caching architectures which runs completely in memory. YCSB [CST+10] is a widely-adopted standard benchmark for CRUD data stores. As shown in Figure 4.8, YMCA consists of a client that implements the YCSB interface for basic CRUD operations, an arbitrary number of cache layers and additional modules for collecting metrics, in particular stale reads, cache misses, and invalidations. Cache layers are stacked onto each other and can model any caching topology (e.g., a CDN or a reverse proxy). Latencies between layers are drawn from pluggable distributions, assuming symmetric latencies.

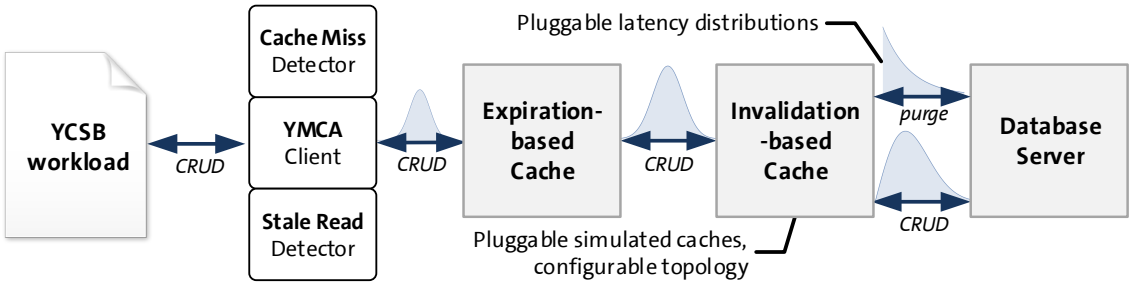


Figure 4.8: Concept of the extensible Monte Carlo simulation framework YMCA.

Overall, YMCA provides a toolbox to analyze caching behavior of multi-layered database infrastructures. The YMCA client tracks and reports stale reads. A read is considered stale, if two conditions are met: 1) there was an acknowledged write with a newer version than the one that was read 2) the read started after the write was acknowledged. Cached data is stored in simulated cache nodes throughout the simulation. The database is maintained as a key-value store. Apart from stale reads, invalidations, latencies, and throughput, YMCA also keeps track of cache hits and misses reported by each cache. In order to simulate long durations, YMCA implements a time scaling mechanism: all latencies and TTL estimations can be scaled by a defined factor. In the following, we assume the setting from Subsection 4.1.7 that includes an infrastructure consisting of a client, a CDN, and the database service. The database service employs the server Cache Sketch to decide whether an update requires an invalidation and then passes cache misses to the TTL estimator to assign the object-specific TTL.

4.3.2 Parameter Optimization for the CATE TTL Estimator

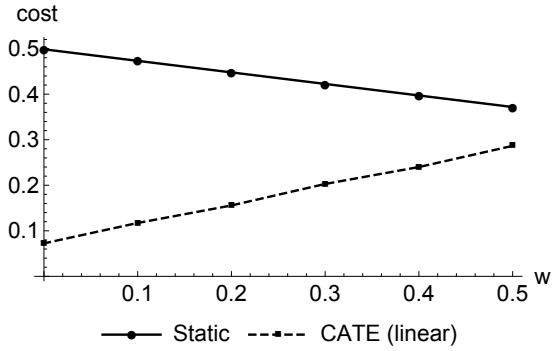
As discussed above, adaptive TTL estimation depends on the *slope* of the ratio function as well as TTL_{max}. Naturally, if the maximum TTL is much longer than the actual simulation, results will show a high cache hit rate and a lot of invalidations compared to a relatively short maximum TTL. Hence, the maximum TTL is set to the duration of the simulation and all other TTLs are estimated in relation to this maximum. In order to optimize these parameters, we use a variation of *maximum descent hill climbing*. Initial slopes of the ratio function are drawn uniformly at random in the $[0,1]$ range.

The **optimization algorithm** tests whether increasing or decreasing the slope provides an improvement of the simplified cost function $(w \cdot \#cachemisses + (1-w) \cdot \#invalidations)/\#ops$ that is to be minimized. The score is calculated as the sum of cache misses and invalidations normalized by the total number of operations. It thus is a simplified version of the score introduced in the model. This simplification is reasonable, because stale reads show a lot of variance (i.e., depending on indeterministic thread-scheduling) while having minimal impacts on the score, thus presenting more of a noise to the optimization. Since the number of invalidations is an approximate indicator of stale reads as well as a measure of the Bloom filter population, we have found the simplified cost function to be a well-working simplification of the original cost function. Depending on the cost of cache misses compared to invalidations (and the subsumed false positive and stale read rates), terms are weighted with $w \in [0,1]$.

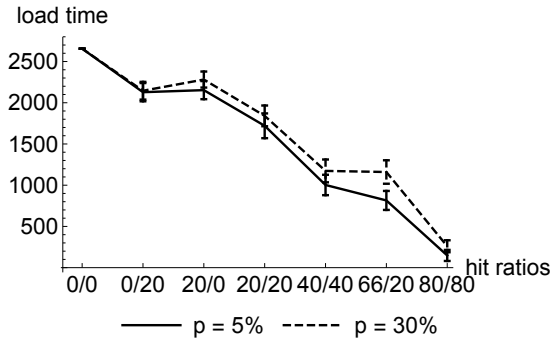
Testing directions of TTL_{max} and *slope* constitutes a super-step, which concludes by persisting the direction of maximum change (towards a lower cost) for the next super-step to start with. The algorithm terminates after a given number of super-steps or when it cannot improve the cost function.

Optimizations were performed for YCSB workloads A [CST+10, Section 4] (write-heavy; read/write balance 50%/50%) and B (read-heavy; read/write balance 95%/5%) with a Zipfian popularity distribution. Each simulation step was run on 100 000 operations for 10 threads, with a targeted throughput of 200 ops/s and a time scaling factor of 50, on the default amount of 1 000 objects. We ran the hill climbing algorithm from 25 starting points. Figure 4.9a and 4.9c show the resulting costs as a function of w for the optimized parameters of CATE, with a linear ratio function compared to the static TTL estimation with a high TTL_{max}. The results demonstrate that CATE performs significantly better than static estimation for applications that do prefer high cache hit rates (workload A). Unsurprisingly, read-heavy workloads leading to many cache hits perform slightly better with a static (maximum) TTL estimation (unless cache misses are weighted very low).

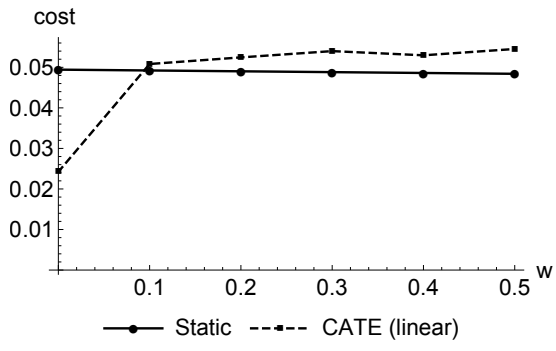
As page load time is arguably the most important web performance metric, we analyzed the gains of *cached initialization* for different browser cache and CDN cache hit rates and two Cache Sketch false positive rates (5% and 30%). The analysis assumes an average web page with 90 resources using 6 connections [GBR14] and that the Cache Sketch is used for every resource. The results shown in Figure 4.9b are as drastic as expected: for



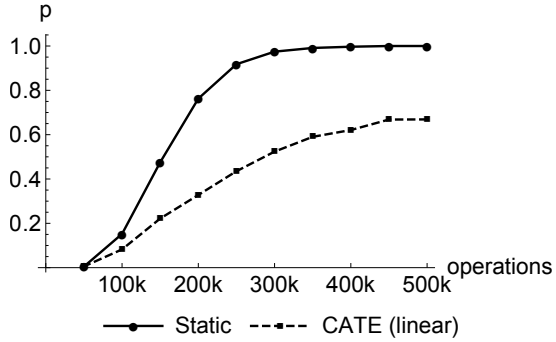
(a) Costs for Workload A.



(b) Page load time improvement through the cached initialization model. Labels on the x-axis indicate the percentual cache hit rate in the browser (first number) and CDN (second number).



(c) Costs for Workload B.



(d) False positive rate for a n=1k, p=0.05 Cache Sketch under Zipf-distributed operations.

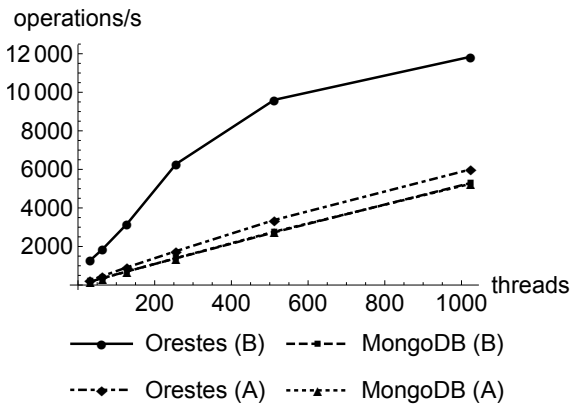
Figure 4.9: YMCA simulation results.

instance, for a cache hit rate of 66% in the browser and 20% in the CDN as described for the Facebook photos workload [HBvR+13], the speedup is over 320% for $p = 0.05$.

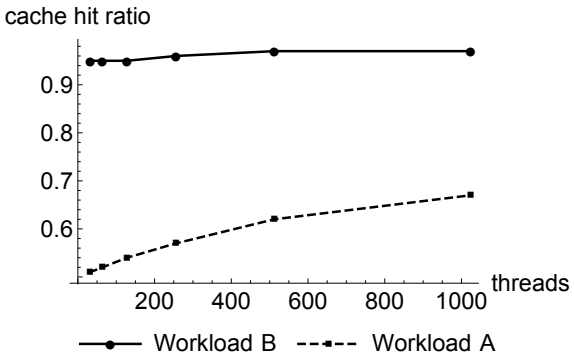
The development of the Cache Sketch false positive rate is shown in Figure 4.9d for 100 000 objects, workload B, a slope optimized for 100 000 operations, and the Bloom filter configured to contain 1 000 elements with $p = 0.05$. As expected, CATE achieves lower false positive rates by decreasing TTLs, when p grows too large. Even though the Cache Sketch is provisioned to only hold 1% of all objects, the static estimator performs surprisingly well, as long as the number of operations is smaller than the number of total objects.

4.3.3 YCSB Results for CDN-Cached Database Workloads

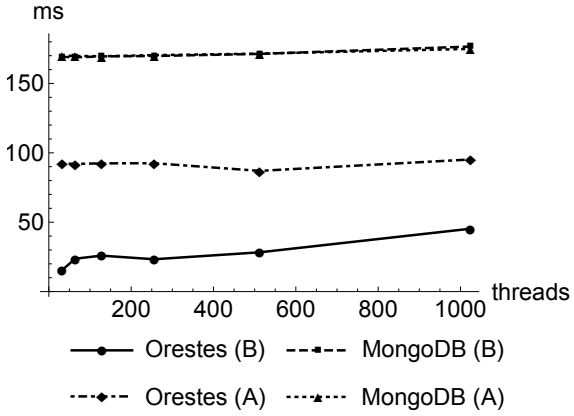
To validate the results in an experimental setup, we conducted the **YCSB benchmark** for the described setup on Amazon EC2, using `c3.8xlarge` instances for the client (*northern California region*) and server (*Ireland*), while caching in the Fastly CDN [Spa17]. We employed the document store MongoDB as a baseline for classic database communication. It was compared to an Orestes server running with MongoDB to add the Cache Sketch and the REST API. The benchmark was performed with the same configuration as the



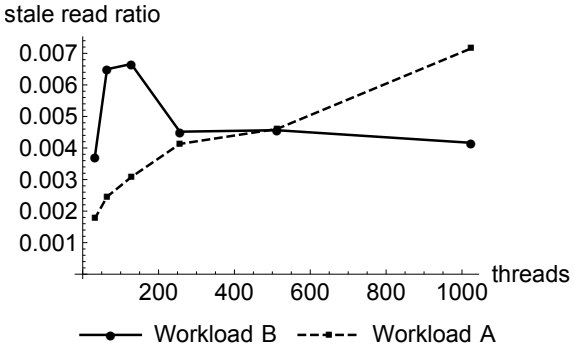
(a) Throughput for a single client.



(b) Cache hit ratios.



(c) Latencies of read operations.



(d) Stale read ratios.

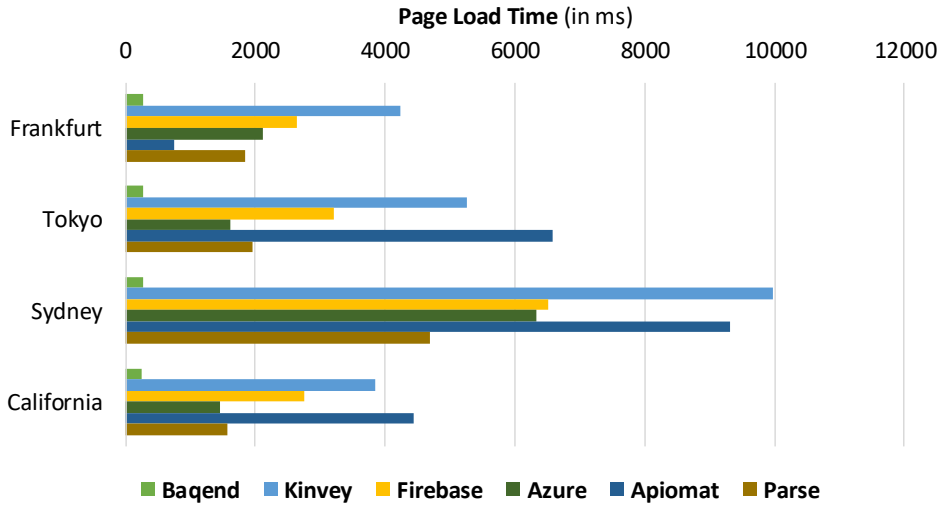
Figure 4.10: Performance and consistency metrics for YCSB with CDN-caching for two different workloads (A and B).

simulation, but using the static TTL estimator. Figure 4.10 shows latency, throughput, cache hit ratios, and stale reads for 32 to 1 024 threads (i.e., YCSB clients).

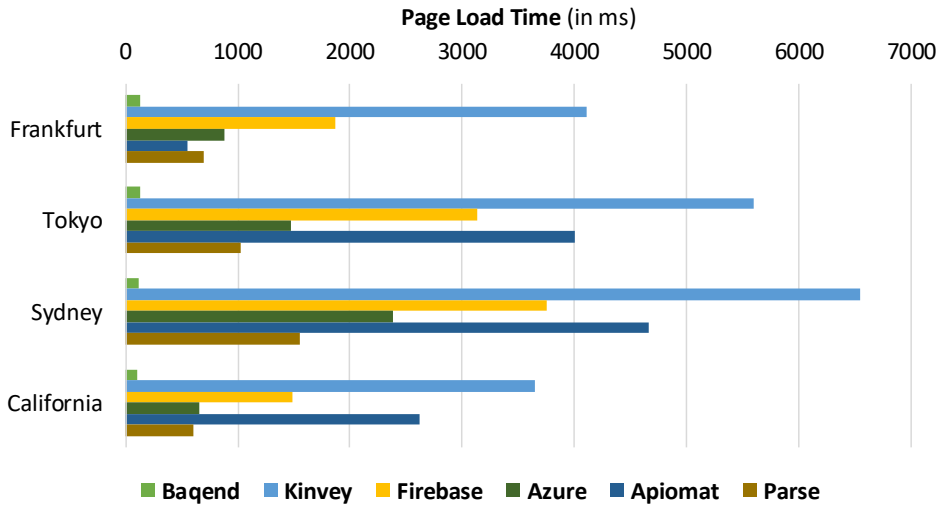
The results reveal the expected behavior: latency and throughput are improved considerably in both workloads, although a slight non-linearity between 512 and 1 024 threads occurs because of thread scheduling overhead of the limited single-machine design of YCSB. MongoDB achieves the same latency and throughput in both workloads, since all operations are bounded by network latency. The very few stale reads show considerable variance and were largely independent from the number of threads, as seen in Figure 4.10d. This fact supports our argument that (Δ_c, p)-atomicity is an appropriate consistency measure and that CDNs are well-suited to answer Cache Sketch-triggered revalidations.

4.3.4 Industry Backend-as-a-Service Evaluation

For a practical performance comparison of BaaS systems, we developed a simple benchmark to test the end-to-end latency of a serverless web application. Here we provide a short summary of the performance of Baqend, the commercial service offering of Orestes. The comparison between our approach and several popular commercial BaaS providers



(a) Performance with a cold browser cache and a warm CDN (first load).



(b) Performance with a warm browser cache and a warm CDN (second load).

Figure 4.11: Page load time comparison for different industry Backend-as-a-Service providers for the same data-driven web application.

(Kinvey, Firebase, Azure Mobile Services, Apiomat, Parse) is open-source and can be validated in a web browser[10].

The benchmark uses the example of a simple news website loaded from different geographical locations (Frankfurt, Tokyo, Sydney, California) with a cold browser cache and a warm CDN cache. Both the web application files (HTML, CSS, and JavaScript) and the actual database objects are fetched from the respective BaaS. The data model is structured by news stream objects that reference individual news. Each news item contains text fields like the title and teaser, as well as the reference to an image object. The data model is adapted to each of the providers abstractions, but conceptually identical. Data is rendered

[10]The benchmark can be run at `http://benchmark.baqend.com/` and the source code of all implementations is published on GitHub [Baq18]

through JavaScript that uses the respective SDKs to fetch the data of 30 news articles with images.

The results are shown in Figure 4.11a. For each location (measured from AWS data centers and a Chrome browser) the page load time (`load` event) is plotted as an average over ten consecutive runs with cold connections and a cold browser cache (first visit). As reads from Baqend rely on invalidation-based CDN caching, the average performance advantage is factor 15.4. As the Cache Sketch becomes effective with expiration-based caching, we also measured the second load. This measurement corresponds to the loading time for a returning visitor with a warm browser cache (see Figure 4.11b). Some of the other providers cache data in the browser, however without the ability to maintain consistency. Nonetheless, the performance advantage of the Cache Sketch is even greater: on second load, the other providers are outperformed by factor of 21.1 on average. This shows that the predicted performance improvements of our proposed caching approach translate into the setting of practical websites and web applications through file and object caching.

4.3.5 Efficient Bloom Filter Maintenance

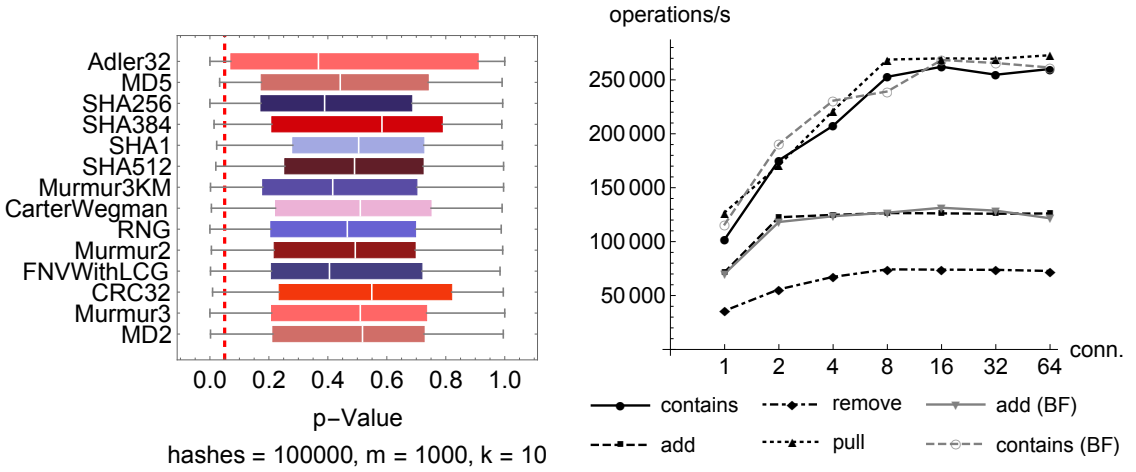
The server Cache Sketch requires an efficient underlying Counting Bloom filter. For this purpose, we developed a Bloom filter framework available as an already widely-used open-source project[11]. It supports normal and Counting Bloom filters as in-memory data structures as well as shared filters backed by the in-memory key-value store Redis [San17]. The library supports the table-based sharding and replication introduced in Section 4.1.6 for high-throughput workloads. The Redis Bloom filter uses the data structures of Redis to maintain an efficient bit vector for the materialized Bloom filter and relies on pipelining and batch transactions to ensure performance and consistency. We chose Redis because of its tunable persistence complemented with very low latency [Car13].

Figure 4.12 shows selected performance characteristics of the Cache Sketch under different hash functions and operations. The uniformity of implemented hash functions for random Strings is evaluated in Figure 4.12a using the p-values for 100 χ^2-goodness-of-fit tests. For random inputs (e.g., UUID object keys) all hash functions perform reasonably well – including simple checksums. However, for keys exhibiting structure, the best trade-off between speed of computation and uniformity is reached by Murmur 3.

Figure 4.12b plots the throughput of the unpartitioned, non-replicated Redis Bloom filters for a growing amount of connections with $m = 100\,000$ and $k = 5$ on an a server with 16 GB RAM and a CPU with four cores and 3 GHz. Read operations (querying, pulling the complete filter) achieve roughly 250 000 ops/s, while write operations (adding, removing) that require some overhead for counter maintenance and concurrency still achieve over 50 000 ops/s resp. 100 000 ops/s.

We also report the performance of Cache Sketch operations on the DBaaS variant of Redis (AWS ElastiCache) for two different instance sizes (see Table 4.2). Each operation

[11] Available at `https://github.com/Baqend/Orestes-Bloomfilter` along with more detailed results.



(a) Quality of Bloom filter hashes for random words (Box-Whisker plot of p-values).

(b) Throughput of the Redis-backed Counting Bloom filter and Bloom Filter (BF).

Figure 4.12: Analysis of the Redis-backed Bloom filters.

type was performed 10M times and averaged over 5 repetitions. The small instance (`cache.m3.large`, 2 cores, 6 GB RAM, medium I/O performance) and the large instance (`cache.r3.8xlarge`, 237 GB RAM, 10 GBit/s I/O) showed less than 30% performance difference. The only exception is the load operation, which fetches the complete Bloom filter. As the operation is bounded by network bandwidth, the larger EC2 instance shows a proportional throughput advantage here. Latencies of all operations in the experiment were consistently below 1 ms.

Cache Sketch	add	remove	pull
In-Memory	4.411M	12.787M	1.311M
Redis (small)	213K	379K	22.8K
Redis (large)	174K	313K	184K

Table 4.2: Throughput of different Cache Sketch implementations in operations/s.

The results provide clear evidence that the Redis-based implementation of the Cache Sketch provides sufficient performance to sustain a throughput of >100 K queries or invalidations per second with a single backing Redis instance. With the *per-table partitioning model* introduced in Section 4.1.6, the Cache Sketch can thus easily support much higher throughput than the Orestes servers and will not become a bottleneck.

In conclusion, the simulations and the experimental evaluation show that the Cache Sketch is able to considerably reduce latency for data management workloads. Through the combination of an efficient Cache Sketch and TTL estimation, the performance benefits are achieved with Δ-atomic reads. In the following, we will extend the scheme from key-based object caching to arbitrary query results. Then, we will discuss an integrated approach for low-latency access to files, objects, and queries.

4.4 Query Caching: Motivation and Problem Statement

In Section 4.1, we showed that the problem of dynamic data in expiration-based caching can be addressed by an appropriate summary data structure combined with server-side invalidations. However, the challenge of caching dynamic data becomes more difficult, if responses can contain arbitrary query results. In the following, we propose an extension to the Cache Sketch approach that does not only account for objects and files, but also the query interfaces provided by DBaas and BaaS systems. The query caching techniques (Quaestor, **Qu**ery **Sto**re)[12] are also part of Orestes [GSW+17].

As an example, consider a social blogging application. To retrieve posts on a particular topic, the client queries the DBaaS:

```
SELECT * FROM posts
WHERE tags CONTAINS 'example'
```

This (pseudocode) query is posed as an HTTP GET request. The web's infrastructure consisting of caches, load balancers, routers, firewalls, and other middleboxes handles the query similar to any other request issued by websites. In particular, any expiration-based caches as well as invalidation-based caches and reverse proxy caches are allowed to directly answer the query, if the DBaaS previously provided a TTL indicating cacheability for a defined time span. This offloads the database system from query processing and greatly reduces end-to-end query latency.

In order to make the Cache Sketch scheme applicable, we need to refine three problems from key-based caching to query result caching:

1. **Invalidation detection.** Does a given update operation change the result set of cached queries?
2. **Cache Coherence.** How can cached queries be kept consistent similar to objects (cf. Section 4.1) when they cannot be invalidated by the DBaaS?
3. **Cacheability.** Which queries are cacheable and what is their optimal TTL?

These challenges are illustrated in Figure 4.13. For every database operation, the backend has to determine whether it invalidates any cached data **(1)**. For object and file caching, this decision is trivial as it only requires to observe each update and delete operation in combination with information from the server Cache Sketch on whether a non-expired TTL has ever been issued. For query caching, invalidation detection is enabled by *InvaliDB*, a scalable subsystem for detecting invalidations of cached query results in real time. In the above example, an invalidation will be triggered when a blog post contained in the query result is changed or a previously non-matching post adds a tag that matches the query predicate.

[12] We will use the names Orestes and Quaestor interchangeably depending on which capabilities of the middleware are in focus.

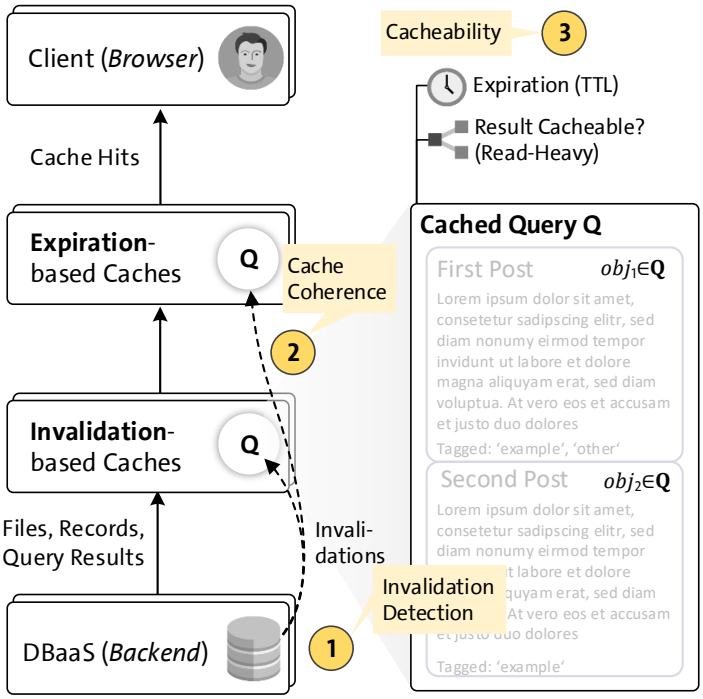


Figure 4.13: The three central challenges of query web caching.

Cache coherence **(2)** of expiration-based caches is based on extending the Cache Sketch, in order to indicate any potentially stale queries. Clients check the Cache Sketch before each query to decide whether cached results are permissible or a revalidation should instead be performed in order to proactively update any stale caches. To this end, queries are represented by a normalized string form that can be looked up in the Cache Sketch.

The cache hit rates are maximized by statistically deriving expiration estimates for query results and deciding which query results are worth caching **(3)**. Query TTL estimation relies on the TTL estimation for individual objects, but is more complex as inter-arrival times of query invalidations potentially depend on any object in the database system.

Traditional web caching does not give any guarantees on freshness when expiration-based caches such as browser caches, are involved. Furthermore, web caches cannot execute any application-specific logic. Instead, they only serve non-expired resources by their unique URL. For queries, this implies that a query can either be served as a list of record URLs (*ID-list*) or as a full result set (*object-list*). There is a difficult trade-off between both representations. When retrieving space-efficient ID-lists, clients have to assemble the result on their own, by fetching each referenced object. The objects are therefore individually cacheable, but more round-trips are necessary to aggregate the result. Object-lists, on the other hand, contain the fully specified objects, so that no additional round-trips are required. However, since caches are unaware of the objects contained in an object-list, objects are re-transmitted for each query result. In addition, object-lists are more prone to frequent invalidations, as any change to a contained object (e.g., a counter) invalidates the complete result.

The proposed query caching approach is a good match for common web workloads that are mostly read-heavy with many clients accessing the same data before it is updated [HN13]. To the best of our knowledge, Quaestor is the first approach that provides fresh query results served over the web caching infrastructure. It can thus improve performance and scalability of database and backend services without requiring additional server infrastructure. The contributions are threefold:

- We propose a comprehensive, service-independent approach for caching **dynamic query results** with rich default consistency guarantees (bounded staleness, monotonic reads and writes, read-your-writes).
- We introduce a scalable middleware infrastructure for maintaining cache coherence through a **query matching pipeline** and Cache Sketches.
- We provide empirical evidence that tremendous **latency improvements** can be obtained through query caching with arbitrarily bounded staleness for web-typical, read-heavy workloads.

In the following, we will present the key techniques used to make query caching feasible on web caches: a cache coherence mechanism, a query invalidation system, and a model for dynamic TTL estimation. Afterwards, an in-depth evaluation of Quaestor is given.

4.5 Cache Coherence for Query Results

To illustrate the value of a cache coherence mechanism, consider query caching with static TTLs as a straw-man solution. In that case, the server would assign a constant, application-defined TTL to each query, so that any web cache may serve the query where staleness is bounded by the TTL. This does not require any query invalidation logic in the client or server, as the regular expiration-based semantics of HTTP web caching are used. The problem of this naive solution is that either many stale queries will occur when the TTL is too high, or cache hit ratios will suffer when the TTL is too low. As in object-based caching, the first step to improving this scheme is adapting the purely static TTLs to the actual frequency of changes for each query. However, even for a better stochastic TTL estimation, stale query results occur for each deviation from the estimate. To address this, the Cache Sketch needs to be extended to capture stale query results.

4.5.1 Cache Sketches for Query Caching

The purpose of the extended Cache Sketch is to answer the question whether a given query is potentially stale. This information allows the server to compensate for TTLs of queries that change before their TTL expires.

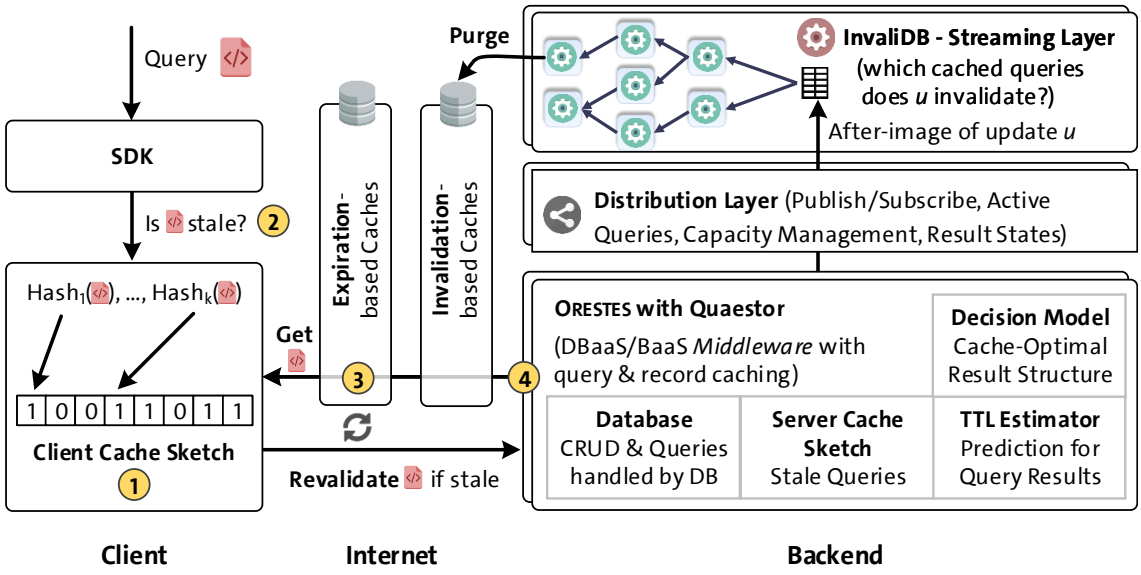


Figure 4.14: Query Caching architecture and request flow for providing cacheable query results.

Request Flow for Queries

Figure 4.14 gives a high-level overview of the query caching architecture with the role of the Cache Sketch for queries. From the perspective of a client performing a query, the request flow is as follows:

1. Upon connection, the client gets a client Cache Sketch (cf. Theorem 4.1) containing freshness information on potentially stale query results. During a client's session, the Cache Sketch is renewed periodically.
2. Before issuing a query, the Cache Sketch is queried by the SDK to decide between a normal cached load and a revalidation request.
3. The caches either serve their cached copy or forward the query upstream.
4. For cache misses and revalidations, the server returns the query result from the database using an appropriate TTL through query TTL estimation (cf. Section 4.6.2) and an appropriate result structure (cf. Section 4.6.3) using a decision model. The query is registered in InvaliDB to detect changes to the delivered query result in real time. If operations on the database implicitly update the query result before the TTL is expired, the query is added to the server Cache Sketch and purged from invalidation-based caches.

Construction and Properties of the Query Cache Sketch

A query or read is performed by querying the client Cache Sketch c_t that was generated at time t. The key is the normalized query string and hashed to the underlying Bloom filter, similar to object IDs. Theorem 4.2 derives the guarantees of the Cache Sketch for queries by generalizing Theorem 4.1 (see page 128) that derived Δ-atomic semantics for object reads.

Definition 4.4. *Let c_{t_3} be the Expiring Bloom Filter generated at time t_3. It contains the normalized query string q of every result $result(q)$ that became stale before it expired in all caches. Formally, this is every q for which holds that $\exists r(q,t_1,TTL), w(x,t_2) : t_1 + TTL > t_3 > t_2 > t_1$. The operation $r(q,t_1,TTL)$ is a query of q at time t_1 with a TTL for the query result and $w(x,t_2)$ is a write happening at t_2 on a record x so that $result(q)$ is invalidated (see notification events add, change, and remove in Section 4.6).*

Theorem 4.2. *A query q performed at time t_4 using the client Cache Sketch c_{t_3} satisfies Δ-atomicity with $\Delta = t_4 - t_3$, i.e., the client is guaranteed to see only query results $result(q)$ that are at most Δ time units stale.*

Proof. Analogous to the proof of Theorem 4.1, consider a query issued at time t_4 using c_{t_3} returning the query result $result(q)$ that was stale for $\Delta > t_4 - t_3$. Therefore, q must have been invalidated at a time $t_2 < t_3$ as otherwise $t_4 - t_2 < \Delta$. Hence, there must have been an earlier query $r(q,t_1,TTL)$ with $t_1 + TTL > t_4 > t_2$ so that $result(q)$ is still cached. By the construction of c_{t_3}, the query is contained in c_{t_3} until $t_1 + TTL > t_4$ and therefore not stale at time t_4 (proof by contradiction). □

The Cache Sketch thus contains all stale queries for one point in time, i.e., queries that became invalid while still being stored in some cache.

Freshness Policies

The achieved freshness is linked to the age of the Cache Sketch. Similar to object caching, the basic way of utilizing the Cache Sketch is to fetch it on page load and use it for the initial resources of the application, e.g., stylesheets and images (*cached initialization*, see Definition 4.1). To maintain Δ-*bounded staleness*, the Cache Sketch is refreshed in a configurable interval of Δ. Clients can therefore precisely control the desired level of consistency for queries, objects, and files. This polling approach for the Cache Sketch resembles Pileus' [TPK+13] method, where clients poll timestamps from all replication sites to determine which replica can satisfy the demanded consistency level. However, the Cache Sketch is significantly more scalable as the freshness information is already aggregated and does not have to be assembled by clients from different caches or replicas.

4.5.2 Consistency

The consistency levels provided by Quaestor are summarized in Figure 4.15. They can be grouped into **default guarantees** that are always met and **opt-in guarantees** that are associated with an overhead and can be enabled per request, session, or application[13].

Default Consistency Guarantees

The central consistency level enabled by the Cache Sketch is Δ-*atomicity* with the application and clients being able to choose Δ. Several additional session consistency guarantees

[13] In Section 2.2.4, formal definitions of the discussed consistency models are given.

are achieved. *Monotonic writes*, i.e., a global order of all writes from one client session, are assumed to be given by the database (e.g., MongoDB) and are not impeded by the Cache Sketch. *Read-your-writes* consistency is obtained by having the client cache its own writes within a session: after a write, the client is able to read her writes from the local cache. *Monotonic read* consistency guarantees that a client will only see monotonically increasing versions of data within a session. This is achieved by having clients cache the most recently seen versions and comparing any subsequent reads to the highest seen version. If a read returns an older version (e.g., from a different cache), the client resorts to the cached version, if it is not contained in the Cache Sketch, or triggers a revalidation otherwise. These session consistency guarantees are maintained by the SDK, transparent for the developers using it.

As discussed in Section 4.1.7, Orestes can expose an eventually consistent data store. The inconsistency window Δ_{DB} of the data store then lowers the Δ-atomicity guarantee. The same holds true, if invalidations are performed asynchronously with lag Δ. However, as the probability that this violates consistency is low, it is a common choice to accept $(\Delta + \Delta_{DB} + \Delta_c)$-atomicity. By choosing a lower Δ, developers can easily compensate both effects. In practice, adjusting Δ to $\Delta - \Delta_c$ allows revalidation requests to be answered by invalidation-based caches instead of the origin servers. This optimization significantly offloads the backend. If however, the exposed database system does not offer strong consistency, but potentially unbounded staleness (e.g., due to asynchronous replication) the Cache Sketch's guarantee becomes a probabilistic consistency level of $(\Delta + \Delta_{DB}, p)$-atomicity (cf. discussion in Section 4.1.7).

Consistency Level	Realization	
Δ-atomicity (staleness never exceeds Δ seconds)	Controlled by age (i.e. refresh interval) of Cache Sketch	Always ✔
Monotonic Writes	Guaranteed by underlying database system	
Read-Your-Writes and **Monotonic Reads**	Written data and most recent read-versions cached in client	
Causal Consistency	If read timestamp is older than Cache Sketch it is given, else revalidation required	Opt-in +
Strong Consistency (Linearizability)	Explicit revalidation (cache miss at all levels)	

Figure 4.15: Consistency levels provided by Quaestor: Δ-atomicity, monotonic writes, read-your-writes, monotonic reads are given by default, causal consistency and strong consistency can be chosen per operation (with a performance penalty).

Opt-in Consistency Guarantees

By allowing additional cache misses, *causal consistency* and even *strong consistency* are possible as an opt-in by the client. With causal consistency, any causally related operations are observed in the same order by all clients [VV16]. With caching, causal consistency can be violated, if of two causally dependent writes one is observed in the latest version and the other is served by a cache. Using the Cache Sketch, any causal dependency younger than the Cache Sketch is observed by each client, as the Cache Sketch acts a **staleness barrier** for the moment in time it was generated: any writes that happened before the generation of the Cache Sketch are visible along with the causal dependencies. However, if a read is newer than the Cache Sketch, causal consistency might be violated on a subsequent second read. Therefore, the client has two options to maintain causal consistency after a read newer than the Cache Sketch is returned[14]. First, The Cache Sketch can be refreshed to reflect recent updates. Second, every read happening before the next Cache Sketch refresh is turned into a revalidation. For strong consistency within a client session, *every* read within that session is performed as a revalidation. In that case, latency is not reduced, but an unnecessary transfer of the object or query result is prevented, if the data is still up-to-date.

All default and opt-in consistency guarantees are identical for objects, files, and queries. From the perspective of the Orestes middleware, a query result is simply a special type of object identified by a query string that changes based on invalidation rules. Therefore, the consistency guarantees provided through the combination of the Cache Sketch and server-side invalidations are the same for all types of cached data delivered by Orestes.

The strongest semantics Orestes can provide are ACID guarantees through distributed cache-ware transactions (see Section 4.8). These optimistic transactions exploit the fact that caching reduces transaction durations and can thereby achieve low abort rates with a variant of backward-oriented optimistic concurrency control. As described in detail in Section 4.8, the key idea is to collect read sets of transactions in the client and validate them at commit time to detect both violations of serializability and stale reads. The scheme is similar to externally consistent optimistic transactions in F1 and Spanner [CDE+13, SVS+13], but can leverage caching and the Cache Sketch to decrease transaction duration for clients connected via wide-area networks.

Additionally, clients can directly subscribe to query result change streams that are otherwise only used for the construction of the Cache Sketch. For this purpose, Orestes exposes a continuous query interface that leverages Websockets [Gri13] to proactively push query result updates to connected end devices. Through this synchronization scheme, the application can define its critical data set through queries and keep it up-to-date in real time. For applications with a well-defined scope of queries, this approach is preferable, while complex web applications will profit from using the Cache Sketch due to lower latency for the initial page load and lower resource usage in the backend.

[14] This can easily be observed based on the *LastModified* field provided in the response for each object.

4.5.3 Cache Sketch Maintenance for Queries

Query caching relies on the server Cache Sketch that stores the Bloom filter and tracks a separate mapping of queries to their respective TTLs. In this way, only non-expired queries are added to the Bloom filter upon invalidation. After their TTL is expired, queries are automatically removed from the Bloom filter. These removals are based on a distributed queue implementation storing the outstanding Bloom filter removals shared across Orestes servers. To achieve this without coordination overhead, the Orestes prototype relies on sorted sets in Redis.

The client-side usage of the Cache Sketch for queries is similar to objects. A stale query is contained in the Cache Sketch until the highest TTL that the server previously issued for that query has expired. While contained, the query always causes a cache miss. To maintain the Cache Sketch in the server, changes to cached query results have to be detected and added in real time, as described in the following section.

4.6 Invalidations and Expirations

To provide server-side query invalidations, Quaestor registers all cached queries in *InvaliDB*[15] which in turn notifies Quaestor as soon as a query result becomes stale. While we use SQL for the sake of clarity in our illustrations, the concept is generic for any database query language. The Orestes prototype with InvaliDB supports MongoDB's query language.

4.6.1 Invalidation Detection

The invalidation pipeline (*InvaliDB*) matches change operations to cached queries. For each cached query, it determines whether an update changes the result set. The invalidator then outputs a set of queries with stale cached query results to Orestes, which sends out invalidations to reverse proxy caches and CDNs. This check is performed by re-evaluating queries on after-images of the relevant database partition in a distributed stream processing system (Storm [Mar14]) co-located with Orestes. The throughput of the invalidation pipeline is the limiting constraint of query caching and determines how many queries can be cached at the same time. Through a *capacity management model*, only queries that are cacheable for a sufficient time span are admitted and prioritized based on the costs of maintaining them.

InvaliDB continuously matches record after-images provided with each incoming write operation (insert, update, delete)[16] against all registered queries. Orestes can subscribe to an arbitrary combination of the following **notification events**, each of which triggers a notification message:

[15] While InvaliDB originates from the development of Orestes, it has primarily been researched by Wolfram Wingerath [EPM+16, GSW+17, WGF+17, WGFR16].

[16] A delete operation provides the identifier of the deleted object and `null` as after-image.

- `match`: an after-image matches a query (stateless since independent of result set)
- `add`: an object enters a result set
- `remove`: an object leaves a result set
- `change`: an object already contained in a result set is updated without altering its query result membership
- `changeIndex` (for sorted queries only): an object already contained in a result set is updated and changes its position within the result
- `all`: any of the above

Of the notification types, only `match` events can be determined in a *stateless* fashion comparing after-images with queries, i.e., without relying on the previous query result. All other notifications are *stateful* and require keeping track of result state in order to determine whether a given after-image changes, enters, or leaves a query result.

To illustrate these different events, consider the query in Figure 4.16 which selects blog posts tagged with the keyword `example`. First, a new blog post is created which is yet untagged and therefore not contained in the result set (box). When an update operation adds the `example` tag to the blog post, it enters the result set which triggers either a `match` or an `add` notification, depending on the parameters provided on subscription. Later, another tag is added which does not affect the matching condition and therefore only changes the object's state, thus entailing another `match` or a `change` notification. When the `example` tag is finally removed from the blog post, the matching condition does not hold anymore and the object leaves the result set, causing a `remove` notification to be sent.

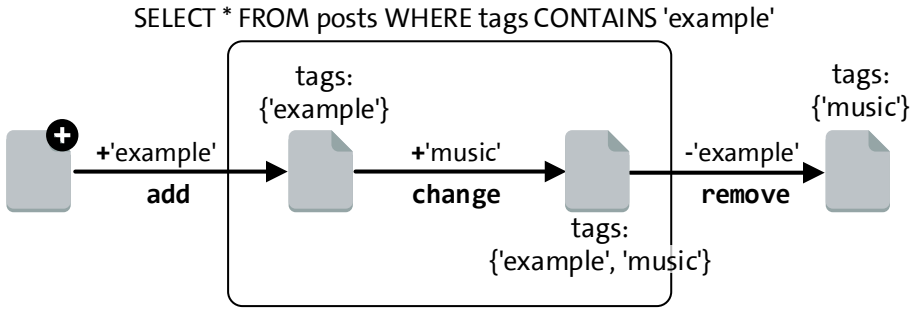


Figure 4.16: Notifications as an object gets updated (figure taken from [GSW+17]).

With respect to query invalidation, only two combinations of event notifications are useful, both of which are stateful since they require knowledge of the result: When the cached query result contains the IDs of the matching objects (ID-list), an invalidation is only required on result set membership changes (`add/remove`). Caching full data objects (object-list), on the other hand, also requires an invalidation as soon as any object in the result set changes its state (`add/remove/change`).

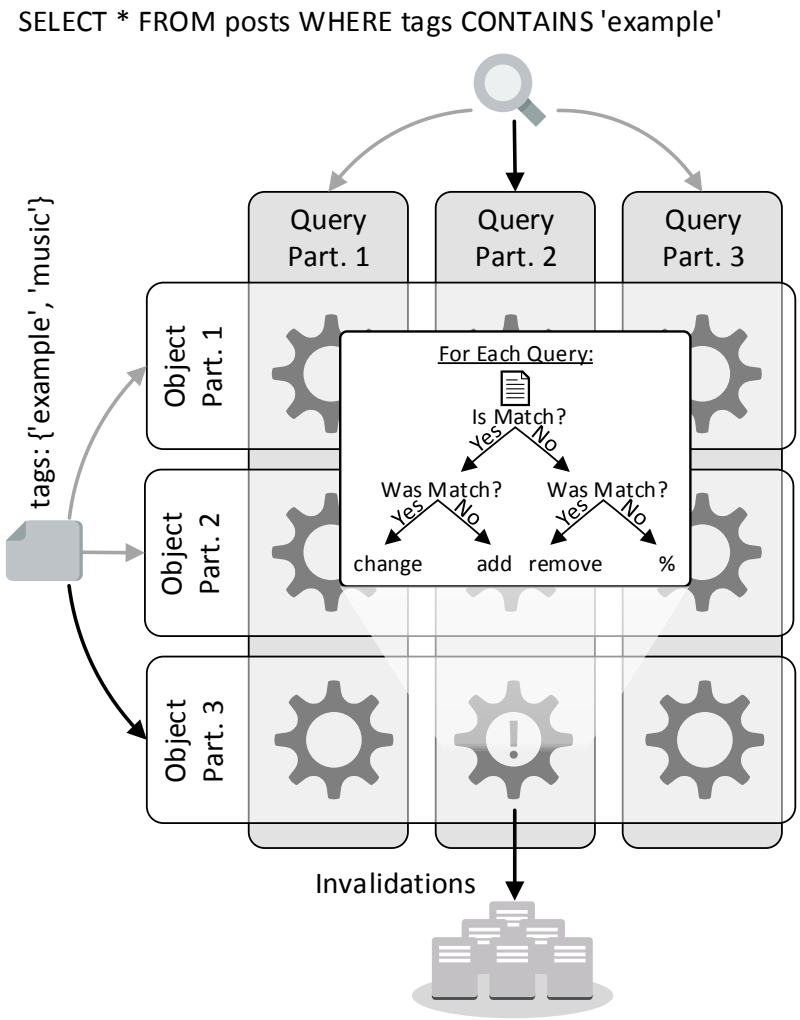


Figure 4.17: InvaliDB workload distribution: every node is only assigned a subset of all queries and a fraction of all incoming updates (figure taken from [GSW+17]).

Workload Distribution

In order to provide the scalable real-time notifications, the InvaliDB prototype relies on three tasks for query ingestion, change stream ingestion, and matching, each of which is distributed over the nodes in the cluster using the Storm real-time computation framework [Mar14, TTS+14]. The matching workload is distributed by hash-partitioning both the stream of incoming data objects and the set of active queries orthogonally to one another, so that every instance of the matching task is responsible for only a subset of all queries (query partitioning) and only a fraction of their result sets (data stream partitioning). The ingestion workload, in contrast, is not partitioned, but scattered across task instances. Every instance of the query and change stream ingestion tasks transactionally pulls newly arrived data items (query activations/deactivations or update operations, respectively) from the source and forwards them according to the partitioning scheme.

Figure 4.17 illustrates workload distribution in a 9-node cluster[17] with three object partitions (lightly shaded rows) and three query partitions (strongly shaded columns). When a query is received by one instance of the query ingestion task, it is forwarded to all matching task instances in its respective query partition (e.g., query partition 2). Since InvaliDB has to be aware of the result sets of all newly added stateful queries in order to maintain their correct state, every new stateful query is initially evaluated on Quaestor and then sent to InvaliDB together with the initial result set.

To rule out the possibility of missing updates in the timeframe between the initial query evaluation (on Orestes) and the successful query activation (on all responsible InvaliDB nodes), all recently received objects are replayed for a stateful query when it is installed. When an update operation is registered by one of the change stream ingestion task instances, this operation and its corresponding after-image are forwarded to all matching task instances in the respective object partition (e.g., object partition 3). If the after-image matches any of the currently active queries, one of the matching task instances in the receiving object partition will register the match. In the example, the one that is responsible for query partition 2 and object partition 3 detects a new match for the example query and therefore sends out an `add` notification.

As a basic rule, all nodes in the cluster are assumed to have equal resources and therefore should also receive the same fraction of the workload. However, if all task instances were distributed as uniformly as possible across all nodes in the cluster, some nodes in the cluster were exclusively concerned with matching, while others were concerned with matching *and* change stream or query ingestion. Therefore, we do not co-locate matching and ingestion task instances on the same nodes, but instead employ a large number of matching-only nodes and a small number of query and change stream ingestion nodes.

Since all matching operations are independent from one another, data objects and queries are hash-partitioned, thus gaining predictable system performance without any hotspots. The sustainable system throughput can be increased by adding more nodes in additional object partitions and, correspondingly, the set of continuously evaluated queries can be increased by adding more nodes in additional query partitions.

Scalability

Since InvaliDB partitions both the change stream *and* the set of all active queries, single-node performance does not limit overall system performance: as long as every query can be handled by a single node, change stream partitioning is not required and the load can be spread across the cluster by simply assigning every node a fair share of all active queries. However, additional change stream partitioning allows distributing responsibility for a single query over several machines and guarantees low latency, even when the resources required for handling individual queries exceed single-node capacity, e.g., due to huge result sets or very low query selectivity at very high update rates. Thus, overall per-

[17] Please note that we omit the parallelism of the data ingestion tasks here in favor of simplicity and only make the distribution of the matching task explicit.

formance is neither bounded by update throughput nor by the number of active queries nor by query selectivity or result set size and scales linearly with the number of cluster nodes (see Section 4.7.4).

Managing Query State

Simple static matching conditions such as `WHERE tags CONTAINS 'example'` are **stateless**, meaning that no additional information is required to determine whether a given after-image satisfies them. As a consequence, the only state required for providing stateful `add`, `remove`, or `change` notifications to stateless queries is the former matching status on a per-record basis. This state can be partitioned by object ID and thus can be easily distributed, just like the computation itself.

With additional `ORDER BY`, `LIMIT`, or `OFFSET` clauses, however, a formerly stateless query becomes **stateful** in the sense that the matching status of a given record becomes dependent on the matching status of other objects. For sorted queries, InvaliDB is consequently required to keep the result ordered and maintain additional information such as the entirety of all items in the offset, but also has to rely on receiving all operations in the same order as the Orestes nodes to prevent missing or false notifications due to out-of-order arrival. To capture result permutations, `changeIndex` events are emitted that represent positional changes within the result. Our current implementation maintains order-related state in a separate processing layer partitioned by query.

Implementation

All current components of the InvaliDB prototype are written in Java. To make our approach towards real-time notifications applicable to a wide range of use cases, we designed InvaliDB with a pluggable query engine, the default supporting MongoDB. We evaluated several other options before implementing our own query engine to copy existing behavior. Specifically, we evaluated using third-party MongoDB-like query engines and instances of MongoDB for query evaluation. However, we had to abandon them due to correctness and performance issues. As a consequence, we implemented a Java-based query matching engine that operates on abstract syntax trees and is able to reuse predicate evaluation across query subscriptions. Communication between Quaestor and InvaliDB is handled through Redis message queues.

At the time of writing, the InvaliDB prototype does not support joins and aggregations. Since Orestes is designed for aggregate-oriented, denormalized NoSQL databases, the capability to pose predicates on nested documents is sufficient to reflect 1:1 and 1:n relationships. Aggregations with groupings are ongoing work and therefore currently uncached.

In summary, InvaliDB provides a scalable stream processing mechanism for detecting query invalidations on top of Orestes. Its central trade-off lies in the partitioning of both queries and changes, which makes most joins inherently expensive, but enables linear

scalability and low latency for filter queries over collections as promoted by many scalable NoSQL systems.

4.6.2 Statistical TTL Estimation

As described in Section 4.2.1, TTL Estimators provide stochastic estimations of cache expiration times for query results. Our mechanism is based on the insight that any cached query result should ideally expire right before its next update occurs, thus achieving maximum cache hit rates while avoiding unnecessary invalidations. The discrepancy between the actual and the estimated TTL directly determines the amount of data considered stale and hence affects the false positive rate of the Cache Sketch. High cache hit rates and an effective Cache Sketch size thus require reliable TTL estimates.

We use a dual strategy for estimating expirations for query results. Initially, TTLs are estimated through the stochastic process of incoming updates. As described in Section 4.2.1, Poisson processes count the occurrences of events in a time interval t characterized by an arrival rate λ. The inter-arrival times of events have an exponential cumulative distribution function (CDF), i.e., each of the identically and independently distributed random variables T_i has the cumulative density $F(x;\lambda) = 1 - e^{(-\lambda x)} \; for \; x \geq 0$ and mean $1/\lambda$. For each database object, Orestes can estimate the rate of incoming writes λ_w in some time window t through sampling.

The result set Q of a query of cardinality n can then be regarded through a set of independent exponentially distributed random variables $T_i, \ldots, T_n$ with different write rates $\lambda_{w1}, \ldots, \lambda_{wn}$. Estimating the TTL for the next change of the result set requires a distribution that models the minimum time to the next write, i.e., $min\{T_1, \ldots, T_n\}$, which is again exponentially distributed with $\lambda_{min} = \lambda_{w1} + \ldots + \lambda_{wn}$ (minimum value distribution [Gal95]). The quantile function then provides estimates that have a probability p of seeing a write before expiration:

$$F^{-1}(p, \lambda_{min}) = \frac{-ln(1-p)}{\lambda_{min}}. \tag{4.4}$$

By varying the quantile, higher/lower TTLs (and thereby cache hit rates) can be traded off against more or fewer invalidations. Alternatively, the TTL can be estimated using the expected time until the next write. This results in always using the observed mean TTL, but in turn does not allow fine-grained adjustments.

For individual records, we always use an estimate based on the approximated write rates. For queries, the Poisson estimate based on the write rates on the keys of the result set is only used as an initial estimate. The initial estimate has a bias, as the used minimum value distribution models the probability of one contained record changing. This subsumes `change` and `remove` events, but not `add` events. Considering potential `add` events is infeasible as it would require knowledge about each database object as well as cardinality

estimates of the query. Therefore, the system quickly needs to converge to a more precise estimate after observing actual invalidations.

If a query result is invalidated, the *actual TTL* of the result can be computed as the difference between the invalidation timestamp and the previous read timestamp. We can hence update the old estimate according to an exponentially weighted moving average (EWMA) closer towards the true TTL:

$$TTL_{query} = \alpha \cdot TTL_{old} + (1-\alpha) \cdot TTL_{actual} \tag{4.5}$$

As an alternative to the EWMA, all strategies from Section 4.2.3 can be employed, too. The current TTL estimate for a query is kept in a shared partitioned data structure called the *active list*, which is accessed by all Quaestor nodes. The key idea of the query TTL estimation model is to make an educated guess about the initial TTL which should then move towards the "true" TTL with some lag after invalidations. TTL estimation is used for queries and records in both expiration- and invalidation-based caches. Note that this does not require clock synchronization, as only relative time spans are used.

4.6.3 Representing Query Results

A cached query result can either be served as a list of record URLs (**ID-list**) or as a full result set (**object-list**). ID-lists are more space-efficient and yield higher per-record cache hit rates as objects of query results are stored and retrieved individually. However, they require more round-trips to assemble the result – the decision which representation to use cannot be made by the cache. Quaestor employs a cost-based decision model in order to weigh fewer invalidations (ID-lists) against fewer round-trips (object-lists) when choosing a result representation.

Cost Model

The goal is to minimize invalidations and query latency while maximizing cache hits. Our decision model weighs the costs of fewer invalidations against lower round-trip costs when deciding between object-lists and ID-lists. Web caches remain unmodified by Orestes, therefore sending a query result in one response (object-list) hides the fact that a result contains separately cacheable objects. With ID-lists, clients load the ID-list first and the individual objects in a subsequent second round-trip, so that caches store individual objects as well as the ID-list itself. For HTTP/1.1, the number of round-trips for ID-lists depends on the result set cardinality (resp. a limit clause) and the number of connections, because only six objects can be transferred in parallel [Gri13]:

$$c_{rtt} = 1 + \left\lceil \frac{min(limit, card(result))}{connections} \right\rceil . \tag{4.6}$$

Through multiplexing, HTTP/2 [IET15] reduces the cost for ID-lists to 2: one request for the ID-list and an arbitrary number of parallel subsequent requests for the referenced objects. For object-lists, the cost is always 1, as they only require a single request. Note that a user-defined limit on the number of records to be returned also impacts the caching decision, because it overrules the original cardinality of the query result. On the other hand, ID-lists have lower invalidation costs, as only `add` and `remove` notifications trigger an invalidation, whereas cached object-lists are also invalidated by `change` notifications. A third hybrid decision is also possible, which is a combination of ID-lists with objects that are likely to be uncached proactively piggybacked through HTTP/2 push. As the hybrid model involves a complex learning problem, we leave it to future work to design and evaluate a respective strategy.

Comparing Costs

The decision to represent cached query results as object-lists or ID-lists depends on the preference between cacheability and round-trip costs. Fewer round-trips result in improved latency, while the ID-list model causes fewer invalidations and thus more cache hits for both queries and record reads. The **query cost equation** hence compares the fraction of saved invalidations (ID-lists) to the fraction of saved round-trip costs (object-lists):

$$asIds := (w \cdot \frac{changes}{removes + adds + changes} > 1 - \frac{1}{c_{rtt}}) \tag{4.7}$$

We also introduce a bias w to express a preference. The decision model provides a straightforward, cheap calculation that only requires Quaestor to track counters on match types and the current result set cardinality. In the following section, we introduce how Quaestor manages local and shared state to provide globally consistent TTLs and decisions on the object-list versus ID-list structure.

4.6.4 Capacity Management

Required matching capacities scale linearly with the number of concurrently cached queries. Based on past cache miss rates and invalidations, only queries that are executed sufficiently often should be activated and passed to InvaliDB. By maintaining node-local, probabilistic heavy-hitter sets of queries that get invalidated too often, queries with moderate expected TTLs can be identified and become eligible for caching.

The distributed caching algorithm that decides how queries are handled is shown in Algorithm 2. Incoming queries are first compared to a local blacklist of *hard uncacheable* queries (line 2): queries are locally marked uncacheable, if their invalidation frequency is above a predefined threshold. In contrast, a query is *soft uncacheable*, it is cacheable per se, but is not admitted at the given time, because InvaliDB has no capacity to match the query. Uncacheable queries are returned to the client as full object-lists after query execution. Clients can mark queries as *hinted* to indicate that their results should always

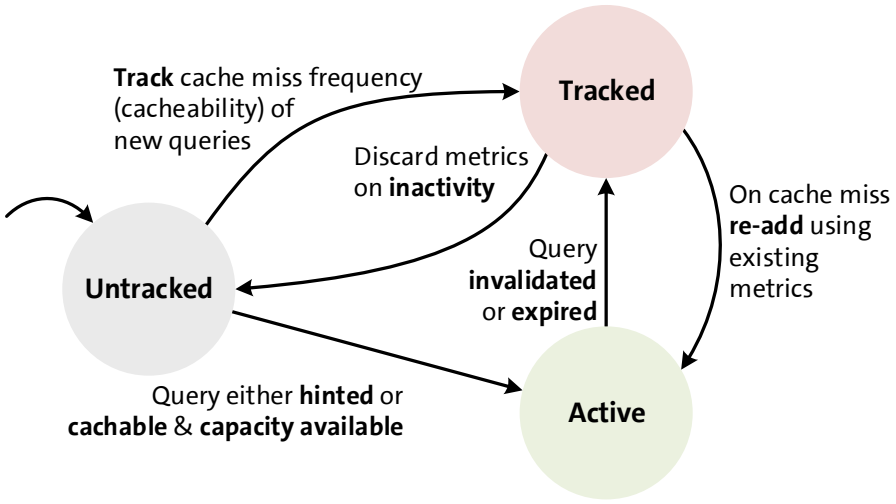


Figure 4.18: Quaestor's query capacity management.

be cached, even if capacity is exceeded[18]. For all queries, we differentiate between three states, as shown in Figure 4.18:

- *Untracked* queries are queries for which there are currently no metrics, either because they are new or because they have not been executed in the last time window.
- *Active* queries are currently cached.
- *Tracked* queries have expired or were invalidated, but their read rates are still tracked. This allows us to adapt the TTL and representation upon reactivation.

Algorithm 2 Distributed query caching algorithm

```
1:  procedure EXECUTE(query) returns result
2:      if query ∈ hardUncacheable return db.execute(query)
3:      metrics ← activeList.getMetrics(query)
4:      if active or free capacity or hinted then
            metrics.state ← active
5:      else if capacity exceeded then
            metrics.state ← tracked
6:      metrics.asIds ← Equation (4.7)
7:      result ← db.execute(query)
8:      if metrics.previousState = untracked then
            TTL ← estimate so that P(T_nextwrite > TTL) = 50%
9:      else
            TTL ← metrics.TTL
10:     if TTL > TTL_min then
            CacheSketch.reportRead(query, TTL)
            set TTL and card(result)
            InvaliDB.subscribe(query, TTL), on invalidation do:
                CacheSketch.reportWrite(query)
                TTL ← α · TTL_old + (1 − α) · TTL_actual
                update card(result) and invalidation counters
11:     return result
```

[18]Hinting is only available for users with administration privileges, in order to prevent end users from degrading overall performance.

If a query is cacheable, the Quaestor node receiving the query modifies the central *active list* of cached queries. It has to perform the state transition of the query, decide how to represent the result, execute the query, and write back the updated metrics. The relevant metrics are the current state, invalidation counters (changes, adds, removes), cache misses, how the query was previously cached, and its previous TTL.

After updating the query state in the **active list** (lines 2 to 6), the query is executed and a TTL for the result set is estimated (lines 8 to 9). For new queries, the Poisson quantile described in Section 4.6.2 is used. TTLs of known queries are adjusted towards the actual TTL upon expiration or invalidation and reused, if the query is re-executed later. If the estimated TTL is larger than TTL_min, the query and its TTL are reported to the central Cache Sketch, the active list, and InvaliDB (line 10). A compare-and-swap approach in the active list is used to prevent race conditions arising from conflicting TTL estimates of concurrent query executions. New queries and those with updated TTLs are subscribed to InvaliDB. Upon each invalidation, affected queries are reported to the Cache Sketch and invalidation counters in the active list are updated. Further, invalidations trigger TTL updates in the metrics component that will impact the TTL for the next incoming cache miss on a query. Finally, the query result is returned to the client. The active list's capacity is limited by the throughput of the given InvaliDB deployment.

Quaestor uses different **admission policies** to accept or reject queries. In a naive approach, the *active list* greedily accepts all new queries until working at capacity. However, a first-come-first-serve order can be problematic depending on the request distribution. If queries that are expensive to match or hard to cache are requested first, InvaliDB's matching capacity will be expended on them. A more refined model than the greedy approach is to only accept queries that have caused at least m cache misses in the last t seconds indicating that caching them would be advantageous. We hence suggest an optimization where queries are not cached on their first read. Instead, Quaestor first registers the read timestamp and begins tracking the number of reads over a timeframe of t seconds. The model therefore pre-filters queries by popularity before making them cacheable.

4.6.5 End-to-end Example

Figure 4.19 gives an end-to-end example of the steps involved in serving cacheable queries. In the depicted setting, the client begins by fetching the Cache Sketch containing a stale query ($q2$) still cached in the client **(1)**. Therefore, when loading the query, the client triggers a revalidation that refreshes the client cache and causes a miss at the invalidation-based cache. Using the active list, the server passes the query to InvaliDB for future change detection, while estimating the TTL and deciding between an ID-list and object-list representation **(2)**.

Before returning the result, the server reports the query to the Cache Sketch, so that every subsequent invalidation within the newly estimated TTL will mark the cached query as stale. The returned result is cached in both caches using the new expiration timestamp.

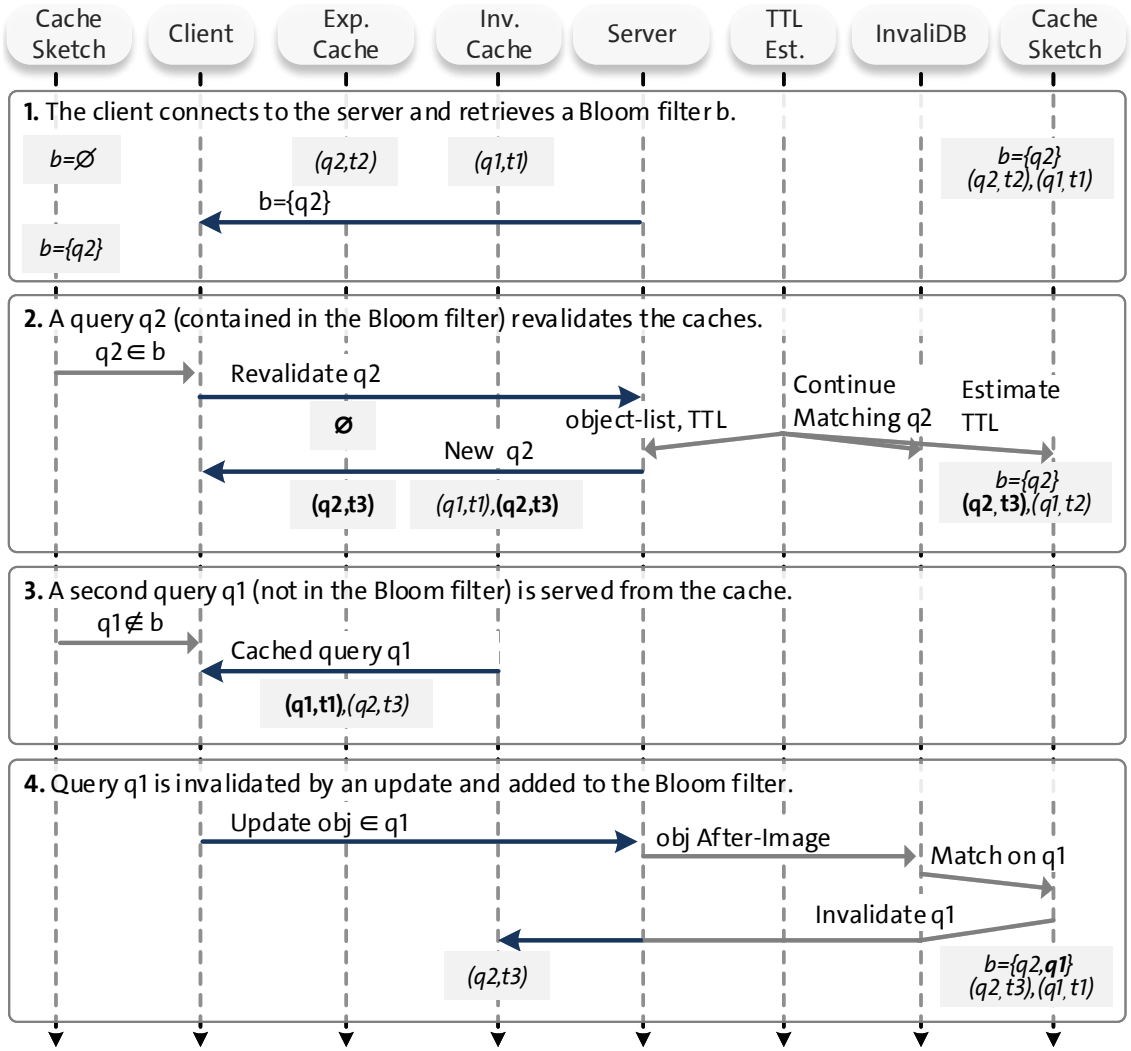


Figure 4.19: End-to-end example of query caching.

When the client performs a query that is not stale ($q1$), the cache can serve the result **(3)**. A change operation to a record contained in that query result is forwarded to the database and the respective after-image is passed to InvaliDB **(4)**. InvaliDB detects the change to the query and reports the invalidation to the Cache Sketch. As the query still has a non-expired TTL, the Cache Sketch adds the stale query and triggers an invalidation to prevent stale reads of the old query result.

4.7 Evaluation of Query Caching

In this section, we demonstrate that Quaestor's scalability is only limited by the write throughput of the underlying database system. We evaluate Quaestor with regard to latency, throughput, and staleness (and hence the effectiveness of TTL estimators) compared to a baseline of just using a CDN, only using a client cache, and no caching at all.

We further demonstrate the linear scalability of InvaliDB and the high throughput of our distributed Cache Sketch implementation.

4.7.1 Experimental setup

Our experimental design is based on the YCSB benchmark [CST+10]. YCSB defines a set of common workloads to evaluate the performance of cloud databases (cf. Section 4.3.1). We implemented a YCSB-style framework that extends the widely-used original benchmark in two aspects: a multi-threading model for massive connection parallelism and a multi-client model to scale the client tier [FWGR14]. As a baseline to our experiments, we used an Orestes deployment with uncached communication, which we deem representative for state-of-the-art database services that do not use web caching (cf. Section 2.2.6).

We evaluated Quaestor on the following EC2 setup: MongoDB was configured in a cluster setting with 3 `m3.xlarge` (4 vCPUs, 15 GB RAM, 2x40 GB SSDs) instances with 2 shard servers and 1 configuration server. Objects were sharded through their hashed primary key. The Cache Sketch as well as the Redis-backed active list were hosted on one `m3.xlarge` instance, respectively. Further, we used 3 Quaestor servers and a varying number of workload-generating client instances (all `m3.xlarge`). To demonstrate the full impact of geographic round-trip latency, Quaestor, MongoDB, and InvaliDB were hosted in a virtual private cloud in the EC2 Ireland region, with workloads being generated from the Northern California region. In the setups using a CDN, Fastly was used (client-CDN round-trip latency 4 ms). Cache misses at CDN edge servers were forwarded to Quaestor nodes in a round-robin manner.

Workloads were specified by defining a discrete multinomial distribution of operation types (reads, queries, inserts, partial updates, and deletes). TCP connections were pre-warmed for 30 seconds on a dummy table. Load was generated using asynchronous requests with 300 HTTP connections per client instance. Each data point was created under 5 minutes of load, which was sufficient to achieve stable and reproducible results. Requests were generated by first sampling an operation type and then sampling the key/-query and table to use (using a Zipfian distribution). For the workloads we analyzed, 10 database tables, each with 10 000 objects, were generated for each run. Each object consisted of 10 fields with randomly drawn integers and strings. Further, 100 distinct queries per table were generated to initially return an average of 10 objects. Throughout the experiment, the number of objects per query result changed, when objects matching the query predicate were updated.

We also extended the YMCA Monte Carlo simulation framework to queries. Simulation is the most reliable method to analyze properties like query staleness as it provides globally ordered event timestamps for each operation and does not rely on error-prone clock synchronization. Further, the simulation enables detailed analysis optimization of various workload parameters such as latency distributions, TTL estimation models, and capacity configurations.

4.7.2 Cloud-Based Evaluation of Query Caching

To demonstrate the effectiveness of Quaestor, we varied typical workload parameters such as incoming connections, the number of queries and objects, and update rates. We studied Quaestor's scalability and performance under high throughput and extended the analysis to more clients and measured staleness using simulation. We did not compare Quaestor to geo-replicated systems (e.g., Pileus) as our main point is to show that commodity web caching highly improves latency with very little staleness and no additional servers. Geo-replication schemes tuned towards one specific geographical setup might still outperform Quaestor.

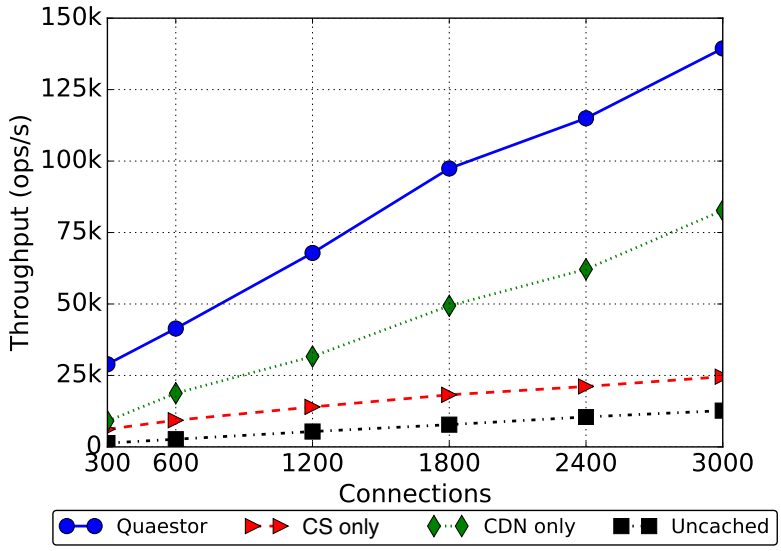


Figure 4.20: Throughput for a varying number of parallel connections comparing uncached database access (*Uncached*), query caching in the CDN (*CDN only*), query caching in the client (*CS only*), and full client and CDN query caching (*Quaestor*).

Read-Heavy Workload

We begin evaluating Quaestor on a read-heavy workload with 99% queries and reads (equally weighted) and 1% writes. Figure 4.20 demonstrates Quaestor's throughput scalability against a baseline without dynamic caching (*Uncached*), a CDN with InvaliDB (*CDN only*), and the client cache based on the Cache Sketch (CS) only (*CS only*). At maximum load (3 000 asynchronous connections delivered by 10 client instances), Quaestor achieves an 11-fold speedup versus an uncached baseline, a 5-fold improvement over the Cache Sketch-based client caches and a 69.5% improvement over a CDN with InvaliDB. Using a CDN with InvaliDB yields superior performance to only using client caches since clients rely on the CDN to fill up their caches quickly.

Client-side Bloom filters were refreshed every second ($\Delta = 1$) to ensure minimal staleness. Figure 4.21 illustrates the latency distribution: while most queries are client cache

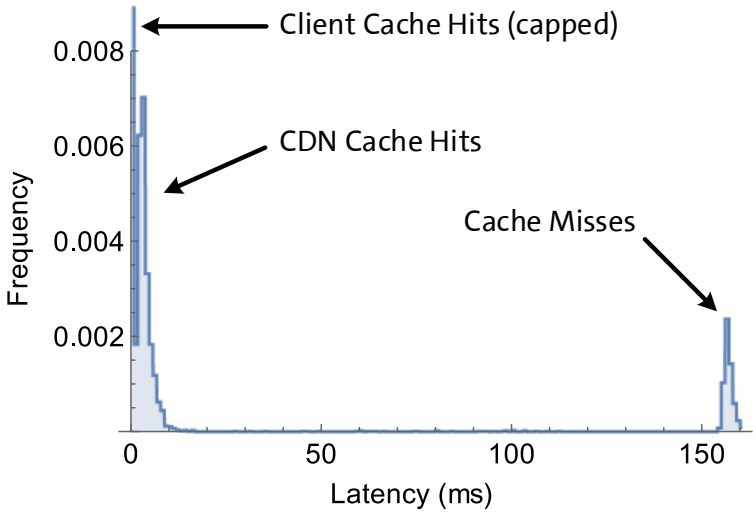


Figure 4.21: Query latency histogram showing peaks for client cache hits, CDN cache hits, and cache misses.

hits with no latency, CDN hits induce an average latency of 4 ms and cache misses 150 ms. Mean round-trip latency between client instances and Quaestor was 145 ms with a variance of 1 ms between runs (error bars omitted due to scale). Please note that linear scalability is not possible, since an increasing number of clients increases the number of updates and thus reduces cacheability.

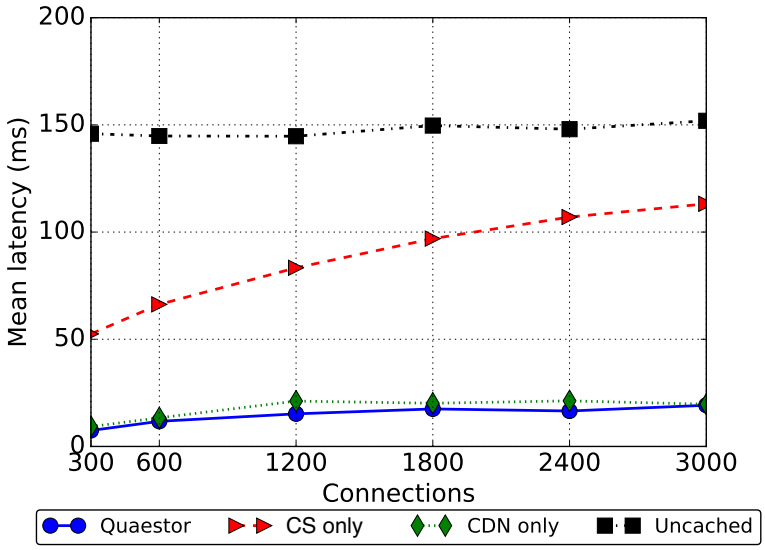


Figure 4.22: Object read latency for a varying number of parallel connections comparing cached to uncached database access.

Figures 4.22 and 4.23 show read and query latency for the same setup. For 3 000 connections, Quaestor achieved a mean query latency of 3.2 ms and a mean read latency of 17.5 ms. As there are 100× more records than queries, cache hit rates for queries are higher and latencies lower. Note that the latency of the variant with the client cache (CS only) increases due to more overhead at the database. In contrast, CDN latency for queries im-

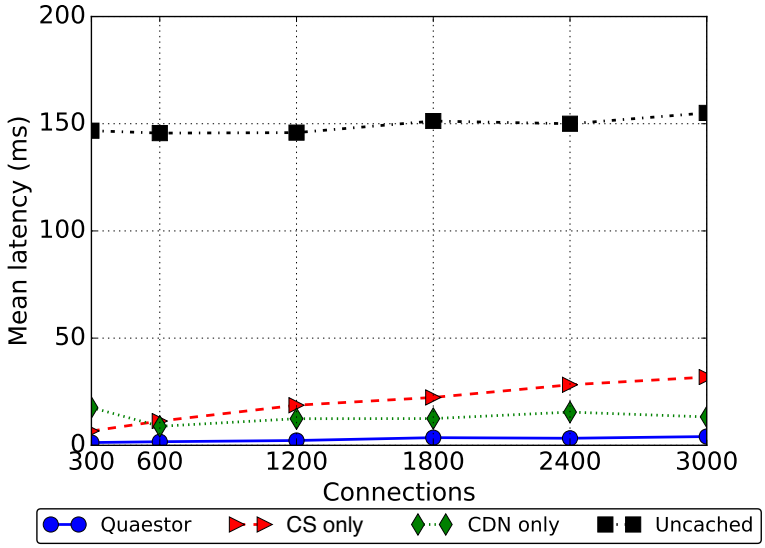


Figure 4.23: Query latency for a varying number of parallel connections comparing cached to uncached database access.

proves initially and remains constant afterwards, because separate clients access the same CDN edge.

It is important to note that the relation between query and read latency depends not only on access distributions, but also on how query predicates "cover" the space of primary keys with respect to the concurrent update operations. That is, if most queries select a key that is also frequently updated, invalidations and thus latency increase. In this workload, query predicates were selected uniformly over the primary keys, but not all primary keys were necessarily covered. With increasing query count, updates are more likely to trigger invalidations, which we demonstrate in the following by varying the number of queries executed by clients.

Varying Query Count

Scalability with regard to query count is governed by the provided InvaliDB configuration (which scales linearly, as shown in Section 4.7.4). We demonstrate the effect of increasing query counts with regard to average request latency and cache hit rates for the same InvaliDB configuration used in the read-heavy workload (8 InvaliDB matching nodes). Figure 4.24 shows how both read and query request latencies are affected by an increasing query count. Read latency improves, because a larger portion of keys is part of a cached query result. When queries that are cached as ID-lists, all records in a result are inserted into the cache as individual entries, thus causing read cache hits by side effect. This improves read latency from initially 20 ms to a mean read latency of 15 ms. The average query latency increases to slightly above 10 ms for larger query counts due to decreasing cache hit rates at the client, as shown in Figure 4.25. Cache hit rates at the CDN are comparably stable, since the concurrent client instances cause sufficient cache hits by

side effect for each other. Ultimately, Quaestor's performance for increasing query counts depends more on the popularity of individual queries and the update rate than on the total number of queries.

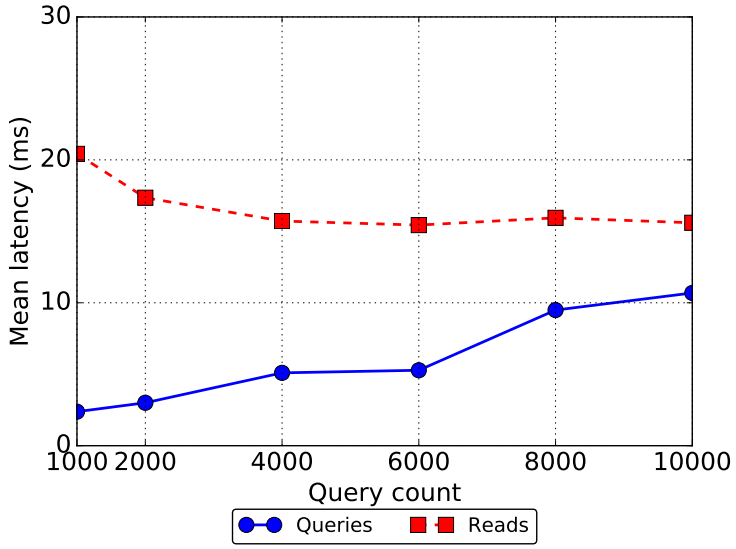


Figure 4.24: Mean latency for reads and queries for different numbers of total queries.

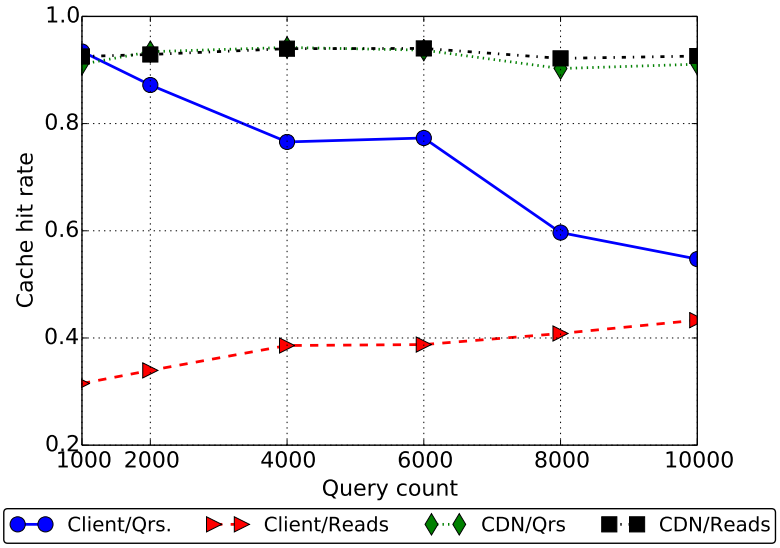


Figure 4.25: Read and query cache hit rates at the client and CDN for different numbers of total queries.

Varying Write Rates

Read-dominant workloads naturally lend themselves to caching, since they allow higher consistency, longer TTLs, fewer invalidations, and less database load. With increasing update rates, throughput is limited by the database. We demonstrate how cache hit rates degrade by increasing update rates (keeping equal read and query rates) in Figure 4.26.

Only 1 200 connections were used to avoid being limited by the write throughput of the MongoDB cluster. Client cache hit rates for both records and queries decrease predictably with increasing update rate. Figure 4.26 shows how staleness (Cache Sketch refresh interval) can be used to mitigate performance degradation in write-heavy scenarios. Notably, the refresh interval has only little impact on cache hit rate degradation. There is no linear correlation between increasing refresh rate and lower latency on higher write rates, because increasing write rates also leads to lower TTLs. Hence, increasing Cache Sketch refreshes above a certain threshold only leads to more staleness without improved client performance.

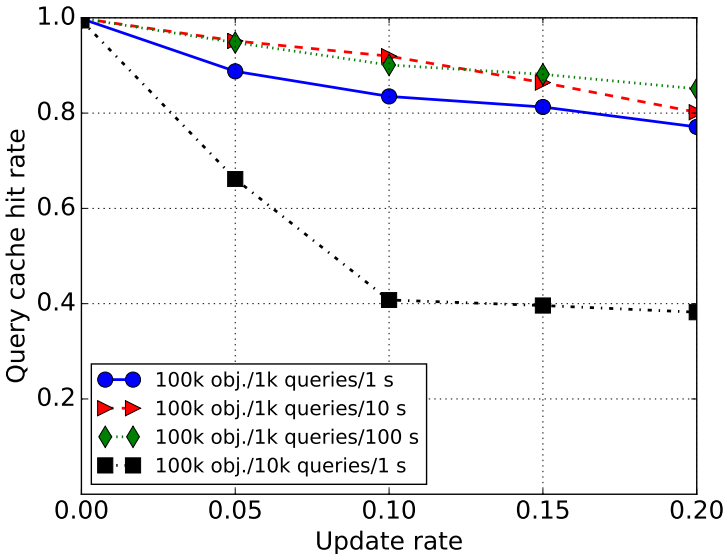


Figure 4.26: Client cache hit rates for queries with varying update rates for different Cache Sketch refresh intervals. The labels indicate the respective number of total objects and queries, as well as the refresh interval.

Varying Object Count

Finally, we investigate Quaestor's performance for varying object counts. Table 4.3 compares latencies for different database sizes indicated by the number of objects. Each collection contains 10 000 objects and is accessed by 100 distinct queries. We increased experiment durations to 600 s and changed the Zipf constant to 0.99 to account for the fact that caches take significantly longer to fill up with increasing object and query counts. Results show that for very small databases and distributions with high Zipf constants, reads and writes concentrate on the same few objects and thus limit cache hit rates. For increasing database sizes, caches take longer to fill up and TTLs have to be adjusted upwards, thus limiting performance during experiments. Nonetheless, query latencies remain below 35 ms, while read latencies slightly suffer from low cache hit rates for the (relatively) short duration of the experiment for higher numbers of total objects.

Objects	Queries	Queries	Reads
10 000	100	13.8 ms	70 ms
100 000	1 000	5.5 ms	40.2 ms
1 million	10 000	11.9 ms	27.2 ms
10 million	100 000	34.8 ms	133 ms

Table 4.3: Average query and read latency for increasing object counts for a request distribution with Zipfian constant 0.99.

Production Results

Baqend currently hosts a range of production applications and has delivered performance improvements to numerous websites. As an example we report the results of the e-commerce company Thinks. While being featured in a TV show with 3.5 million viewers, the shop had to provide low latency to potential customers. By relying on Orestes to cache all static data (e.g., files) and dynamic query results (e.g., articles with stock counters) the website achieved sub-second loads while being requested by 50 000 concurrent users (>20 000 HTTP requests per second). The business effect was measurable: the shop achieved a conversion rate of 7.8%, which is roughly 3 times above the industry average [Cha17]. Usually, such a request volume requires massive scale in the backend. However, since the CDN cache hit rate was 98%, the load could be handled by 2 DBaaS servers and 2 MongoDB shards.

4.7.3 Simulation-Based Evaluation of Query Caching

In the following, we analyze client-side staleness, TTL Estimation, and the decision model through Monte Carlo simulation of Quaestor.

The EC2-based evaluation showed Quaestor under maximum load, using relatively few client instances with many parallel connections. To analyze staleness and the decision model, we use a more web-typical configuration of many clients (100) with fewer HTTP connections per client (6) in the simulation. The simulation detects staleness (i.e., any violations of linearizability [GLS11]) in the client caches and the CDN. Client-side staleness is bounded by the Cache Sketch refresh interval. Upon every Cache Sketch renewal, clients revalidate stale cache entries identified by the filter. CDN staleness is primarily governed by invalidation latency. In our experiments, CDN staleness was constantly below 0.1% with 100 ms mean invalidation latency (10 ms variance).

A stale read at the client cache occurs, if the version read from a cache v_r is older than the last acknowledged write version v_w (possibly written by another client). A stale read at the CDN occurs when a write v_w has been acknowledged, but the corresponding record has not been invalidated from the CDN, yet, and a later read retrieves version v_r older than the written version ($v_r < v_w$).

Figure 4.27 illustrates the relationship between Bloom filter refresh rate and client staleness. Staleness initially increases fast between 1 s and 10 s refresh rate, but is limited by

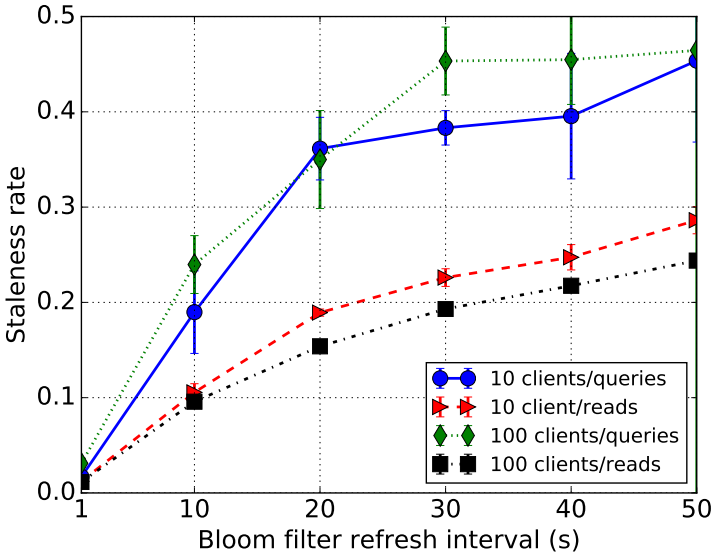


Figure 4.27: Stale read and stale query rates for 10 and 100 clients with different refresh intervals.

two factors for higher refresh intervals. First, every time a client begins an update operation it invalidates the corresponding record from its own cache. Second, client staleness rates are limited by cache hit rates, which were up to 60% for records and up to 95% for queries in the benchmark, thus explaining the difference between record and query staleness.

TTL Estimation

We also used simulation to compare our query TTL estimation scheme against the *true TTL* for every query, which we define as the time period a query could have been cached until invalidation. Figure 4.28 shows the cumulative distribution functions (CDFs) for estimated and true TTLs for a 1% write rate for 10 minutes. The CDF comparison shows the expected result of having a similar distribution for the majority of TTLs and larger errors on the unpredictable long tail of the access distribution. Due to the learning phase and the bias of initial estimates, there is a systematic underestimation that could be tuned by adjusting the EWMA constant α of the TTL estimator. As query caching assumes that relevant queries are executed often and by many clients, the high accuracy on the short tail is a highly relevant property of the TTL estimation scheme.

Decision Model

The decision model enables dynamic representations for query results that depend on result size, the number of connections, and different matching events. Object-lists offer minimal latency due to a single round-trip, ID-lists offer better cache utilization due to not requiring invalidation on change events. As discussed in Section 4.6.3, the bias w allows expressing a preference over the representation. We analyzed the advantage of using

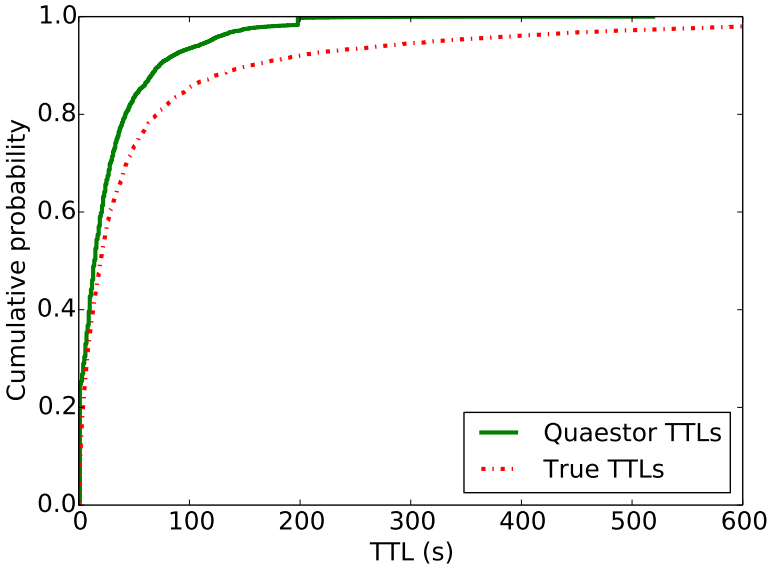


Figure 4.28: CDF of the query TTL estimation scheme compared with the CDF of the *true* TTL as measured in the simulation.

ID-lists in a setting with 10% write rate where many invalidations could be avoided. The caching decision was evaluated by introducing a probability p of changing an attribute that is not part of the query predicate and hence only causes change events, thus decreasing invalidations for ID-lists with increasing p.

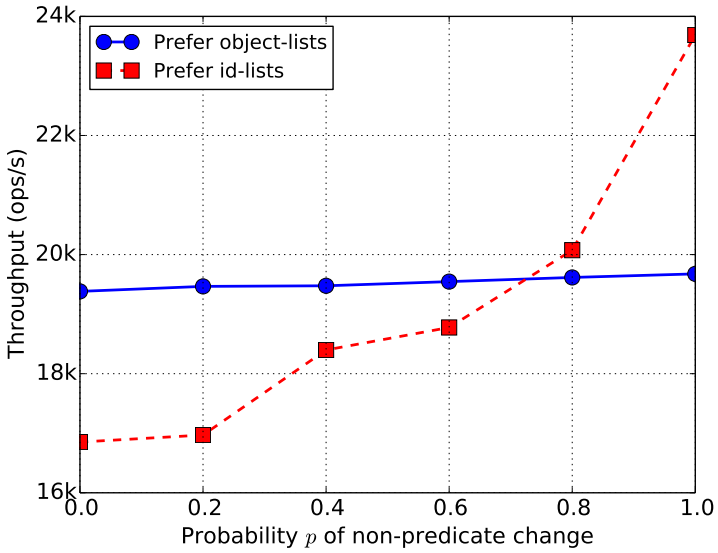


Figure 4.29: Throughput of object/ID-lists for an increasing probability of non-predicate changes for 100 simulated clients (600 connections total).

Figure 4.29 illustrates that object-lists are a better choice for workloads where fields in the query predicate are changed, while ID-lists gain considerably from avoiding invalidations on non-predicate changes. Figure 4.30 compares query staleness for increasing p, which demonstrates that ID-lists can reduce staleness considerably. We also observed a slight

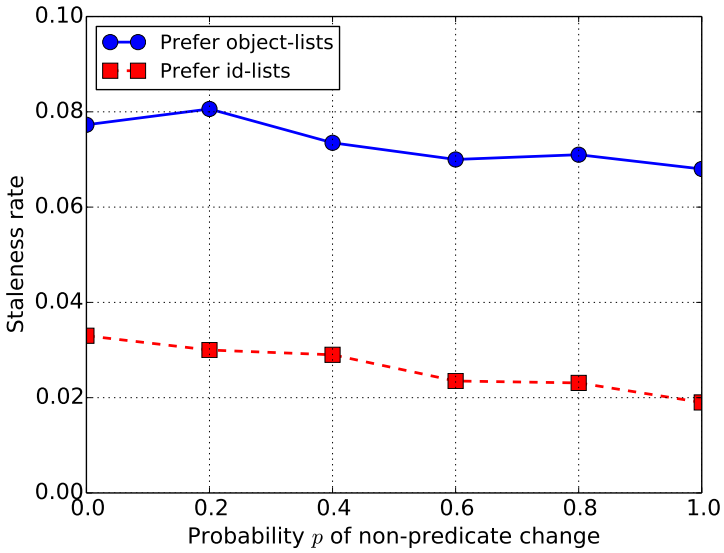


Figure 4.30: Query staleness for object/ID-lists. At higher probability p of non-predicate changes, ID-lists avoid more invalidations and thus achieve lower staleness.

performance increase for higher p when using object-lists, which is caused by the lower probability of non-predicate changes causing multiple invalidations.

4.7.4 InvaliDB

To demonstrate the scalability of our real-time matching approach, we measured sustainable matching throughput and match latency for differently sized InvaliDB deployments on Amazon EC2. Our test setup comprised one client machine, one Orestes server, one Redis server, and an InvaliDB cluster. As a baseline, we evaluated the InvaliDB deployment with only a single node for query matching and then doubled both the number of active queries and the number of matching nodes with every subsequent experiment series. Every deployment had a single node dedicated to query and change stream ingestion. The Redis server hosting the message queues for communication between InvaliDB and the Quaestor server as well as all InvaliDB nodes were `c3.large` instances with 2 vCPUs (Xeon E5-2680 v2, Ivy Bridge) and 3.75 GB RAM each. Every matching node hosted two separate matching task instances (one per vCPU) on two separate JVMs. The Quaestor server was a `c3.xlarge` instance and did not become a bottleneck.

Workload

For every InvaliDB configuration, we performed a series of experiments, each of which consisted of two phases: In the **preparation phase**, any still-active queries from earlier experiments were removed and queries for the upcoming one were activated. In the subsequent 2-minute **measurement phase**, the client machine performed 1 000 insert operations per second against the Quaestor server and measured notification latency as the

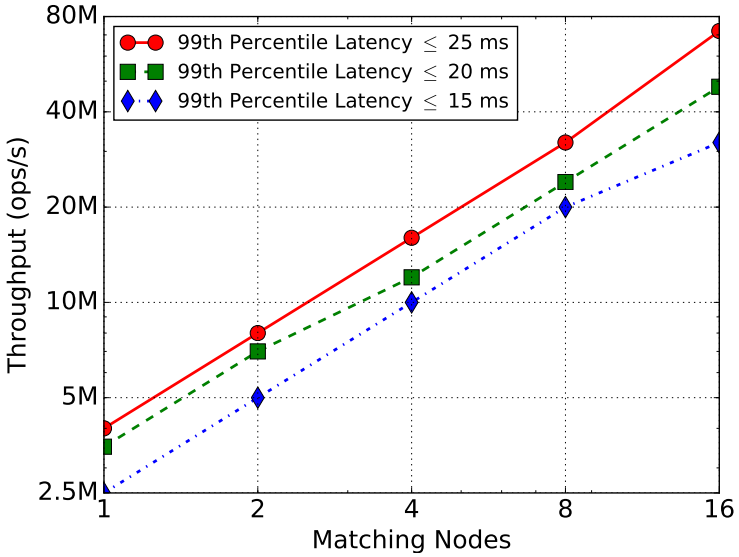


Figure 4.31: Notification latency as a function of the executed matching operations per second for InvaliDB clusters employing 1, 2, 4, 8, and 16 matching nodes (figure taken from [GSW+17]).

difference between the timestamp of notification arrival and of the point in time directly before sending the corresponding insert statement.

We chose the same constant update throughput of 1 000 inserts per second for all experiment series, but varied the number of active queries relatively to the number of matching nodes in each InvaliDB cluster, so that all clusters were exposed to the same **relative load**. We started each experiment series with 500 active queries per node and increased their number by the same amount until the system was saturated and incoming operations started queueing up. Thus, the cluster with only 1 matching node started with 500 active queries, whereas the 16-node cluster started with 8 000.

Evaluation

To demonstrate the efficiency and scalability of InvaliDB, we measured notification latency under increasing load for 5 InvaliDB clusters employing between 1 and 16 matching nodes. The line plot in Figure 4.31 illustrates that matching throughput scales linearly with the number of matching nodes, even under tight latency bounds. Please note that we plotted the number of active queries on a logarithmic scale for better readability, since both the number of matching nodes and the number of active queries and thus overall system load were doubled with every experiment series.

All clusters achieved 99th percentile latencies below 20 ms up to 3 million and below 30 ms up to 4 million ops/s per node, while huge latency spikes marked system capacity at roughly 5 million ops/s per node. Peak latencies never exceeded 100 ms under load of 3 million ops/s per node or less. Table 4.4 shows that all clusters displayed almost identical latency characteristics under an identical relative load of 3 million ops/s per node and thus

demonstrates that InvaliDB provides predictable low latency, irrespective of the number of matching nodes in the cluster.

		1 node	2 nodes	4 nodes	8 nodes	16 nodes
	ops/s	3M	6M	12M	24M	48M
latency (ms)	avg	8.5	8.8	8.9	8.8	8.5
	std. dev.	2.6	2.7	2.8	3.5	2.7
	95 perc.	13.0	13.0	13.0	13.0	13.0
	99 perc.	15.9	16.0	17.0	18.0	16.0
	max	38.0	65.0	68.0	84.0	58.0

Table 4.4: Details on the latency characteristics of the different InvaliDB clusters at 3 million matching operations per second per node (data taken from [GSW+17]).

4.7.5 Evaluation Summary

In summary, we have shown that Quaestor can provide substantial, scalable query latency improvements which are primarily governed by write rates. In particular, we have shown that the combination of the Cache Sketch and CDNs provides a super-linear performance increase that cannot be achieved by either one of the components. Up to tenfold performance improvements were obtained for different workloads by caching queries and objects in the CDN and in clients. Further, we showed that the Cache Sketch was able to achieve low stale reads rates with arbitrary fine-tuning of the latency-staleness trade-off available to the application. Our result confirmed that the query TTL estimation scheme converges towards the correct TTL for frequent queries and the decision model exploits the workload to optimize cache hit rates through an optimal query representation. We also demonstrated the feasibility of scalable invalidation detection for queries at consistent low latency, irrespective of cluster size. Sustainable throughput was measured above 4 million ops/s per EC2 `c3.large` instance and scaled linearly with the number of instances.

4.8 Cache-Aware Transaction Processing

As derived in the previous sections, the Cache Sketch approach offers tunable consistency guarantees that applications can choose at granularity down to single reads and queries. However, grouping multiple operations into a transaction for correctness is an indispensable capability of various data management problems [SSS15, WV02, BN09]. In Section 2.2.7, we identified latency as a pivotal problem for abort rates of both optimistic and pessimistic transactions. The central observation was that the transaction runtime due to external aborts is directly linked to the latency of operations performed in the context of that transaction.

We identified multiple challenges for distributed transaction processing that Orestes can address:

1. For distributed environments, the **latency of transactional operations** has to be tackled as it drastically affects both pessimistic and optimistic concurrency control.
2. State-of-the-art concurrency control schemes do not support the use of web caching. A **cache-aware transaction scheme** that enables the use of expiration-based and invalidation-based caches is missing.
3. Most current NoSQL database systems offer superior scalability and performance for web applications, but lack the refined semantics of ACID transactions. Therefore, **polyglot persistence transactions** should enable ACID guarantees as an opt-in feature without compromising other system properties.
4. Latency problems usually arise due to the distribution of users. Traditional transaction APIs assume access from application servers in a three-tier architecture, but **transactional access in BaaS** applications has not been addressed.

In this section, we will extend the single-object concurrency abstractions of Orestes (conditional and partial updates described in Section 3.5.7) to general-purpose ACID transactions with low latency[19]. While the identified problems exist for both optimistic and pessimistic transactions, the challenge of cache-awareness rules out pessimistic strategies as a potential solution (cf. Section 2.2.7). First, we will therefore quantify the problem of aborts in optimistic transactions through a stochastic analysis of rollback probabilities and transaction runtimes.

4.8.1 The Abort Rate Problem of Optimistic Transactions

In order to motivate the benefits of low read latency in transaction processing, we analyze the abort probability of optimistic transactions in a theoretical model (cf. [Wit16]). The drawback of optimistic concurrency control is that long-running transactions accessing many objects have a high abort probability. The main factors determining the abort probability are [GHKO81]:

1. The **runtime of the transaction** and the related size of the read set
2. The **update frequency** of accessed objects
3. The **access distribution** for reads and writes (e.g., Zipfian or uniform)

The **abort probability** of non-cached optimistic transactions is determined by the update probabilities of the objects in the read set. For a baseline analysis, we consider updates to be uniformly distributed over a set of N objects and performed at a constant rate r. Time is discretized into steps of length s (e.g., 1 ms). The question whether an object is updated in a certain time-step is a Bernoulli trial with a success probability of $p = \frac{r}{N}$. The

[19] This section is based on results obtained in a master thesis by Witt, supervised in the context of this thesis [Wit16].

corresponding binomial experiment $Y \sim B(d,p)$ expresses the probability that k objects are updated in l time steps:

$$P(Y=k) = \binom{l}{k} p^k \cdot (1-p)^{l-k} \tag{4.8}$$

$$P(Y=0) = (1-p)^l = \left(\frac{N-r}{N}\right)^l \tag{4.9}$$

We further assume that a transaction successively reads objects with a uniform popularity distribution over the same set of N objects and that a read operation takes l time steps. Thus, l represents round-trip latency of reads. Each transaction reads n objects. The transaction model is illustrated in Figure 4.32.

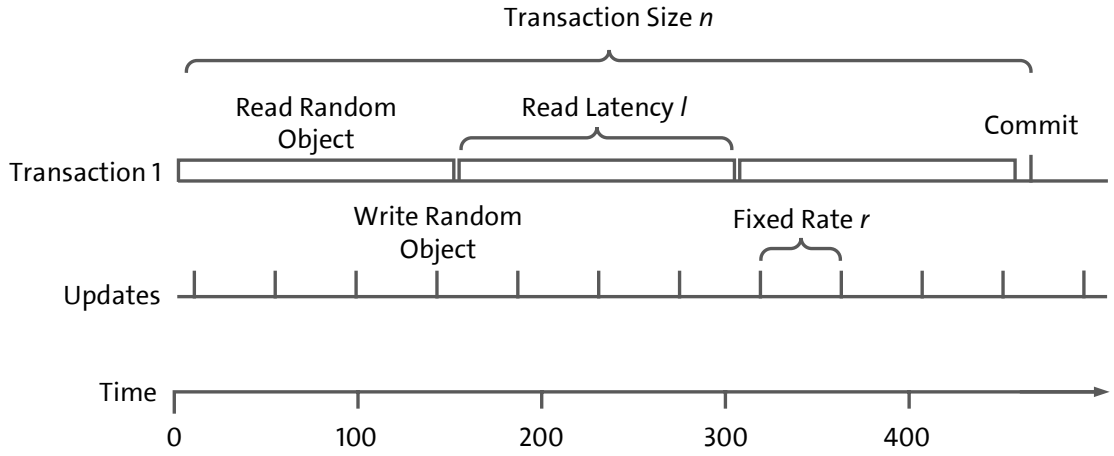


Figure 4.32: The transaction model parameters for the stochastic analysis.

Intuitively, the ith read has a vulnerability window of $(n-i+1)\cdot l$ time steps in which it can be overwritten and hence cause the overall transaction to abort[20]. The corresponding conflict probability for the ith read is $P(Y=0)^{(n-i+1)\cdot l}$. For example, the first read has a probability of $P(Y=0)^{n\cdot l}$ to cause the whole transaction to abort. As before, the random variable A describes the outcome of a transaction (abort or commit). The probability of a successful commit for a transaction is:

$$P(A=0) = \prod_{i=1}^{n} P(Y=0)^i = \prod_{i=1}^{n} \left(\frac{N-r}{N}\right)^{i\cdot l} = \left(1-\frac{r}{N}\right)^{\frac{1}{2}l\cdot n\cdot(n+1)} \tag{4.10}$$

Figure 4.33 shows the abort probability as a function of the transaction size, update rate, and read latency. It is clearly visible that lower read latency leads to drastically reduced abort rates.

[20]Without loss of generality, we assume reads to logically happen at the begin of an invocation. Shifting the linearization point of the read between invocation and response only changes the result by a fixed factor and can therefore be ignored.

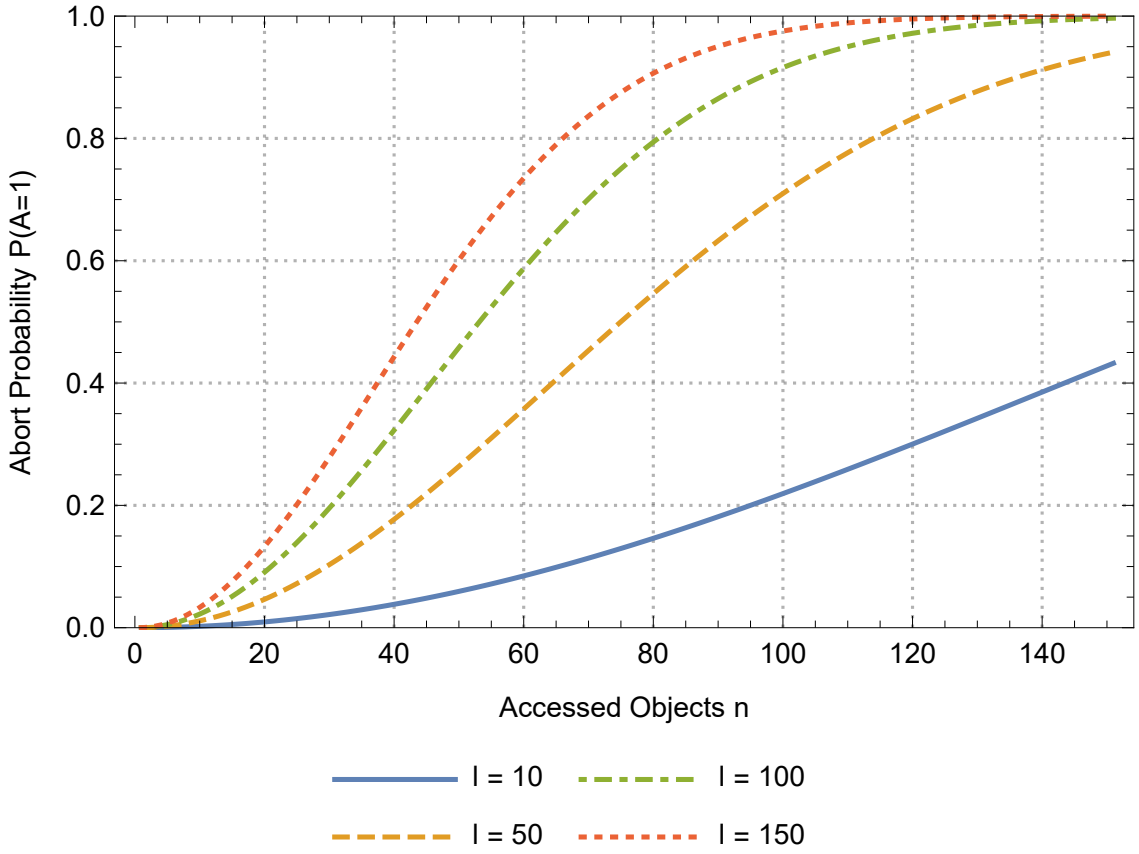


Figure 4.33: The abort probability for an increasing number of accessed objects at different read latencies l for $N = 10000$ objects in the database, time steps of $s = 1ms$, and 50 writes per second ($r = 0.02$).

Transaction Duration with Retries

Most applications require a failed transaction to be retried, after rollback due to an external abort. The number of aborts R before a successful commit is described by a geometric distribution:

$$P(R = k) = P(A = 0) \cdot P(A = 1)^k \tag{4.11}$$

The geometric distribution yields the expected number of aborts $E[R]$:

$$E[R] = \frac{1 - P(A = 0)}{P(A = 0)} \tag{4.12}$$

The duration of a transaction execution is $t = l \cdot n$. Let $T = (R+1) \cdot t$ be the random variable that describes the duration of a transaction including retries. Due to linearity of expected values, $E[T]$ can be derived as follows:

$$E[T] = (E[R] + 1) \cdot l \cdot n = \left(\frac{1 - P(A = 0)}{P(A = 0)} + 1 \right) \cdot l \cdot n \tag{4.13}$$

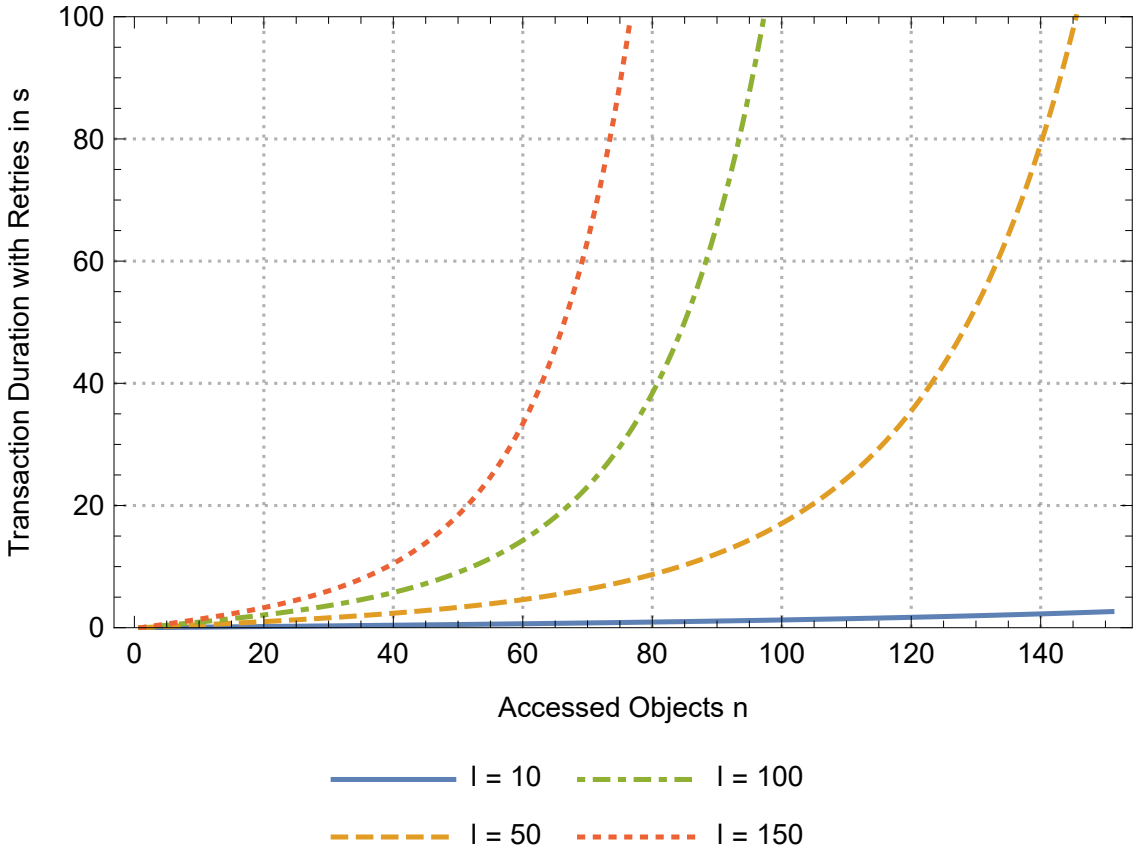


Figure 4.34: Transaction runtime with retries for an increasing number of accessed objects at different read latencies l for $N = 10000$ objects in the database, time steps of $s = 1$ ms, and 50 writes per second ($r = 0.02$).

Figure 4.34 shows the transaction duration including retries for the same parameters as the abort analysis (see Figure 4.33). If the application requires retries of externally aborted transactions, the impact of read latency is thus even higher.

The above discussion and analysis reveal that neither pessimistic nor optimistic protocols are well-suited for long-running transactions and thus high-latency environments. As the system can control neither the update frequency, access distribution, nor read set size, the only way to reduce aborts is to minimize transaction duration by improving the latency of each operation.

4.8.2 DCAT: Distributed Cache-Aware Transactions

We introduce distributed cache-aware transactions (DCAT) as an extension to Orestes that uses the Cache Sketch approach to reduce latency and thus abort rates. The algorithm is based on backward-oriented concurrency control, since in contrast to other pessimistic and optimistic schedulers, it allows integrating web caches into the transaction processing (see Section 2.2.7).

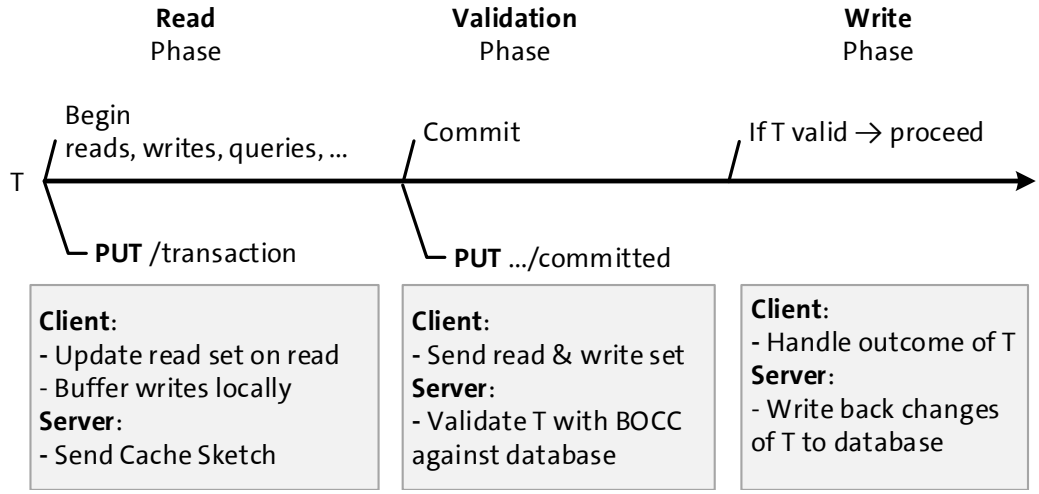


Figure 4.35: The three phases of an optimistic DCAT transaction.

Latency of interactive transactions is almost exclusively governed by the number of round-trips between client and server. DCAT minimizes round-trips by serving read operations from nearby caches and buffering write operations at the client. In the ideal case, a transaction requires two round-trips (fetching the Cache Sketch and committing the transaction), which is independent of the number of accessed objects.

The two operations that DCAT exposes towards the client via REST are `begin` and `commit` in order to define the transaction boundaries. The three phases of a DCAT transaction are depicted in Figure 4.35. The overall transaction structure is similar to BOCC [KR81], and respectively the variant BOCC+ [Rah88] that is based on version numbers to prevent false aborts. In the read phase, the client signals the start of the transaction to the server which in response sends the newest Cache Sketch. During the read phase all writes in the client are buffered locally and all read versions are tracked in the read set. If the transaction aborts internally, the client simply discards its local state. To commit, the read and write set are transferred to the server, which uses this information to validate whether the transaction can commit or whether it has to be aborted (validation phase). If the validation succeeds, transactional changes are made permanent and visible in the database during the write phase.

The detailed transaction scheme involving processing steps in the client and server is shown in Figure 4.36. The steps of the transaction are:

1. The client initiates a transaction with the begin operation which fetches the Cache Sketch from one (of potentially various) Orestes servers and creates a transaction context to handle further operations.
2. Read operations and queries during the transaction are preferably served from caches, using the fresh Cache Sketch to bound the staleness to the time elapsed since transaction begin. This mechanism is essential to DCAT, as the Cache Sketch conceptually provides an up-to-date snapshot view on all caches linked to the transaction begin.

3. Write operations are buffered in the transaction context and not sent to the server. Instead, the write set is applied atomically in the write phase, depending on the successful validation.
4. The commit operation sends the read and write sets to one of the Orestes servers for it to handle the commit process. This requires the commit procedure to be stateless, so that any server can process the validation and write phase for a committing transaction. We achieve statelessness through a decoupled coordination service that runs independently from the Orestes servers.
5. The Orestes server is responsible for validating transaction read sets and applying write sets to the database. Following BOCC+, the server has to ensure mutual exclusion of the commit process between overlapping transactions. Therefore, it acquires coordination locks in the coordinator.
6. Once the critical section of the commit process is entered, the validation procedure compares the versions from the read set to the current database state. The validation is successful, if all versions coincide. After successful validation, the write set is applied to the database and coordination locks are released. An unsuccessful validation results in a transaction abort. In this case, the write set is discarded and the coordination locks are released afterwards.
7. The server returns the result of the transaction to the client. If the transaction failed, the client can decide to re-execute the transaction by starting at step 1.

4.8.3 Server-Side Commit Procedure

The commit procedure has to ensure isolation of transactions by testing for any violations of serializability that may have arisen from concurrent transactions or cached reads. As most NoSQL systems do not support a native transaction concept (cf. Chapter 3), a delib-

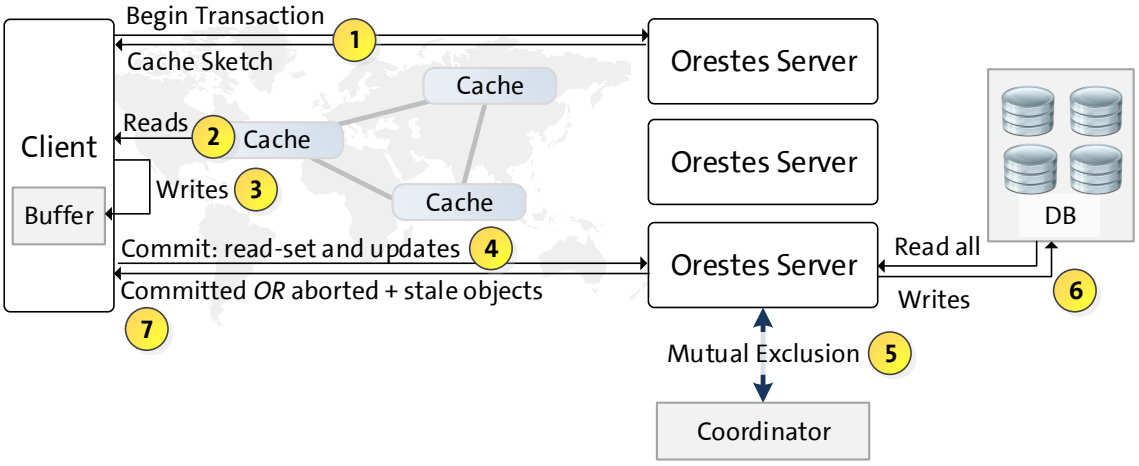


Figure 4.36: DCAT transaction concept and the steps involved at the client and server.

erate design choice was to build on abstractions in the Orestes middleware[21] in order to be applicable to many data stores. The only requirement towards the database system is the option for linearizable read operations. This is necessary for correctness when comparing versions of the read set to the current state of the database.

Algorithm 3 shows the DCAT commit procedure. First, validation locks are acquired in the coordinator to prevent validation of transactions that access the same objects (line 2). Our prototype implementation of the coordinator is based on a single-server Redis, but could be scaled easily, as validation locks may be hash-sharded across multiple servers. Next, the commit procedure ensures that the *bucket-level permissions* for all buckets accessed in the write set are met for the user executing the transaction (line 3). Similarly, the server-side *FaaS business logic* is executed to check whether any of the individual handlers aborts the transaction (line 4).

Algorithm 3 DCAT commit procedure.

```
1:  procedure COMMIT(tid, readSet,writeSet)→{COMMITTED,ABORTED}
2:      coordinator.acquireLocks(tid,readSet,writeSet)
3:      if !db.hasBucketPermission(tid,bucket) for each bucket ∈ writeSet then return ABORTED
4:      if db.executeBusinessLogic(writeSet) raises Exception then return ABORTED
5:      for (id,version) ∈ readSet⋃writeSet do          ▷ Check for stale reads and conflicts
6:          if db.get(id).version ≠ version then return ABORTED
7:          if !db.hasObjectPermission(tid,id) then return ABORTED
8:      coordinator.log(tid,readSet,writeSet)                          ▷ Ensure durability
9:      db.update(writeSet)                                ▷ Make private changes visible
10:     coordinator.releaseLocks(tid,readSet,writeSet)
11:     return COMMITTED
```

The main validation step verifies that the version is still up-to-date for each object that was read or written by the transaction [Rah88] (lines 5 to 7). If one of the objects is stale, a conflict occurred and a rollback of the transaction is necessary. Similar to other globally distributed, transactional systems (e.g, Spanner [CDE+13]), the validation accepts schedules that are in the class of commit order-preserving conflict serializable (COCSR) schedules, also known as external consistency [Coo13] or strict serializability [WV02]. In particular, this means that some conflict serializable (CSR) schedules are not accepted by the DCAT commit procedure[22]. Besides the version check, it is also ensures that the transaction possesses the object-level permissions required to access each of the objects.

After the validation steps, durability is ensured by logging the transaction as committed (line 8). Afterwards, all updates from the write set are applied to the database and the locks can safely be released as the transaction is committed (lines 9 to 11).

[21] For systems with transaction support, the commit procedure can make use of short-lived validation transactions as an alternative to the BOCC procedure in Orestes [BBB+17].

[22] As an illustrating example, consider two transactions t_1 and t_2 producing the interleaved schedule $b_1 b_2 r_1(x) r_2(x) w_2(x) c_2 w_1(y) c_1$ (for notation see [WV02, Ch. 3]). There is only one conflict from t_1 to t_2 on object x. While this schedule is in CSR, it would be aborted by DCAT as the validation would detect that t_1 read an outdated version of x, hence violating COCSR.

In summary, the DCAT commit procedure validates access rights and serializability and applies successful transactions to the database using a mutual exclusion mechanism provided independently from underlying database systems. As servers can crash at any time, a recovery mechanism is required to achieve fault tolerance.

Recovery

To enable recovery of successfully validated transactions that failed during their application to the database, read and write sets are logged. The **Roll-Forward Recovery** [PV97] uses these log entries to apply the write set to the database and release the locks. Once the log entry for a transaction is persisted, the transaction will commit eventually (without requiring *rollbacks*). A crashed transaction can, however, block all overlapping transactions until its recovery. The recovery procedure can be triggered by two mechanisms. First, any transaction that tries to commit and is unable to obtain the respective validation locks, will retry the lock acquisition. Upon repeated failure, the recovery process is activated, to check whether a transaction crashed during validation and has to be completed from its log entry. This allows the blocked transaction to proceed. Second, the recovery periodically scans the log for pending writes to proactively finish any orphaned transactions.

Correctness

DCAT generates only schedules that are in COCSR (correctness) and is able to produce any valid COCSR schedule (completeness). COCSR is a restriction of CSR demanding that if an edge from n_i to n_j exists in the conflict graph (a conflict from t_i to t_j), then t_i must commit before t_j [WV02, p. 102f]. While COCSR accepts fewer schedules, it is particularly useful for distributed data management. If local transactions on a cluster executing with COCSR are composed in a global transaction, that global transaction is conflict serializable [WV02, p. 678]. This makes it straightforward to compose multiple DCAT transactions to global transactions, for example in the context of polyglot persistence.

We prove **correctness** and **completeness** of DCAT regarding COCSR in Theorem 4.3 and Theorem 4.4.

Theorem 4.3. *Let S be the set of schedules generated by the DCAT commit procedure. Every produced schedule is in COCSR: $S \subseteq COCSR$ (correctness).*

Proof. Consider a produced schedule with two transactions t_i and t_j ($i \neq j$) that exhibit a conflict from t_i to t_j on object x, both completed successfully with t_j committing before t_i (violation of COCSR). As the coordinator ensures mutual exclusion, t_i and t_j cannot have been validated concurrently, as otherwise they would not be in conflict. Therefore, t_j must have been validated first, since by assumption it committed first. As there is a conflict from t_i to t_j on object x, either

(i) t_i has written x that was later read by t_j.

(ii) t_i has written x that was later written by t_j.

(iii) t_i read x that was later written by t_j.

By construction of the DCAT protocol, (i) and (ii) are impossible, as writes of t_i are kept locally until it has committed, but t_j committed first. t_j can therefore neither read nor overwrite an x updated by t_i. For case (iii), t_i must have read x in version k and t_j must have updated x to version $n > k$. However, in that case, validation of t_i in Algorithm 3 on line 5 will abort, as the commit procedure observes the outdated version. Thus, all three potential causes of the conflict are impossible (proof by contradiction). □

Theorem 4.4. *Let S be the set of schedules generated by the DCAT commit procedure. There is no schedule in COCSR that is not produced by DCAT: $S \supseteq COCSR$ (completeness).*

Proof. DCAT aborts a transaction t_i, iff t_i read or wrote an object x in version k that a second transaction t_j updated to version $n > k$ afterwards (conflict t_i to t_j on x). This implies that t_j has committed before t_i, as otherwise its write would not be visible to t_i. A schedule with a conflict from t_i to t_j where t_j has committed before t_i is not in COCSR according to its definition [WV02, p. 102f]. Therefore, DCAT only aborts schedules that are not in COCSR and is thus complete. □

With the set of schedules produced by DCAT being both complete and correct regarding the serializability class COCSR, DCAT provides the full set of ACID guarantees:

Atomicity. DCAT transactions are either applied completely or not at all. This is ensured by the logging step performed before any objects are written. If the log entry was written completely, the transaction will eventually become visible with its complete write set. The recovery process guarantees that only if all of the transaction's writes are applied, other transactions can obtain validation locks for access to the same objects. If the log was not written completely, the transaction is not applied at all. As no undo recovery is necessary, atomicity cannot be affected by potential cascading aborts [WV02].

Consistency. DCAT does not currently support *automatic* or *declarative* consistency constraints (e.g., uniqueness and referential integrity). Therefore, we only consider *logical consistency* [SSS15, p. 10] which is a property in the responsibility of application developers. A logically consistent transaction produces a consistent database state, if it is based on a consistent state at transaction begin and both isolation and atomicity are guaranteed for its execution. Since isolation and atomicity are provided, logically consistent DCAT transactions will keep the database consistent.

Isolation. As proven in Theorem 4.3 and Theorem 4.4 DCAT ensures isolation of transactions by only allowing schedules that are in COCSR. As the DCAT validation operates on object level, it is able to detect all concurrency anomalies that arise from serializability violations on single objects: dirty writes, dirty reads, lost updates, non-repeatable reads, as well as read an write skew. However, since queries are only tracked in the form of the specific objects returned in the produced result set, the

phantom problem can occur. For example, if a transaction queries all bank accounts that were created in 2018, a concurrent transaction could insert a new matching account object without causing a conflict in validation. In future work, DCAT can be extended to queries, by explicitly tracking them in the read set in order to validate them in the commit procedure.

Durability. DCAT transactions are durable, once their write phase has completed. Durability is ensured by the underlying database and an option available in almost any database system.

The major non-functional properties of DCAT are low latency, elastic scalability, database independence, and fault tolerance. **Low latency** is achieved by leveraging the Cache Sketch approach for transactional reads and queries. In the best case, all operations during a transaction can be answered by web caches, so that only the begin and commit require a client-server round-trip. As the stochastic analysis revealed (cf. Section 4.8.1), read latency is the determining factor for the abort probability, which implies that DCAT can drastically increase the success probability of transactions. **Elastic scalability** is enabled by building on the distributed Orestes architecture (cf. Chapter 3). Each Orestes server can handle transaction commits. The only potential scalability bottleneck is the coordinator that therefore can be partitioned over the keyspace of objects. As Orestes relies on range- and hash-sharded NoSQL database systems, scalability of the writing commit procedure is also given. The coordination mechanism is decoupled from the database and only requires linearizability. Thus, DCAT is **database-independent**. The **fault tolerance** model is inherited from Orestes: according to the CAP theorem [GL02], certain networks partitions render DCAT unavailable, as linearizability is required[23]. However, DCAT is fault-tolerant with respect to server failures through its recovery mechanism and it increases read availability by allowing reads and queries to be answered from caches.

4.8.4 Cache-Aware RAMP Transactions

We have furthermore investigated the use of Read Atomic Multi-Partition (RAMP) transactions [BFG+16] in combination with DCAT as an optimization to reduce abort rates of read transactions and to further improve latency as well as throughput.

RAMP transactions "enforce atomic visibility while offering excellent scalability, guaranteed commit despite partial failures [...] and minimized communication between servers" [BFG+14, p. 1]. The central idea of RAMP is to provide an isolation level called *read atomic* (RA) isolation, that itself is weak, but sufficient for some applications. By combining the scalable, non-blocking RAMP protocol with the strong semantics of DCAT,

[23] Davidson et al. showed that serializability in general cannot be achieved in the presence of network partitions [DGMS85].

applications can trade transactional isolation guarantees against additional latency benefits and RAMP's guaranteed commit[24].

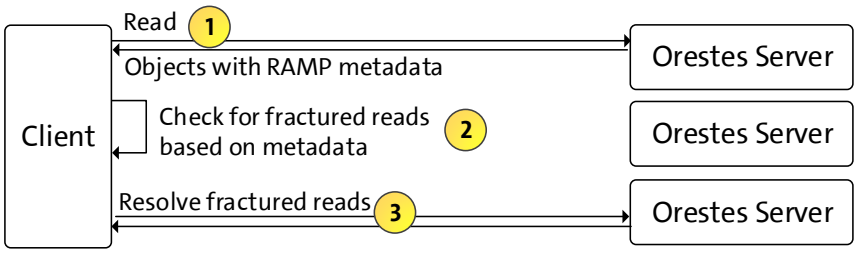


Figure 4.37: Execution steps of a read-only RAMP transaction: reading annotated objects, client-side validation, and resolution of fractured reads.

RAMP transactions guarantee read atomic (RA) isolation, which is met, if neither **fractured read anomalies** occur, nor reads of uncommitted, aborted, or intermediate data. A fractured read happens, if transaction t_i writes version x_m and y_n and transaction t_j reads version x_m and y_k with $k < n$ [BFG+14, p. 6]. Intuitively, a fractured read occurs, if only parts of a committed transaction's write set are observed. The other key properties of RAMP transactions are scalability, minimized server communication, and a guaranteed commit, i.e., RAMP transactions do not abort or block.

RAMP prevents fractured reads for multi-partition transactions by validating read operations at the client side and loading missing versions in a second round-trip as depicted in Figure 4.37. In the first step, a RAMP transaction reads the objects from the server including additional metadata stored with each object. This object metadata references all other objects that were written together with the object. The metadata is used in the second step to validate the read set for potential fractured read anomalies. If fractured reads were found, the client resolves them by requesting missing object versions from the server. Transactional writes (not shown in the figure) are applied in bulk at transaction commit. The commit procedure annotates the object metadata in a two-phase write process that contacts each accessed object partition [BFG+14].

To combine the benefits of RAMP transactions with the latency improvements and ACID semantics of DCAT, RAMP transactions are used for read-only transactions that only require read atomic isolation. DCAT, on the other hand, is used whenever full ACID semantics are needed and especially to avoid anomalies not prevented by RAMP (e.g., lost updates [BFG+14, p. 9]). The combination thus is an option to mitigate the abort rate problem for long-running read-only transactions.

The Cache Sketch scheme can be applied to RAMP transactions in order to improve read latency. Instead of contacting the server in step one (Figure 4.37), objects can be **served from caches** and validated at the client side (step two). The server only has to be con-

[24]RAMP transactions are guaranteed to commit, as they are *coordination-free*. A coordination-free execution ensures that a transaction cannot be blocked by other transactions and will commit, if the system partition of each accessed object can be reached [Bai15].

tacted for resolving potential fractured reads. In the best case, RAMP transactions combined with Cache Sketch-based caching therefore do not contact the server at all.

4.8.5 Evaluation

In order to evaluate the benefits of DCAT for both latency and transaction abort rates, we conducted a Monte Carlo simulation based on our YMCA framework (cf. Section 4.3.1) to compare optimistic BOCC+ transactions with and without caching. The simulation executes read-only transactions on a system with a baseline traffic of 50 writes and 1 500 reads per second ($\approx 95\%$ reads), distributed uniformly over a set of 10 000 database objects. As in the evaluation of Quaestor, CDN cache hits induce 4 ms latencies and cache misses 150 ms. We compare results for latency, abort rate, and runtime of a transaction, retried until their eventual success, each as a function of the transaction size.

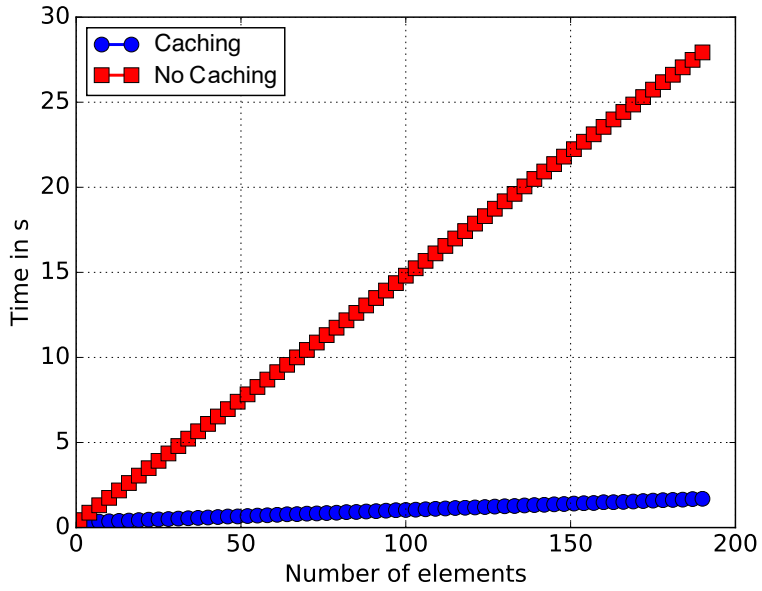


Figure 4.38: Transaction duration as a function of transaction size (data taken from [Wit16]).

Figure 4.38 shows the immense benefit caching has on transaction execution time. While standard transactions need about 150 ms for each read operation, cached transactions achieve about 10 ms per read and thus are on average 15 times faster. The performance of uncached transactions quickly becomes prohibitively high, DCAT remains usable up to a high number of objects used in a transaction.

The latency benefit has a substantial effect on the transaction abort rate as shown in Figure 4.39. The short execution time improves the abort probability while the increased risk of loading stale objects from the cache is balanced by fetching the Cache Sketch at transaction begin. Note that the empirical results correspond to the stochastic model as shown in Figure 4.33.

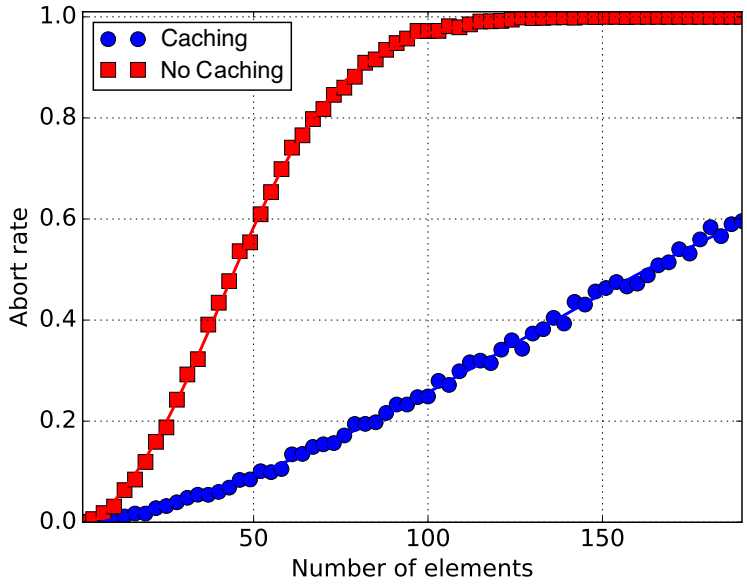


Figure 4.39: Abort rate as a function of transaction size (data taken from [Wit16]).

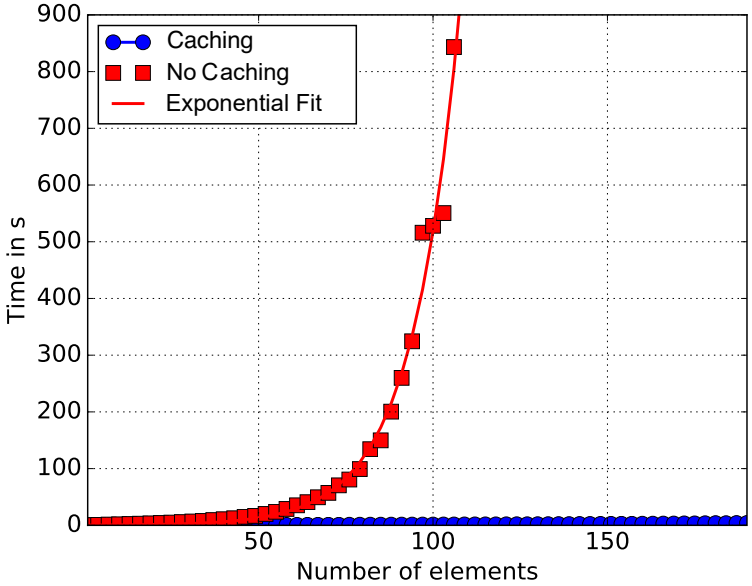


Figure 4.40: Runtime as a function of transaction size (data taken from [Wit16]).

The practical effect of the abort rate improvement is shown in Figure 4.40, which illustrates transaction runtime with retries, measured from the begin of a transaction until successful completion (i.e., commit). In this setting, clients retry failed transactions, until they commit. While non-cached transactions with more than 100 read objects practically never succeed, cache-aware transactions are much faster and succeed within a few tries.

Besides the simulation-based evaluation of DCAT, we analyzed throughput, latency, and scalability for a cloud-based Orestes deployment. The throughput of Orestes scales linearly up to a point of roughly 8 servers, where the underlying, sharded MongoDB cluster be-

comes bottlenecked due to an implementation issue in its `findAndModify` operation used for applying write sets[25]. Overall, the latency overhead compared to non-transactional read and write operations in Orestes was approximately 40%. The single-node Redis coordinator showed consistent low latency and was able to sustain a throughput of approximately 4 000 transaction validations per second. In future work, the practical evaluation can be extended to confirm the simulation results for various types of transactional application workloads.

In summary, the evaluation provides evidence for our claim that caching can significantly improve latency and abort probabilities of optimistic transactions in the context of cloud data management. The Cache Sketch approach lowers transaction durations by roughly an order of magnitude and thus combines the performance benefits of modern NoSQL systems with the strong guarantees of ACID transactions.

4.9 Summary

In this Chapter, we investigated the applicability of web caching for mutable records, files, and query results. To this end, we derived a novel caching approach for dynamic data that improves loading times in web applications. We rely on three pivotal ideas to make this possible: (1) the Cache Sketch as a compact client representation for stale data, (2) online TTL estimators for objects and queries, and (3) invalidation detection in real-time. We implemented these techniques in the Orestes middleware to offer low latency for data management through caching with client-defined staleness bounds as well as several practically useful client-centric consistency guarantees.

First, we proposed the Cache Sketch as a data structure that ensures Δ-atomicity and enables clients to read every non-stale object from expiration-based caches (e.g., browser caches). The database service maintains the Cache Sketch as a Bloom filter of potentially stale objects. At the server-side, the Cache Sketch minimizes the number of cache invalidations by deciding, whether update operations require purging of invalidation-based caches (e.g., CDNs). To minimize the Cache Sketch size, invalidation costs, and cache misses, we proposed the concept of TTL estimators that produce expiration dates based on the frequency by which resources are accessed. To reason about the performance and consistency implications, the YCSB Monte Carlo Caching Simulator offers a generic framework for analyzing different workloads, caching architectures, and Cache Sketch parameters. Our simulations and cloud-based results for object-based caching showed a substantial performance uplift of 4 to 10 times for typical database workloads.

Next, we extended the approach to query result caching in the Quaestor architecture. We designed a TTL estimation scheme that is able to handle the composition of objects to result sets in order to provide accurate predictions for future query invalidations. We

[25] The `findAndModify` operation limits overall system throughput as it obtains unnecessarily coarse-grained locks [Mon18].

introduced a cost-based decision model to optimize the query result representation according to a trade-off between invalidations, cache hits, and the false positive rate of the Cache Sketch. To tackle invalidation detection, we introduced a scalable architecture for matching cached queries to incoming updates in realtime. Evaluation results demonstrate Quaestor's effectiveness in reducing query latency by up to an order of magnitude while strictly limiting staleness.

Last, we applied the Cache Sketch to transaction processing by designing distributed cache-aware transactions (DCAT). We provided a stochastic model to quantify the impact of latency on abort probabilities and found that latency is the key to improving optimistic transactions for distributed environments. To this end, we derived a concurrency control scheme that provides COCSR transactions while using the Cache Sketch scheme for latency reduction. In the evaluation, we demonstrated that our caching approach can make previously infeasible transactions fast enough for interactive workloads.

5 Towards Automated Polyglot Persistence

In this chapter, we present a solution for automated polyglot persistence based on service level agreements (SLAs). The goal is to unburden applications of the complexities associated with manual polyglot persistence by automating the choice of a well-suited database system. This vision of completely automated polyglot persistence complements our previously presented contributions on low-latency cloud data management by making the performance improvements applicable to arbitrary systems while also tackling the remaining issue of insufficient backend performance.

We introduce the Polyglot Persistence Mediator (PPM) as a middleware to database systems that is capable of mapping schemas annotated with SLAs to different systems. A scoring algorithm provides a routing model that is used at runtime to forward data and operations to the most suitable system. Thus, the PPM can transparently apply the Cache Sketch approach to diverse data stores while consolidating them behind a single interface for applications. For a typical polyglot persistence scenario, the PPM can improve write throughput by 50-100% and simultaneously reduce read and query latency drastically.

5.1 Motivation

As described in Chapter 2, polyglot persistence is the concept of using different database systems within a single application domain, addressing different functional and non-functional needs with each system [SF12]. While virtually any multi-purpose application could benefit from polyglot persistence, there are currently some obvious drawbacks. Designing and implementing an application on multiple databases is considerably harder than just using one backend. Application demands frequently exceed capabilities of single databases. At the same time, overhead of configuration, deployment, and maintenance increases drastically with each database system used. Today, superior polyglot persistence solutions are therefore often abandoned for lack of know-how and resources.

Recently, cloud providers enabled software engineers to develop and deploy applications at fast paces through Backend-, Platform-, and Infrastructure-as-a-Service systems. However, using state-of-the-art cloud services, tenants still have to make the choice of using

a certain database [HIM02, DAEA13, CJP+11]. To solve this dilemma, we will present the Polyglot Persistence Mediator (PPM) as a new kind of middleware layer. Employing a service level agreement (SLA), developers can define specific requirements for their schemas. Schemas are centrally managed by the Orestes middleware, which maps them to individual database systems. Even though many NoSQL systems do not employ explicit schemas[1], we assume the presence of a central schema (cf. Chapter 3) in order to compute a database-independent evaluation of requirements.

For illustration, consider an example use case of an online newspaper. Usually, there is a list of the most popular articles. To enable a ranking, counters for article impressions need to be written up to many thousand times per second, whereas the article itself is rarely changed after publication. Therefore, if impression counters and articles were stored together, counter writes would eventually slow down article reads. On the other hand, if both were stored separately, the application would lose the ability to sort articles by impressions. The solution is to use schema annotations for write throughput and sorting, so that the PPM could for example decide to store counters in a sorted set in Redis (achieving much higher throughput), while article reads are directed to MongoDB (supporting complex ad-hoc queries).

In the following, we will first apply the NoSQL Toolbox classification from Chapter 3 to annotation-based declaration of functional and non-functional requirements. This enables us to derive the concept of an automated choice of backends. We will then explain the architecture of the mediator and evaluate its prototype in an experimental case study. Last, we will discuss a number of potential enhancements and further lines of research.

5.2 Concept: Choosing Database Systems by Requirements

The polyglot persistence mediation is structured in three phases. In the **requirements** phase (1), schemas are annotated with desired SLAs. In the **resolution** phase (2), the cloud provider computes a suitable mapping of requirements to databases. Throughout the **mediation** phase (3), operations and queries are rewritten and distributed to different database systems [WFGR15].

Automated polyglot persistence requires formal decision criteria. To this end, we first propose a classification of functional and non-functional requirements with suitable places for schema-based annotations. Table 5.1 provides an overview of these requirements building on the NoSQL Toolbox. On the highest level, requirements are divided into *binary* and *continuous* requirements. Binary requirements support yes-or-no decisions. For example, a database either supports server-side joins or it does not. This might be subjective for some non-functional requirements like scalability, since there is no agreed-upon way of measuring them.

[1] Some NoSQL databases like DynamoDB [Dyn17], OrientDB [Tes13], and Cassandra [LM10] support explicit schemas without a middleware.

Continuous requirements like write latency, on the other hand, can be evaluated by comparing specific values to the context of an application. For instance, low latency for an interactive website usually would not constitute low latency for high-frequency trading. Therefore, automated polyglot persistence needs to provide a rich syntax for SLAs that can be parameterized with application-specific goals.

Annotation	Type	Annotated at
Read Availability	Continuous	Field/Bucket/DB
Write Availability	Continuous	Field/Bucket/DB
Read Latency	Continuous	Field/Bucket/DB
Read Throughput	Continuous	Field/Bucket/DB
Write Latency	Continuous	Field/Bucket/DB
Write Throughput	Continuous	Field/Bucket/DB
Data Scalability	Non-Functional	Field/Bucket/DB
Write Scalability	Non-Functional	Field/Bucket/DB
Read Scalability	Non-Functional	Field/Bucket/DB
Elasticity	Non-Functional	Field/Bucket/DB
Durability	Non-Functional	Field/Bucket/DB
Linearizability	Non-Functional	Field/Bucket
Sequential Consistency	Non-Functional	Field/Bucket
Δ- and k-Atomicity	Non-Functional	Field/Bucket
PRAM Consistency	Non-Functional	Field/Bucket
Read-your-Writes	Non-Functional	Field/Bucket
Causal Consistency	Non-Functional	Field/Bucket
Writes follow reads	Non-Functional	Field/Bucket
Monotonic Read	Non-Functional	Field/Bucket
Monotonic Write	Non-Functional	Field/Bucket
Scan Queries	Functional	Field
ACID Transactions	Functional	Bucket/DB
Sorting	Functional	Field
Range Queries	Functional	Field
Point Lookups	Functional	Field
Conditional Updates	Functional	Field
Joins	Functional	Bucket/DB
Filter Queries	Functional	Bucket/DB
Analytics and Aggregation	Functional	Field/Bucket/DB
Full-text Search	Functional	Field
Atomic Updates	Functional	Field/Bucket

Table 5.1: Proposed SLA annotations (cf. Section 2.2.4 and 3.1).

5.2.1 Defining Requirements Through SLAs

Figure 5.1 provides an overview of the initial requirements phase that yields a schema annotated with application-specific requirements expressed as SLAs. On a high level, tenants define schemas consisting of databases, buckets, and fields (cf. Chapter 3).

Tenants may then define annotations that can be used to annotate complete databases, buckets, or attributes of a bucket. For instance, annotating a single field to support joins would not be useful. Binary functional and non-functional requirements (capabilities) annotated at a certain hierarchy level result in constraints that have to be met by ev-

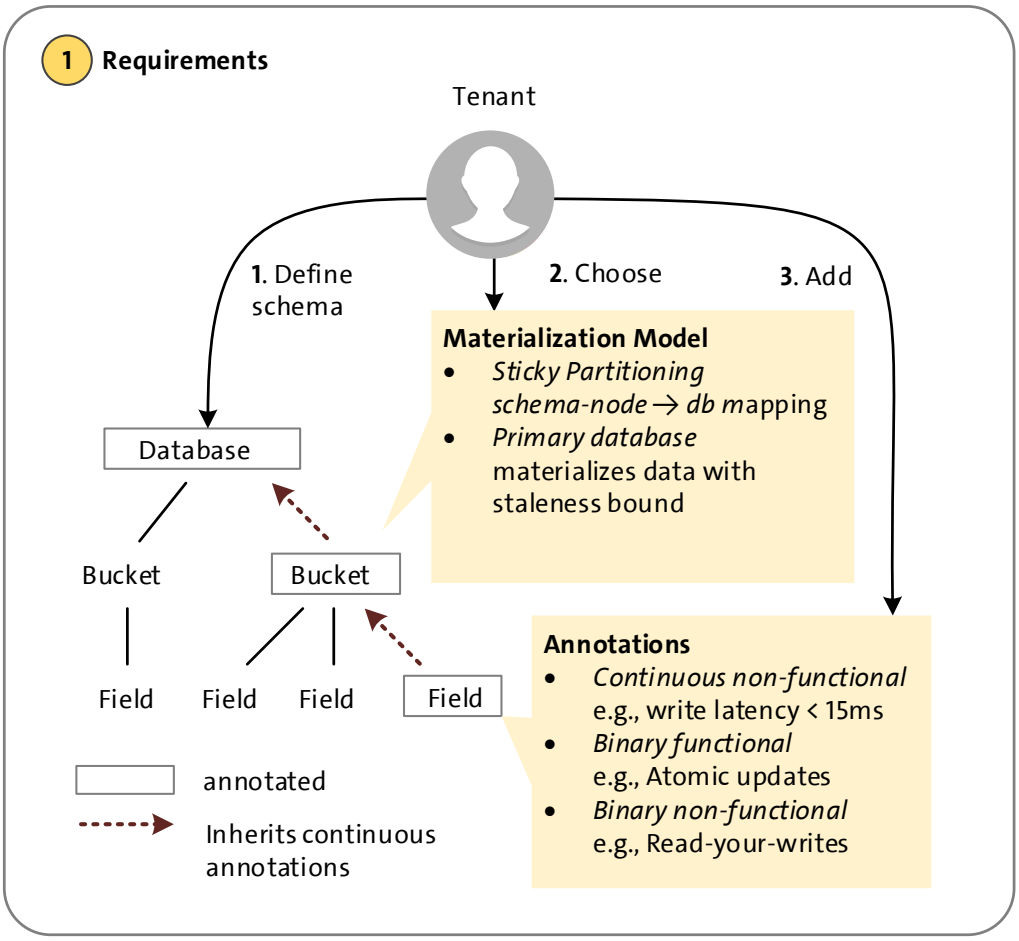


Figure 5.1: First phase of meditation: schema-based SLA annotations.

ery entity on and below that level. Continuous requirements are simply pushed down to the field level: for example, if a complete database is annotated to support 99.5% of read availability, every field in that database inherits that annotation. Enforcing continuous requirements on higher levels (i.e., buckets) is not feasible, since this would require knowledge about the distribution of this requirement on all its child entities.

An annotation consists of an arbitrary number of binary requirements and continuous non-functional requirements specified either through utility functions or specific goals on the requirement (e.g., latency below 15 ms).

Finally, using polyglot persistence, tenants also have to choose a materialization model:

Sticky Partitioning. Instructs the mediator to always route operations for a schema-node to the same database.

Primary Database. Defines a read-only master copy to which the mediator periodically materializes data stored in other databases. A staleness bound defines the maximum tolerable delay Δ between materializations. Under the primary database model, applications must thus tolerate eventual consistency with Δ-atomicity, i.e., the possibility of reading a value that has been stale for at most Δ time units. This model allows

to unite complex, slightly stale queries in the primary database with high throughput and low latency for simple updates and queries performed in other databases and is therefore the default in Orestes.

In the following section, we will demonstrate how the SLA specifications translate to routing decisions by introducing a scoring model. We will also elucidate how annotations combined over different hierarchy levels can be resolved.

5.2.2 Scoring Databases against SLA-Annotated Schemas

The DBaaS/BaaS provider is responsible for resolving the requirements to a database mapping as shown in Figure 5.2. Annotations are resolved by the provider by first comparing the specified binary requirements with all currently available systems. If no system can provide the desired combination, the provider can either reject the annotation right away or try to provision another type of database. In this context, a system is one specific deployment of a database, i.e., an IaaS provider might manage a number of different configurations of the same database. All databases capable of delivering the binary requirements are then scored to find an optimal setup for a specific tenant.

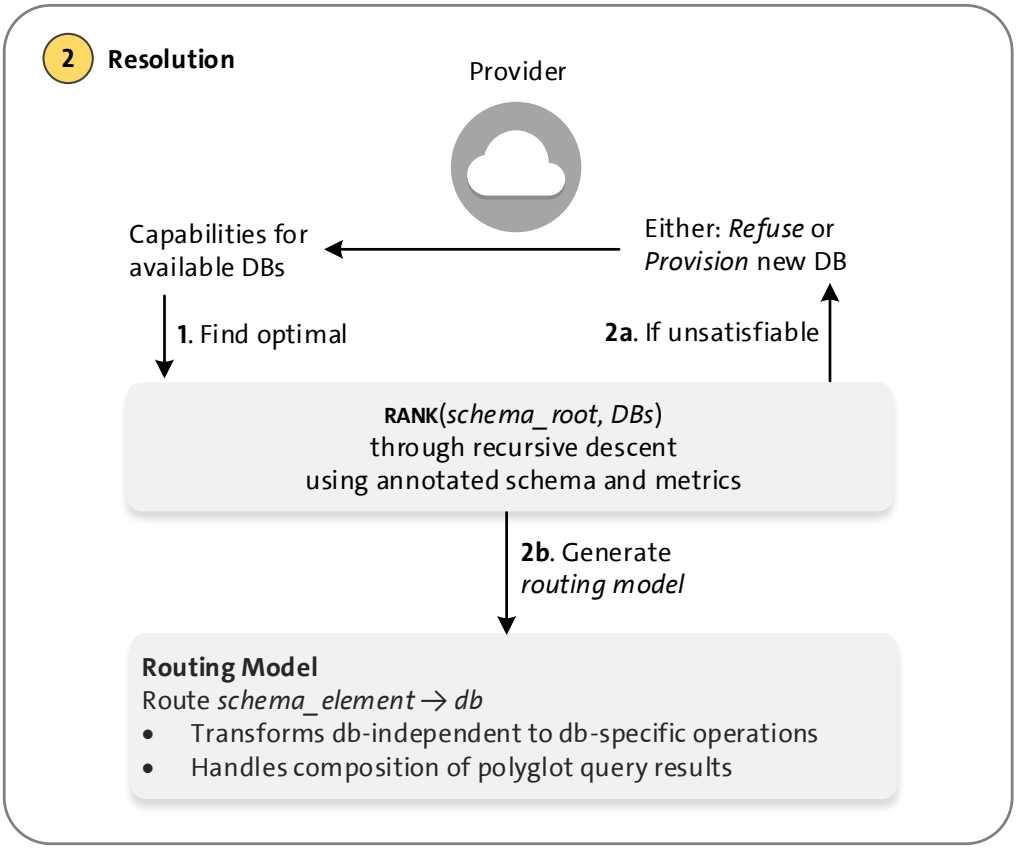


Figure 5.2: Second phase of meditation: scoring of available systems.

The evaluation of annotations is performed recursively over all hierarchy levels of the schema, as shown in Algorithm 4. The scoring algorithm starts at the database (root) level

of the schema. First, it excludes all databases incompatible with the current node's constraints (i.e., its binary requirements), so child nodes cannot choose databases their parent nodes do not support. Field nodes simply calculate their scores according to the scoring model (cf. Equation 5.1). Other nodes recursively calculate mappings of databases to scores for their child nodes. The algorithm then intersects among the resulting databases (line 7) to find systems that can support all requirements of the child nodes and averages over the resulting scores of each database of each child node. When the recursion returns to the current node and it is annotated (line 8), it adds the optimal mapping to the routing model. Finally, aggregated scores are returned.

Consider the following example: a bucket is annotated for object-level atomic updates. One field of this bucket requires a certain level of write throughput, another one has an annotation for read latency. At the root level, there is nothing to do, so the algorithm turns to bucket nodes. For the specific bucket, all databases incapable of ensuring atomicity are removed. Individual field-level annotations are now evaluated and each returns a set of databases capable of supporting both their binary and continuous requirements. The recursion then returns and computes the intersection of each field node, determining which databases can support the required combination of throughput and latency best. Finally, the bucket node adds the result to its routing model so that all data items will be stored in the according database. Annotating a node with a binary requirement at database or bucket level means that all items of that database or bucket will be stored in the same data store. The algorithm ensures compatibility of annotations along the schema hierarchy. It should be noted that schema or annotation changes may require repartitioning, the details of which we leave to future work.

Algorithm 4 Scoring algorithm for input schema node

```
1:  procedure RANK(node, DBs) returns {db → score}
2:      drop db ∈ DB if not node.annotations ⊆ db.capabilities
3:      if node is field then
4:          scores ← {db ∈ DBs → score(db, node)}
5:      else
6:          childScores ← {(child, db, score) |
                child ∈ node.children and
                (db, score) ∈ RANK(child, DBs)}
7:          scores ← db, avg(score) from childScores
                group by db
                having count(child) = |node.children|
8:      if node is annotated then
9:          add (node → argmax_db scores) to routingModel
10:     return scores
```

The score of a database is calculated by adding individual scores for each continuous non-functional requirement $cn \in CN$. Tenants can also assign arbitrary weights to each

requirement to model relative importance. The total score is then normalized by the sum of weights:

$$score(db) = \frac{\sum_{i=1}^{|CN|} w_i \cdot f_i(metric(cn_i))}{\sum_{i=1}^{|CN|} w_i} \tag{5.1}$$

For our scoring model, we propose two alternatives. First, we consider requirement-specific, normalized **utility functions**. A utility function maps values of a particular metric to the utility this requirement has for a specific use case:

$$f(metric) \rightarrow utiltiy \in [0,1] \tag{5.2}$$

Figure 5.3 shows two examples of requirement-specific utility. For instance, an interactive application may consider any latency below 20 ms to be acceptable, with a linear decrease to zero utility at 50 ms. On the other hand, availability might be scored by a sigmoid function, indicating that availability below a certain threshold is of little utility and then drastically increases in utility up to some point of saturation (e.g., 98% vs 99.99% of availability). Practically, users could interactively manipulate these functions in a service dashboard.

Normalization of utility functions helps computing a unified score and defining SLA violations. For instance, an SLA may include multiple thresholds with different consequences. First, monitoring such thresholds helps providers to better understand their setups and the impact of changes on their performance. Falling under a certain threshold could also result in auto-scaling the respective database. Second, violations may trigger compensations for tenants in the form of partial refunds or service credits.

The second scoring model does not require users to specify a mapping of values to utilities. Instead, they simply specify goal values for each requirement. Goals are then compared against current metrics of the system in a manner of **performance indexing** [LS13]:

$$f(cn_i) \;=\; \frac{goal(cn_i)}{metric(cn_i)} \quad (or\ inverse) \tag{5.3}$$

For instance, a goal of 50 milliseconds in latency compared to an actual average latency of 20 milliseconds would result in a score of 2.5. Thanks to the collected metrics, arbitrary SLA models may be defined (e.g., pricing models based on deviations [Bas12]).

After computing scores, the database with the maximum score will be selected to store the annotated field. This decision is made based on both the current and historic values, using a weighted moving average of all the metrics collected by the provider (i.e., calculating either performance indices or current values of all utility functions). If annotations are allowed to be changed later, the provider has to support live data migration between different databases.

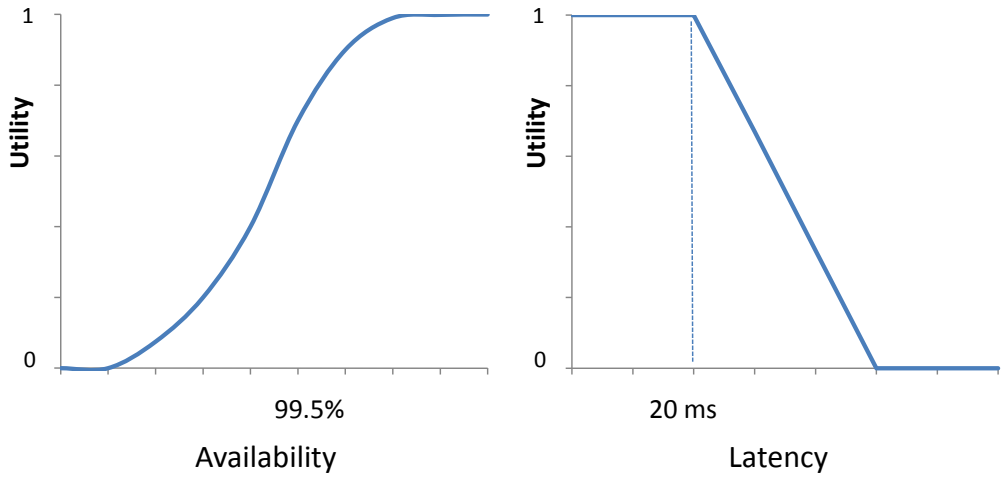


Figure 5.3: Examples of normalized utility functions.

5.2.3 Mediation

The Polyglot Persistence Mediator acts as a broker between applications and databases. Applications use the unified REST API (cf. Section 3.5.2) to issue queries, CRUD operations, transactions, and other operations to the mediator in a database-agnostic fashion. Based on the routing model, the mediator selects the appropriate database and transforms the incoming operations to database-specific operations:

$$transform(agnosticOperation, db) \rightarrow dbOperations \tag{5.4}$$

As an example, consider the addition of a value to an array-valued field, for which the resolution step determined MongoDB as most appropriate for the tenant's annotations. The mediator looks up the affected field in the routing model which yields *MongoDB*. Next, the mediator queries its transformation rule repository to map the incoming operation *push(obj, field, val)* to the specific operation {*$push:* { *field: val* }} and forwards it to the selected MongoDB cluster. In general, an operation can be transformed into a set of operations to account for potential data model transformations and denormalization (e.g., secondary index maintenance).

The mediator is stateless and can thus be replicated arbitrarily for linear scalability. Updates to the routing model (schema changes) can be shared through coordination services (e.g., Zookeeper [HKJR10]), consensus protocols (e.g., Paxos [Lam01], Raft [OO13]) or classic 2PC with different availability/consistency trade-offs [BBC+11, BVF+12]. The mediator also manages the aforementioned materialization model.

The PPM continuously monitors operations issued to backend databases to **report metrics** for throughput, latency, and availability, aggregating them into means, modes, weighted moving averages, percentiles, and standard deviations. The metrics are used for scoring in the resolution step as well as the detection of SLA violations. Scoring thus reflects the

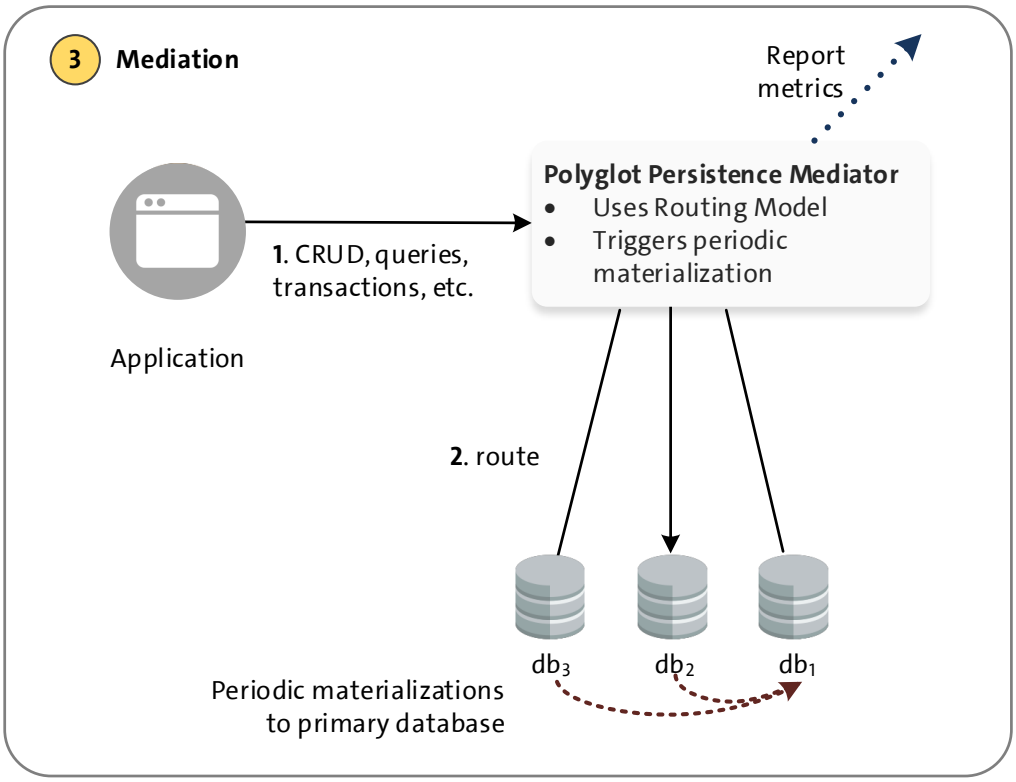


Figure 5.4: Third phase: polyglot persistence mediation.

actual system state. Of course, when a new database is provisioned, metrics are unknown. They can either be estimated using a system performance model of the database or by running a synthetic workload or historic traces [CST+10, SSS+08].

The mediator allows two deployment models:

Cloud Broker. The PPM can be deployed on-premise (e.g., in a private cloud), relying on local database deployments and Database-as-a-Service offerings of public cloud providers (e.g., Amazon RDS, DynamoDB, S3, Elasticache, Windows Azure Table Storage [DHJ+07, CWO+11]). In this case, some flexibility is lost, as public cloud providers will not adjust deployments to optimize scores nor will they allow SLAs based on scores and utility functions.

Public Database-as-a-Service. A public cloud provider could adopt the mediator and expose it as a "meta" database or backend service to achieve better internal resource utilization while offering richer SLAs and functionalities.

5.2.4 Architecture of the Polyglot Persistence Mediator

We implemented a prototype of the PPM as part of the Orestes middleware. Thus, applications can use the unified REST API interface which maps all client-side operations to the appropriate server-side system. As Orestes implements a powerful hybrid schema system that is independent of individual database technologies, the PPM can offer polyglot persistence on field level by adding annotations as schema metadata. In our experiments,

we used MongoDB as the principal storage facility (i.e., the primary copy materialization model) and Redis as a caching layer to accelerate writes, which constitutes a typical polyglot persistence scenario. The PPM analyses annotations (using simple performance indices) and routes partial updates according to the annotations (write throughput, write latency, etc.) either to the primary database (MongoDB) or the caching layer (Redis).

The PPM contains a module that schedules materialization between databases while providing strong guarantees through strictly ordered materialization and an upper bound on the materialization time for any particular object. Similar measures are taken to ensure that delete operations are carried out for all databases containing fragments of an object. Orestes inherits the availability model of the underlying systems. If a database node fails, all writes and materializations to that respective partition or database will fail, too.

5.3 Experimental Case Study

We performed experimental evaluations using various distributed Amazon EC2 setups. We implemented a custom benchmark client that collects metrics similar to the Yahoo Cloud Serving Benchmark (YCSB) [CST+10], i.e., averages, minimums, maximums, and percentiles on latency as well as average throughput.

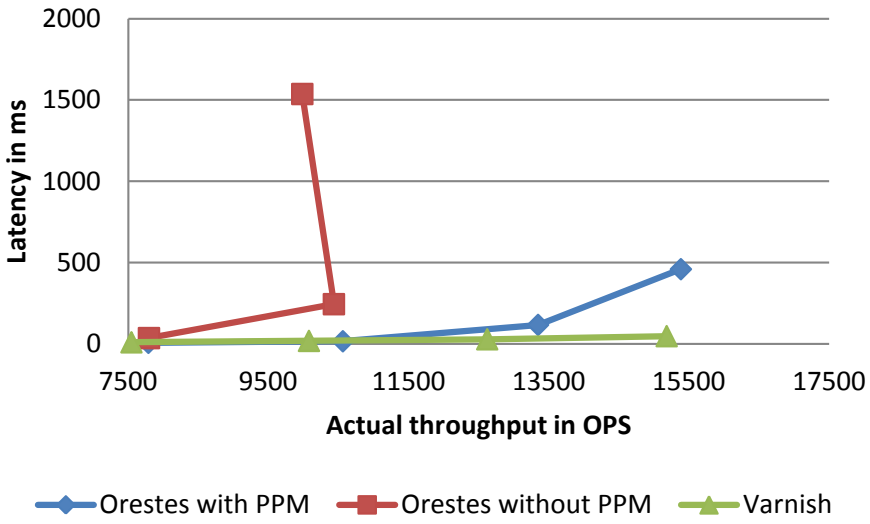
The first test scenario encompasses a single-client/single-server setup for the introductory scenario of impression counters in an online newspaper. Access patterns were generated according to a Zipf distribution. Materialization intervals were set to 60 seconds.

Figure 5.5a demonstrates typical latency behavior for a medium range of throughputs. MongoDB and Redis each ran on a separate *m1.large* instance, whereas the benchmark client and the Orestes server with and without PPM each ran on a *c3.4xlarge* instance to account for the fact that the main performance overhead occurs between benchmark client and Orestes server. MongoDB instances were equipped with 1 000 provisioned IOPS for their main storage volumes and the write concern was set to *acknowledged*.

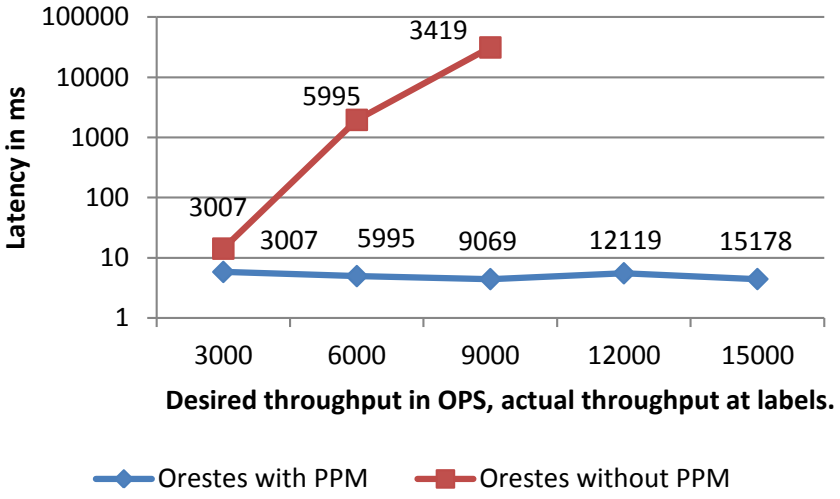
For the benchmark, 100 000 update operations were executed on 100 different articles in MongoDB with impression counters annotated for high throughput and thus maintained in Redis. The line chart shows that the Orestes middleware supported by the PPM performs significantly better than Orestes without a PPM[2]. We found that the throughput-limit corresponds to an average latency of approximately 500 ms. In this setup, the PPM achieved a 50% increase in write throughput while maintaining lower latencies. For an HTTP baseline, we used a Varnish cache [Kam17] to benchmark performing GET requests against a static resource. At low throughputs, the PPM constantly achieved better latencies than both Varnish and Orestes without a PPM.

Next, we demonstrate the results for a setup of 12 benchmark clients, 4 servers, and 1 server for each database (all *m1.large*). Figure 5.5b shows results for a read-only bench-

[2] In this context, Orestes without a PPM means that all database operations were routed to a single database (i.e., MongoDB).



(a) Latency behavior for a single-client/single-server setup.

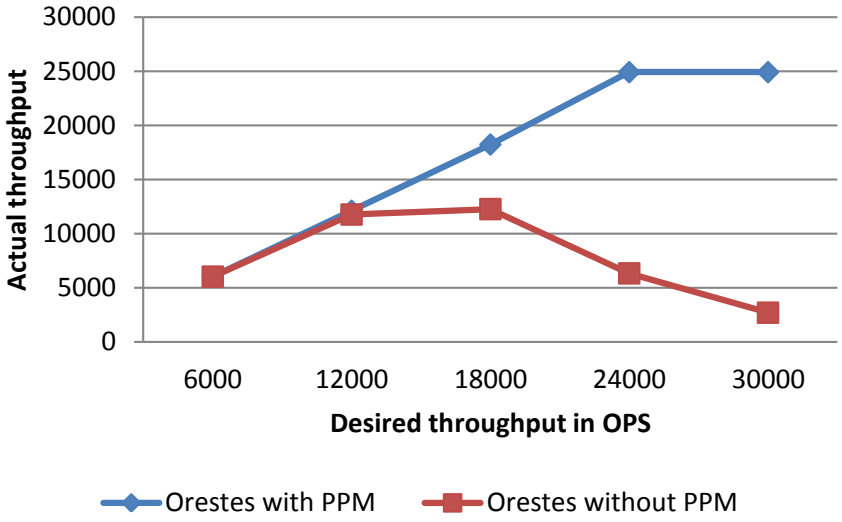


(b) Read-only benchmark results for 12 clients and 4 servers.

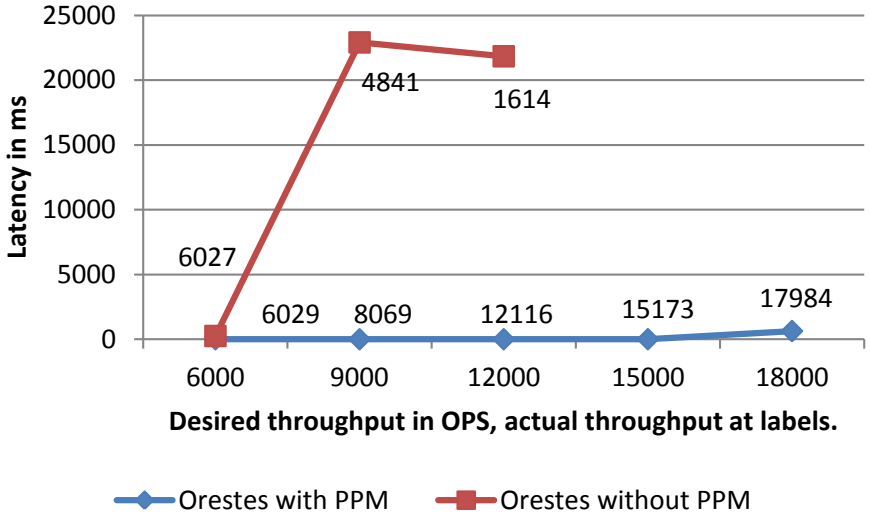
Figure 5.5: Evaluation of the Polyglot Persistence Mediator for a single-node setup and a read-only setup with multiple client and server nodes.

mark. In the use case of top-listed articles, we were interested only in the top-10 articles in the sorted set. While articles were stored in MongoDB, PPM-supported reads were routed to the Redis set. Reads without the PPM were simply executed on MongoDB. Visibly, the persistence mediator provided a consistent latency under 10 ms up to 15,000 reads per second on the sorted set. Please note that the mediator would still route reads on other fields or the complete object to MongoDB (since we only annotated a specific attribute).

Figure 5.6a demonstrates the same setup in a write-only scenario. Comparing desired and actual throughputs, we can see that MongoDB can serve up to 12 000 writes per second,



(a) Write-only benchmark results, 12 clients and 4 servers



(b) Mixed benchmark results, 12 clients and 4 servers, 95% reads.

Figure 5.6: Evaluation of the Polyglot Persistence Mediator for a write-only and a mixed workload.

while the persistence mediator reaches 25 000. Hence, employing the PPM with MongoDB as our default storage, we effectively doubled write throughput for a common scenario.

5.4 Outlook

While the PPM offers a general methodology for handling polyglot persistence in an automated fashion, it can be extended in many potential ways.

5.4.1 Scoring and Database Selection

The ability of a cloud provider to maintain its SLA guarantees heavily depends on the scoring, i.e., the cost-utility ratio. Therefore, it is crucial for the provider to select database configurations that indeed fulfill the requirements, i.e., consistently achieve high scores. One line of further research therefore is the estimation of future scores based on historic metrics. The scoring could then be adapted to prefer selections that have expected high scores in the future. There are many potential statistical and machine learning techniques to achieve this. For instance, reinforcement learning (e.g., Q-learning [DKM+11]) could be used to learn from past selection decisions based on the utility achieved for the provider (i.e., minimizing SLA violations and maximizing resource utilization). Similarly, time series analysis (e.g., ARMA [PHS+09]) and regression can be employed to estimate the future development of metrics. Validating these different prediction models will also require new synthetic or production workload traces that capture the dynamics of multiple consolidated tenants.

5.4.2 Workload Management and Multi-Tenancy

The PPM could improve performance by actively scheduling requests. Requests that pertain to throughput-oriented annotations can be improved by batching, while latency-sensitive requests can be scheduled to experience minimum queuing delays. This workload management has to be implemented per database and depends on the multi-tenancy strategy (e.g., private process or private schema, see Section 2.2.6) employed by the mediator to ensure sufficient isolation between tenants.

5.4.3 Polyglot Setups

As a practical question, the PPM needs to consider which system setups might provide optimal polyglot persistence functionality. For our experiments, Redis was used as an on-demand caching layer to complement MongoDB as the primary data store. But many other combinations are useful, too. For instance, object stores (e.g., S3 [Ama17a]) can be used to store binary data (e.g., images), while shared-nothing file systems such as HDFS [Whi15] and Tachyon [LGZ+13] could be used for fields with analytic potential, leveraged through platforms such as Hadoop [Whi15] and Spark [ZCF+10]. Wide-column stores (e.g., HBase [Hba17], Accumulo [CRW15]) and table stores (e.g., DynamoDB [Dyn17], Azure Tables [CWO+11]) can similarly be used to store analyzable, structured, write-heavy data. Dynamo-style systems (e.g., Riak [Ria17], Cassandra [LM10]) can be leveraged to keep fields that require high availability, but can tolerate temporary inconsistencies. A full-text search annotation could be handled by adding a distributed search platform (e.g., Solr [Luc17], ElasticSearch [Ela17]) as an additional primary copy that the mediator exploits for search queries. The mediator can also feed modifying operations into InvaliDB for continuous query processing, which should then allow different query languages applicable to polyglot data.

5.4.4 Adaptive Repartitioning

Changing existing annotations at runtime implies that in order to adapt, data might have to be migrated to a more suitable database system. In contrast to existing migration approaches such as Zephyr [EDAE11], Albatross [DNAE11], Dolly [CSSS11], and Slacker [BCM+12] that migrate complete databases, it is sufficient for the PPM to migrate on a field or bucket level. The PPM could thus perform migration through iterative state replication [EDAE11]: to transfer a snapshot to the newly selected database, incoming requests are still applied to the old database, while buffering them for migration. Once bootstrapping of the snapshot completes, buffered requests are applied. Afterwards, the PPM can route all requests to the new database. One major challenge is to either obtain a consistent snapshot from distributed databases without downtime or design a migration algorithm that tolerates snapshot inconsistencies.

5.5 Summary

In this chapter, we introduced the idea of a Polyglot Persistence Mediator. The PPM is an extension of Orestes that enables tenants of DBaaS/BaaS systems to leverage automated polyglot persistence on a declarative basis using schema annotations. Tenants can precisely specify their requirements. Through annotations, they can define utility functions for non-functional requirements (e.g., availability and latency) as well as necessary binary properties (e.g., object-level atomicity). The mediator scores available database systems and selects the optimal system for each part of the tenant's schema and automatically routes data and operations accordingly. Exploiting the proposed annotations, providers are free to define which requirements they provide SLAs on and tenants can employ annotations in an opt-in fashion, maximizing flexibility on both ends. We provided evidence that tremendous improvements in latency and throughput can be obtained and outlined the remaining challenges for providing a general-purpose Polyglot Persistence Mediator.

6 Related Work

This chapter provides a discussion of related work. We start with caching and replication as the two primary techniques for low latency. Next, we discuss related transaction protocols and Database-as-a-Service approaches. This thesis is inspired by the idea of geo-replication and influenced by various techniques from related work, such as Bloom filters for compact digests, expiration-based caching for passive replication, as well as continuous queries for cache invalidations. We believe that Orestes adds a very useful design choice for low-latency, data-centric cloud services and support this claim by comparing its characteristics and trade-offs to related work.

6.1 Caching

Related work on caching can be categorized by several dimensions (see Figure 6.1). The first dimension is the *location* of the cache. We focus on caches relevant for cloud and database applications, in particular, server-side and database caching, reverse and forward proxy caching (mid-tier), and client caching [LLXX09]. The second dimension is the *granularity* of cached data. In particular, these are files, database records and pages, query results, and page fragments [APTP03a, APTP03b, AJL+02, CLL+01a, CZB99, CRS99, DDT+04, LC99, LN01]. The third dimension is the *update* strategy that determines the provided level of consistency [Cat92, GS96, NWO88, LC97, CL98, BAK+03, BAM+04].

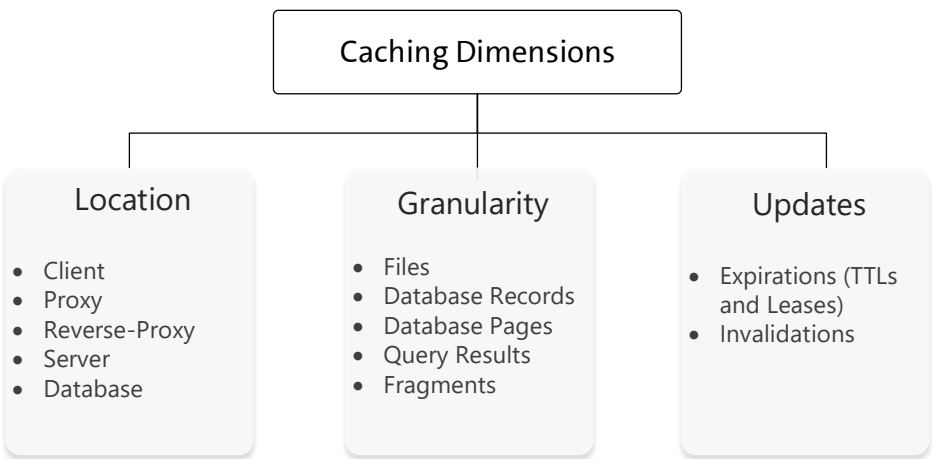


Figure 6.1: The three central dimensions of caching.

Besides these major dimensions, there are smaller distinctions. The cache replacement strategy defines how the limited amount of storage is best allocated to cached data [PB03, DFJ+96]. The initialization strategy determines whether the cache is filled on-demand or proactively[1] [ABK+03, LGZ04, LKM+02, BAK+03, LR00, LR01a]. The update processing strategy indicates whether changes to cached data are replacements, incremental changes, or based on recomputation [Han87, BCL89, AGK95, LR01a, IC98, BLT86]. This work is orthogonal to these minor dimensions, but heavily based on the different locations (end-to-end), caching granularities (files, query results, and records), and update strategies (expiration and invalidation). We will discuss other work on caching that employs specific combinations and compare it to our end-to-end approach.

Table 6.1 summarizes major related work according to the dimensions *location* and *updates*. In the following, we will first discuss server-side and client-side application caching as well as database caching and contrast both to web caching approaches. Next, we will show the different methods for cache coherence that can be grouped into expiration-based and invalidation-based approaches. Finally, we will review related work on caching query and search results and discuss summary data structures for caching.

	Expiration-based	Invalidation-based	Hybrid
Client	Browser cache [IET15], ORMs [Rus03a, ABMM07, CSH+16], ODMs [SHKS15], User Profiles [BR02], Alex Protocol [GS96], CSI [RXDK03], Service Workers [Ama16]	Avoidance-based Algorithms [ÖV11, FCL97, WN90]	Client-Server Databases [KK94, ÖDV92, CALM97], Oracle Result Cache [Ora17]
Mid-Tier	HTTP proxies [IET15], PCV [KW97], ESI [TWJN01]	PSI [KW98], CDNs [PB08, FFM04, Fre10]	Leases [Vak06], Volume Leases [YADL99, YADL98], TTR [BDK+02]
Server and DB	Incremental TTLs [AAO+12],	CachePortal [CLL+01b], DCCP [KLM97], Reverse Proxies [Kam17], Ferdinand [GMA+08], Facebook Tao [BAC+13], Cache Hierarchies [Wor94], DBProxy [APTP03a], DBCache [BAM+04], MTCache [LGZ04], WebView [LR01a]	Memcache [Fit04, NFG+13, XFJP14], Redis [Car13], IMDGs [ERR11, Lwe10], CIP [BBJ+10], Materialized Views [LLXX09]

Table 6.1: Selected related work on caching classified by location and update strategy.

[1] Proactive filling of the cache is also referred to as *materialization* [LR00].

6.1.1 Server-Side, Client-Side, and Web Caching

Server-Side Caching

Caching is often a primary concern in distributed backend applications. Numerous caching systems have been developed to allow application-controlled storage and queries of volatile data. Typically, they are employed as **look-aside** caches storing hot data of the underlying database system, with the application being responsible for keeping the data up-to-date. Among the most popular of these systems is Memcache, an open-source, in-memory hash table with a binary access protocol introduced by Fitzpatrick in 2004 [Fit04]. Memcache does not have any native support for sharding, but there are client-side libraries for distribution of records over instances using consistent hashing. Facebook, for example, uses this approach for their high fan-out reads of pages [NFG+13, XFJP14] and their social media graph [BAC+13]. The key-value store Redis is used for similar caching scenarios, enabling more advanced access patterns with its support for data structures (e.g., hashes, lists, sorted sets) instead of opaque data values [Car13]. In contrast to Memcache, Redis additionally supports different levels of persistence and an optimistic batch transaction model. Considerable research went into the optimization of these caches in terms of hashing performance [FAK13], fair cache resource sharing between clients [PLZ+16], and optimal memory allocation [CEAK16]. For the Java programming language, a standard caching API has been defined and implemented by various open-source and commercial caching projects [Luc14]. For server-side caching with higher persistence guarantees, key-value stores such as Riak [Ria17], Voldemort [ABD+12], Aerospike [Aer18], HyperDex [EWS12], and DynamoDB [Dyn17] are suitable.

In-memory data grids (IMDGs) [RRND15, p. 247] are distributed object stores used for state management and caching in Java and .Net applications. Industry products include Oracle Coherence, VMware Gemfire, Alachisoft NCache, Gigaspaces XAP, Hazelcast, Scaleout StateServer, Terracotta, JBoss Infinispan, and IBM eXtreme Scale [ERR11, Lwe10]. Compared to key-value caches, IMDGs offer the advantage of tightly integrating into the application's programming language and its class and object models. In this respect, IMDGs are similar to object-oriented database management systems (OODBMSs), as they expose native data types (e.g., maps and lists). Additionally, distributed coordination abstractions such as semaphores, locks, and atomic references as well execution of MapReduce jobs are typically supported. IMDGs are also used in related research projects (e.g., CloudSim [KV14]), due to the simple abstractions for shared distributed state. An IMDG is not a good fit for a database system in Orestes, because the abstraction level is too language-specific and persistence is not guaranteed.

Server-side caching with key-value stores and IMDGs is a proven technique for reducing backend processing latency by offloading persistent data stores in I/O-bound applications. This comes at a cost, however: the application has to maintain the caches using domain-specific logic. The complexities of maintaining consistency and retrieving cached data are thus left to application developers. In contrast to server-side caching solutions, this work

aims to provide low *end-to-end* latency by exploiting existing HTTP caching infrastructures and automating the process of caching for the application. Nonetheless, Orestes builds on key-value caches, in particular Redis, both for maintenance of shared state and for coordination of transaction commits.

Client-Side Database Caching

Client-side database caching has been discussed for decades in the database community [LLXX09, BAK+03, LKM+02, LGZ04]. In this case, the term "client" does not refer to a browser or mobile device, but to a server node of a backend application. In the context of distributed object databases, object-based, page-based, and hybrid approaches have been studied [KK94, ÖV11, CALM97]. Object-based buffer management has the advantage of a lower granularity, allowing for higher concurrency in the client for transactional workloads. Page-based buffers are more efficient when queries tend to access all objects within a page and imposes less messaging overhead. This caching model is fundamentally different from web caching, as the client buffer has to support the transaction model of the database system. As the cache is retained across transaction boundaries (inter-transaction caching), the problem of transactional isolation is closely tied to that of cache consistency [CFLS91, BP95]: neither transactions from the same client nor transactions from different clients are allowed to exhibit anomalies caused by stale reads and concurrent buffer updates.

Cache consistency algorithms from the literature can be classified as *avoidance-based* or *detection-based* [ÖV11, FCL97, WN90]. The idea of avoidance-based cache consistency is to prevent clients from reading stale data. This can be achieved by having writing clients ensure that any updated objects are not concurrently cached by any other client. Detection-based algorithms allow reading stale data, but perform a validation at commit time to check for violations of consistency. The second dimension of cache consistency algorithms is their approach to handling writes. Writes can be *synchronous*, meaning that at the time a client issues a write, the write request is sent to the server. The server can then, for example, propagate a write lock to all clients holding a cached copy of the written object (Callback-Read Locking [FC92]). With *asynchronous* writes, clients still inform the server about each write, but optimistically continue processing until informed by the server. This can lead to higher abort rates [ÖVU98]. In the *deferred* scheme, clients batch write requests and send them at the end of each transaction, thus reducing the write message overhead. Avoidance-based deferred algorithms typically suffer from high abort rates as well [FC92].

There are commercial relational database systems that offer client-side caching, for example, Oracle implements a client- and server-side result cache [Ora17]. The protocols and algorithms for client-side caching in databases serve the purpose of reducing the load on the database system, thereby decreasing backend processing latency. However, they are not applicable to end-to-end latency reduction in cloud data management, as web and mobile clients cannot exploit this form of caching. The overhead of locks distributed over

potentially hundreds of thousands of clients and the complexity of client-specific state in the database server impose a prohibitive overhead.

We use a detection-based approach, since with web caching it can never be guaranteed that no stale cached data is read. Similar to Özsu et al. [ÖDV92], we employ invalidations to minimize the probability of stale reads. In contrast to this model, expiration-based caches can neither execute custom consistency logic nor receive server-side invalidations which motivates the introduction of our Cache Sketch approach. As any optimistic protocol, our DCAT transaction protocol performs conflict checks at commit time, making this work a *detection-based deferred consistency scheme*. Adya et al. [AGLM95] have proposed a similar scheme called Adaptive Optimistic Concurrency Control (AOCC). It also relies on a backward-oriented validation step [ABGS86, CO82, LW84], but serializes transactions in timestamp order of the committing server. Effectively, AOCC performs timestamp ordering [SS94] with a two-phase commit protocol [Lec09] and thus accepts a smaller class of schedules than DCAT which is based on BOCC+ [KR81, Rah88]. Moreover, instead of relying on version numbers, AOCC servers maintain a set of metadata items for each of the client's cached data. AOCC was designed for the case of very few clients: the metadata maintained in each server increases with both the number of cached records and the number of clients, making it unable to scale for web scenarios with many clients.

Caching in Object-Relational and Object-Document Mappers

Due to the reasons laid out above, most persistence frameworks today rely on programmatic control of object-based caches with no support from the database system. With the increasing adoption of scalable NoSQL systems, the landscape of mappers bridging the **impedance mismatch** between the data model of the database system and the application has grown [SHKS15]. In fact, many applications do not use any native database system API, but instead rely on the convenience of object mappers such as Hibernate, DataNucleus, Kundera, EclipseLink, Doctrine, and Morphia [TGPM17]. In case of Java, the Java Persistence API (JPA) standard [DeM09] is considered state-of-art superseding the older Java Data Objects API (JDO) [Rus03a].

Both JPA and JDO and the equivalent technology from Microsoft called Entity Framework [ABMM07], support the notion of a first-level (L1) and a second-level (L2) cache. The L1 cache is exclusive to a persistence context and ensures that queries and lookups always resolve to the same object instances. The L2 is shared across contexts to leverage access locality between different contexts, processes, or even machines. The L2 interface is pluggable, so various options from in-process storage to Memcache- or IMDG-backed distributed implementations are available. Both L1 and L2 caches are write-through caches that directly reflect any updates passing through them. However, if data is changed from different contexts or even different clients, the L1 and L2 caches suffer from stale reads. The application has to explicitly flush or bypass these caches in order to prevent violations of consistency.

The first iterations of the Orestes prototype were based on JDO and JPA for the client API and the OODBMSs db4o [Db417] and Versant [Ver17] for data storage. It quickly became apparent that the missing capability of both systems to shard and replicate data limited the capabilities of the overall system. Additionally, JDO and JPA are mostly employed in three-tier applications where clients are located close to the database. In these settings, the performance-critical latency of HTTP requests cannot be optimized with web caching. Paired with the continued shift to single-page applications [MP14] and cloud-based services consumed directly in browsers and mobile devices, we chose to adopt core JPA concepts such as the L1 cache and the persistence life cycle in a JavaScript persistence API. In contrast to object mappers, the synchronization of the L1 cache is fully automatic in Orestes, as it exploits the Cache Sketch in the same ways as expiration-based caches (e.g., the browser cache). An L2 cache is not required, as regular web caches take its place. Chen et al. [CSH+16] recently proposed CacheOptimizer as a tool to help developers find the optimal cache configuration of their mapper to increase throughput. However, developers and application architects still face the problem of reasoning about the isolation and consistency guarantees offered for their specific application workloads. Orestes abstracts from the complexities introduced by caching in mappers and offers rigorous consistency guarantees.

Web Caching

In related work, web caches are either treated as a storage tier for immutable content or as a means of content distribution for media that do not require freshness guarantees [HBvR+13, Fre10]. Web caches are further defined by their implementation of the HTTP caching standards [IET15]. They can be employed in every location on the end-to-end path from clients to server. The granularity is typically files, though this is up to the application. Updates are purely expiration-based.

The applicability of web caching schemes is closely tied to web workloads and their properties. Breslau et al. were the first to systematically analyze how Zipf-distributed access patterns lend themselves for limited storage capacities of web caches [BCF+99, HL08, WF11]. Across six different traces, they found a steep average exponent of 0.85. Zipf-distributed popularity is closely related to our proposed capacity management scheme: even if only a small subset of “hot” queries can be actively matched against update operations, this is sufficient to achieve high cache hit rates.

The literature on workload characterization presents mixed conclusions. Based on an analysis of school logs, Gewertzman et al. [GS96], and Besavros [Bes95] found that most popular files tend to remain unchanged. Labrindis et al. [LR01b] and Douglis et al. [DFKM97], however, concluded that there is a strong correlation between update frequency and popularity of files. In another analysis of a more diverse set of university and industry traces conducted by Breslau et al. [BCF+99], the correlation between popularity and update rate was found to be present, but weak. In our work, we therefore do not assume a correlation

for the estimation of TTLs and the capacity model, but instead rely on individual treatment of cache miss and update rates.

Another question particularly important for simulations is, how the arrival processes of reads, writes, and queries can be modeled stochastically. Poisson processes with exponentially distributed inter-reference times are most widely used [Tot09, WAWB05, VM14]. However, homogeneous Poisson processes do not capture any rate changes (e.g., increased popularity) or seasonality (e.g., massive frequent changes upon deployments). Session-based models describe web traffic as a combination of individual user sessions. Session inter-arrival times typically follow a Poisson process, while inter-click times follow heavy-tailed distributions like Weibull, LogNormal, and Pareto distributions [Den96, Gel00]. For all Poisson-like workloads, TTL estimators will exhibit high error rates due to the high variance of the exponential distribution. As the Cache Sketch "corrects" estimation errors, the concrete workload and estimation accuracy only affect the false positive and cache hit rates. This is in stark contrast to pure TTL-based cache coherence schemes [GS96, LLXX09, BR02, RS03, KR01] which will exhibit high staleness rates, if workloads are inherently unpredictable.

Many optimizations of web caches have been studied. This includes cache replacement schemes [CI97], cooperative caching [RL04, RLZ06, TC03], and bandwidth-efficient updates [MDFK97]. As we assume unmodified web caches, this line of work does not have a direct impact on the design of Orestes. In the past twenty years, numerous cache prefetching schemes have been proposed for browser, proxy, and CDN caches [PM96, Bes96, KLM97, MC+98]. Today, these schemes are not widely used in practice due to the overhead in the cache and excess network usage caused by wrong prefetching decisions. Prefetching at the level of caches slightly affects cache hit rates, but is orthogonal to Orestes, so we do not discuss it in detail here.

6.1.2 Cache Coherence: Expiration-Based and Invalidation-Based Caching

Cache coherence is a major concern for any caching approach. Similar to distributed databases, there is an inherent trade-off in caching approaches between throughput and latency on the one side and ease-of-use and provided correctness guarantees on the other. Often in practice, developers even have to bypass caching manually in order to achieve the desired consistency level [NFG+13, ABK+15].

Expiration-Based Caching

In the literature, the idea of using an expiration-based caching model has previously been explored in the context of file and search result caching [DFJ+96, APTP03a, LGZ04, BAM+04, KFD00, KB96, HKM+88, Mog94]. Expiration-based caching (also referred to as pull-based caching [LLXX09]) can be categorized into *TTL-based*, *lease-based*, and *piggybacking* strategies. Expiration-based caching usually involves *asynchronous* validation of cached entries, i.e., the freshness is validated when cached data is expired. *Synchronous*

validation (polling-every-time [LLXX09]) only reduces bandwidth, but not latency, which makes it inapplicable for the goal of this work.

Leases

The lease model is a concept from the distributed file systems literature [HKM+88,Mog94] originally proposed by Gray et al. [GC89]. A lease grants access to a local copy of an object until a defined expiration time [Vak06]. It is therefore similar to a lock, however combined with a limited validity to mitigate the problem of client failures and deadlocks. For the duration of the lease, the holder has to acknowledge each server-side invalidation in order to maintain strong consistency. A lease combines the concepts of expiration-based and invalidation-based cache coherence: while the lease is still active, the client will receive invalidations, afterwards the client has to acquire a new lease which is accompanied by a renewed expiration time [Vak06].

A central problem of leases is that long leases may incur high waiting times for updates, if a client does not respond, whereas short leases imply a large control message overhead and increase latency. A major refinement of the basic lease scheme addressing this problem are volume leases proposed by Yin et al. [YADL99, YADL98]. A volume groups related objects together and introduces a coarser level of granularity. Clients need to have both an active volume and object lease in order to perform an object read. By giving volume leases short expiration times and object leases longer expiration times, writes experience shorter delays and the message overhead for object lease renewals is reduced. By additionally incorporating access metrics, adaptive leases introduced by Duvuri et al. [DST03] can further optimize the read-versus-write latency trade-off by dynamically calculating lease durations.

The lease model is not well-suited for client caches in the web. Especially with mobile devices and high-traffic websites, leases on objects will usually expire, as client connectivity is intermittent and potentially thousands of clients will hold leases on the same object. The effect would therefore be similar to a TTL-based model, where the server has to delay writes until the respective TTL is expired.

However, clients can achieve a model similar to leases using real-time queries in Orestes. In order to lower the Δ of the Cache Sketch's Δ-atomicity guarantee, clients can register a real-time query to receive notifications upon changes of currently used objects. These notification are still asynchronous (similar to CDN invalidations), but usually received in the order of one round-trip latency (cf. Section 4.7). Implicitly, each continuous query in Orestes has an assigned expiration to limit resource consumption, which makes this approach very similar to an opt-in lease. However, the central difference is that subscribed clients do not need to acknowledge updates, so writes do not get blocked. This advantage comes at the cost of lowering linearizability in the classic lease-model to (Δ, p)-atomicity because of notification latency.

Piggybacking

Piggybacking schemes batch together validations or invalidations and transfer them in bulk. Krishnamurthy et al. [KW97] proposed Piggyback Cache Validation (PCV). PCV is designed for proxy caches to decrease staleness by proactively renewing cached data. Each time a proxy cache processes a request for an origin server, the local cache is checked for objects from that origin that are either expired or will be expired soon. The revalidation requests for these objects are then batched and attached (piggybacked) with the original request to the origin server. With sufficient traffic to frequently piggyback revalidations, this can reduce latency and staleness as cached data is refreshed before it is requested by a client. Piggyback Server Invalidation (PSI) [KW98] follows a similar idea: when the server receives a revalidation request based on the version, the server additionally piggybacks a list of resources that have been invalidated since that modification, too. PCV and PSI can be combined in a hybrid approach [KW99, CKR98]. The idea is to use PSI, if little time has passed since the last revalidation, and PSV otherwise as the overhead of invalidation messages is smaller, if few objects have changed.

These piggybacking schemes only work for shared caches (proxy caches, ISP caches, reverse proxy caches) and require modifications of the caching logic of HTTP [FR14]. As a central premise of this work is to keep the semantics of HTTP caching intact to potentially exploit every web cache, Orestes does not use PCV and PSI. Instead, piggybacking is used for transparent renewals of the Cache Sketch: messaging overhead and request latency are reduced.

Time-to-Live (TTL)

TTLs are usually assumed to be implicit, i.e., they are not explicitly defined by the application as they are not known in advance [LLXX09]. HTTP adopted the TTL model as it is the most scalable and simple approach to distribute cached data in the web [FGM+99, IET15]. The core of every TTL scheme is the **latency-recency trade-off**. Cao et al. [BR02] propose to employ user profiles for browsers that express the preference towards either higher recency or lower latency. Fixed TTL schemes that neither vary in time nor between requested objects/queries lead to a high level of staleness [Wor94]. We think this approach is not applicable in the modern web, as users expect maximum performance without noticeable staleness. It therefore becomes the task of the application and the cloud services to minimize and hide any occurring staleness.

A popular and widely used TTL estimation strategy is the Alex protocol [GS96] (also referred to as Adaptive TTL [RS03, Wan99, KW97, CL98]) that originates from the Alex FTP server [Cat92]. It calculates the TTL as a percentage (e.g., 20%) of the time since the last modification, capped by an upper TTL bound. Simulations have shown that for certain workloads this scheme can contain the staleness rate to roughly 5% [GS96]. In an AT&T trace analyzed by Feldmann et al. [FCD+99] for a low percentage of 20%, the overall staleness for the Alex protocol was 0.22%. On the other hand, 58.5% of all requests were revalidations on unchanged resources. The Alex protocol is similar to the query TTL

update strategy in Orestes, but has the downside of neither converging to the actual TTL nor being able to give estimates for new queries.

Alici et al. proposed an adaptive TTL computation scheme for search results on the web [AAO+12]. In their incremental TTL model, expired queries are compared with their latest cached version. If the result has changed, the TTL is reset to a minimum TTL, otherwise the TTL is augmented by an increment function (linear, polynomial, exponential) that can either be configured manually or trained from logs. Though the model is adaptive, it requires offline learning and assumes a central cache co-located with the search index. If the time of an invalidation is known (e.g., in a database setting instead of a search engine application), TTLs can potentially be computed more precisely than in their scheme, which only relies on subsequent reads to detect staleness and freshness.

The central difference between TTL-based schemes from related work and our Cache Sketch approach is that over- or underestimating TTLs only reduces efficiency, but does not affect the correctness guarantees provided by Orestes. When used as the only means of cache coherence, the above approaches exhibit potentially high levels of staleness that are only bounded by the maximum permissible TTLs. We do, however, strongly build upon previous work on TTL-based caching, as the quality of TTL estimates directly influences the effectiveness of the Cache Sketch and the cache hit ratio.

Invalidation-Based Caching

Arguably, invalidations are the most intuitive mechanism to deal with updates of cached data. In this case, the server is responsible for detecting changes and distributing invalidation messages to all caches that might have cached that data. Invalidation-based caching can either be invalidation-only or update-based [LLXX09]. In the invalidation-only scheme, stale content is only evicted from the cache and reloaded upon the next cache miss. With the update-based approach, new versions are proactively pushed to caches. Almost every CDN works with the invalidation-only scheme in order to limit network overhead [PB08]. A notable exception is the academic Coral CDN, which is mainly designed for static, non-changing content and hence supports the update-based model [FFM04, Fre10].

Candan et al. [CLL+01b] first explored automated invalidation-based web caching with the CachePortal system that detects changes of HTML pages by analyzing corresponding SQL queries. CachePortal is a reverse proxy cache with two major components. The *sniffer* is responsible for logging incoming HTTP requests and relating them to SQL queries detected at the JDBC database driver level to produce a query-to-URL mapping. The *invalidator* monitors update operations and detects which queries are affected to purge the respective URLs. The authors find the overhead of triggers or materialized views prohibitive and hence rely on a different approach. For each incoming update, a polling query is constructed. The polling query is either issued against the underlying relational database or an index structure maintained by the invalidator itself. If a non-empty result is returned, the update changes the result set of a query and a URL invalidation is triggered.

The number of polling queries is proportional to both the number of updates and the number of cached queries. CachePortal therefore incurs a very high overhead for caching on the database and the invalidator.

Unlike InvaliDB, the load on the invalidator cannot be scaled horizontally. Furthermore, the approach is strictly specific to a fixed set of technologies (JDBC, Oracle RDBMS, BEA Weblogic application server) and only covers reverse proxy caching. Furthermore, the mapping from HTTP requests to queries breaks under concurrent access, as it is based on observing queries within a time window. If multiple users request different resources at the same time, the mapping is flawed. Orestes, on the other hand, eliminates the indirection by exposing queries and objects directly via HTTP, so that the mapping is always unambiguous.

Dilley et al. [KLM97] proposed the invalidation-based protocol DOCP (Distributed Object Consistency Protocol). The protocol extends HTTP to let caches subscribe to invalidations. DOCP therefore presents an effort to standardize invalidation messages, which are provided through custom and vendor-specific approaches in practice (e.g., the HTTP PURGE method [Kam17]). The authors call the provided consistency level *delta-consistency* which is similar Δ-atomicity, i.e., all subscribed caches will have received an invalidation of a written data item at most *delta* seconds after the update has been processed. Our Websocket-based query subscription mechanism is more powerful than DOCP as it allows subscriptions to an arbitrary number of conditions and queries multiplexed over a single connection to the origin. It is also update-based, while DOCP is invalidation-only.

Worrel [Wor94] studied hierarchical web caches to derive more efficient cache coherence schemes. He designed an invalidation protocol specifically suited for hierarchical topologies and compared it to fixed TTL schemes w.r.t. server load, bandwidth usage, and staleness. He found the scheme to be superior in terms of staleness and competitive to TTLs in server load and bandwidth usage. In Orestes, the topology of caches is unknown and hence our invalidation scheme is agnostic regarding the internal structure of CDNs and interception caches. A particular problem of deep hierarchies is the *age penalty* problem studied by Cohen et al. [CK01]: older content in the upper levels of the hierarchy propagates downstream and negatively impacts dependent caches. Orestes addresses this problem by decoupling the age of cached data from freshness through proactive use of the Cache Sketch.

An alternative to cache invalidation was proposed by the Akamai founders Leighton and Lewin [LL00]. The idea is to include a hash value of the content in the URL, so that upon changes the old version does not get invalidated, but instead is superseded by a new URL containing the new fingerprint (*cache busting*). This approach is widely used in practice through build tools such as Grunt, Gulp, and Webpack. The downside is that this scheme only works for embedded content that does not require stable URLs (e.g., images and scripts). In particular, it cannot be applied to database objects, query results, and HTML

pages. Furthermore, it only allows for invalidation at application deployment time and not at runtime.

Edge Side Includes (ESI) [TWJN01] take the approach of Leighton and Lewon a step further by shifting template-based page assembly to the edge, i.e., CDN caches. ESI is a simple markup language that allows to describe HTML pages using inclusion of referenced fragments that can be cached individually. Rabinovich et al. [RXDK03] proposed to move ESI assembly to the client arguing that the rendering of ESI on the edge adds to the presumed main bottleneck of last-mile latency [Nag04]. While ESI has not gained relevance for the browser, the underlying idea is now widely used in practice [BPV08]. Every single-page application based on Ajax and MVC-frameworks for rendering employs the idea of assembling as website from individual fragments usually consumed from cloud-based REST APIs. Orestes is fundamentally based on this paradigm, as it enables the Backend-as-a-Service model on top of existing database systems with fine-granular caching of database records and query results.

Bhide et al. [BDK+02] also proposed a scheme to combine invalidation- and expiration-based caching in proxies. They argue that web workloads are inherently unpredictable for the server and therefore propose a **Time-to-Refresh** (TTR) computed in the proxy to replace TTLs. TTRs are computed for each data item based on previous changes and take a user-provided *temporal coherency requirement* into account that expresses the tolerable staleness based on data values (e.g., the stock price should never diverge by more than one dollar). TTRs therefore dynamically reflect both the rate of change (as expressed in TTLs) and the desired level of coherence. Bhide et al. present algorithms to mix the communication-intensive expiration-based revalidations through TTRs with invalidations. Orestes, in contrast, uses a coherency requirement based on time (Δ-atomicity) which is easier to reason about and applicable to all data types. Furthermore, the trade-off towards lower TTRs is very different, as the Cache Sketch refreshes only impose little overhead compared to per-object revalidations. Additionally, invalidations in Orestes are not specific to each proxy cache, which eliminates the need to avoid costly invalidations and associated server-side state.

Browser Caching

Traditionally, browsers only supported transparent caching at the level of HTTP, as specified in the standard [FGM+99]. The only recent additions to the original caching model are means to specify that stale content may be served during revalidation or unavailability of the backend [Not10], as well as an immutability flag to prevent the browser from revalidating upon user-initiated page refreshes [McM17]. For workloads of static content, Facebook reported that the browser cache served by far the highest portion of traffic (65.5%) compared to the CDN (20.0%) and reverse proxy caches (4.6%) [HBvR+13].

Two extensions have been added to browsers in order to facilitate offline website usage and application-level caching beyond HTTP caching. *AppCache* was the attempt to let the server specify a list of cacheable resources in the cache manifest. The approach

suffered from various problems, the most severe being that no resource-level cache coherence mechanism was included and displaying non-stale data required refreshing the manifest [Ama16]. To address these problems, *Service Workers* were proposed. They introduce a JavaScript-based proxy interface to intercept requests and programmatically define appropriate caching decisions [Ama16]. Still, cache coherence is not in the scope of Service Workers and has to rely on application-specific heuristics. A set of best practices for developing with Service Workers was published by Google and termed *Progressive Web Apps* [Mal16]. Orestes can leverage Service Workers to not only employ the Cache Sketch for data requested via explicit JavaScript API calls, but also cache HTML documents, scripts, stylesheets, and images implicitly fetched by the browser.

To structure client-side data beyond a hash table from URLs to cached data and enable processing of the data, three techniques have been proposed and partly standardized [Cam16] (cf. Section 2.4.2). *LocalStorage* provides a simple key-value interface to replace the use of inefficient cookies. *Web SQL Database* is an API that exposes access to an embedded relational database, typically SQLite. The specification is losing traction and will likely be dropped [Cam16, p. 63]. *IndexedDB* is also based on an embedded relational database system. Data is grouped into databases and object stores that present unordered collections of JSON documents. By defining indexes on object stores, range queries and point lookups are possible via an asynchronous API. These storage capabilities of the browsers can be leveraged in Service Workers and hence Orestes can provide offline query processing on records previously fetched through the CRUD API and queries (cf. [Twe17, Sch17]).

Web Performance

A central finding of performance in modern web applications is that perceived speed and page load times (cf. Section 2.4.1) are a direct result of physical **network latency** [Gri13]. The HTTP/1.1 protocol that currently forms the basis of the web and REST APIs suffers from inefficiencies that have partly been addressed by *HTTP/2* [IET15]. Once adopted by caches, CDNs, and end devices, its push model will allow to simplify the query result representation in Orestes to use ID-lists without performance downsides: along the ID-list the referenced objects get pushed a single round-trip instead of two. As operations are furthermore multiplexed through a single TCP connection, any refreshes of the Cache Sketch can be performed without causing head-of-line blocking [Gri13] for other requests such as queries and CRUD operations.

Wang et al. [WKW16] explored the idea of offloading the client by preprocessing data in proxies with higher processing power. Their system Shandian evaluates websites in the proxy and returns them as a combination of HTML, CSS, and JavaScript including the heap to continue the evaluation. For slow Android devices, this scheme yielded a page load time improvement of about 50%. Shandian does, however, require a modified browser which makes it inapplicable for broad usability in the web. The usefulness of the offloading is also highly dependent on the processing power of the mobile device, as the

proxy-side evaluation blocks delivery and introduces a trade-off between increased latency and reduced processing time.

Netravali et al. proposed Polaris [NGMB16] as an approach to improve page load times. The idea is to inject information about dependencies between resources into HTML pages, as well as JavaScript-based scheduling logic that loads resources according to the dependency graph. This optimization works well in practice, because browsers rely on heuristics to prioritize fetching of resources. By an offline analysis of a specific server-generated website, the server can determine actual read/write and write/write dependencies between JavaScript and CSS ahead of time and express them as a dependency graph. This allows parallelism where normally the browser would block to guarantee side-effect free execution. Depending on the client-server round-trip time and bandwidth, Polaris yields a page load time improvement of roughly 30%. The limitations of the approach are that it does not allow non-determinism and that dependency graphs have to be generated for every client view. For personalized websites, this overhead can be prohibitive. Furthermore, the approach assumes server-side rendering as opposed to Orestes that is designed for single-page applications.

6.1.3 Query-Level Caching

Query caching has previously been tackled from different angles in the context of distributed database systems [DFJ+96, APTP03a, LGZ04, BAM+04, KFD00, KB96], mediators [LC99, CRS99, ACPS96], data warehouses [DRSN98, KP01, LKAP01], peer-to-peer systems [GMA+08, PH03, KNO+02], and web search results [BLV11, CJP+10, BBJ+10, AAO+11]. Most of this work is focused on the details of answering queries based on previously cached results, while only few approaches also cover cache coherence which is in the focal point of this thesis.

Peer-to-Peer Query Caching

Garrod et al. have proposed Ferdinand, a proxy-based caching architecture forming a peer-to-peer distributed hash table (DHT) [GMA+08]. When clients query data, the proxy checks a local, disk-based map from query strings to result sets. If the result is not present, a lookup in another proxy is performed according to the DHT scheme. The consistency management is based on a publish/subscribe invalidation architecture. Each proxy subscribes to multicast groups corresponding to the locally cached queries. A limiting assumption of Ferdinand is that updates and queries follow a small set of fixed templates defined by the application. This is required to map updates and queries to the same publish/subscribe topics, so that only relevant updates will be received in each caching proxy.

Orestes does not limit applications to the use of prepared queries. Instead, InvaliDB decouples the problem of query matching from broadcasting invalidations. Peer-to-peer query caching has also been employed for reducing traffic in file sharing protocols [PH03], as well as to distributed OLAP queries [KNO+02]. IPFS [Ben14] also employs a peer-to-peer

approach with DHTs to cache file chunks across many users. In contrast to Orestes, it cannot accelerate the delivery of web content, as the overhead of metadata lookups is prohibitive for low latency.

Mediators

In contrast to reverse proxies that can serve any web application, mediators are typically designed to handle one specific use case, type of data, or class of data sources. Related work in this area is mostly concerned with constructing query plans using semantic techniques to leverage both locally cached data from the mediator as well as distributed data sources [LC99, CRS99, ACPS96]. To comply with the web caching model, Orestes cannot evaluate query plans in caches. Therefore, we instead reuse previous query results and individual records without semantic analysis of query predicates.

Query Caching Proxies and Middlewares

DBProxy, DBCache, and MTCache [APTP03a, LGZ04, BAM+04] rely on dedicated database proxies to generate distributed query plans that can efficiently combine cached data with the original database. However, these systems need built-in tools of the database system for consistency management and are less motivated by latency reduction than by reducing query processing overhead in the database similar to materialized views [Shi11].

DBProxy [APTP03a] is designed to cache SQL queries in a proxy, similar to a reverse proxy cache or a CDN. DBProxy adapts the schema as new queries come in and learns query templates by comparing queries to each other. When a query is executed in the database, results are stored in DBProxy. To reuse cached data, DBProxy performs a containment check that leverages the simplicity of templates to lower the complexity of traditional query containment algorithms [APTP03b]. DBProxy receives asynchronous updates from the database system and hence offers Δ-atomicity by default. The authors describe monotonic reads and strong consistency as two potential options for reducing staleness in DBProxy, but do not evaluate or elaborate on the implications. DBProxy assumes that the application runs as a Java-based program in the proxy and enhances the JDBC driver to inject the caching logic. The authors do not discuss the impact of transactional queries on correctness when they are invisible to the database system.

DBCache [BAM+04, LKM+02, BAK+03] and *MTCache* [LGZ04] are similar approaches that employ nodes of relational database systems for caching (IBM DB2 and Microsoft SQL Server, respectively). Both systems rewrite query plans to exploit both local and remote data. In DBCache, the query plan is called a *Janus* plan and consists of a probe query and a regular query. The probe query performs an existence check to determine whether the local tables can be used for the query. Afterwards, a regular query containing a clause for both local and remote data is executed. Cache coherence is based on the DB2 replication interface that asynchronously propagates all updates of a transaction. MTCache uses the corresponding asynchronous replication mechanism from Microsoft SQL Server. It maintains the cache as a set of materialized views and performs cost-based optimization

on query templates to decide between local and remote execution. Due to their strong relation to database replication protocols, DBCache and MTCache are effectively lazily populated read replicas. Orestes is orthogonal to caching proxies and mid-tier caches as they serve the purpose of database offloading and can be combined with our end-to-end caching for latency reduction.

Labrinidis et al. proposed *WebViews* as a technique of caching website fragments [LR01a, LR00]. A WebView refers to HTML fragments generated by database queries, e.g., a styled table of stock prices. Through a cost-based model, WebViews are either materialized in the web servers, in the database, or not at all. The authors found that materialization in the web servers is generally more effective than materialization in the database by at least a factor of 10, since it incurs fewer round-trips to the database. This is in accordance with Orestes where query results are cached in arbitrary caches as near as possible to the client to avoid network round-trips.

Search Result Caching

According to Bortnikov et al. [BLV11], caching approaches for search results can be classified into coupled and decoupled design. In a *decoupled design* (e.g., [CJP+10]), the caches are independent from the search index (i.e., the database), while a *coupled design* is more sophisticated and actively uses the index to ensure cache coherence. Blanco et al. investigated query caching in the context of incremental search indices at Yahoo and proposed a coupled design [BBJ+10]. To achieve cache coherence, their cache invalidation predictor (CIP) generates a synopsis of invalidated documents in the document ingestion pipeline. This summary is checked before returning a cached search query to bypass the cache when new indexed document versions are available. Unlike our evolving Cache Sketch, the synopses are immutable, created in batch, and only used to predict likely invalidations of server-side caches.

Bortnikov et al. [BLV11] improved the basic CIP architecture using realistic workloads, more efficient cache replacement algorithms and optimizations to deal with less popular documents. Alici et al. [AAO+11] were able to achieve comparable invalidation accuracy using a timestamp-based approach where an invalidation is detected by having the cache distribute the timestamp metadata of a cached query to all responsible search servers. These confirm freshness, if they did not index updated document versions, nor new documents that also match the search term, based on the respective timestamps. The broadcast is less expensive than reevaluating the query, but not suitable for latency reduction in a web caching scenario.

6.1.4 Summary Data Structures for Caching

Bloom Filters for Caching

A related system using Bloom filters for web caching is Summary Cache proposed by Fan et al. [FCAB00] which employs Bloom filters as metadata digests in cooperative web caches.

The approach, however, is fundamentally different to ours, as these summaries ("cache digests") are generated in intervals to communicate the set of locally available cached data to cooperating web caches. In the context of Summary Cache, Counting Bloom filters were introduced in the literature for the first time. Since each server has to delete URLs from the Bloom filter when they are replaced from the cache, a removal operation is necessary. In this setting, considerations about the optimal Bloom filter size and invalidations are not required as the Bloom filter only serves as a means of bandwidth reduction.

Recently, cache fingerprinting has been proposed for improving HTTP/2 push [ON16]. The idea is to construct a digest of the browser cache's contents – similar to Summary Cache – to efficiently identify resources that are already available in the client and therefore do not have to be pushed by the server. Instead of Bloom filters, Golomb-compressed sets (GCS) [PSS09] are used. GCS exploit the fact that in a Bloom filter with only one hash function, the differences between values follow a geometrical distribution [MU05]. This pattern can be optimally compressed using Golomb-coding, yielding a smaller size than Bloom filters. The fingerprinting scheme has not been standardized, yet, but an experimental implementation is available in the H2O web server [Hev16].

In NoSQL systems, Bloom filters are frequently used to accelerate storage engines based on log-structured merge-trees (LSMs). Google's BigTable [CDG+08] pioneered the approach that has been adopted in various systems (e.g., Cassandra, LevelDB, HyperDex, WiredTiger, RocksDB, TokuDB) [LM10, EWS12, GD11]. In BigTable, data is stored in immutable SSTables located on disk. In order to avoid disk I/O for the lookup of a key for each SSTable, a Bloom filter is loaded into memory. Only when the check is positive, the SSTable is queried on disk. In contrast to the Cache Sketch, the Bloom filters only need to be constructed once, as on-disk data is immutable. The Orestes prototype extends the Bloom filter hash function implementation (Murmur) of Cassandra's LSM-engine to Counting Bloom filters and JavaScript-based Bloom filters [GSW+15].

Alternatives to Bloom Filters

Space-efficient alternatives to Bloom filters have been proposed. While Golomb-coded sets [PSS09] achieve slightly smaller sizes, they are not suited for Orestes, as fast $O(1)$ lookups are not possible. Mitzenmacher [Mit02] proposed Compressed Bloom Filters. They are based on the observation that a very sparse Bloom filter can be efficiently compressed by an entropy encoding like Huffman and arithmetic codes [Rom97]. However, due to the size of the uncompressed filter, memory consumption is infeasible for the client. Using several Blocked Bloom filters with additional compression would mitigate this problem, but increase the complexity and latency of lookups [PSS09]. Cuckoo filters have been proposed by Fan et al. [FAKM14] as a more space-efficient alternative to Counting Bloom filters. However, they are not applicable for the Server Cache Sketch as the number of duplicate entries is strictly bounded. If Cuckoo Filters were used, updates to the Cache Sketch would fail during phases of many writes, implying later false negatives. Matrix filters proposed by Porat et al. [PPR05, Por09] achieve the lower limit of required space

for a given false positive rate. This advantage is contrasted by linear lookup time and a complex initial construction of the data structure. As Bloom filters are already within a factor of 1.44 of the theoretical lower bound of required space [BM03] and offer $O(1)$ inserts and lookups, they are the best choice for the Cache Sketch. Kirsch et al. [KM06] showed that the use of a linear combination of two independent hash functions reduces the amount of costly hash computations without loss of uniformity. The Cache Sketch implements this optimization. An overview of other Bloom filter variants and applications is given by Broder and Mitzenmacher [BM03] and Tarkoma et al. [TRL12].

In summary of the above, Orestes separates itself from previous work on caching in multiple aspects. First, it uses existing HTTP infrastructure and does not require custom caching or replication servers. Employing stochastic models, this work provides fine-grained TTL estimates for query results, records, and files. The expiration- and invalidation-based caching models in Orestes are combined by a scalable real-time query invalidation pipeline. Furthermore, the cost-based optimization and flexibility of the Cache Sketch yield a tunable trade-off between latency, consistency, and server load by adapting to the workload at runtime.

6.2 Geo-Replication

6.2.1 Replication and Caching

The goal of replication is to increase read scalability and to decouple reads from writes to offload the database and reduce latency. Replication can protect the system against data loss. In case of geographically distributed replicas (geo-replication), read latency for distributed access from clients is improved, too [BBC+11, SVS+13, KPF+13, LFKA13, CDE+12, CDE+13, SPAL11, CDG+08, DHJ+07, CRS+08, TPK+13]. In this setting, a central constraint is that intra-data center latencies are small (<5 ms), while inter-data center communication is expensive (50-150 ms) [AEM+13]. Caching can be viewed as an alternative to replication. With caching, data is fetched and stored on-demand, while with geo-replication the complete data set is synchronized between multiple replica sites, incurring higher management overhead. If replicas are allowed to accept writes (multi-master), considerable coordination is required to guarantee consistency. Orestes' caching distinguishes between caches that require expensive updates (invalidation-based caches) and passive caches that do not incur any overhead to the server (expiration-based caches).

Charron-Bost et al. [CBPS10, Chapter 12] and Öszu and Valduriez [ÖV11, Chapter 13] provide a comprehensive review of replication techniques. We will focus on a discussion of exemplary, influential geo-replicated systems and outline how their trade-offs differ from the trade-offs inherent to Orestes. The advantage of geo-replication is that consistency and transactional isolation levels can be chosen through the replication protocol and be tuned for the respective database system. A major downside is that usually these systems either have to perform multiple synchronous wide-area round-trips for consistent

updates [BBC+11, KPF+13] or only provide eventual consistency without recency guarantees [CDG+08, DHJ+07, CRS+08].

6.2.2 Eager Geo-Replication

Through eager geo-replication as implemented in Megastore [BBC+11], Spanner [CDE+13, CDE+12], and F1 [SVS+13] as well as in MDCC [KPF+13] and Mencius [MJM08], applications achieve strong consistency at the cost of higher write latencies (typically 100 ms [CDE+12] to 600 ms [BBC+11]).

Megastore

Baker et al. [BBC+11] came to the conclusion, that the cost of strong consistency and ACID transactions in highly distributed systems is often acceptable in order to empower developers. Megastore's data model is based on entity groups, that represent fine-grained, application-defined data partitions (e.g., a user's message inbox). Transactions are supported per co-located entity group, each of which is mapped to a single row in BigTable that offers row-level atomicity. Transactions spanning multiple entity groups are possible, but not encouraged, as they require expensive 2PC [Lec09].

Megastore (also available as a DBaaS called Google Cloud Datastore) uses synchronous wide-area replication. The replication protocol is based on Paxos consensus [Lam98] over positions in a shared write-ahead log. Megastore uses the Multi-Paxos [Lam01] optimization to achieve best-case performance of one wide-area round-trip per write as opposed to two round-trips with regular Paxos. This replication protocol has been improved by Kraska et al. [KPF+13] in MDCC (Multi-Data Center Consistency). They include two additional Paxos optimizations (fast and generalized Paxos) and reduce conflicts by leveraging commutativity of certain updates.

To allow consistent local read operations, Megastore tracks the replication status of each entity group in a per-site coordinator. In order for the coordinator to reflect the latest state of each entity group, the Paxos replication not only has to contact a quorum as in the original protocol, but has to wait for acknowledgments from each replica site. This implies that lower latency for consistent reads is achieved at the expense of slower writes.

The authors report average read latencies of 100 ms and write latencies of 600 ms. These numbers illustrate the considerable cost of employing synchronous wide-area replication. The high latency of writes is critical, as Megastore employs a form of optimistic concurrency for writes on the same entity group: if two writes happen concurrently during replication, only one will succeed. This limits the throughput to $1/l_w$, where l_w is the write latency, i.e., about 10 writes per second in the best case. Both read and write latencies are significantly lower in our approach and write throughput is not limited by latency.

Spanner and F1

Spanner [CDE+13, CDE+12] evolved from the observation that Megastore's guarantees – though useful – come at performance penalty that is prohibitive for some applications. Spanner is a multi-version database system that unlike Megastore provides efficient cross-shard ACID transactions. The authors argue: "We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions" [CDE+12, p. 4]. Spanner automatically groups data into partitions (tablets) that are synchronously replicated across sites via Paxos and stored in Colossus, the successor of GFS [GGL03]. Transactions in Spanner are based on two-phase locking and 2PC executed over the leaders for each partition involved in the transaction. Spanner serializes transactions according to their global commit times[2]. To make this feasible, Spanner introduces TrueTime, an API for high precision timestamps with uncertainty bounds implemented using atomic clocks and GPS. Each transaction is assigned a commit timestamp from TrueTime. Using the uncertainty bounds, the leader can wait until the transaction is guaranteed to be visible at all sites before releasing locks. This also enables efficient read-only transactions that can read a consistent snapshot for a certain timestamp across all data centers without any locking.

Mahmoud et al. [MNP+13] proposed an optimization for faster commits that integrates local 2PC in data centers with a Paxos consensus as to whether the transaction should commit (*replicated commit* protocol). This reduces commit latency, but comes at the cost of high read latency, since every read needs to contact a majority of data centers to only read committed data.

F1 [SVS+13] and its commercial version Cloud Spanner [Bre17] build on Spanner to support SQL-based access for Google's advertising business. To this end, F1 introduces a hierarchical schema based on Protobuf, a rich data encoding format similar to Avro and Thrift [Kle17]. To support both OLTP and OLAP queries, it uses Spanner's abstractions to provide consistent indexing. A lazy protocol for schema changes allows non-blocking schema evolution [RRS+13]. Besides pessimistic Spanner transactions, F1 supports optimistic transactions that use a similar commit procedure as our DCAT approach. Each row bears a version timestamp that is used at commit time to perform a short-lived pessimistic transaction to validate a transaction's read set. In contrast to Orestes, optimistic transactions in F1 still suffer from the abort rate problem, as the read phase is latency-bound and the commit requires slow, distributed Spanner transactions (cf. Section 4.8.1).

According to the CAP theorem [Bre00], Spanner and F1 cannot be highly available systems. Brewer [Bre17] argues that in practice, however, they behave as highly available systems through engineering best practices. For example, Cloud Spanner does not rely on the public Internet to perform geo-replication, but instead transfers data over private, redundant networks owned and operated by Google.

[2] This is termed *external consistency* by the Spanner authors and known in the literature as *strict serializability* or *commit order-preserving conflict serializable (COCSR)* [WV02].

CockroachDB [Coc17] is an open-source, geo-replicated, relational database system based on the design of Spanner and F1. To support commodity hardware, CockroachDB does not use TrueTime, but instead uses NTP synchronization with hybrid logical clocks[3] [KDM+14]. As a consequence, CockroachDB cannot provide strict serializability for transactions, only serializability[4]. The transaction protocol is based on an underlying key-value store that is replicated using Raft consensus [OO13] for groups of keys. Atomicity is achieved through a locking protocol on per-record metadata, similar to Percolator [PD10]. Isolation is implemented as multi-version timestamp ordering [WV02] per consensus group and 2PC across groups. Read-write and write-write conflicts therefore cause transaction aborts, if the operations are not ordered according to the transaction begin timestamps. Like Spanner and F1, CockroachDB is prone to high read, write, and transaction latency due to synchronous geo-replication and 2PC.

Summing up, strict serializability is an important property for applications. Without this guarantee, blind writes (e.g., inserting a comment record) can be delayed arbitrarily and may never become visible. Spanner and F1 achieve strict serializability by delaying transaction commits and using high-precision clocks, while CockroachDB sacrifices the guarantee for performance reasons. Orestes, on the other hand, provides strict serializability through its optimistic protocol executed at a single site, thus preventing timing constraints between distributed data centers.

6.2.3 Lazy Geo-Replication

With lazy geo-replication as in Dynamo [DHJ+07], BigTable/HBase [CDG+08, Hba17], Cassandra [LM10], MongoDB [CD13], CouchDB [ALS10], Couchbase [LMLM16], Espresso [QSD+13], PNUTS [CRS+08], Walter [SPAL11], Eiger [LFKA13], and COPS [LFKA11] stale reads are allowed, but the system performs better and remains available during partitions.

Eventually Consistent Geo-Replication

Most asynchronously replicated NoSQL systems support using their intra-data center replication protocol for cross-data center replication. In contrast to systems with transparent geo-replication, the application needs to be explicitly configured to route read and write requests to the correct data center. MongoDB [CD13] allows tagging shards with a zone parameter to allocate data to regions based on properties (e.g., an “address” field in user documents). It also supports distributing replicas within a replica set over multiple locations. However, this comes at a cost, as replicas from another data center can be elected to masters upon network partitions and transient failures. Couchbase [LMLM16] uses the asynchronous Memcache replication protocol for geo-replication. Most RDBMSs in-

[3] Hybid logical clocks combine the benefits of logical clocks [Lam78] for simple tracking of causality with physical clocks that are within a defined drift from real time by merging both.

[4] In particular for individual operations (transactions with a single read or write), the lack of strict serializability implies that linearizability is not guaranteed in CockroachDB.

clude only limited support for geo-distributed deployments, mostly directly based on their asynchronous intra-data center replication protocols (e.g., in MySQL, MySQL Cluster, and PostgresSQL [Pos17]).

CouchDB [ALS10] has a multi-master replication protocol that was designed for heavily geo-distributed setups from device-embedded instances to multiple data centers. As writes are allowed on each slave, conflicts are tracked using hash histories [AEM+13], an alternative to vector clocks [DHJ+07] for causality tracking. Quorum systems such as Dynamo, Cassandra, and Riak [LM10, DHJ+07] require location-awareness for each key's preference list, i.e., the information on whether the responsible database nodes are local to the data center or connected through wide-area networks. Cassandra, for example, supports configuring remote site behavior through topology strategies and per operation quorums. These quorums define whether data is replicated purely asynchronously (e.g., for analytics) or whether a remote cluster has to participate in the overall quorum ("EACH_QUORUM") [CH16]. Riak distinguishes between a source cluster for operational workloads and sink clusters that do not participate in quorums and only asynchronously receive writes from the source cluster.

BigTable and HBase [CDG+08, Hba17] are synchronously replicated within a data center at the file system level (GFS and HDFS [GGL03], respectively), but offer asynchronous wide-area replication, mainly for purposes of disaster recovery. LinkedIn's Espresso is a document store that uses asynchronous master-slave replication within a data center built on top of a change data capturing system called Databus [DBS+12]. Subscribers to this replication bus can be placed in remote data centers.

PNUTS

Causal consistency is the strongest level of consistency achievable without inter-data center coordination [LFKA11]. Yahoo's PNUTS system [CRS+08] was influential in this respect, as it combines stronger consistency with a geo-replicated design. PNUTS leverages the observation that updates for a particular record tend to originate from the same region. Therefore, the primary is chosen per record ("record-level mastering"). Updates are propagated through an asynchronous pub/sub message broker that enforces a serial order for updates on the same key which guarantees causal consistency per key (termed "timeline consistency"). Reads can be directed to any replica, if timeline consistency is sufficient ("read-any"), or explicitly request monotonic reads ("read-critical") or strong consistency ("read-latest") else. In each region, records are range-sharded and stored in MySQL. The design of PNUTS presents a compromise between multi-master and master-slave replication. It decouples failures of primaries for different records and achieves low latency, if the primary only receives writes from nearby clients.

Eiger, COPS, and Walter

Eiger [LFKA13] and COPS [LFKA11] are two approaches for providing full causal consistency for asynchronous replication. Eiger and COPS have strong similarities, their major

difference is that causality tracking in COPS is based on per-record metadata, while Eiger tracks dependencies between operations. COPS introduces the notion of *causal+ consistency* that combines causal consistency with guaranteed convergence of writes. While COPS is not the first system to provide causal+ consistency for geo-replication, it is the first that is not based on unscalable use of the database log like Bayou [DPS+94] and PRACTI [BDG+06]. The key idea of COPS is to have clients attach metadata of causally relevant read operations to each write operation. During replication at a remote site, a write is only applied if all causal dependencies have also been applied already. To ensure convergence, conflicting writes are resolved using a commutative and associative handler (e.g., last-writer-wins). COPS also introduces a two-phase commit algorithm for read-only transactions that only see causally consistent records.

Walter [SPAL11] extends the COPS approach for causality tracking to transactions, by introducing *Parallel Snapshot Isolation* as an isolation level that relaxes snapshot isolation to allow different transaction orderings on different sites. Bailis et al. have proposed *bolt-on causal consistency* [BGHS13] that provides causal consistency at the client side. The idea is similar to the concept behind COPS: writes are only made visible for reads, once their casual dependencies are available. However, as this safety guarantee is not paired with a liveness guarantee, clients can end up reading very stale data.

The main problem of all geo-replication schemes for causal consistency is that either potential causality is tracked which imposes a large overhead or that developers are faced with the burden of explicitly declaring causal relationships. In Orestes, the key insight for providing efficient causal consistency is that the Cache Sketch represents a natural time bound for all records that must satisfy causal consistency, since the primary site guarantees causal consistency.

Pileus

Our work has similarities with Pileus [TPK+13] proposed by Terry et al. from Microsoft Research. Pileus also achieves low latency, single round-trip writes, and bounded staleness. It is based on an SLA concept, in which developers can annotate consistency levels and latency bounds with utility values. For example, an application could specify that up to 5 minutes of staleness are tolerable and then define the monetary value of requests that return in 200 ms, 400 ms or 600 ms. Pileus has a key-value data model with CRUD-based access. It employs a primary site for updates and all geo-replicated secondary sites are asynchronously replicated.

Clients are responsible for selecting a replica by evaluating the SLA and returning the sub-SLA (a combination of a consistency and latency requirement with a utility) that has the highest utility multiplied by the probability of meeting the consistency and latency requirement. Data is then read from the replica that maximizes the selected sub-SLA. The decision whether a replica can satisfy a consistency requirement is based on computing a minimum acceptable read timestamp that indicates how far a replica is allowed to lag behind the primary without violating the consistency level. To make this feasible, clients

need to frequently collect information about network latency and replication lag from all replicas. The consistency levels are similar to Orestes (strongly, eventual, monotonic reads, Δ-atomic, causally consistent). However, Pileus assumes a higher Δ (typically minutes [TPK+13, p. 316]), as otherwise polling from replicas becomes inefficient and strict clock synchronization would be required.

In a follow-up work, Pileus was extended by the ability for dynamic primary/secondary reconfiguration in order to maximize global utility of SLAs in a system called Tuba [AT14]. Here, a configuration service periodically collects observed latencies and SLA violations from clients and selects a new configuration with the best utility-to-cost ratio. Potential reconfigurations include adding or switching a primary site and changing the replication factor and replica locations. Clients need to always be aware of configuration changes in order not to perform strongly consistent reads and writes on non-primaries. Compared to Pileus, Tuba increases the probability of strongly consistent reads from 33% to 55%.

The major differences between Pileus/Tuba and Orestes are:

- Orestes supports **queries**, while Pileus is limited to CRUD.
- Our approach to low latency relies on standard **web caches** instead of custom replicas.
- Staleness bounds in Orestes do not depend on **clock synchronization**.
- Cache Sketches **scale** to an arbitrary number of caches, as staleness information is consolidated in one Bloom filter instead of being polled from each replica.

Tao

Tao is an example of a system that combines geo-replication with caching. Bronson et al. [BAC+13] describe the system that stores Facebook's multi-petabyte social graph. The data is held in a sharded MySQL which is asynchronously replicated across data centers. Caching is performed at two levels of cache tiers. The *leader cache tier* is located in front of MySQL and is allowed to perform writes on it. Multiple *follower cache tiers* service requests to their nearest application servers and forward requests to the leader if necessary. Each tier consists of many modified Memcache [Fit04] servers with custom memory allocation and LRU cache eviction schemes [XFJP14, NFG+13]. Tiers are sharded through consistent hashing to avoid reshuffling of data in case of failures. To mitigate popularity-induced hotspots, each shard inside a tier can be master-slave replicated. The tiers behave like synchronous write-through caches, i.e., when a write request arrives at a follower tier's Memcache shard, it is forwarded to the respective leader shard. If the current data center is the master for that data item, the write is performed on the corresponding MySQL shard. Otherwise, it is forwarded to the leader tier of the master data center. When the write is complete, invalidation messages are issued to every cache holding that data item. Cache coherence is thus asynchronous, i.e., there are no consistency guarantees, but anecdotally the lag is in the order of one second [BAC+13]. Tao handles roughly 1 billion reads per second with a read-heavy workload (over 99% reads).

Lu et al. [LVA+15] performed extensive consistency checking for Tao's two-level caching architecture by sampling requests. They analyzed violations of linearizability, read-your-writes consistency, and per-object sequential consistency. For Facebook's workload, the violation are reported to be very rare (e.g., 0.00151% in case of linearizability). The authors attribute this to the fact that writes are very rare and only 10%-25% of all objects experience both reads and writes. The effects on transactional isolation were not measured, as the distributed nature of transactions made a tracing and checking approach impossible. In contrast to Orestes, Tao does not provide consistency guarantees (but only violation probabilities) and also does not allow developers to fine-tune performance against consistency on a per-operation basis.

Tunable Consistency and the Latency-Consistency Trade-Off

The idea of exposing tunable consistency to developers is also found in other systems. In many applications some operations need to be performed with strong consistency (e.g., password checking), while eventual consistency is acceptable for others (e.g., adding a product to the shopping cart). Both Twitter and Facebook have sub-systems providing strong consistency for operations on critical data [Sch16, LVA+15]. In Google's Megastore [BBC+11], weakly consistent reads are allowed for performance reasons despite strongly consistent updates. In Gemini [LPC+12], red (strongly consistent) and blue (weakly consistent) operations are distinguished for geo-replicated storage. Gemini maximizes the use of fast, locally executed blue operations by determining when an operation is commutative to every potentially concurrent operation.

Kraska et al. [KHAK09] proposed to attach SLAs to objects in order to include the cost as an optimization factor for cloud-based storage systems (*consistency rationing*). The two SLA classes A and C reflect data that is always handled with strong or weak consistency, respectively, while class B is continuously optimized according to a cost function. Florescu and Kossmann [FK09] argue that most cloud-based applications are not concerned with the concrete level of consistency, but the overall cost of the application.

Application complexity is usually increased by different consistency choices [LLC+14]. Guerraoui et al. [GPS16] proposed *Correctables* as a new programming model that abstracts different consistency levels. The main idea is to provide a Promise-based [LS88b] interface that can either directly execute an operation at the desired consistency level or return multiple results with increasing consistency and delay. For example, in a ticket checkout process, a potentially stale stock counter could be returned first to proceed, when it is sufficiently high, with the option to abort shortly afterwards, if the actual stock value is already zero. A similar scheme is used in Meteor [HS16], to hide potentially slow write operations from users (*latency compensation*). *Correctables* could be combined with Orestes to return results that are Δ-atomic first, while in the background, a linearizable result is fetched.

Geo-Replica Placement

In contrast to caching, the decision *where* to replicate data involves intimate knowledge of workloads and access patterns. Web caching is inherently more adaptive than replication, as data is materialized on demand and as near to the client as possible. Wu et al. [WBP+13] have proposed SPANStore to address replica-placement in multi-cloud environments. SPANStore minimizes the cost of a data storage deployment based on application requirements such as latency SLOs and desired consistency levels. For each access set of an application's workload, a placement manager decides where to store data and from where to serve reads and writes. To provide transparency to the application, a client library proxies access to the different cloud data centers.

The problem of geo-replication was also studied for transactional workloads by Sharov et al. [SSMS15]. They proposed a replication framework for transactional, highly distributed workloads that minimizes latency through appropriate primary and replica placement. Zakhary et al. [ZNAE16] described a similar approach for majority-based replication. They employ a cache-like "optimistic read" optimization: instead of always reading from a majority of replicas, a passive replica (effectively a client cache) can be used and reads can be validated before transaction commit. This idea is similar to DCAT, but only leverages a single cache.

Consistency

Consistency in replicated storage systems has been studied in both theory [GLS11] and practice [BVF+12, LVA+15, Ber14]. An up-to-date and in-depth discussion of consistency in distributed systems and databases is provided by Viotti and Vukolic [VV16]. Their two main observations are that there is a complex relationship between different consistency levels and that similar guarantees are often named differently across research communities. Similar to asynchronously replicated systems [DHJ+07, CRS+08, LM10], Orestes trades consistency against performance by invalidating asynchronously and allowing stale reads. We studied the strict staleness bounds imposed by the Cache Sketch through the Monte Carlo simulation framework YMCA (cf. Section 4.3.1) that is similar to PBS proposed by Bailis et al. [BVF+12].

Lee et al. [LPK+15] proposed to decouple the problem of consistency from database system design through a system called RIFL (Reusable Infrastructure for Linearizability). RIFL builds on remote procedure calls (RPCs) with at-least-once semantics (i.e., invocations with retries) and enhances them to exactly-once semantics which are sufficient to guarantee linearizability. To this end, each request is assigned a unique identifier and a persistent log guarantees that completed requests will not be re-executed[5]. The authors report a write overhead of their implementation in RAMCloud [OAE+11] of only 4% compared to the base system without RIFL. The exactly-once semantics also simplify the implementation of transactions. Their approach builds on Sinfonia [AMS+07], an in-memory service

[5] The idea of building distributed transactions on a shared log is also found in Calvin [TDW+12] (cf. p. 241).

System	Concurrency Control	Isolation	Granularity	Commit Protocol
Megastore [BBC+11]	OCC	SR	Entity Group	Local
G-Store [DAEA10]	OCC	SR	Entity Group	Local
ElasTras [DAEA13]	OCC	SR	Entity Group	Local
Cloud SQL Server [BCD+11]	PCC	SR	Entity Group	Local
Spanner [CDE+12]	PCC	SR/SI	Multi-Shard	2PC
F1 [SVS+13]	PCC or OCC	SR/SI	Multi-Shard	2PC
Percolator [PD10]	OCC	SI	Multi-Shard	2PC
MDCC [KPF+13]	OCC	RC	Multi-Shard	2PC-like
TAPIR [ZSS+15]	TO	SR	Multi-Shard	2PC-like
CloudTPS [WPC12]	TO	SR	Multi-Shard	2PC
Cherry Garcia [DFR15a]	OCC	SI	Multi-Shard	Client-coord.
Omid [GJK+14]	MVCC	SI	Multi-Shard	Local
FaRMville [DNN+15]	OCC	SR	Multi-Shard	Local
RAMP [BFG+14]	Custom	Read-Atomic	Multi-Shard	Client-coord.
Walter [SPAL11]	PCC	Parallel SI	Multi-Shard	2PC
H-Store/VoltDB [KKN+08]	Deterministic CC	SR	Multi-Shard	Local
Calvin [TDW+12]	Deterministic CC	SR	Multi-Shard	Local
Orestes with DCAT	OCC	SR	Multi-Shard	Custom

Table 6.2: Related transactional systems and their concurrency control protocols (*OCC*: optimistic concurrency control, *PCC*: pessimistic concurrency control, *TO*: timestamp ordering, *MVCC*: multi-version concurrency control), achieved isolation level (*SR*: serializability, *SI*: snapshot isolation, *RC*: read committed), transaction granularity, and commit protocol.

infrastructure that provides a mini-transaction primitive for atomic cross-node memory access. A central limitation of RIFL is its assumption that clients are reliable and do not lose their state upon crashes. In the web this assumption does not hold and hence RIFL cannot be applied to the same BaaS context as Orestes.

In summary, geo-replication and caching are two very useful techniques for low latency that can be combined. Geo-replication is more powerful to provide protection against disaster scenarios and can reduce latency for strongly consistent reads. Orestes, on the other hand, has a higher number of potential replicas, since each web cache can answer reads. Furthermore, both reads and writes require at most one wide-area round-trip and cache coherence imposes only little overhead. Another advantage is that transaction processing in Orestes can leverage cached/replicated data without cross-data center transaction protocols.

6.3 Transaction Processing

In this section, we will review related work on transaction processing for cloud data management and NoSQL databases. Table 6.2 summarizes important systems for distributed transaction processing and their central properties. We will give a short discussion of each approach and summarize the differences to Orestes with DCAT at the end of the section.

6.3.1 Entity Group Transactions

The approaches can be distinguished by their scope and the degree to which they exploit data locality. Megastore [BBC+11] made the concept of *entity groups* popular that define a set of records that can be accessed in the same transactional context. Megastore's transaction protocol suffers from low throughput per entity group as discussed in the previous section.

In G-Store [DAEA10], entity groups (termed key groups) are created dynamically by the system as opposed to statically through schema-based definitions as in Megastore. Each group has a dedicated master that runs the transactions in order to avoid cross-node coordination. Ownership of a group can be transferred to a different master using a protocol similar to 2PC. G-Store assumes a stable mapping of records to groups, as otherwise many migrations are required to run transactions. The master uses optimistic concurrency to run transactions locally on a single group.

Microsoft's Cloud SQL Server [BCD+11] is also based on entity groups, which are defined through a partition key. Unlike the primary key, a partition key is not unique and identifies a group of records that can be updated in single transactions. A similar concept is employed in Cassandra, Twitter's Manhattan, Amazon DynamoDB, and Windows Azure Table Services [CWO+11, LM10] to enable local sorting or multi-record atomic updates. By introducing the partition key, the concurrency control protocol of Microsoft SQL Server can remain unchanged and still serve multi-tenant workloads, as long as the data per partition key does not exceed the limits of a single database node.

ElasTras [DAEA13] is a DBaaS architecture that builds on entity groups and optimistic concurrency per group managed by an *owning transaction manager*. The central assumption is that either each tenant is so small that data fits into a single partition or that larger databases can be split into independent entity groups. ElasTras employs the *mini-transactions* concept by Aguilera et al. [AMS+07] to support transactions across nodes for management operations like schema changes. ElasTras supports elasticity through a live-migration protocol (Albatross [DNAE11]) that iteratively copies entity groups to new nodes in a multi-step process. This is in contrast to Orestes where we see elastic scalability as a property that should be provided by the underlying database system (e.g., MongoDB) using sharding instead of being added on top of a single-server database system design. ElasTras' largest practical downside is that it assumes completely static entity groups, which is prohibitive for real-world application [CDE+12].

In Orestes, transactions are not constrained to entity groups. While entity groups are a lightweight solution for enabling transactions using well-known single-node concurrency control schemes, scalability is limited and the scope of transactions is limited, too. Instead, we employ multi-tenancy at a higher level, allowing each tenant to scale horizontally across database nodes. If tenants only work on a small data set, transactions will still be constrained to a single database node and hence be less prone to latency stragglers.

6.3.2 Multi-Shard Transactions

As reviewed in the previous section, Spanner [CDE+12], MDCC [KPF+13], CockroachDB [Coc17], and F1 [SVS+13] implement transactions on top of eager geo-replication by trading correctness and fault tolerance against increased latency, whereas Walter [SPAL11] relaxes isolation to increase the efficiency of geo-replication.

FaRMville is a multi-shard transaction approach that has similarities to our approach and was independently proposed by Dragojevic et al. [DNN+15]. The design itself is based on DRAM memory and RDMA (Remote Direct Memory Access) for very low latency. RAM is made persistent through per-rack batteries for uninterrupted power supply. The transaction protocol uses optimistic transactions over distributed shard servers. To this end, the write set is locked by a coordinator executing the commit procedure. The versions of the read set are then validated for freshness and changes are persisted to a transaction log and each individual shard. Using the high-performance hardware setup, FaRMville achieves 4.5 million TPC-C *new order* transactions per second[6].

FaRMville follows a very different design goal of optimizing intra-data center latency for transactions executed from application servers. In contrast, Orestes is designed for remote web clients executing the transactions to support the Backend-as-as-Service model. The motivating idea is similar, though: while FaRMville minimizes abort rates by low latency storage hardware within a data center, we minimize it through caching. Nonetheless, FaRMville could be combined with our Cache Sketch approach, as writes in our scheme are buffered in clients and read sets – including potentially cached reads – are validated at commit time.

TAPIR (Transactional Application Protocol for Inconsistent Replication) [ZSS+15] is based on the observation that replication and transaction protocols typically do the same work twice when enforcing a strict temporal order. The authors propose a consensus-based replication protocol that does not enforce ordering unless explicitly necessary. TAPIR only uses a single consistent operation: the prepare message of the 2PC protocol. All other operations are potentially inconsistent. TAPIR achieves strict serializability using optimistic multi-version timestamp ordering based on loosely synchronized clocks, where the validation happens on read- and write sets at commit time. The authors show that commit latency can be reduced by 50% compared to consistent replication protocols. TAPIR assigns transaction timestamps in clients, but assumes a low clock drift for low abort rates. This is impossible for the web-based use cases of Orestes, where browsers and mobile devices can exhibit arbitrary clock drift [Aki15, Aki16]. Unlike Orestes, TAPIR also suffers from high client-server latencies and exhibits more aborts caused by timestamp ordering.

[6] The achieved transaction throughput is above the highest-ranking TPC-C result at that time, but below the performance of the coordination-free approach by Bailis et al. [BFF+14]

6.3.3 Client-Coordinated Transactions

Percolator [PD10], Omid [GJK+14], and the Cherry Garcia library [DFR15a] are approaches for extending NoSQL databases with ACID transactions using client coordination. While Omid and Percolator only address BigTable-style systems, Cherry Garcia is similar to Orestes in regard to its support for heterogeneous data stores.

Google published the design of its real-time web crawler and search index Percolator [PD10]. Percolator is implemented as an external protocol on top of BigTable. It uses several metadata columns to implement a locking protocol with snapshot isolation guarantees. A client-coordinated 2PC enables multi-key transactions using a timestamp service for transaction ordering. Percolator's protocol is designed for high write throughput instead of low latency reads in order to accommodate massive incremental updates to the search index: latency is reported to be in the order of minutes. The client coordination, multi-round-trip commits and writes, and the lack of a deadlock detection protocol make it unsuitable for access across high-latency WANs.

Omid [GJK+14] also provides snapshot isolation for transactions with a lock-free middleware for multi-version concurrency control on top of a slightly modified HBase. It relies on a central *Transaction Status Oracle* (SO) (similar to the earlier ReTSO work [JRY11]) for assigning begin and commit timestamps to transactions and to perform a snapshot isolation validation at commit time. Omid is designed for application servers, where status information of the SO can be replicated into the servers to avoid most of the round-trips. However, for a highly distributed scenario as addressed by Orestes, relying on a single centralized SO limits scalability and incurs expensive wide-area round-trips for distant application servers.

In his PhD thesis, Dey proposes the Cherry Garcia library [Dey15] for transactions across heterogeneous cloud data stores. The library requires the data store to support strong consistency, multi-versioning, and compare-and-swap updates (e.g., as in Windows Azure Storage [CWO+11]). Similar to Percolator [PD10] and ReTSO [JRY11], the transaction protocol identifies read sets based on transaction begin timestamps and write sets based on transaction commit timestamps, with the metadata maintained in the respective data stores [DFR15a]. For the generation of sequentially ordered transaction timestamps, Cherry Garcia either requires a TrueTime-like API [CDE+12] with error bounds or a centralized timestamp oracle [GJK+14]. In the two-phase transaction commit of Cherry Garcia, the client checks for any write-write and read-write conflicts and makes uncommitted data visible to other transactions. Cherry Garcia is not well-suited for low-latency, as a read potentially requires multiple round-trips to determine the latest valid version suitable for a read, thus increasing the probability of transaction aborts during validation.

Like Percolator, Omid, and Cherry Garcia, Orestes does not modify the underlying database system. All three approaches assume, however, that the client coordinating the transaction is a server in a three-tier application. Unlike Orestes, they are not suited for web and mobile clients participating in transactions, since the latency overhead would be pro-

hibitive for starting transactions, reading and writing, as well as coordinating the commit. DCAT addresses this problem by caching reads, buffering writes, and only contacting the server for commits. Also, DCAT does not burden the primary database system with maintenance of transactional metadata, but instead employs much faster transaction validation and commits using a coordination service.

RAMP (Read Atomic Multi-Partition) by Bailis et al. [BFG+14] also realizes client-coordinated transactions. RAMP only offers a weak isolation level (*read atomic*) in order to be always available, even under network partitions. As discussed in Section 4.8.4, RAMP transactions can be combined with DCAT in order to merge their respective strengths: Orestes provides improved performance and opt-in ACID semantics, whereas RAMP offers a lightweight mechanism to prevent the *fractured read* anomaly. The client is responsible for the RAMP validation and to resolve missing dependencies, so that it is fully compatible with our web caching.

6.3.4 Middleware-Coordinated Transactions

An alternative to embedding transaction processing in the database system or the involved clients is to provide a transactional middleware that accepts transactions from applications and executes them over non-transactional database systems.

CloudTPS [WPC12] is a transaction middleware for web applications. It supports cross-shard transactions using a two-level architecture. In order to avoid a bottleneck through a single coordinator, CloudTPS employs Local Transaction Managers (LTMs) that manage mutually disjoint partitions of the underlying database. Isolation is implemented through timestamp ordering [WV02]. Each LTM executes a sub-transaction of the global transaction and ensures that local commits are properly ordered. A 2PC executed by a designated LTM over all other participating LTMs ensures atomicity of the global commit. Transactions are executed non-interactively in the middleware and have to be predefined at each LTM as a Java function. All keys accessed in a transaction have to be declared at transaction begin, so that the responsible LTMs are known in advance.

As timestamp ordering is susceptible to conflicts, transactions in CloudTPS have to be short-lived and only access a limited set of keys (excluding range and predicate queries). Instead of persisting each write to the underlying storage system, LTMs hold the data independently, distributed through consistent hashing and replicated across multiple LTMs. Periodically, data is persisted to the storage system. Orestes is less prone to aborts, as the BOCC+ concurrency control scheme generates a larger set of permissible schedules than timestamp ordering. Also, DCAT transactions are interactive and ad-hoc, allowing any number of records to be part of the transaction without prior declaration.

Xi et al. [XSL+15] proposed a scheme to effectively combine pessimistic and optimistic concurrency control algorithms. Their system Callas groups transactions by performance characteristics and applies the most appropriate concurrency control mechanism to each.

This is enabled by a two-tiered protocol that applies locking across groups and arbitrary schemes within a group of similar characteristics.

Deuteronomy [LLS+15] follows the idea of separating data storage (data component, DC) and transaction management (transaction component, TC). Similar to Orestes, it also relies on heterogeneous database systems. The authors demonstrate that building on a high-performance key-value store, a throughput of over 6M operations per second can be achieved on scale-up hardware with an appropriate TC. Scalability, however, is limited to the threads of the underlying NUMA (Non-Uniform Memory Access) machines. Therefore, Deuteronomy is not suited for scale-out architectures as addressed by DCAT.

Hekaton [DFI+13], the storage engine of Microsoft SQL Server [Gra97], is another example for the wide-spread use of optimistic transactions in the industry. The authors introduce a new multi-version, optimistic concurrency control scheme for serializability that is optimized for OLTP workloads in main memory. Besides the validation of the read set as in DCAT, Hekaton also validates commit dependencies introduced by concurrent operations during the validation phase. While this optimization increases concurrency and hence throughput, it also introduces cascading aborts. DCAT only exposes committed data and cannot use multi-versioning, as caches cannot perform version checks required for correct visibility.

6.3.5 Deterministic Transactions

H-Store [KKN+08] and its commercial successor VoltDB [SW13] are horizontally scalable main-memory RDBMSs. Sometimes, this new class of scale-out relational databases is referred to as *NewSQL* [GHTC13]. Other examples of the NewSQL movement are Clustrix [Clu17], a MySQL-compatible, scalable RDBMS and NuoDB [Nuo17], an RDBMS built on top of a distributed key-value store.

VoltDB is based on eager master-slave replication and shards data via application-defined columns (similar to MongoDB). Transactions are defined at deployment time as stored procedures written in Java or SQL. Each shard has a Single Partition Initiator (SPI) that works off a transaction queue for that partition in serial order. As data is held in memory, this lack of concurrency is considered an optimization to avoid locking overhead [HAMS08]. Single-shard transactions are directly forwarded to SPIs and do not require additional concurrency control as the execution is serial. Read-only transactions can directly read from any replica without concurrency control (called *one-shot*). Multi-shard transactions are sequenced through a Multi Partition Initiator (MPI) that creates a consensus among SPIs for an interleaved transaction ordering. During execution, cross-shard communication is required to distribute intermediate results. Written data is atomically committed through 2PC. VoltDB scales well for workloads with many single-shard transactions. For multi-shard transactions serialized through the MPI, however, the consensus overhead causes throughput to decrease with increasing cluster size.

Calvin [TDW+12] is a transaction and replication service for enhancing available database systems with ACID transactions. Transactions in Calvin have to be run fully server-side (written in C++ or Python) and must not introduce non-determinism, similar to H-Store and VoltDB [KKN+08, SW13]. This permits Calvin to schedule the order of transactions before their execution. Client-submitted transactions are appended to a shared replicated log that is similar to the Tango approach [BZM+13]. To achieve acceptable performance despite this centralized component, requests are batched, persisted to a storage back-end (e.g., Cassandra), and the batch identifiers are replicated via Paxos. The scheduler relies on the log order to create a deadlock-free, deterministic ordering of transactions using two-phase locking. As each transaction's read and write sets have to be declared in advance, allocation of locks can be performed before the transaction begin (*preclaiming* [WV02]). Transactions execute locally on each shard by exchanging the read sets with other shards and only writing local records. While Calvin achieves high throughput in TPC-C benchmarks, its model is strictly limited to deterministic, non-interactive transactions on pre-defined read and write sets, which eliminates most forms of queries. DCAT, in contrast, allows clients to dynamically create their read sets at runtime based on other reads and queries. Furthermore, there is an inherent trade-off between commit latency and throughput introduced by the batching interval of the shared log.

Orestes supports interactive, potentially non-deterministic transactions, which are more flexible than stored procedures. However, executing server-side transactions could be an interesting fall-back for high-contention transactions, that repeatedly abort. In that case, Orestes could revert to server-side transaction execution as proposed in H-Store and Calvin.

6.3.6 Comparison with DCAT

Our DCAT approach for serializable, multi-shard transactions has several similarities with related work (cf. [Wit16]):

- **Low-latency reads** are the central goal, as in systems and approaches with eager geo-replication (Megastore [BBC+11], Spanner [CDE+13, CDE+12], F1 [SVS+13], MDCC [KPF+13], Mencius [MJM08], Replicated Commit [MNP+13], and CockroachDB [Coc17]).
- DCAT relies on **optimistic concurrency control**, similar to Megastore [BBC+11], F1 [SVS+13] , G-Store [DAEA10] , Percolator [PD10], MDCC [KPF+13], Cherry Garcia [DFR15a], H-Store/VoltDB [KKN+08] , TAPIR [ZSS+15], and FaRMville [DNN+15].
- To minimize round-trips induced by transactional updates, the write set is **buffered** in the client, similar to FaRMville [DNN+15], F1 [SVS+13], and Cherry Garcia [DFR15a].
- Orestes with DCAT enables **polyglot persistence** by allowing transactions to be processed on top of non-transactional database systems, similar to Cherry Garcia

[DFR15a], Omid [GJK+14], CloudTPS [WPC12], Calvin [TDW+12], and Deuteronomy [LLS+15].

- The DCAT commit procedure requires **two phases** (validation and writing), similar to Spanner [CDE+12], F1 [SVS+13], Percolator [PD10], MDCC [KPF+13], CloudTPS [WPC12], TAPIR [ZSS+15], Cherry Garcia [DFR15a], FaRMville [DNN+15], and Walter [SPAL11].

In summary, our approach is a novel research perspective on low-latency transaction processing that builds on caching instead of geo-replication. To our knowledge, it is the first approach to leverage web caching for transactions. By accelerating the reads during optimistic transactions and combining them with a lightweight, single-site commit procedure, DCAT avoids the problem of slow wide-area coordination associated with geo-replication. DCAT therefore broadens the design space of transactions for cloud data management and offers a solution to the abort rate problem of optimistic transactions.

6.4 Database-as-a-Service and Polyglot Persistence

The DBaaS model promises to shift the problem of configuration, scaling, provisioning, monitoring, backup, privacy, and access control to a service provider [CJP+11]. Hacigumus et al. [HIM02] coined the term DBaaS and argued that it provided a new paradigm for organizations to alleviate the need for purchasing expensive hardware and software to build a scalable deployment. Lehner and Sattler [LS13] and Zhao et al. [ZSLB14] provide a comprehensive overview of current research and challenges introduced by the DBaaS paradigm.

The DBaaS model emerged as a useful service category offered by PaaS and IaaS providers and is therefore mainly rooted in industry. Table 6.3 summarizes selected commercial systems and groups them by important properties such as data model, sharding strategy, and query capabilities. All systems except Cloudant are based on proprietary REST APIs and details about their internal architectures are not published. Another observation is that fine-grained SLAs are not provided, due to the difficulty of satisfying tenant-specific requirements on a multi-tenant infrastructure. We address this problem through the Polyglot Persistence Mediator.

6.4.1 Multi-Tenancy and Virtualization

Most related work focuses on specific aspects of DBaaS models. Multi-tenancy and virtualization are closely related, as resource sharing between tenants requires some level of virtualization of underlying resources (the schema, database process, operating system, computing hardware, and storage systems). The trade-off between performance and isolation for multi-tenant systems has been studied extensively [ASJK11, AGJ+08, AJKS09, KL11, SKM08, WB09, JA07]. Orestes employs containerization (based on Docker [Mer14]) for lightweight shared-machine multi-tenancy. Virtualization can also be a

System	Data Model	CAP	Queries/ Indexing	Replication	Sharding	Trans-actions	SLAs
Cloudant [BGH+15]	Document Store	AP	Incremental MR Views	Lazy, Local & Geo	Hashing	No	No
DynamoDB [Dyn17]	Wide-Column	CP	Local & Global Index	Eager, Local	Hashing	No	No
Azure Tables [CWO+11]	Wide-Column	CP	By key, Scans	Eager, Local	Hashing	No	99.9% Uptime
Google Cloud DataStore [Dat17, BBC+11]	Wide-Column	CP	Local & Global Index	Eager, Geo	Entity Groups	Per Group	No
S3, Azure Blobs, GCS [Ama17a]	Blob-Store	AP	No	Lazy, Local & Geo	Hashing	No	99.9% Uptime (S3)

Table 6.3: Selected industry DBaaS systems and their main properties: data model, category according to the CAP theorem, support for queries and indexing, replication model, sharding strategy, transaction support, and service level agreements.

means to enable elasticity, by live-migrating tenants based on workloads and requirements [EDAE11, BCM+12, DNAE11, DAEA13, DEAA09]. Orestes does not perform live migration and instead relies on the database layer to shard data and on statelessness to replicate the application tier.

6.4.2 Database Privacy and Encryption

Since a DBaaS is hosted by third party, security and privacy are particularly important. Several researchers have proposed solutions to prevent attackers and providers from analyzing data stored in a DBaaS system. A survey of the field is provided by Köhler et al. [KJH15]. The ideal solution for DBaaS privacy is fully homomorphic encryption, which enables arbitrary computations on encrypted data stored in the database. Though Gentry [Gen09] proposed a scheme in 2009, the performance overhead is still prohibitive for use in real-world application.

The naive approach to ensure data confidentiality is to perform queries only in the client, so that data can be fully encrypted. This approach is used in ZeroDB [EW16]. The obvious limitation is that the client and network quickly become the bottleneck: in ZeroDB, the query logic is executed in the client and each descent in the B-tree requires one round-trip, leading to very high latency. MIT's CryptDB project [PRZB11, Pop14] is based on a layered encryption scheme, where different encryption levels enable different query operators, e.g., homomorphic encryption for sum-based aggregation and deterministic encryption for equality predicates. CryptDB assumes a database proxy out of the threat scope that is responsible for rewriting queries with the appropriate keys before forwarding

them to the database holding the encrypted data. The MySQL-based prototype exhibited a processing overhead of 26% compared to native access, but latency was increased by an order of magnitude. The problem of CryptDB is that the vulnerability is only moved into the proxy that is co-located with application servers and therefore typically cloud-hosted, too. Nonetheless, first commercial DBMSs have implemented explicitly declared encryption levels for queries on encrypted data, e.g., Microsoft SQL Server supporting random and deterministic encryption [Alw17].

The problem of vulnerable proxies in CryptDB was addressed in a follow-up system called Mylar [KFPC16,PZ13,PSV+14]. Mylar implements multi-key keyword search on encrypted data with a middleware operating only on encrypted data without access to keys. The browser is responsible for encrypting and decrypting data based on user keys. Data is stored and encrypted using the key of the user owning the record. The core idea of the encrypted keyword search is that clients generate an encrypted token for search that works on any record irrespective of the key it was encrypted with. When a user grants access to another user, a delta value is constructed in a way that allows the server to transform tokens without leaking data. The downside of Mylar is that it only enables keyword search. Performance is further limited, as the server has to scan every record for a token comparison, only per-record duplicates of keywords can be indexed. Nonetheless, Mylar is an important step towards secure sharing of information between application users and it is also notable for providing security against attacks of both middleware and database.

Due to the efficiency and applicability constraints of Mylar, CryptDB, ZeroDB, and fully homomorphic encryption, Orestes does not process queries on encrypted data. However, the Polyglot Persistence Mediator could be extended to interpret privacy SLAs and map the respective operations and queries to an encrypted data store. Furthermore, the performance of the above system could significantly be improved by applying the Cache Sketch to speed up reads and queries.

Relational Cloud is a visionary architecture for a secure, scalable, and multi-tenant DBaaS by Curino et al. [CJP+11]. It proposes to use private database virtualization for multi-tenancy and CryptDB for privacy. Access to the database is handled through a JDBC driver which directs requests to load-balancing frontend servers that partition data across back-end servers to store the actual data in CryptDB. The partitioning engine Schism [CJZM10] is based on workload graphs: whenever two tuples are accessed within a transaction, the weight of their edge is increased. By finding a partitioning of tuples with a minimal cut, cross-node transactions are minimized. The partitioning rules are compacted and generalized by training a decision tree that is used in frontend servers for routing. The consolidation engine Kairos [CJMB11] monitors workloads and outputs a mapping from virtual machines to physical nodes in order to optimize combined resource requirements of multiple tenants. In contrast to Orestes, Relational Cloud is fundamentally based on live migration to ensure both resource efficiency and an optimal partitioning, whereas our idea is to scale transactions and data storage independently. Furthermore, the Polyglot

Persistence Mediator does not infer a partitioning solely based on transaction co-access, but relies on the application to define more fine-grained SLAs to support this process.

6.4.3 Service Level Agreements (SLAs)

Various approaches have been proposed for SLAs in cloud services and DBaaS systems [CAAS07, ZSLB14, ABC14, Bas12, XCZ+11, TPK+13, LBMAL14, PSZ+07, Sak14]. Traditionally, this topic has been tackled in the context of workload management for mainframe systems, to optimize simple performance metrics like query response time [CDF+07, LS13]. Many approaches rely on the underlying virtualization environment to enforce SLAs by means of live migration, e.g., Zephyr [EDAE11], Albatross [DNAE11], Dolly [CSSS11], and Slacker [BCM+12]. Baset [Bas12] reviews SLAs of commercial cloud providers like AWS, Azure, and Rackspace and concludes that performance-based SLAs are not guaranteed by any provider. Furthermore, the burden of providing evidence of SLA violations rests on the customer.

Xiong et al. have proposed ActiveSLA [XCZ+11] as an admission control framework for DBaaS systems. By predicting the probability of a query completing before its deadline, a cost-based decision on allowing or rejecting the query can be made using the SLA. Chi et al. [CMH11] have proposed a similar approach that uses an SLA-based scheduler *iCBS* to minimize expected total costs. Sakr et al. [SL12] presented the CloudDB AutoAdmin framework that monitors SLAs of cloud-hosted databases and triggers application-defined rules upon violations to help developers build on SLAs. Armbrust et al. [ACK+11] proposed the SQL extension PIQL (Performance Insightful Query Language) that predicts SLA compliance using a query planner which is aware of developer-provided hints. Instead of choosing the fastest plan, the optimizer only outputs plans where the number of operations is known in advance. Lang et al. [LSPK12] formulate the SLA problem for DBaaS systems as an optimization task of mapping client workloads to available hardware resources. In particular, they provide a way for DBaaS providers to choose the class of hardware that best suits the performance SLOs of their tenants.

To our best knowledge, Orestes is the first DBaaS approach to combine service level agreements with schema design for database-driven applications. Instead of focusing on a specific performance SLA as common in most related work, we propose to express each functional and non-functional data management requirement based on our NoSQL classification scheme.

6.4.4 Resource Management and Scalability

Resource Allocation and Workload Characterization

Problems closely related to SLAs are resource and storage allocation [MRSJ15, SLG+09], pricing models [LS13, p. 145], and workload characterization [GKA09, GMU+12]. As Orestes is built on the Infrastructure-as-a-Service abstraction level, we consider low-level

hardware and virtual machine allocation schemes to be out of scope for this work. Further, our approach to workload characterization is indirect: we observe the effects of workloads through the overall SLA compliance and perform the appropriate scaling or migration logic accordingly.

Auto-Scaling and Elasticity

For providing elasticity, DBaaS systems have to automatically scale in and out to accommodate the current and future mix of tenant workloads. The ability to forecast workloads enables the most efficient forms of auto-scaling, as the service does not have to react to overload situations and SLA violations, but can instead proactively adjust its capacities. Kim et al. [KWQH16] and Lorido-Botran et al. [LBMAL14] provide an overview of commonly employed workload predictors and auto-scaling techniques from the literature. Related work on auto-scaling can be grouped into approaches for threshold-based rules (e.g., [HMC+12, HGGG12, KF11, MBS11, GSLI11, CS13]), reinforcement learning (e.g., [DRM+10, BHD13, TJDB06, BRX13, XRB12]), queuing theory (e.g., [USC+08, VPR07, ZCS07]), time series analysis and prediction (e.g., [CDM11, GGW10, SSGW11, FLWC12, IKLL12, PN09]), control theory (e.g., [PHS+09, XZF+07, BGS+09, ATE12, PH09]), and database live-migration (e.g., [EDAE11, DNAE11, CSSS11, BCM+12, DAEA13]). While auto-scaling does not replace capacity planning, it significantly increases flexibility as the cloud infrastructure can be adapted at runtime.

Orestes is not fixed to a single auto-scaling technique. In practice, rule-based systems based on parameters such as CPU load, I/O operations, and memory usage are often sufficient and therefore the default in Orestes [LBMAL14]. Complex, proactive models are usually stronger for sudden surges in demand, but most of the algorithms proposed in the literature strongly depend on a certain workload type. A special architectural trait of Orestes is its integration into CDNs. This potentially allows scaling based on end-user metrics such as HTTP latency. Orestes already uses the same channel for high-availability: if the CDN reports a failed request, the management server is contacted by the CDN, a health check is performed, and a failover to a new Orestes container is initiated. Marcus and Papaemmanouil [MP17] argue that scalability and query planning decisions for cloud data management should not depend on humans or simple rules but instead harness machine learning techniques, in particular reinforcement learning. We follow a similar idea in the context of learning query result TTLs by applying deep reinforcement learning [SGDY16].

6.4.5 Benchmarking

Different benchmarks have been proposed to evaluate latency, throughput, consistency, and other non-functional properties of distributed and cloud databases [DFNR14, CST+10, CST+10, PPR+11, BZS13, BKD+14, BT11, BK13, BT14, Ber15, Ber14] .

Performance Benchmarking

The *Yahoo Cloud Serving Benchmark* (YCSB) [CST+10] was published in 2010 and is the de-facto standard for benchmarking NoSQL systems. YCSB is designed to measure throughput and latency for CRUD and scan operations performed against different data stores [FWGR14, WFGR15]. The main shortcoming is the missing distribution of workload generation to prevent clients from becoming the actual bottleneck. The second problem is that YCSB's thread-per-request model incurs high overhead and increases latency [FWR17]. We addressed this problem in YMCA by scaling the client tier and using an asynchronous workload generation, while keeping the well-known semantics and workloads of YCSB.

While YCSB's generic workloads make it easily applicable to any data store, its lack of application-specific workloads render the results hard to interpret. Particularly in contrast to the widely used TPC benchmarks [PF00] for RDBMSs, YCSB neither covers queries nor transactions. *BG* [BG13] was proposed as an alternative to YCSB that models interactions in a social network. BG not only collects performance indicators, but also measures the conformance to application-specific SLAs and consistency. The *Under Pressure Benchmark* (UPB) [FMdA+13] is based on YCSB and quantifies the availability of replicated data stores by comparing the performance during normal operation with the performance during node failures. Since neither YCSB, BG, nor UPB allow simulation of database and network architectures, we developed YMCA. It therefore allows to test assumptions about performance metrics such as throughput, latency, and cache hit rates without having to deploy a distributed system, first.

Consistency Benchmarking

As consistency is one of the central properties that many cloud data management systems trade against other non-functional properties for performance reasons, various benchmarks have been proposed to quantify eventual consistency and staleness. Wada et al. [WFZ+11] proposed a methodology to measure the staleness of reads for cloud databases based on a single reader and writer. As reader and writer rely on simple timestamps for consistency checks, the strategy is highly dependent on clock synchronization and unsuitable for geo-replicated systems. Bermbach et al. [BT11, BT14] extended the approach by supporting multiple, distributed readers frequently polling the data store. This uncovered a pattern for the staleness windows of Amazon S3. However, the scheme still assumes clock synchronization and therefore might lead to questionable results [BZS13].

Golab and Rahman et al. [GLS11, RGA+12] argue that a consistency benchmark should not introduce a workload that stresses the system artificially, but should rather extend existing workloads to also capture staleness information. The authors propose an extension of YCSB that tracks timestamps and uses them to compute an empirical Δ for the Δ-atomicity of the underlying data store by finding the maximum time between two operations that yielded a stale result. YCSB++ [PPR+11] circumvents the problem of clock synchronization by relying on a centralized Zookeeper instance for coordination of readers and writers

to measure consistency. As a consequence, YCSB++ can only provide a lower bound for the inconsistency window. Furthermore, we discovered that the implementation of the coordination is flawed, leading to false experimental results [WFGR15].

Bailis et al. proposed the *Probabilistically Bounded Staleness* (PBS) [BVF+12, BVF+14] prediction model to estimate the staleness of Dynamo-style systems based on messaging latencies between nodes. PBS relies on a Monte Carlo simulation sampling from latency distributions to calculate the probability of a stale read for a given time after a write ($(\Delta,t)-$atomicity). As PBS only works for Dynamo-style systems and does not include caching, we adopted this approach for YCSB workloads and arbitrary topologies of database nodes and caches in YMCA (cf. Section 4.3.1). This allows us to study staleness introduced not only by replication, but also by invalidation-based and expiration-based caching. Furthermore, the simulation frees the analysis from the trade-off between errors introduced by clock drift and imprecision introduced by coordination delay, as exact simulation times can be used.

Any database system can potentially be provided in the form of a DBaaS. However, low-latency access, elastic scalability, polyglot persistence, cross-database transactions, and efficient multi-tenancy play important roles for scalable web applications and have only partly been addressed by related work so far. In this thesis, we seek to provide a comprehensive methodology for DBaaS environments that combines these properties.

6.4.6 Database Interfaces and Polyglot Persistence

REST APIs

Most cloud services, including DBaaS and BaaS systems, use REST APIs to ensure interoperability and accessibility from heterogeneous environments. Originally proposed as an architectural style by Fielding [Fie00], REST now commonly refers to HTTP-based interfaces. HTTP [FGM+99] emerged as the standard for distributing information on the Internet. Originally, it was employed for static data, but now serves sophisticated use cases from web and mobile application to Internet of Things (IoT) applications. The growing adoption of HTTP/2 [IET15] solving the connection multiplexing problem of HTTP/1.1 facilitates this movement. For web applications, REST and HTTP have largely replaced RPC-based approaches (e.g., XML RPC or Java RMI [Dow98]), wire protocols (e.g., PostgreSQL protocol [Pos17]), and web services (specifically, SOAP and WS-* standards family [ACKM04]).

Google's GData [Gda17] and Microsoft's OData (Open Data Protocol) [Oda17] are two approaches for standardized REST/HTTP CRUD APIs that are used by some of their respective cloud services. Many commercial DBaaS systems offer custom REST APIs tailored for one particular database (e.g., DynamoDB, Cloudant). A first theoretic attempt for a unified DBaaS REST API has been made by Haselman et al. [HTV10] for RDBMSs. Dey [Dey15] proposed REST+T as a REST API for transactions. In REST+T, each object

is modeled as a state machine modified through HTTP methods. It cannot be applied to DCAT, as the implicit state machine forbids object caching.

Orestes differentiates itself from other DBaaS/BaaS REST APIs in that it actively incorporates infrastructure support (caching and load balancing) as well as complex data management concepts (schema management and transactions). Unlike the Orestes REST API, related DBaaS APIs do not make use of caching and do not support the BaaS model where arbitrary clients are authenticated and are then granted access to the database.

Backend-as-a-Service

According to Roberts [Rob16], *serverless* architectures are applications that depend on cloud services for server-side logic and persistence. The two major categories of serverless services are Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS). Both approaches are rooted in commercial cloud platforms rather than research efforts.

FaaS refers to stateless, event-triggered business logic executed on a 3rd-party platform [Rob16]. Industry offerings are AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions. While FaaS offers a very simple and scalable programming model, its applicability is limited by the lack of persistent state. The major difference between FaaS and Platform-as-a-Service lies in the ability of FaaS to seamlessly scale on a per-request basis, as no application server infrastructure (e.g., Rails, Django, Java EE) is required. The term "BaaS" refers to services that enable the development of rich client applications through database access, authentication and authorization mechanisms, as well as SDKs for websites and mobile apps. BaaS therefore is a natural extension of DBaaS towards scenarios of direct client access Apps without intermediate application servers.

Orestes merges capabilities of FaaS with BaaS by a tier of stateless REST servers that provide FaaS capabilities through handlers and API endpoints coupled to the Backend-as-a-Service interfaces for persistence, user management, etc (see Section 3.5.5).

Many commercial BaaS platforms are available (e.g., Firebase, Kinvey, and Azure Mobile Services). Most of these platforms are based on proprietary software and unpublished architectures which hinders a comparison. However, different open-source BaaS platforms have been developed. They typically consist of an API server (e.g., Node.js or Java) for BaaS functionality and user-submitted code and a NoSQL database system for persistence (e.g., MongoDB, Cassandra or CouchDB).

Meteor [HS16] is a development framework and server for running real-time web applications. It is based on MongoDB and directly exposes the MongoDB query language to JavaScript clients for both ad-hoc and real-time queries. Node.js-based application servers run custom code and standard APIs, e.g., for user login. Scalability is limited, as each application server subscribes to the MongoDB replication log (*oplog tailing*)[7], in order to match subscribed queries to updates. Each server therefore has to maintain the

[7] Historically, there is another approach called poll-and-diff that relies on periodic query execution for discovery of result changes. However, poll-and-diff does not scale with the number of real-time query subscribers.

aggregate throughput of a potentially sharded MongoDB cluster. As the replication log furthermore only contains partial information on updates, the application servers need to perform additional database queries to check for a match. The Meteor architecture is therefore unsuitable for query caching and real-time queries as employed in Orestes (cf. [WGW+18]).

Deployd [Dep17], Hoodie [Hoo17], and Parse Server [Par17] are based on Node.js, too. Deployd [Dep17] is a simple API server for common app functionalites and a simple, MongoDB-based CRUD persistence API. It is focused on simplicity and is neither horizontally scalable nor multi-tenant. Hoodie [Hoo17] is a BaaS that combines CouchDB and a client-side CouchDB clone called PouchDB for offline-capable apps with synchronization. Through CouchDB change feeds, clients can subscribe to simple CRUD events with limited querying capabilities. Hoodie is focused on offline-first applications and offers no support for data and request scalability.

Parse Server [Par17] is an open-source implementation of the Parse platform that was acquired by Facebook in 2013 and later discontinued [Lac16]. It has extensive mobile SDKs that go beyond wrapping the REST API and also provide widgets and tooling for building the frontend. Parse Server is based on Node.js and MongoDB and supports file storage, a JSON CRUD API, user management, access control, and real-time queries that are functionally similar to those provided by InvaliDB (cf. Section 4.6), but without support for ordering [Wan16]. The real-time query architecture relies on broadcasting every update to every server that holds WebSocket connections to clients through a single Redis instance. This does not allow the system to scale upon increasing update workloads beyond single-server capacity. Parse Server does not expose many data management abstractions such as indexes, partial updates, concurrency control, and schemas, making it unsuitable for performance-critical and advanced applications. In particular, latency of HTTP requests is not reduced through caching. However, in order to prevent the browser from performing two round-trips due to cross-origin pre-flight requests, REST semantics are violated and every interaction is wrapped in an HTTP POST and GET request [Gri13].

BaasBox [Bas17] and Apache Usergrid [Use17] are open-source Java-based BaaS platforms. BaasBox [Bas17] is a simple single-server platform based on the multi-model database OrientDB [Tes13]. Its main capabilities are CRUD-based persistence and a simple social media API for app development. Apache Usergrid [Use17] is a scalable BaaS built on Cassandra and geared towards mobile applications. Through a REST API and SDKs, it supports typical features such as user management, authorization, JSON and file storage, as well as custom business logic expressed in Java. Multi-tenancy is achieved through a shared database model by running private API servers for each tenant, while consolidating rows in a single Cassandra cluster. Query support is limited due to Cassandra's architecture and there are no consistency guarantees nor multi-key transactions. Both BaasBox and Usergrid are designed for mobile applications, so they do not address the latency and performance requirements of websites as the Orestes middleware does.

The major difference between other BaaS platforms and Orestes is the capability to support large-scale, low-latency web applications. While many native mobile applications can tolerate high network latencies to some extent by packaging as much data as possible into the binary release, the user satisfaction in web applications is typically governed by page load time, so that the amount of overall payload has to be minimized. This point is addressed through our caching approach. The need for scalability to support both high request loads, real-time requirements, and large volumes of user-generated content are addressed through our architecture that decouples data storage from stateless application logic. The ability to scale real-time queries with respect to both update throughput and query concurrency is also novel. Furthermore, Orestes is the only BaaS to expose ACID transactions and explicit fine-grained control over consistency levels.

Polyglot Persistence

The term polyglot persistence was introduced by Leberknight [Leb08] and later popularized by Fowler [SF12]. Most web-scale architectures are heavily based on polyglot persistence both within application components as well as across different applications. Twitter uses Redis [San17] for storing tweets, a custom eventually consistent wide-column store named Manhattan [Sch16] for user and analytics data, Memcache [Fit04] for caching, a custom graph store called FlockDB as well as MySQL, HDFS, and an object store [Has17]. Google, Facebook, and Amazon are also recognized for their broad spectrum of employed database systems. While there is no shortage of polyglot persistence architectures in practice, little research has gone into addressing the problem of how to design, implement, and maintain polyglot persistence architectures.

Object-relational (OR) and object-document (OD) mappers are important classes of tools that limit vendor lock-in and minimize impedance mismatch [Mai90, Amb12]. By abstracting from implementation details of database systems, they facilitate polyglot persistence. Popular mappers are Hibernate, DataNucleus, Kundera, EclipseLink, OpenJPA, Entity Framework, Active Record, Spring Data, Core Data, Doctrine, Django, and Morphia [IBNW09, TGPM17, DeM09]. Torres et al. [TGPM17] provide a comprehensive overview of mappers and propose a catalog of criteria to evaluate their capabilities (e.g., metadata extraction, foreign key support and inheritance). Störl et al. [SHKS15] reviewed mappers specifically targeted to NoSQL databases. The authors observed that, while basic CRUD functionality works well across all analyzed mappers, query expressiveness vastly differs. This is a consequence of providing high-level query languages in the mapper, that potentially cannot be mapped to the limited querying capabilities of the underlying database system and therefore has to be emulated client-side. Also, the authors observed that the overhead introduced by some mappers is significant, in particular for updates and deletes. Wolf et al. [WBGsS13] describe the steps required to adapt traditional OR-mappers such as Hibernate to key-value stores. Their effort makes it obvious that there is a significant feature gap between state-of-the-art mapper abstractions and capabilities found in low-level data stores.

Contrasting the above-discussed mappers, Orestes moves the abstraction layer from a client persistence library into the middleware. We are convinced that this approach increases flexibility, as the middleware has more information available (e.g., workloads, SLAs, database and cluster state). To abstract from the heterogeneity of different database systems, our unified REST API takes the place of the OR/OD mapper API and provides a coherent interface for heterogeneous persistence frameworks usable from various programming languages.

Multi-model databases address polyglot persistence of data models and seek to provide them in a single data store. This imposes heterogeneous requirements on a single database system and hence implies tremendous engineering challenges. ArangoDB [Ara17] and OrientDB [Tes13] are two examples of systems that provide main APIs for storing and querying documents, but also support graph traversal and key-value storage. While these systems simplify operations by integrating polyglot capabilities into single systems, there are more sophisticated solutions available for each of the supported polyglot models. Several RDBMSs also incorporate non-relational data models such as XML and JSON [Cro06] as a data type with SQL extensions to modify and query its contents. The major limitation of multi-model approaches is that the data model is only one of many requirements that necessitate polyglot persistence (e.g., scalability and latency). Many requirements are directly tied to replication, sharding, and query processing architectures and therefore are very difficult to consolidate in a single system. For this reason, our Polyglot Persistence Mediator leverages the broad landscape of data management systems and reduces the challenge to providing a mapping between application requirements and individual systems.

In summary, Orestes builds on related work on DBaaS systems and scalable cloud services. It consolidates well-known techniques with novel methods for polyglot persistence, SLAs, latency requirements, and scalable transactions in an end-to-end cloud data management approach.

7 Conclusions

This thesis addresses low latency for cloud data management by proposing a novel caching approach for dynamic data. In Chapter 1, we identified four central challenges contributing to the latency problem: latency of dynamic data, direct client access, transaction abort rates, and polyglot persistence. Chapter 2 discussed these challenges in the context of backend, network, and frontend performance. In the following, we analyzed and categorized data management requirements in Chapter 3, deriving the cloud platform design Orestes which combines rigorous consistency guarantees and rich client interfaces with low latency. As a centerpiece of our framework, we presented the Cache Sketch approach in Chapter 4 to make cache coherence of dynamic data feasible on the web. In Chapter 5, we extended the idea through a Polyglot Persistence Mediator for mapping requirements to suitable systems. We surveyed related work in Chapter 6 and discussed how different paradigms for scalable, low-latency data management align to our work.

In this chapter, we conclude the dissertation by summarizing our main contributions. We then discuss opportunities for future work on cloud data management with low latency. We close with our thoughts on the role that latency may play for the future of the web.

7.1 Main Contributions

In this dissertation, we introduced Orestes as an approach to reduce latency on the web through a novel cache coherence strategy for dynamic data. We showed that high latency in data management (cf. Challenge C_1) can be reduced to the problem of serving replicated or cached data from nearby locations while maintaining configurable levels of consistency. To translate advantages in low-latency data access to faster applications, we addressed the challenge of providing direct client access to cloud-based DBaaS systems (cf. Challenge C_2). Since transactions are a key abstraction in data management that is particularly latency-sensitive, we introduced a scheme to achieve low abort rates through optimistic transactions over cached data (cf. Challenge C_3). Finally, we showed that NoSQL database systems can be accurately classified through the NoSQL Toolbox that provides decision support for choosing a system. Based on the toolbox, we demonstrated that polyglot persistence can be achieved in an automated, declarative fashion for heterogeneous NoSQL database systems (cf. Challenge C_4).

Throughout our work, we found that the separated worlds of web and database technology can profit immensely from each other. In particular, the web caching model that was long thought of as irreconcilable with modern data management is, in fact, an ideal basis for data-driven, geo-distributed applications. As our goal has been to make our findings easily applicable in practice, we reinforced that the Backend- and Database-as-a-Service models should be provided as a middleware to tap into the huge potential of the current database ecosystem. In this section, we summarize the main contributions of this work and the lessons learned from building Orestes and its commercial implementation Baqend.

7.1.1 Object, File, and Query Caching

To minimize latency between users and cloud services, we proposed to use the web caching infrastructure with its broad distribution of available caches. As the web caching model assumes mostly static data, we derived a dual strategy for maintaining cache coherence for dynamic data. On the one hand, the developed cloud service is responsible for proactively updating invalidation-based caches that can be controlled through a server (e.g., CDNs). On the other hand, we assign the task of updating expiration-based caches (e.g., browser caches) to the client through a probabilistic *Cache Sketch* data structure based on Bloom filters. To make the scheme feasible, we solved *invalidation detection* for objects, files, and queries, i.e., the problem of triggering a cache purge, whenever an update potentially affects the consistency of data stored in caches. The efficiency of the scheme was further improved through *TTL estimation* to deliver cacheable responses with an expiration time that is close to the expected time until the next invalidation. Our approach offers many useful client-centric consistency guarantees out-of-the-box (e.g., monotonic reads and Δ-atomicity) and allows tuning staleness bounds through the Cache Sketch. As strong semantics are often required in critical parts of applications, we derived a solution to combine low latency with optimistic ACID transactions in DCAT (*Distributed Cache-Aware Transactions*).

Our proposed Cache Sketch scheme is a new means for any cloud service to reduce latency with fine-grained control of the exposed level of consistency. It is deployed in production for thousands of applications. To validate our approach, we showed that performance of common web workloads is improved by an order of magnitude.

7.1.2 Backend-as-a-Service

Latency improvements in data management only become effective for users, if end devices are enabled to access cloud-hosted data directly in a two-tier architecture, as backend latency has little effects on end-to-end latency. Therefore, we proposed a unified REST API for cloud data management that combines individual system capabilities in a coherent interface. We argued that many functional and non-functional properties can be provided in a generic fashion as a *Database/Backend-as-a-Service middleware*. With Orestes, we demonstrated that the unified REST API can be offered in an extensible, scalable, and

highly available manner on top of different NoSQL database systems. Besides our central contribution of low latency, the design enhances data stores with various other capabilities, including authentication and access control, schema management, transactions, real-time queries, and a co-located Function-as-a-Service engine. Orestes thus enables aggregate-oriented data stores to be offered as a low-latency Backend-as-a-Service.

We are convinced that in research, data management should be addressed as the interaction of requirements with system implementation techniques. We investigated this idea by constructing the *NoSQL Toolbox* that puts requirements in relation with schemes for sharding, replication, storage, and queries. By covering potential data management requirements in a single unified REST API, database system implementations can focus on specific query, consistency, and transaction requirements. Many non-functional requirements – in particular low latency – are therefore solved in a *database-independent* middleware for all stores instead of individually for different stores.

7.1.3 Polyglot Persistence Mediation

Based on the lessons learned from designing and implementing Orestes, we concluded that combining multiple database systems should be an automated process that is based purely on requirements. Therefore, we introduced the *Polyglot Persistence Mediator* (PPM) to orchestrate data stores based on schemas annotated with SLAs. The PPM can thus route data and queries to systems that are equipped to fulfill the demanded functional and non-functional requirements (e.g., data scalability and conditional updates). The main idea is that application architects only have to define what they expect from their data systems and that the mediator chooses the appropriate systems and enforces SLAs. To select a set of applicable data stores, a ranking algorithm descends through the annotated schema and maps portions of the schema to available systems. Data is then transparently partitioned on field, bucket, or database level and operations are rewritten by the PPM.

We showed that automated polyglot persistence can outperform one-size-fits-all solutions by 50-100% in latency and throughput. As the trend towards polyglot persistence continues, we are convinced that automating and optimizing the distribution of data fragments to heterogeneous database systems, is one of the central research challenges in cloud data management. The PPM is a conceptual framework and prototype to start this new line of scientific work.

7.2 Future Work

This work opens several promising directions for future research on low-latency cloud data management. These range from extensions to the overall caching scheme to new opportunities arising from the approach itself. In the following, we will discuss four relevant areas of future work.

7.2.1 Caching for Arbitrary Websites, APIs, and Database Systems

This thesis is based on the assumption of a DBaaS/BaaS model for cloud data management. However, the Cache Sketch can be applied to any type of data, in particular unstructured resources of websites. Service Workers (cf. Section 2.4.2) offer a novel browser technique for implementing client-side proxies that can modify the networking behavior of end user devices [Ama16]. This allows decoupling the idea of caching dynamic data from database interfaces and extending it to arbitrary requests of existing websites by leveraging Orestes and its Cache Sketch as a proxy/CDN.

Cache Sketches for APIs and Website Resources. Websites and APIs can be enhanced to serve data through Orestes without changing the given application frontend and backend. To this end, a configurable Service Worker has to reroute requests that would otherwise go to a slow backend, so that requests are instead delivered from web caches managed by Orestes. Resource that have not been cached before can be fetched from the original backend, so that Orestes can apply TTL estimates. Invalidations and Cache Sketch maintenance can be performed by comparing the cached version in Orestes to the original version. In the frontend, the Cache Sketch logic has to be applied in the Service Worker by orchestrating its dedicated cache to return resources that are not stale. By defining or learning a working set of critical resources, the website furthermore becomes offline-capable. To facilitate offline operations, the Cache Sketch guarantees prompt and fine-grained synchronization whenever the device resumes connection.

Learning TTLs for Unstructured Data. The new challenge for applying the Cache Sketch to external data sources is the difficulty of unknown semantics: without knowing the structure of data, a TTL estimation and a decision on cacheability is difficult to make (cf. Section 4.1). Therefore, a novel TTL estimation process is required that incorporates the evolution of the cached resources over time and across applications, without any prior assumptions. This task can be formulated as a regression and classification problem requiring an application-independent machine learning approach. Besides learning the optimal TTLs, content optimizations such as minification, transcoding, and concatenation can also be optimized based on real-user performance monitoring.

Consistency Guarantees. As updates to external data sources are unobservable to Orestes, either invalidations have to be reported through an API (e.g., from shop and content management systems) or Orestes has to periodically crawl and refresh currently cached data. Through the centralized storage of metadata, periodic refreshes can be specified by the application based on content types, URL patterns, and other metadata. This improves upon the purely URL-based invalidation model of CDNs [PB07]. The consistency guarantees in the refresh model are defined by the Δ-atomicity of the Cache Sketch combined with the refresh interval. Within these bounds, the system can learn to adapt to the actual frequency with that content changes in a stochastic model to optimize the probabilistic (Δ, p)-atomicity (cf. Section 2.2.4). Furthermore, the optimization problem of trading cache misses against refresh overhead arises as small intervals can impose

substantial overhead on the original data source. The system therefore has to learn distinguishing between infrequently accessed resources and the critical working set of a REST API or website.

Customization and Segmentation through Dynamic Blocks. A primary challenge for proxy-based caching are personalized server-generated pages, e.g., a user-specific shopping promotion based on tracking data. This problem can be addressed by the Cache Sketch through a concept for dynamic blocks: if developers can declare portions of the page as user- or segment-specific through selectors, Orestes can deliver a cached anonymous version of the page immediately while fetching the customized version in the background. When the slower response arrives, the anonymous dynamic blocks are replaced by the customized version in the client. Thus, the frontend starts rendering early and swaps the dynamic blocks later. The scheme potentially accelerates even highly customized websites with complex business logic (e.g., e-commerce platforms) because a cached generic page can be loaded faster than customized views, so that dependencies (e.g., images and scripts) can be resolved while the dynamic content is still being loaded. An important question for future work therefore is, whether dynamic blocks can be detected automatically by online analysis of the workload and page structure.

Low Latency for RDBMSs. While the proxy model with Service Workers enables the use of Cache Sketches for websites and REST APIs, applying the techniques to relational database systems requires further work. The central problem for caching is that every query result may have a different form, including joined attributes and aggregations. Inexpensive object-based caching thus is the exception, whereas SQL query results constitute most of the requests. Therefore, Orestes needs to support real-time matching of SQL queries in order to perform appropriate cache invalidations and to support the continuous query API (cf. Section 4.6). This implies the question of how to perform joins and aggregations efficiently in a streaming fashion for cached queries. To limit the problem of matching overhead, identifying underlying SQL query templates used in the application may be an essential optimization to reduce resources for different instantiations of the same query pattern. To combine relational with non-relational modeling, the polyglot schema of Orestes has to be extended to support complex integrity constraints and referential actions. Optimistic DCAT transactions can be optimized for relational systems by employing server-side pessimistic transactions for the short-lived optimistic commit procedure. By combining the performance and scalability benefits of NoSQL databases with the high level of expressiveness in RDBMSs, Orestes could become a general-purpose middleware for low latency data management.

7.2.2 Reinforcement Learning of Caching Decisions

The problem of optimizing TTLs and deciding which resources to cache is pivotal to the effectiveness and size of the Cache Sketch. Therefore, in future work, this problem may be

addressed through a comprehensive machine learning approach that adapts to the actual workload characteristics of applications.

Predictive Online Learning. The current TTL estimation approach for query results and objects in Orestes is *backward-oriented*: by continuously monitoring actual TTLs, the system adapts to new conditions. However, any spikes or patterns are only detected after they arise. To tackle this problem, a *forward-oriented* or predictive approach is required that extrapolates from the current workload to adapt to patterns such as seasonal spikes before they occur. To this end, prior work on regression, predictive analytics, and time series models [LBMAL14] can be extended to this new context of deriving caching decisions from read, query, and update operations over time. In addition, the challenges of selecting the right model for a type of application and finding an online solution or an optimal frequency of model retraining have to be tackled, too.

Reinforcement Learning of TTLs at Runtime. Training a predictive model for TTL estimation is difficult in a real-world production setting, because the system exhibits several hidden parameters such as cache hit rates, read frequencies, and the state of expiration-based caches. This issue can be tackled by posing the problem as a reinforcement learning scenario: though certain parameters are unobservable, several proxy values (e.g., invalidation rate and system load) indicate the effectiveness of TTL estimates and cacheability decisions. By treating TTL estimates as the output of a reinforcement learning model with a delayed reward, a generic online solution may be derived in follow-up research. While we have shown the basic feasibility of this idea [SGDY16], it remains to be shown that (deep) reinforcement learning can achieve performance comparable to that of pre-trained predictive models for both object and query result caching.

Learning Cacheability. Besides learning optimal TTLs, it is crucial to optimize the false positive rate of the Bloom filter by only caching data that has the potential to achieve a high cache hit rate. The required computing capacity for query matching scales linearly with the number of cached query results. Therefore, the optimization problem of selecting the most rewarding queries for caching has to be addressed. Under a given matching throughput of query-to-object comparisons per second, the relevant subset of queries has to be identified and maintained for changing workloads. This problem is inherently similar to reinforcement learning of TTLs, as decisions on cacheability are reflected in a later reward for global system performance parameters and SLA compliance.

Predictive Pushing. Apart from data management, learning can also improve networking and protocol usage. The number of network round-trips before the first meaningful paint in the browser is critical for user-perceived performance. One central optimization to minimize the number of round-trips is to push resources with HTTP/2 that the client is likely to request. However, determining what to push is inherently difficult as pushed resources slow down other resources when they are already cached in the client or not requested at all. Therefore, treating the pushing strategy as an optimization problem is difficult, as the state of the client's cache is unknown and resources that might be re-

quested (e.g., referenced objects in a query result) are non-deterministic. The problem can be tackled through a reinforcement learning process that decides what to push based on historic performance metrics reported by clients.

Model Training through Monte Carlo Simulation. To precisely evaluate the performance of caching and learning algorithms, knowledge on hidden system parameters is required. In future work, our approach of the YCSB Monte Carlo Simulator (YMCA, cf. Section 4.3.1) can be extended from simple client-server setups and predefined distributions to real-world network topologies with accurate latency distributions based on network protocol traces. This not only allows offline training of predictive models, but also enables pre-testing any TTL estimator and cache coherence scheme for a potentially broad set of application workloads and network topologies.

7.2.3 Fully Automatic Polyglot Persistence

Our proposed mediator approach offers extensive opportunities for future work in order to make it applicable to sophisticated data management problems.

Measurement and Enforcement of SLAs. The first step towards automated polyglot persistence is to not only map the desired SLAs to database systems, but to monitor compliance of each SLO in realtime. This enables reacting to violations of SLAs, so that the system can maximize the overall fulfillment of SLAs and make predictions on the costs of providing a particular SLA. The optimization goal of minimizing expected SLA violation costs $\sum_{n=0}^{N} P(violation_n) \cdot penalty(violation_n)$ could potentially be tackled through various time series prediction methods. However, since the mediator has detailed knowledge on the employed system, it could potentially learn a system model for each type of database system that maps a set of SLAs under a given system configuration to a probability of violation. In future research, it should therefore be evaluated whether optimal SLA fulfillment is rather achieved through workload management at the level of the PPM or through configuration changes and scaling actions on the underlying database systems (cf. Section 5.4). While we presented the PPM in the context of a cloud service orchestrating different database systems, it could also be employed as a meta-DBaaS that incorporates and refines the SLA models of employed DBaaS systems. Thus, the following research question emerges: how can composition relationships of SLAs within a schema be treated in modeling and enforcement, in order to meet the top-level SLA definition of the application?

Live Migration. In order to prevent imminent SLA violations caused by system overload, the mediator has to perform live migrations. While Orestes provides offloading of critical reads during migration, writes need to be handled explicitly by the PPM, while migrations are ongoing. Therefore, prior work on live migration [EDAE11, BCM+12, DNAE11, DAEA13, DEAA09, SKD17] has to be extended to polyglot settings by supporting multi-model data transformation, field-level migrations, and rewriting of queries during the migration process. Further, the risk of violating SLAs during migration has to be learned in order to find the right point in time for a migration.

Polyglot Auto-Scaling. Similar to live migration, prior work on auto-scaling [PN09, HMC+12, KF11, GSLI11, CS13, BHD13, TJDB06, XRB12, USC+08, CDM11, SSGW11] is substantially limited in polyglot persistence architectures as each system is treated individually. Therefore, incorporating the possibility of scaling in and out multiple NoSQL database systems in the context of a single application workload constitutes an important area of future work. Ideally, auto-scaling measures should be tightly integrated into the SLA monitoring and workload management process, to actively prevent SLA violations through scaling as well as optimizing resource utilization for a multi-tenant service.

Visual Configuration and Decision Guidance. A key element of making polyglot persistence practically relevant is ease of use in the various application development phases from design to implementation. This is particularly true for small applications where in-depth knowledge of database technologies cannot be presumed. Visual modeling tools can simplify the usage of the PPM, by offering tunable "knobs" to define SLA trade-offs in the form of simple, visually presented utility functions (e.g., the usefulness of latency over throughput). The simple presentation of complex functional and non-functional system traits like consistency is an interesting research problem in the intersection of human-computer interaction and data management. Another problem that can be supported by visual tools is the definition of polyglot schemas. Presenting both the schema-based SLAs as well as hints for the physical placement of schema components visually might simplify the use of different data stores.

Polyglot Data Analytics and Queries. As soon as data is split across multiple systems, analytics, machine learning, OLAP, and data science applications cannot easily make use of the complete data set. In the primary database model where the PPM asynchronously materializes all data to a defined system for a complete view (cf. Section 5.2.1), data analytics can be performed on the primary system. However, if real-time or near-real-time analytics are required, computations and queries are only feasible when split across the partitions, as typically done in Big Data frameworks like Hadoop's YARN [Whi15] and Spark's RDDs [ZCD+12]. The scheduling and query rewriting needs to be performed in the PPM, which is only possible, when the various challenges of virtual integration in a polyglot context are solved, in particular, the mapping between different query languages and compensating for the lack of expressiveness in low-level query languages. The same problem applies to OLTP workloads, where system-agnostic query languages like GraphQL [Gra17] might help to build a standard interface for polyglot queries with non-trivial expressiveness.

7.2.4 Polyglot, Cache-Aware Transactions

We proposed DCAT as a general-purpose, optimistic transaction approach that is applicable to any data store supporting linearizability. The idea of combining server-side validation with latency reduction based on Cache Sketches can be extended in several ways.

Constraint Checking and Full Query Support. In future work, DCAT should be extended to constraint checking during the commit procedure. This would allow to guarantee global correctness invariants that are otherwise impossible to maintain (e.g., referential integrity). Furthermore, explicit support for queries is necessary in order to prevent the phantom problem which is currently not prevented by DCAT (cf. Section 4.8). Further research on scaling the coordinator of the validation should analyze whether the centralized approach can be improved in latency and scalability by a sharded lock allocation scheme.

Exploiting Commutativity. An important performance optimization is the consideration of partial update operations in the commit procedure. As certain partial updates commute with each other and do not rely on object versions (e.g., counter increments and decrements), the potential for lock conflicts during validation can be reduced to lower the overall abort rate. In the best case, a transaction contains only mutually commutative updates which can be executed concurrently, without any locking at all.

Multi-System Polyglot Transactions. As Orestes unifies access to different systems, transactions are exposed in a coherent fashion irrespective of the underlying databases. In future work, the use of multiple data stores within a single transaction should be explicitly supported. For developers, this should be transparent, as the distribution of data is implicitly based on the schema and its annotations. Polyglot DCAT transactions would thus allow Orestes and the PPM to organize the data independently from transaction scopes.

Automatic Transaction Protocol Selection. For transactions with high contention, the pessimistic protocols can achieve lower abort rates than an optimistic commit procedure. By detecting such cases and reverting to a pessimistic protocol, performance could improve substantially. Furthermore, if relaxed transaction properties are sufficient for a particular transaction, an automatic choice between several client-, middleware-, and server-coordinated protocols could be made [XSL+15, WPC12, BFG+14, Dey15, GJK+14, LLS+15, KKN+08]. A central future research goal therefore is the automatic selection of the most appropriate transaction protocol to minimize transaction latency and external aborts. The decision can be based on both the outcome of a single transaction (e.g., repeated retries) as well as the overall transaction workload of an application or even a multi-tenant service. By treating the transaction protocol selection as a machine learning problem, Orestes could potentially select the most appropriate approach based on performance, success probability, resource usage, and the required level of transaction guarantees. For example, when excessive deadlocks compromise pessimistic transactions, Orestes would switch to an optimistic commit and revert when write hotspots lead to repeatedly failed validations. This approach combined with the PPM could eventually hide the choice of a database system or transaction protocol behind a purely declarative interface without compromising performance. Effectively, developers would be enabled to build transactional applications that guarantee high throughput, low latency, and scalability without having to know or choose the underlying transaction implementation.

7.3 Closing Thoughts

High latency is a fundamental challenge in our increasingly connected, digital society. With ever-growing scale and distribution, the web is facing a performance problem seemingly imposed by simple rules of physics: the speed of light as the upper bound for the speed of information propagation. Though caching is arguably the most wide-spread performance optimizations in computer science, loading times on computers and mobile devices are omnipresent in our everyday lives. The missing piece in transforming the web towards instant response times is fast access to dynamically changing pieces of data. In the past, volatile content such as personalized shopping recommendations, tailored news streams, and social media updates could not be tackled by caching, making them completely subject to the physical distance between a user and a service.

In this dissertation, we introduce Orestes to address latency by bringing web and database technology together. Orestes disentangles physical network latency from application performance, by caching dynamic data in distributed web caches available all over the world. For caching of queries, reads, and files, we derived means for explicit control over both consistency requirements and ACID transactions. Orestes solves cache coherence in a database-independent fashion by exposing cloud data management through a unified REST interface. It enhances NoSQL systems to become full-fledged Database/Backend-as-a-Service systems which profoundly increases developer productivity compared to traditional software architectures. We argued that not only low latency can be accomplished in a generic fashion, but even the choice of the most suitable database technology can be automated by an SLA-aware Polyglot Persistence Mediator. We provided empirical evidence that across a range of workloads, our approach scales horizontally and improves latency by an order of magnitude over the current state of the art.

In times where the web has become an indispensable cornerstone of society, people are accustomed to slow-loading websites, and developers accept spending copious amounts of time optimizing performance. The techniques introduced in this thesis enable a web with imperceptible delays. The dramatic latency improvements may open up use cases that were not yet conceivable for performance reasons. Caching of dynamic data potentially makes the difference between an application with sub-second loading times and one that users will never use. With Orestes, we address a severe problem that the database community has been facing recently. While newly proposed data stores usually excel in one particular performance property, none excel in all. With this dissertation, we argue that tackling latency in cloud data management is a challenge that can be solved once for all systems through a database-independent middleware that combines their strengths. We are convinced that further research on this idea within the web and database fields can eventually eliminate noticeable loading times entirely.

Bibliography

[AA17] Joshua Bell Ali Alabbas. Indexed Database API 2.0. `https://w3c.github.io/IndexedDB/`, 2017. (Accessed on 07/14/2017).

[AAB05] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. On the Availability of Non-strict Quorum Systems. In Pierre Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2005.

[AAO+11] Sadiye Alici, Ismail Sengor Altingovde, Rifat Ozcan, Berkant Barla Cambazoglu, and Özgür Ulusoy. Timestamp-based result cache invalidation for web search engines. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 973–982. ACM, 2011.

[AAO+12] Sadiye Alici, Ismail Sengor Altingovde, Rifat Ozcan, B. Barla Cambazoglu, and Özgür Ulusoy. Adaptive time-to-live strategies for query result caching in web search engines. In *European Conference on Information Retrieval*, pages 401–412. Springer, 2012.

[Aba12] D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45(2):37–42, 2012.

[ABC14] Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. Impact of Response Latency on User Behavior in Web Search. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, pages 103–112, Gold Coast, Queensland, Australia, 2014. ACM.

[ABD+12] Aditya Auradkar, Chavdar Botev, Shirshanka Das, Dave De Maagd, Alex Feinberg, Phanindra Ganti, Lei Gao, Bhaskar Ghosh, Kishore Gopalakrishna, Brendan Harris, Joel Koshy, Kevin Krawez, Jay Kreps, Shi Lu, Sunil Nagaraj, Neha Narkhede, Sasha Pachev, Igor Perisic, Lin Qiao, Tom Quiggle, Jun Rao, Bob Schulman, Abraham Sebastian, Oliver Seeliger, Adam Silberstein, Boris Shkolnik, Chinmay Soman, Roshan Sumbaly, Kapil Surlaker, Sajid Topiwala, Cuong Tran, Balaji Varadarajan, Jemiah Westerman, Zach White, David Zhang, and Jason Zhang. Data Infrastructure at LinkedIn. In Anastasios Ke-

mentsietsidis and Marcos Antonio Vaz Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 1370–1381. IEEE Computer Society, 2012.

[ABGS86] Divyakant Agrawal, Arthur J. Bernstein, Pankaj Gupta, and Soumitra Sengupta. Distributed Multi-Version Optimistic Concurrency Control for Relational Databases. In *Spring COMPCON'86, Digest of Papers, Thirty-First IEEE Computer Society International Conference, San Francisco, California, USA, March 3-6, 1986*, pages 416–421. IEEE Computer Society, 1986.

[ABK+03] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache Tables: Paving the Way for an Adaptive Database Cache. In *VLDB*, pages 718–729, 2003.

[ABK+15] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[ABMM07] Atul Adya, José A Blakeley, Sergey Melnik, and S Muralidhar. Anatomy of the ado. net entity framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 877–888. ACM, 2007.

[ABN+95] H. Attiya, A. Bar-Noy, et al. Sharing memory robustly in message-passing systems. *JACM*, 42(1), 1995.

[ACK+11] Michael Armbrust, Kristal Curtis, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. PIQL: success-tolerant query processing in the cloud. *PVLDB*, 5(3):181–192, 2011.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. Web services. In *Web Services*, pages 123–149. Springer, 2004.

[ACPS96] Sibel Adali, K. Selçuk Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 137–148. ACM Press, 1996.

[ADE12] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. *Data Management in the Cloud: Challenges and Opportunities*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[Ady99] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[AEM+13] Divyakant Agrawal, Amr El Abbadi, Hatem A. Mahmoud, Faisal Nawab, and Kenneth Salem. Managing Geo-replicated Data in Multi-datacenters. In Aastha Madaan, Shinji Kikuchi, and Subhash Bhalla, editors, *Databases in*

Networked Information Systems - 8th International Workshop, DNIS 2013, Aizu-Wakamatsu, Japan, March 25-27, 2013. Proceedings, volume 7813 of *Lecture Notes in Computer Science*, pages 23–43. Springer, 2013.

[Aer18] Aerospike. `http://www.aerospike.com/`, 2018. (Accessed on 05/11/2018).

[AG17] Nick Antonopoulos and Lee Gillam, editors. *Cloud Computing: Principles, Systems and Applications (Computer Communications and Networks)*. Springer, 2nd ed. 2017 edition, 7 2017.

[Agg06] Charu C. Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *Proceedings of the 32nd international conference on Very large data bases*, pages 607–618. VLDB Endowment, 2006.

[AGJ+08] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1195–1206, 2008.

[AGK95] Brad Adelberg, Hector Garcia-Molina, and Ben Kao. Applying Update Streams in a Soft Real-Time Database System. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 245–256. ACM Press, 1995.

[AGLM95] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 23–34. ACM Press, 1995.

[AJKS09] Stefan Aulbach, Dean Jacobs, Alfons Kemper, and Michael Seibold. A comparison of flexible schemas for software as a service. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 881–888. ACM, 2009.

[AJL+02] Jesse Anton, Lawrence Jacobs, Xiang Liu, Jordan Parker, Zheng Zeng, and Tie Zhong. Web caching for database applications with Oracle Web Cache. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 594–599. ACM, 2002.

[Aki15] Tyler Akidau. The world beyond batch: Streaming 101. *O'Reilly Media*, August 2015. Accessed on 08/21/2017.

[Aki16] Tyler Akidau. The world beyond batch: Streaming 102. *O'Reilly Media*, January 2016. Accessed on 08/21/2017.

[All10] Subbu Allamaraju. *Restful web services cookbook: solutions for improving scalability and simplicity*. " O'Reilly Media, Inc.", 2010.

[ALO00] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized Isolation Level Definitions. In David B. Lomet and Gerhard Weikum, editors, *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 67–78. IEEE Computer Society, 2000.

[ALS10] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB - The Definitive Guide: Time to Relax*. O'Reilly, 2010.

[Alw17] Always Encrypted (Database Engine). `https://msdn.microsoft.com/en-us/library/mt163865.aspx`, 2017. (Accessed on 05/20/2017).

[Ama16] Sean Amarasinghe. *Service worker development cookbook*. 2016. OCLC: 958120287.

[Ama17a] Amazon Simple Storage Service (S3). `//aws.amazon.com/documentation/s3/`, 2017. (Accessed on 07/28/2017).

[Ama17b] Amazon Web Services AWS – Server Hosting & Cloud Services. `https://aws.amazon.com/de/`, 2017. (Accessed on 05/20/2017).

[Amb12] Scott Ambler. *Agile database techniques: Effective strategies for the agile software developer*. John Wiley & Sons, 2012.

[AMS+07] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[Amu17] Mike Amundsen. *RESTful Web Clients: Enabling Reuse Through Hypermedia*. O'Reilly Media, 1 edition, 2 2017.

[Ang17] Angular Framework. `https://angular.io/`, 2017. (Accessed on 05/26/2017).

[Apa17a] Apache jclouds. `https://jclouds.apache.org/`, 2017. (Accessed on 06/05/2017).

[Apa17b] Apache Libcloud. `http://libcloud.apache.org/index.html`, 2017. (Accessed on 06/05/2017).

[APB09] Mark Allman, Vern Paxson, and Ethan Blanton. TCP congestion control. Technical report, 2009.

[App17] App Engine (Google Cloud Platform). `https://cloud.google.com/appengine/`, 2017. (Accessed on 05/20/2017).

[APTP03a] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proceedings of the ICDE*, pages 821–831, 2003.

[APTP03b] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. Scalable template-based query containment checking for web semantic caches. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 493–504. IEEE Computer Society, 2003.

[AR17] Ejaz Ahmed and Mubashir Husain Rehmani. Mobile Edge Computing: Opportunities, solutions, and challenges. *Future Generation Comp. Syst.*, 70:59–63, 2017.

[Ara17] ArangoDB. `https://www.arangodb.com/documentation/`, 2017. (Accessed on 05/20/2017).

[Arc18] HTTP Archive. `http://httparchive.org/trends.php`, 2018. Accessed: 2018-07-14.

[ASJK11] Stefan Aulbach, Michael Seibold, Dean Jacobs, and Alfons Kemper. Extensibility and Data Sharing in evolving multi-tenant databases. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 99–110. IEEE Computer Society, 2011.

[AT14] Masoud Saeida Ardekani and Douglas B. Terry. A Self-Configurable Geo-Replicated Cloud Storage System. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 367–381. USENIX Association, 2014.

[ATE12] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In Filip De Turck, Luciano Paschoal Gaspary, and Deep Medhi, editors, *2012 IEEE Network Operations and Management Symposium, NOMS 2012, Maui, HI, USA, April 16-20, 2012*, pages 204–212. IEEE, 2012.

[AWS17] AWS Elastic Beanstalk - PaaS Application Management. `https://aws.amazon.com/de/elasticbeanstalk/`, 2017. (Accessed on 05/20/2017).

[Azu17] Microsoft Azure: Cloud Computing Platform & Services. `https://azure.microsoft.com/en-us/`, 2017. (Accessed on 05/20/2017).

[BAC+13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, and others. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[Bac17] Backbone.js. `http://backbonejs.org/`, 2017. (Accessed on 05/26/2017).

[Bai15] Peter Bailis. *Coordination Avoidance in Distributed Databases*. PhD thesis, University of California, Berkeley, USA, 2015.

[BAK+03] Christof Bornhövd, Mehmet Altinel, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, page 662. ACM, 2003.

[BAM+04] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *Data Engineering*, 27(2):11–18, 2004.

[Baq18] News BaaS Benchmark. `https://github.com/Baqend/news-benchmark`, 2018. (Accessed on 09/08/2018).

[Bas12] Salman A. Baset. Cloud SLAs: present and future. *ACM SIGOPS Operating Systems Review*, 46(2):57–66, 2012.

[Bas17] The BaasBox server. `https://github.com/baasbox/baasbox`, 2017. (Accessed on 05/20/2017).

[BBB+17] David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a SQL system. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 331–343. ACM, 2017.

[BBC+11] J. Baker, C. Bond, J.C. Corbett, JJ Furman, A. Khorlin, J. Larson, J.M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, volume 11, pages 223–234, 2011.

[BBJ+10] Roi Blanco, Edward Bortnikov, Flavio Junqueira, Ronny Lempel, Luca Telloli, and Hugo Zaragoza. Caching search engine results over incremental indices. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 82–89. ACM, 2010.

[BCD+11] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL server for cloud computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1255–1263. IEEE, IEEE, 2011.

[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, IEEE, 1999.

[BCL89] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.

[BCM+12] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. Cut me some slack: Latency-aware live migration for databases. In *Proceedings of the 15th international conference on extending database technology*, pages 432–443. ACM, 2012.

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 164–177, 2003.

[BDF+13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. *Proceedings of the VLDB Endowment*, 7(3), 2013. 00001.

[BDF+15] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. pages 1–16. ACM Press, 2015.

[BDG+06] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In Larry L. Peterson and Timothy Roscoe, editors, *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings*. USENIX, 2006.

[BDK+02] Manish Bhide, Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant Shenoy. Adaptive push-pull: Disseminating dynamic web data. *IEEE Transactions on Computers*, 51(6):652–668, 2002.

[Bec00] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[Bel10] Mike Belshe. More Bandwidth Doesn't Matter (much). Technical report, Google Inc., 2010.

[Ben14] Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014.

[Ber99] Philip A. Bernstein. Review - A majority consensus approach to concurrency control for multiple copy databases. *ACM SIGMOD Digital Review*, 1, 1999.

[Ber14] David Bermbach. *Benchmarking Eventually Consistent Distributed Storage Systems*. KIT Scientific Publishing, Karlsruhe, Baden, 2014.

[Ber15] David Bermbach. An Introduction to Cloud Benchmarking. In *2015 IEEE International Conference on Cloud Engineering, IC2E 2015, Tempe, AZ, USA, March 9-13, 2015*, page 3. IEEE Computer Society, 2015.

[Bes95] Azer Bestavros. Demand-based document dissemination to reduce traffic and balance load in distributed information systems. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing, SPDP 1995, San Antonio, Texas , USA, October 25-28, 1995*, pages 338–345. IEEE, 1995.

[Bes96] A. Bestavros. Speculative data dissemination and service to reduce server load, network traffic and service time in distributed information systems. In *Proc. Twelfth Int. Conf. Data Engineering*, pages 180–187, February 1996.

[BFF+14] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3):185–196, 2014.

[BFG+13] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. HAT, not CAP: Highly available transactions. In *Workshop on Hot Topics in Operating Systems*, 2013.

[BFG+14] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable Atomic Visibility with RAMP Transactions. In *ACM SIGMOD Conference*, 2014.

[BFG+16] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with RAMP transactions. *ACM Trans. Database Syst.*, 41(3):15:1–15:45, 2016.

[BG13] Sumita Barahmand and Shahram Ghandeharizadeh. BG: A benchmark to evaluate interactive social networking actions. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

[BGH+15] Christopher D Bienko, Marina Greenstein, Stephen E Holt, Richard T Phillips, et al. *IBM Cloudant: Database as a Service Advanced Topics*. IBM Redbooks, 2015.

[BGHS13] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[BGS+09] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, San Diego, California, 2009. USENIX Association.

[BHD13] Enda Barrett, Enda Howley, and Jim Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, 25(12):1656–1674, 2013.

[BJ87] Ken Birman and Thomas Joseph. *Exploiting virtual synchrony in distributed systems*, volume 21. ACM, 1987.

[BK13] David Bermbach and Jörn Kuhlenkamp. Consistency in Distributed Storage Systems - An Overview of Models, Metrics and Measurement Approaches. In Vincent Gramoli and Rachid Guerraoui, editors, *Networked Systems - First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers*, volume 7853 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2013.

[BK14] Peter Bailis and Kyle Kingsbury. The network is reliable. *Queue*, 12(7):20, 2014.

[BKD+13] David Bermbach, Jörn Kuhlenkamp, Bugra Derre, Markus Klems, and Stefan Tai. A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores. In *2013 IEEE International Conference on Cloud Engineering, IC2E 2013, San Francisco, CA, USA, March 25-27, 2013*, pages 114–123. IEEE Computer Society, 2013.

[BKD+14] David Bermbach, Jörn Kuhlenkamp, Akon Dey, Sherif Sakr, and Raghunath Nambiar. Towards an Extensible Middleware for Database Benchmarking. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking. Traditional to Big Data - 6th TPC Technology Conference, TPCTC 2014, Hangzhou, China, September 1-5, 2014. Revised Selected Papers*, volume 8904 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2014.

[Blo70] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently Updating Materialized Views. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986.*, pages 61–71. ACM Press, 1986.

[BLV11] Edward Bortnikov, Ronny Lempel, and Kolman Vornovitsky. Caching for Realtime Search. In Paul D. Clough, Colum Foley, Cathal Gurrin, Gareth J. F. Jones, Wessel Kraaij, Hyowon Lee, and Vanessa Murdock, editors, *Advances in Information Retrieval - 33rd European Conference on IR Research, ECIR 2011, Dublin, Ireland, April 18-21, 2011. Proceedings*, volume 6611 of *Lecture Notes in Computer Science*, pages 104–116. Springer, 2011.

[BM03] Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485–509, 2003.

[BM13] Arshdeep Bahga and Vijay Madisetti. *Cloud Computing: A Hands-on Approach*. CreateSpace Independent Publishing Platform, 2013.

[BMZA12] Flavio Bonomi, Rodolfo A. Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In Mario Gerla and Dijiang Huang, editors, *Proceedings of the first edition of the MCC workshop on Mobile cloud computing, MCC@SIGCOMM 2012, Helsinki, Finland, August 17, 2012*, pages 13–16. ACM, 2012.

[BN09] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 2009.

[BP95] Alexandros Biliris and Euthimios Panagos. A High Performance Configurable Storage Manager. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 35–43. IEEE Computer Society, 1995.

[BPV08] Rajkumar Buyya, Mukaddim Pathan, and Athena Vakali, editors. *Content Delivery Networks (Lecture Notes in Electrical Engineering)*. Springer, 2008 edition, 9 2008.

[BR02] Laura Bright and Louiqa Raschid. Using Latency-Recency Profiles for Data Delivery on the Web. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 550–561. Morgan Kaufmann, 2002.

[Bre00] Eric A. Brewer. Towards Robust Distributed Systems., 2000.

[Bre17] Eric Brewer. Spanner, TrueTime and the CAP Theorem. Technical report, 2017.

[BROL14] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *VLDB*, 7(13), 2014.

[BRX13] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. Coordinated Self-Configuration of Virtual Machines and Appliances Using a Model-Free Learning Approach. *IEEE Trans. Parallel Distrib. Syst.*, 24(4):681–690, 2013.

[BT11] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In Karl M. Göschka, Schahram Dustdar, and Vladimir Tosic, editors, *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing, MW4SOC 2011, Lisbon, Portugal, December 12-16, 2011*, page 1. ACM, 2011.

[BT14] David Bermbach and Stefan Tai. Benchmarking Eventual Consistency: Lessons Learned from Long-Term Experimental Studies. In *2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, March 11-14, 2014*, pages 47–56. IEEE Computer Society, 2014.

[BVF+12] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. Technical Report 8, 2012.

[BVF+14] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. *The VLDB Journal*, 23(2):279–302, April 2014.

[BWA13] Meenakshi Bist, Manoj Wariya, and Amit Agarwal. Comparing delta, open stack and Xen Cloud Platforms: A survey on open source IaaS. In *Advance Computing Conference (IACC), 2013 IEEE 3rd International*, pages 96–100. IEEE, 2013.

[BWT17] David Bermbach, Erik Wittern, and Stefan Tai. *Cloud Service Benchmarking - Measuring Quality of Cloud Services from a Client Perspective*. Springer, 2017.

[BYV+09] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[BZM+13] Mahesh Balakrishnan, Aviad Zuck, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, and Tao Zou. Tango: distributed data structures over a shared log. pages 325–340. ACM Press, 2013.

[BZS13] David Bermbach, Liang Zhao, and Sherif Sakr. Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*, volume 8391 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2013.

[CAAS07] Ítalo S. Cunha, Jussara M. Almeida, Virgílio A. F. Almeida, and Marcos Santos. Self-Adaptive Capacity Management for Multi-Tier Virtualized Environments. In *Integrated Network Management, IM 2007. 10th IFIP/IEEE International Symposium on Integrated Network Management, Munich, Germany, 21-25 May 2007*, pages 129–138. IEEE, 2007.

[CALM97] Miguel Castro, Atul Adya, Barbara Liskov, and Andrew C. Myers. HAC: hybrid adaptive caching for distributed storage systems. In Michel Banâtre, Henry M. Levy, and William M. Waite, editors, *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 102–115. ACM, 1997.

[Cam16] Raymond Camden. *Client-side data storage: keeping it local*. O'Reilly, Beijing, first edition edition, 2016. OCLC: ocn935079139.

[Car13] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[Cas81] Marco A. Casanova. *The Concurrency Control Problem for Database Systems*, volume 116 of *Lecture Notes in Computer Science*. Springer, 1981.

[Cat92] Vincent Cate. Alex-a global filesystem. In *Proceedings of the 1992 USENIX File System Workshop*, pages 1–12. Citeseer, 1992.

[CBPS10] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.

[CCDM13] Jerry Chu, Yuchung Cheng, Nandita Dukkipati, and Matt Mathis. Increasing TCP's initial window. 2013.

[CCRJ14] Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan, and Arvind Jain. Tcp fast open. Technical report, 2014.

[CD13] Kristina Chodorow and Michael Dirolf. *MongoDB - The Definitive Guide*. O'Reilly, 2013.

[CDE+12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 261–264. USENIX Association, 2012.

[CDE+13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, 2013.

[CDF+07] Pierre Cassier, Annamaria Defendi, Dagmar Fischer, John Hutchinson, Alain Maneville, Gianfranco Membrini, Caleb Ong, and Andrew Rowley. *System Programmer's Guide To–Workload Manager*. IBM, 2007.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[CDM11] Eddy Caron, Frédéric Desprez, and Adrian Muresan. Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients. *J. Grid Comput.*, 9(1):49–64, 2011.

[CEAK16] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[CFLS91] Michael J. Carey, Michael J. Franklin, Miron Livny, and Eugene J. Shekita. Data caching tradeoffs in client-server DBMS architectures. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991.*, pages 357–366. ACM Press, 1991.

[CGH+17] Paris Carbone, Gábor E. Gévay, Gábor Hermann, Asterios Katsifodimos, Juan Soto, Volker Markl, and Seif Haridi. Large-Scale Data Stream Processing Systems. In Albert Y. Zomaya and Sherif Sakr, editors, *Handbook of Big Data Technologies*, pages 219–260. Springer, 2017.

[CH16] Jeff Carpenter and Eben Hewitt. *Cassandra: The Definitive Guide*. " O'Reilly Media, Inc.", 2016.

[Cha15] Lee Chao. *Cloud Computing Networking: Theory, Practice, and Development*. Auerbach Publications, 2015.

[Cha17] Dave Chaffey. Ecommerce conversion rates. *smartinsights.com*, 2017. accessed: 2017-05-15.

[CI97] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *1st USENIX Symposium on Internet Technologies and Systems, USITS'97, Monterey, California, USA, December 8-11, 1997*. USENIX, 1997.

[CJMB11] Carlo Curino, Evan P. C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 313–324. ACM, 2011.

[CJP+10] Berkant Barla Cambazoglu, Flavio Paiva Junqueira, Vassilis Plachouras, Scott A. Banachowski, Baoqiu Cui, Swee Lim, and Bill Bridge. A refreshing perspective of search engine caching. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 181–190. ACM, 2010.

[CJP+11] Carlo Curino, Evan PC Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational Cloud: A Database-as-a-Service for the Cloud. In *Proc. of CIDR*, 2011.

[CJZM10] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceed-*

ings of the VLDB Endowment, 3(1-2):48–57, 2010.

[CK01] Edith Cohen and Haim Kaplan. The Age Penalty and Its Effect on Cache Performance. In Tom Anderson, editor, *3rd USENIX Symposium on Internet Technologies and Systems, USITS'01, San Francisco, California, USA, March 26-28, 2001*, pages 73–84. USENIX, 2001.

[CKR98] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. In *SIGCOMM*, pages 241–253, 1998.

[CL98] Pei Cao and Chengjie Liu. Maintaining Strong Cache Consistency in the World Wide Web. *IEEE Trans. Computers*, 47(4):445–457, 1998.

[CLL+01a] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling Dynamic Content Caching for Database-Driven Web Sites. In Sharad Mehrotra and Timos K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 532–543. ACM, 2001.

[CLL+01b] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling Dynamic Content Caching for Database-driven Web Sites. In *SIGMOD*, pages 532–543, New York, NY, USA, 2001. ACM.

[Clo17a] Cloud Application Platform - Devops Platform | Cloud Foundry. `https://www.cloudfoundry.org/`, 2017. (Accessed on 06/05/2017).

[Clo17b] Cloud Application Platform | Heroku. `https://www.heroku.com/`, 2017. (Accessed on 05/20/2017).

[Clo17c] vCloud Suite, vSphere-Based Private Cloud: VMware. `http://www.vmware.com/products/vcloud-suite.html`, 2017. (Accessed on 06/05/2017).

[Clu17] Clustrix: A New Approach to Scale-Out RDBMS. `http://www.clustrix.com/wp-content/uploads/2017/01/Whitepaper-ANewApproachtoScaleOutRDBMS.pdf`, 2017. (Accessed on 05/20/2017).

[CMH11] Yun Chi, Hyun Jin Moon, and Hakan Hacigümüs. iCBS: Incremental Cost-based Scheduling under Piecewise Linear SLAs. *PVLDB*, 4(9):563–574, 2011.

[CO82] Stefano Ceri and Susan S. Owicki. On the Use of Optimistic Methods for Concurrency Control in Distributed Databases. In *Berkeley Workshop*, pages 117–129, 1982.

[Coc17] CockroachDB - the scalable, survivable, strongly-consistent SQL database. `https://github.com/cockroachdb/cockroach`, 2017. (Accessed on 05/20/2017).

[Coo13] Brian F. Cooper. Spanner: Google's globally-distributed database. In Ronen I. Kat, Mary Baker, and Sivan Toledo, editors, *6th Annual International Systems*

and Storage Conference, SYSTOR '13, Haifa, Israel - June 30 - July 02, 2013, page 9. ACM, 2013.

[CRF08] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable Isolation for Snapshot Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, New York, NY, USA, 2008. ACM.

[Cro06] Douglas Crockford. JSON: Javascript object notation. *URL http://www. json. org*, 2006.

[CRS99] Boris Chidlovskii, Claudia Roncancio, and Marie-Luise Schneider. Semantic Cache Mechanism for Heterogeneous Web Querying. *Computer Networks*, 31(11-16):1347–1360, 1999.

[CRS+08] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[CRW15] Aaron Cordova, Billie Rinaldi, and Michael Wall. *Accumulo: Application Development, Table Design, and Best Practices*. " O'Reilly Media, Inc.", 2015.

[CS13] Emiliano Casalicchio and Luca Silvestri. Autonomic management of cloud-based systems: the service provider perspective. In *Computer and Information Sciences III*, pages 39–47. Springer, 2013.

[CSH+16] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. CacheOptimizer: helping developers configure caching frameworks for hibernate-based database-centric web applications. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 666–677. ACM, 2016.

[CSSS11] Emmanuel Cecchet, Rahul Singh, Upendra Sharma, and Prashant Shenoy. Dolly: virtualization-driven database provisioning for the cloud. In *ACM SIGPLAN Notices*, volume 46, pages 51–62. ACM, 2011.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[CWO+11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, and Huseyin Simitci. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, ACM, 2011.

[CZB99] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: caching dynamic contents on the Web. *Distributed Systems Engineering*, 6(1):43–50, 1999.

[DAEA10] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174. ACM, 2010.

[DAEA13] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems*, 38(1):1–45, April 2013.

[Dat17] Google Cloud Datastore. `https://cloud.google.com/datastore/docs/concepts/overview`, 2017. (Accessed on 05/20/2017).

[DB13] Regine Dörbecker and Tilo Böhmann. The Concept and Effects of Service Modularity - A Literature Review. In *46th Hawaii International Conference on System Sciences, HICSS 2013, Wailea, HI, USA, January 7-10, 2013*, pages 1357–1366. IEEE Computer Society, 2013.

[Db417] db4o. `https://github.com/lytico/db4o`, 2017. (Accessed on 05/20/2017).

[DBS+12] Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh, Balaji Varadarajan, Sunil Nagaraj, David Zhang, Lei Gao, Jemiah Westerman, Phanindra Ganti, and others. All aboard the Databus!: Linkedin's scalable consistent change data capture platform. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 18. ACM, 2012.

[DDT+04] Anindya Datta, Kaushik Dutta, Helen M. Thomas, Debra E. VanderMeer, and Krithi Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: An approach and implementation. *ACM Trans. Database Syst.*, 29(2):403–443, 2004.

[Dea09] Jeff Dean. Designs, lessons and advice from building large distributed systems, 2009. Keynote talk at LADIS 2009.

[DEAA09] Sudipto Das, Amr El Abbadi, and Divyakant Agrawal. ElasTraS: An Elastic Transactional Data Store in the Cloud. *HotCloud*, 9:131–142, 2009.

[DeM09] Linda DeMichiel. JSR 317: Java Persistence 2.0. *Java Community Process, Tech. Rep*, 2009.

[Den96] Shuang Deng. Empirical model of WWW document arrivals at access link. In *Communications, 1996. ICC'96, Conference Record, Converging Technologies for Tomorrow's Applications. 1996 IEEE International Conference on*, volume 3, pages 1797–1802. IEEE, 1996.

[Dep17] Deployd: a toolkit for building realtime APIs. `https://github.com/deployd/deployd`, 2017. (Accessed on 05/20/2017).

[Dey15] Akon Samir Dey. Cherry Garcia: Transactions across Heterogeneous Data Stores. 2015.

[DFI+13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 international conference on Management of data*, pages 1243–1254. ACM, 2013.

[DFJ+96] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. Semantic Data Caching and Replacement. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 330–341. Morgan Kaufmann, 1996.

[DFKM97] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey C. Mogul. Rate of Change and other Metrics: a Live Study of the World Wide Web. In *1st USENIX Symposium on Internet Technologies and Systems, USITS'97, Monterey, California, USA, December 8-11, 1997*. USENIX, 1997.

[DFNR14] Anamika Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*, pages 223–230. IEEE, 2014.

[DFR15a] A. Dey, A. Fekete, and U. Röhm. Scalable distributed transactions across heterogeneous stores. In *2015 IEEE 31st International Conference on Data Engineering*, pages 125–136, April 2015.

[DFR15b] Akon Dey, Alan Fekete, and Uwe Rohm. REST+T: Scalable Transactions over HTTP. pages 36–41. IEEE, March 2015.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, San Francisco, CA, 2004. USENIX Association.

[DGMS85] Susan B Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys (CSUR)*, 17(3):341–370, 1985.

[DHJ+07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SOSP*, volume 14 of *17*, pages 205–220. ACM, 2007.

[DKM+11] Xavier Dutreilh, Sergey Kirgizov, Olga Melekhova, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, pages 67–74, 2011.

[DLNW13] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[DNAE11] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment*, 4(8):494–505, 2011.

[DNN+15] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 54–70. ACM, ACM Press, 2015.

[Dom18] Jörn Christopher Domnik. Integration der Baqend Backend-as-a-Service APIs in Android. Masterarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, 2018.

[Dow98] Troy Bryan Downing. *Java RMI: remote method invocation*. IDG Books Worldwide, Inc., 1998.

[DPS+94] Alan J. Demers, Karin Petersen, Mike Spreitzer, Doug Terry, Marvin Theimer, and Brent B. Welch. The Bayou Architecture: Support for Data Sharing Among Mobile Users. In *First Workshop on Mobile Computing Systems and Applications, WMCSA 1994, Santa Cruz, CA, USA, December 8-9, 1994*, pages 2–7. IEEE Computer Society, 1994.

[DPS13] Erik Dahlman, Stefan Parkvall, and Johan Skold. *4G: LTE/LTE-advanced for mobile broadband*. Academic press, 2013.

[DRM+10] Xavier Dutreilh, Nicolas Rivierre, Aurélien Moreau, Jacques Malenfant, and Isis Truck. From Data Center Resource Allocation to Control Theory and Back. In *IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5-10 July, 2010*, pages 410–417. IEEE Computer Society, 2010.

[DRSN98] Prasad Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. Caching Multidimensional Queries Using Chunks. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 259–270. ACM Press, 1998.

[DST03] Venkata Duvvuri, Prashant J. Shenoy, and Renu Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. *IEEE Trans. Knowl. Data Eng.*, 15(5):1266–1276, 2003.

[Dyn17] DynamoDB. `http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html`, 2017. (Accessed on 05/20/2017).

[EDAE11] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 301–312. ACM, 2011.

[EGLT76] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, 1976.

[Ela17] Elasticsearch: Open Source Distributed Real Time Search & Analytics. `http://www.elasticsearch.org/`, 2017. (Accessed on 05/26/2017).

[Emb17] Ember.js Framework. `https://www.emberjs.com/`, 2017. (Accessed on 05/26/2017).

[EPM+16] EDBT, Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amelie Marian, Letizia Tanca, Ioana Manolescu, Kostas Stefanidis, International Conference on Extending Database Technology, and EDBT. *Advances in database technology - EDBT 2016 19th International Conference on Extending Database Technology, Bordeaux, France, March 15-18, 2016: proceedings*. University of Konstanz, University Library, Konstanz, 2016. OCLC: 957156764.

[ERR11] Mohamed El-Refaey and Bhaskar Prasad Rimal. Grid, soa and cloud computing: On-demand computing models. *Computational and Data Grids: Principles, Applications and Design: Principles, Applications and Design*, page 45, 2011.

[ESW78] Robert S. Epstein, Michael Stonebraker, and Eugene Wong. Distributed Query Processing in a Relational Data Base System. In Eugene I. Lowenthal and Nell B. Dale, editors, *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 31 - June 2, 1978*, pages 169–180. ACM, 1978.

[Eve14] Tammy Everts. State of the Union for Ecommerce Page Speed & Web Performance (Winter 2013-14). `https://blog.radware.com/applicationdelivery/applicationaccelerationoptimization/2014/02/report-sotu-for-ecommerce-page-speed-web-performance-winter-2013-14/`, 2014. (Accessed on 05/26/2017).

[Eve16] Tammy Everts. *Time Is Money: The Business Value of Web Performance*. O'Reilly Media, 2016.

[EW16] Michael Egorov and MacLane Wilkison. ZeroDB white paper. *arXiv preprint arXiv:1602.07168*, 2016.

[EWS12] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.

[EWS13] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Multi-key transactions for keyvalue stores. *United Networks, LLC, Tech. Rep*, 5, 2013.

[FAK13] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 371–384. USENIX Association, 2013.

[FAKM14] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. pages 75–88. ACM Press, 2014.

[Far06] Dan Farber. Google's Marissa Mayer: Speed wins. `http://www.zdnet.com/article/googles-marissa-mayer-speed-wins/`, 2006. (Accessed on 05/26/2017).

[FC92] Michael J. Franklin and Michael J. Carey. Client-Server Caching Revisited. In M. Tamer Özsu, Umeshwar Dayal, and Patrick Valduriez, editors, *Distributed Object Management, Papers from the International Workshop on Distributed Object Management (IWDOM), Edmonton, Alberta, Canada, August 19-21, 1992*, pages 57–78. Morgan Kaufmann, 1992.

[FCAB00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM TON*, 8(3):281–293, 2000.

[FCD+99] Anja Feldmann, Ramón Cáceres, Fred Douglis, Gideon Glass, and Michael Rabinovich. Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments. In *Proceedings IEEE INFOCOM '99, The Conference on Computer Communications, Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies, The Future Is Now, New York, NY, USA, March 21-25, 1999*, pages 107–116. IEEE, 1999.

[FCL97] Michael J. Franklin, Michael J. Carey, and Miron Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Trans. Database Syst.*, 22(3):315–363, 1997.

[FFM04] Michael J. Freedman, Eric Freudenthal, and David Mazieres. Democratizing Content Publication with Coral. In *NSDI*, volume 4, pages 18–18, 2004.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol–HTTP/1.1, 1999. *URL http://www. rfc. net/rfc2616. html*, 1999.

[Fid87] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. 1987.

[Fie00] R. T Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, Citeseer, 2000.

[Fir16] Maximiliano Firtman. *High Performance Mobile Web: Best Practices for Optimizing Mobile Web Apps*. O'Reilly Media, 1 edition, 9 2016.

[Fit04] Brad Fitzpatrick. Distributed caching with Memcached. *Linux journal*, 2004(124):5, 2004.

[FK09] Daniela Florescu and Donald Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.

[FLO+05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[FLR+14] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud Computing Patterns - Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014.

[FLWC12] Wei Fang, ZhiHui Lu, Jie Wu, and ZhenYin Cao. RPPS: A novel resource prediction and provisioning scheme in cloud data center. In Louise E. Moser, Manish Parashar, and Patrick C. K. Hung, editors, *2012 IEEE Ninth International Conference on Services Computing, Honolulu, HI, USA, June 24-29, 2012*, pages 609–616. IEEE Computer Society, 2012.

[FMdA+13] Alessandro Gustavo Fior, Jorge Augusto Meira, Eduardo Cunha de Almeida, Ricardo Gonçalves Coelho, Marcos Didonet Del Fabro, and Yves Le Traon. Under pressure benchmark for DDBMS availability. *JIDM*, 4(3):266–278, 2013.

[For12] A Behrouz Forouzan. *Data communications & networking*. Tata McGraw-Hill Education, 2012.

[FR14] Roy Fielding and J Reschke. RFC 7234: Hypertext Transfer Protocol (HTTP/1.1): Caching. Technical report, IETF, 2014.

[Fre10] Michael J. Freedman. Experiences with CoralCDN: A Five-Year Operational View. In *NSDI*, pages 95–110, 2010.

[FRT92] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. Concurrency Control for High Contention Environments. *ACM Trans. Database Syst.*, 17(2):304–345, 1992.

[FSSF01] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. The Dos and Don'ts of Client Authentication on the Web. In *USENIX Security Symposium*, pages 251–268, 2001.

[FWGR14] Steffen Friedrich, Wolfram Wingerath, Felix Gessert, and Norbert Ritter. NoSQL OLTP Benchmarking: A Survey. In Erhard Plödereder, Lars Grunske, Eric Schneider, and Dominik Ull, editors, *44. Jahrestagung der Gesellschaft für*

Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland, volume 232 of *LNI*, pages 693–704. GI, 2014.

[FWR17] Steffen Friedrich, Wolfram Wingerath, and Norbert Ritter. Coordinated Omission in NoSQL Database Benchmarking. In Bernhard Mitschang, Norbert Ritter, Holger Schwarz, Meike Klettke, Andreas Thor, Oliver Kopp, and Matthias Wieland, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Workshopband*, volume P-266 of *LNI*, pages 215–225. GI, 2017.

[GAAU15] Younghwan Go, Nitin Agrawal, Akshat Aranya, and Cristian Ungureanu. Reliable, Consistent, and Efficient Data Sync for Mobile Apps. In Jiri Schindler and Erez Zadok, editors, *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 359–372. USENIX Association, 2015.

[Gal95] R.G. Gallager. *Discrete Stochastic Processes*. The Springer International Series in Engineering and Computer Science. Springer US, 1995.

[GBR14] Felix Gessert, Florian Bücklers, and Norbert Ritter. ORESTES: a Scalable Database-as-a-Service Architecture for Low Latency. In *CloudDB 2014, Data Engineering Workshops (ICDEW)*, pages 215–222. IEEE, 2014.

[GC89] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In Gregory R. Andrews, editor, *Proceedings of the Twelfth ACM Symposium on Operating System Principles, SOSP 1989, The Wigwam, Litchfield Park, Arizona, USA, December 3-6, 1989*, pages 202–210. ACM, 1989.

[GD11] Sanjay Ghemawat and Jeff Dean. LevelDB. `http://leveldb.org`, 2011.

[Gda17] Google Data APIs. `https://developers.google.com/gdata/`, 2017. (Accessed on 05/26/2017).

[Gel00] Erol Gelenbe. *System performance evaluation: methodologies and applications*. CRC press, 2000.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[GFW+14] Felix Gessert, Steffen Friedrich, Wolfram Wingerath, Michael Schaarschmidt, and Norbert Ritter. Towards a Scalable and Unified REST API for Cloud Data Stores. In Erhard Plödereder, Lars Grunske, Eric Schneider, and Dominik Ull, editors, *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, volume 232 of *LNI*, pages 723–734. GI, 2014.

[GGL03] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43, 2003.

[GGW10] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: predictive elastic resource scaling for cloud systems. In *Proceedings of the 6th International Conference on Network and Service Management, CNSM 2010, Niagara Falls, Canada, October 25-29, 2010*, pages 9–16. IEEE, 2010.

[GH02] Rachid Guerraoui and Corine Hari. On the consistency problem in mobile distributed computing. In *Proceedings of the 2002 Workshop on Principles of Mobile Computing, POMC 2002, October 30-31, 2002, Toulouse, France*, pages 51–57. ACM, 2002.

[GHa+96] Jim Gray, Pat Hell and, et al. The dangers of replication and a solution. *SIGMOD Rec.*, 25(2):173–182, June 1996.

[GHKO81] Jim Gray, Pete Homan, Henry F. Korth, and Ron Obermarck. A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System. In *Berkeley Workshop*, page 125, 1981.

[GHTC13] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):22, 2013.

[GJK+14] Ferro Daniel Gómez, Flavio Junqueira, Ivan Kelly, Benjamin Reed, and Maysam Yabandeh. Omid: Lock-free Transactional Support for Distributed Data Stores. In *ICDE*, 2014.

[GJP11] K. Gilly, C. Juiz, and R. Puigjaner. An up-to-date survey in web load balancing. *World Wide Web*, 14(2):105–131, 2011.

[GKA09] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Storage workload characterization and consolidation in virtualized environments. In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*. Citeseer, 2009.

[GL02] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[GL06] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[GLPT76] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In G. M. Nijssen, editor, *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, pages 365–394. North-Holland, 1976.

[GLS11] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. Analyzing consistency properties for fun and profit. In *ACM PODC*, pages 197–206. ACM, 2011.

[GMA+08] Charles Garrod, Amit Manjhi, Anastasia Ailamaki, Bruce Maggs, Todd Mowry, Christopher Olston, and Anthony Tomasic. Scalable query result caching for web applications. *Proceedings of the VLDB Endowment*, 1(1):550–561, 2008.

[GMU+12] Ajay Gulati, Arif Merchant, Mustafa Uysal, Pradeep Padala, and Peter J. Varman. Workload dependent IO scheduling for fairness and efficiency in shared storage systems. In *19th International Conference on High Performance Computing, HiPC 2012, Pune, India, December 18-22, 2012*, pages 1–10. IEEE Computer Society, 2012.

[GÖ10] Lukasz Golab and M. Tamer Özsu. *Data Stream Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[Goo17a] Google Cloud Computing, Hosting Services & APIs | Google Cloud Platform. `https://cloud.google.com/`, 2017. (Accessed on 05/20/2017).

[Goo17b] Google Cloud Prediction API. `https://cloud.google.com/prediction/docs/`, 2017. (Accessed on 06/18/2017).

[Gos05] John Gossmann. Introduction to Model/View/ViewModel pattern for building WPF apps. `https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/`, 08 2005. (Accessed on 05/26/2017).

[GPS16] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 169–184. USENIX Association, 2016.

[GR15a] Felix Gessert and Norbert Ritter. Polyglot Persistence. *Datenbank-Spektrum*, 15(3):229–233, November 2015.

[GR15b] Felix Gessert and Norbert Ritter. Skalierbare NoSQL- und Cloud-Datenbanken in Forschung und Praxis. In *Datenbanksysteme für Business, Technologie und Web (BTW 2015) - Workshopband, 2.-3. März 2015, Hamburg, Germany*, pages 271–274, 2015.

[GR16] Felix Gessert and Norbert Ritter. Scalable Data Management: NoSQL Data Stores in Research and Practice. In *32nd IEEE International Conference on Data Engineering, ICDE 2016*, 2016.

[Gra97] Jim Gray. Microsoft SQL Server. January 1997.

[GRA+14] Wojciech M. Golab, Muntasir Raihan Rahman, Alvin AuYoung, Kimberly Keeton, and Indranil Gupta. Client-Centric Benchmarking of Eventual Consis-

tency for Cloud Storage Systems. In *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*, pages 493–502. IEEE Computer Society, 2014.

[Gra17] GraphQL. `https://facebook.github.io/graphql/`, 2017. (Accessed on 05/25/2017).

[Gri13] Ilya Grigorik. *High performance browser networking*. O'Reilly Media, [S.l.], 2013.

[GS96] James Gwertzman and Margo I Seltzer. World Wide Web Cache Consistency. In *USENIX ATC*, pages 141–152, 1996.

[GSHA11] Ajay Gulati, Ganesha Shanmuganathan, Anne M. Holler, and Irfan Ahmad. Cloud Scale Resource Management: Challenges and Techniques. In Ion Stoica and John Wilkes, editors, *3rd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'11, Portland, OR, USA, June 14-15, 2011*. USENIX Association, 2011.

[GSLI11] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. Exploring alternative approaches to implement an elasticity policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 716–723. IEEE, 2011.

[GSW+12] Tian Guo, Upendra Sharma, Timothy Wood, Sambit Sahu, and Prashant J. Shenoy. Seagull: Intelligent Cloud Bursting for Enterprise Applications. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 361–366. USENIX Association, 2012.

[GSW+15] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. The Cache Sketch: Revisiting Expiration-based Caching in the Age of Cloud Data Management. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"*. GI, 2015.

[GSW+17] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Erik Witt, Eiko Yoneki, and Norbert Ritter. Quaestor: Query Web Caching for Database-as-a-Service Providers. *Proceedings of the VLDB Endowment*, 2017.

[GTS+02] D. Gourley, B. Totty, M. Sayer, A. Aggarwal, and S. Reddy. *HTTP: The Definitive Guide*. Definitive Guides. O'Reilly Media, 2002.

[GWFR16] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. NoSQL database systems: a survey and decision guidance. *Computer Science - Research and Development*, November 2016.

[GWR17] Felix Gessert, Wolfram Wingerath, and Norbert Ritter. Scalable Data Management: An In-Depth Tutorial on NoSQL Data Stores. In *BTW (Workshops)*, volume P-266 of *LNI*, pages 399–402. GI, 2017.

[HA90] Phillip W Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 302–309. IEEE, 1990.

[Ham07] James Hamilton. On designing and deploying internet-scale services. In *21st LISA*. USENIX Association, 2007.

[HAMS08] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992, 2008.

[Han87] Eric N. Hanson. A Performance Analysis of View Materialization Strategies. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 440–453. ACM Press, 1987.

[Här84] Theo Härder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2):111–120, 1984.

[Har14] Dickt Hardt. The OAuth 2.0 authorization framework (2012). *URL: https://tools. ietf. org/html/rfc6749. html*, 2014.

[Has17] Mazdak Hashemi. The Infrastructure Behind Twitter: Scale. `https://blog.twitter.com/2017/the-infrastructure-behind-twitter-scale`, 2017. (Accessed on 05/25/2017).

[HB09] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[Hba17] HBase. `http://hbase.apache.org/`, 2017. (Accessed on 05/25/2017).

[HBvR+13] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of Facebook photo caching. In *SOSP*, pages 167–181, 2013.

[HDF13] Kai Hwang, Jack Dongarra, and Geoffrey C Fox. *Distributed and cloud computing: from parallel processing to the internet of things*. Morgan Kaufmann, 2013.

[Hel07] Joesph Hellerstein. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, November 2007.

[Hev16] H2o Server. `https://h2o.examp1e.net/configure/http2_directives.html`, 2016. (Accessed on 05/26/2017).

[HGGG12] Rui Han, Li Guo, Moustafa Ghanem, and Yike Guo. Lightweight Resource Scaling for Cloud Applications. In *12th IEEE/ACM International Symposium*

on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012, pages 644–651. IEEE Computer Society, 2012.

[Hil16] Tony Hillerson. *Seven Mobile Apps in Seven Weeks: Native Apps, Multiple Platforms*. Pragmatic Bookshelf, 2016.

[HIM02] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing database as a service. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 29–38, 2002.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[HL08] R. T. Hurley and B. Y. Li. A Performance Investigation of Web Caching Architectures. In *Proceedings of the 2008 C3S2E Conference*, C3S2E '08, pages 205–213, Montreal, Quebec, Canada, 2008. ACM.

[HMC+12] Masum Z. Hasan, Edgar Magana, Alexander Clemm, Lew Tucker, and Sree Lakshmi D. Gudreddi. Integrated and autonomic cloud resource scaling. In Filip De Turck, Luciano Paschoal Gaspary, and Deep Medhi, editors, *2012 IEEE Network Operations and Management Symposium, NOMS 2012, Maui, HI, USA, April 16-20, 2012*, pages 1327–1334. IEEE, 2012.

[HN13] J. Huang and D. Nicol. Trust mechanisms for cloud computing. *Journal of Cloud Computing*, 2, 2013.

[Hoo17] GitHub - hoodiehq/hoodie: A backend for Offline First applications. `https://github.com/hoodiehq/hoodie`, 2017. (Accessed on 05/25/2017).

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[HS07] Marc Hadley and P Sandoz. JSR 311: Java api for RESTful web services. *Technical report, Java Community Process*, 2007.

[HS16] Stephan Hochhaus and Manuel Schoebel. *Meteor in action*. Manning Publ., 2016.

[HTV10] T. Haselmann, G. Thies, and G. Vossen. Looking into a REST-Based Universal API for Database-as-a-Service Systems. In *CEC*, pages 17–24, 2010.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[HW03] Gregor Hohpe and Bobby Woolf. Enterprise Integration Pattern. *Addison-Wesley Signature Series*, 2003.

[HYA+15] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of "big data" on cloud computing: Review and open research issues. *Inf. Syst.*, 47:98–115, 2015.

[IBM17] IBM Bluemix – Cloud-Infrastruktur, Plattformservices, Watson, & weitere PaaS-Lösungen. `https://www.ibm.com/cloud-computing/bluemix`, 2017. (Accessed on 05/20/2017).

[IBNW09] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A Classification of Object-Relational Impedance Mismatch. pages 36–43. IEEE, 2009.

[IC98] Arun Iyengar and Jim Challenger. Data Update Propagation: A Method for Determining How Changes to Underlying Data A ect Cached Objects on the Web. Technical report, Technical Report RC 21093 (94368), IBM Research Division, Yorktown Heights, NY, 1998.

[IET15] IETF. RFC 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2). 2015.

[IKLL12] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Comp. Syst.*, 28(1):155–162, 2012.

[Int17] Ecma International. ECMAScript 2017 Language Specification. 6 2017. (Accessed on 09/22/2017).

[JA07] Dean Jacobs and Stefan Aulbach. Ruminations on Multi-Tenant Databases. In Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus, editors, *Datenbanksysteme in Business, Technologie und Web (BTW 2007), 12. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Proceedings, 7.-9. März 2007, Aachen, Germany*, volume 103 of *LNI*, pages 514–521. GI, 2007.

[Jet] Jetty - Servlet Engine and Http Server. `https://www.eclipse.org/jetty/`. (Accessed on 04/30/2018).

[Joy17] Joyent | Triton. `https://www.joyent.com/`, 2017. (Accessed on 06/05/2017).

[JRY11] Flavio Junqueira, Benjamin Reed, and Maysam Yabandeh. Lock-free transactional support for large-scale storage systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W 2011), Hong Kong, China, June 27-30, 2011.*, pages 176–181. IEEE, 2011.

[Kam17] Poul-Henning Kamp. Varnish HTTP Cache. `https://varnish-cache.org/`, 2017. (Accessed on 04/30/2017).

[KB96] Arthur M. Keller and Julie Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *VLDB J.*, 5(1):35–47, 1996.

[KDM+14] S Kulkarni, M Demirbas, D Madeppa, A Bharadwaj, and M Leone. Logical physical clocks and consistent snapshots in globally distributed databases, 2014.

[KF11] Pawel Koperek and Wlodzimierz Funika. Dynamic Business Metrics-driven Resource Provisioning in Cloud Environments. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics - 9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011. Revised Selected Papers, Part II*, volume 7204 of *Lecture Notes in Computer Science*, pages 171–180. Springer, 2011.

[KFD00] Donald Kossmann, Michael J. Franklin, and Gerhard Drasch. Cache investment: integrating query optimization and distributed data placement. *ACM Trans. Database Syst.*, 25(4):517–558, 2000.

[KFPC16] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-End Integrity Protection for Web Applications. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 895–913. IEEE Computer Society, 2016.

[KHAK09] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: pay only when it matters. *Proceedings of the VLDB Endowment*, 2(1):253–264, 2009.

[KHR02] Dina Katabi, Mark Handley, and Charles E. Rohrs. Congestion control for high bandwidth-delay product networks. In Matthew Mathis, Peter Steenkiste, Hari Balakrishnan, and Vern Paxson, editors, *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 19-23, 2002, Pittsburgh, PA, USA*, pages 89–102. ACM, 2002.

[KJH15] Jens Köhler, Konrad Jünemann, and Hannes Hartenstein. Confidential database-as-a-service approaches: taxonomy and survey. *Journal of Cloud Computing*, 4(1):1, 2015.

[KK94] Alfons Kemper and Donald Kossmann. Dual-Buffering Strategies in Object Bases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 427–438. Morgan Kaufmann, 1994.

[KKN+08] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E.P.C. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance,

distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[KL11] Tim Kiefer and Wolfgang Lehner. Private Table Database Virtualization for DBaaS. In *IEEE 4th International Conference on Utility and Cloud Computing, UCC 2011, Melbourne, Australia, December 5-8, 2011*, pages 328–329. IEEE Computer Society, 2011.

[KLAR10] Heba Kurdi, Maozhen Li, and HS Al-Raweshidy. Taxonomy of Grid Systems. In *Handbook of research on P2P and grid systems for service-oriented computing: Models, Methodologies and Applications*, pages 20–43. IGI Global, 2010.

[Kle17] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 1 edition edition, January 2017.

[KLL+97] David R. Karger, Eric Lehmanl, Tom Leightonl, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[KLM97] Tom M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *1st USENIX Symposium on Internet Technologies and Systems, USITS'97, Monterey, California, USA, December 8-11, 1997*. USENIX, 1997.

[KM06] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. In *Algorithms–ESA 2006*, pages 456–467. Springer, 2006.

[KNO+02] Panos Kalnis, Wee Siong Ng, Beng Chin Ooi, Dimitris Papadias, and Kian-Lee Tan. An adaptive peer-to-peer network for distributed caching of OLAP results. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 25–36. ACM, 2002.

[Kos00] Donald Kossmann. The State of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[KP+88] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[KP01] Panos Kalnis and Dimitris Papadias. Proxy-server architectures for OLAP. In Sharad Mehrotra and Timos K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 367–378. ACM, 2001.

[KPF+13] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *EuroSys*, pages 113–126. ACM, 2013.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[KR01] B. Krishnamurthy and J. Rexford. Web Protocols and Practice, HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement. *Recherche*, 67:02, 2001.

[KR10] James F Kurose and Keith W Ross. *Computer networking: a top-down approach*, volume 5. Addison-Wesley Reading, 2010.

[Kre14] Jay Kreps. Questioning the Lambda Architecture. `https://www.oreilly.com/ideas/questioning-the-lambda-architecture`, 2014. (Accessed on 09/23/2018).

[Kri13] Raffi Krikorian. Timelines at Scale. `http://infoq.com/presentations/Twitter-Timeline-Scalability`, 2013. (Accessed on 04/30/2017).

[Kri15] Michael Krigsman. Research: 25 percent of web projects fail. `http://www.zdnet.com/article/research-25-percent-of-web-projects-fail/`, Dec 2015. (Accessed on 04/30/2017).

[KV14] Pradeeban Kathiravelu and Luís Veiga. An Adaptive Distributed Simulator for Cloud and MapReduce Algorithms and Architectures. In *Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2014, London, United Kingdom, December 8-11, 2014*, pages 79–88. IEEE Computer Society, 2014.

[KW97] Balachander Krishnamurthy and Craig E. Wills. Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web. In *1st USENIX Symposium on Internet Technologies and Systems, USITS'97, Monterey, California, USA, December 8-11, 1997*. USENIX, 1997.

[KW98] Balachander Krishnamurthy and Craig E. Wills. Piggyback Server Invalidation for Proxy Cache Coherency. *Computer Networks*, 30(1-7):185–193, 1998.

[KW99] Balachander Krishnamurthy and Craig E. Wills. Proxy Cache Coherency and Replacement - Towards a More Complete Picture. In *Proceedings of the 19th International Conference on Distributed Computing Systems, Austin, TX, USA, May 31 - June 4, 1999*, pages 332–339. IEEE Computer Society, 1999.

[KWQH16] In Kee Kim, Wei Wang, Yanjun Qi, and Marty Humphrey. Empirical Evaluation of Workload Forecasting Techniques for Predictive Cloud Resource Scaling. In *9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 1–10. IEEE Computer Society, 2016.

[Lac16] Kevin Lacker. Moving On. *Parse Blog*, January 2016. Accessed on 12/09/2017.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[Lam86a] Leslie Lamport. On interprocess communication. part I: basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[Lam86b] Leslie Lamport. On interprocess communication. part II: algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[Lan01] Douglas Laney. 3d Data Management: Controlling Data Volume, Velocity, and Variety. Technical report, META Group, February 2001.

[LBMAL14] Tania Lorido-Botran, Jose Miguel-Alonso, and JoseA. Lozano. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

[LC97] Chengjie Liu and Pei Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the 17th International Conference on Distributed Computing Systems, Baltimore, MD, USA, May 27-30, 1997*, pages 12–21. IEEE Computer Society, 1997.

[LC99] Dongwon Lee and Wesley W. Chu. Semantic Caching via Query Matching for Web Sources. In *Proceedings of the 1999 ACM CIKM International Conference on Information and Knowledge Management, Kansas City, Missouri, USA, November 2-6, 1999*, pages 77–85. ACM, 1999.

[LCSA99] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing Persistent Objects in Distributed Systems. In Rachid Guerraoui, editor, *ECOOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings*, volume 1628 of *Lecture Notes in Computer Science*, pages 230–257. Springer, 1999.

[Leb08] Scott Leberknight. Polyglot Persistence. `http://www.sleberknight.com/blog/sleberkn/entry/polyglot_persistence`, 2008. (Accessed on 04/30/2017).

[Lec09] Jens Lechtenbörger. Two-Phase Commit Protocol. In LING LIU and M.TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 3209–3213. Springer US, 2009.

[Len02] Maurizio Lenzerini. Data integration: A theoretical perspective. In Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis, editors, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246. ACM, 2002.

[LFKA11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[LFKA13] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 313–328, 2013.

[LGZ04] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. Mtcache: Transparent mid-tier database caching in SQL server. In Z. Meral Özsoyoglu and Stanley B. Zdonik, editors, *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 177–188. IEEE Computer Society, 2004.

[LGZ+13] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Eric Baldeschwieler, Scott Shenker, and Ion Stoica. Tachyon: Memory Throughput I/O for Cluster Computing Frameworks. *memory*, 18:1, 2013.

[Lin06] Greg Linden. Make Data Useful. `http://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt`, 2006. (Accessed on 04/30/2017).

[LKAP01] Thanasis Loukopoulos, Panos Kalnis, Ishfaq Ahmad, and Dimitris Papadias. Active caching of on-line-analytical-processing queries in WWW proxies. In Lionel M. Ni and Mateo Valero, editors, *Proceedings of the 2001 International Conference on Parallel Processing, ICPP 2002, 3-7 September 2001, Valencia, Spain*, pages 419–426. IEEE Computer Society, 2001.

[LKM+02] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 600–611. ACM, 2002.

[LL00] F Thomson Leighton and Daniel M Lewin. Global hosting system, August 22 2000. US Patent 6,108,703.

[LLC+14] Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 281–292. USENIX Association, 2014.

[LLS+15] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High Performance Transactions in Deuteronomy. In *CIDR 2015,*

Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings. www.cidrdb.org, 2015.

[LLXX09] Alexandros Labrinidis, Qiong Luo, Jie Xu, and Wenwei Xue. Caching and Materialization for Web Databases. *Foundations and Trends in Databases*, 2(3):169–266, 2009.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[LMLM16] Sarath Lakshman, Sriram Melkote, John Liang, and Ravi Mayuram. Nitro: A fast, scalable in-memory storage engine for nosql global secondary index. *PVLDB*, 9(13):1413–1424, 2016.

[LN01] Qiong Luo and Jeffrey F. Naughton. Form-Based Proxy Caching for Database-Backed Web Sites. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 191–200. Morgan Kaufmann, 2001.

[LPC+12] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 265–278. USENIX Association, 2012.

[LPK+15] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John K. Ousterhout. Implementing linearizability at large scale and low latency. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 71–86. ACM, 2015.

[LPS09] Mihai Letia, Nuno Preguiça, and Marc Shapiro. CRDTs: Consistency without concurrency control. *arXiv preprint arXiv:0907.0929*, 2009.

[LR00] Alexandros Labrinidis and Nick Roussopoulos. WebView Materialization. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 367–378. ACM, 2000.

[LR01a] Alexandros Labrinidis and Nick Roussopoulos. Adaptive WebView Materialization. In *WebDB*, pages 85–90, 2001.

[LR01b] Alexandros Labrinidis and Nick Roussopoulos. Update Propagation Strategies for Improving the Quality of Data on the Web. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao,

and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 391–400. Morgan Kaufmann, 2001.

[LS88a] Richard J Lipton and Jonathan S Sandberg. *PRAM: A scalable shared memory*. Princeton University, Department of Computer Science, 1988.

[LS88b] Barbara Liskov and Liuba Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 260–267. ACM, 1988.

[LS13] Wolfgang Lehner and Kai-Uwe Sattler. *Web-Scale Data Management for the Cloud*. Springer, New York, auflage: 2013 edition, April 2013.

[LSPK12] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. Towards Multi-tenant Performance SLOs. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 702–713. IEEE Computer Society, 2012.

[Luc14] Gregory Robert Luck. The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 107. `https://www.jcp.org/en/jsr/detail?id=107`, 2014. (Accessed on 04/30/2017).

[Luc17] Apache Lucene - Apache Solr. `http://lucene.apache.org/solr/`, 2017. (Accessed on 04/30/2017).

[LVA+15] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: measuring and understanding consistency at Facebook. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 295–310. ACM, 2015.

[LW84] Ming-Yee Lai and W. Kevin Wilkinson. Distributed Transaction Management in Jasmin. In Umeshwar Dayal, Gunter Schlageter, and Lim Huat Seng, editors, *Tenth International Conference on Very Large Data Bases, August 27-31, 1984, Singapore, Proceedings.*, pages 466–470. Morgan Kaufmann, 1984.

[Lwe10] Bernhard Lwenstein. *Benchmarking of Middleware Systems: Evaluating and Comparing the Performance and Scalability of XVSM (MozartSpaces), JavaSpaces (GigaSpaces XAP) and J2EE (JBoss AS)*. VDM Verlag, 2010.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mö17] Konstantin Möllers. A Graphical Metamodelling Framework for the Web Applied on Backend-as-a-Service Systems. Masterarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany,

7 2017.

[MAD+11] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11, 2011.

[Mai90] David Maier. Representing database programs as objects. In *Advances in database programming languages*, pages 377–386. ACM, 1990.

[Mal16] Ivano Malavolta. Beyond native apps: web technologies to the rescue!(keynote). In *Proceedings of the 1st International Workshop on Mobile Development*, pages 1–2. ACM, 2016.

[Mar14] Nathan Marz. History of Apache Storm and lessons learned. *Thoughts from the Red Planet*, 10 2014. Accessed: 2015-12-17.

[MB16] San Murugesan and Irena Bojanova. *Encyclopedia of Cloud Computing*. John Wiley & Sons, 2016.

[MBS11] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Enacting SLAs in clouds using rules. In *European Conference on Parallel Processing*, pages 455–466. Springer, 2011.

[MC+98] Evangelos P Markatos, Catherine E Chronaki, et al. A top-10 approach to prefetching on the web. In *Proceedings of INET*, volume 98, pages 276–290, 1998.

[McK16] Martin McKeay. Akamai's State of the Internet Report Q4 2016. Technical report, Akamai, 2016.

[McM17] Patrick McManus. Using Immutable Caching To Speed Up The Web. `https://hacks.mozilla.org/2017/01/using-immutable-caching-to-speed-up-the-web/`, 2017. (Accessed on 04/30/2017).

[MDFK97] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In Christophe Diot, Christian Huitema, Scott Shenker, and Martha Steenstrup, editors, *Proceedings of the ACM SIGCOMM 1997 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, September 14-18, 1997, Cannes, France.*, pages 181–194. ACM, 1997.

[Mee12] Patrick Meenan. Speed Index - WebPagetest Documentation. `https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index`, 2012. (Accessed on 07/16/2017).

[Mer14] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[MG09] Peter Mell and Tim Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[Mic16] Microsoft. TypeScript Language Specification. `https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md`, 01 2016. (Accessed on 10/06/2017).

[Mil68] Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277. ACM, 1968.

[Mit02] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604–612, 2002.

[MJM08] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machine for WANs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384. USENIX Association, 2008.

[MKC+12] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, et al. SFS: random write considered harmful in solid state drives. In *FAST*, 2012.

[MNP+13] Hatem A. Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-Latency Multi-Datacenter Databases using Replicated Commit. *PVLDB*, 6(9):661–672, 2013.

[Mog94] Jeffrey C. Mogul. Recovery in spritely NFS. *Computing Systems*, 7(2):201–262, 1994.

[Mon17] MongoDB. `https://www.mongodb.com/`, 2017. (Accessed on 06/18/2017).

[Mon18] FAQ: Concurrency — MongoDB Manual 3.0. `https://docs.mongodb.com/v3.0/faq/concurrency/`, 2018. (Accessed on 05/27/2018).

[MP14] M Mikowski and J Powell. Single Page Applications, 2014.

[MP17] Ryan Marcus and Olga Papaemmanouil. Releasing Cloud Databases from the Chains of Performance Prediction Models. In *CIDR*, 2017.

[MRSJ15] Gabor Madl, Ramani Routray, Yang Song, and Rakesh Jain. Account clustering in multi-tenant storage management environments. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 1698–1707. IEEE, 2015.

[MSL+11] Prince Mahajan, Srinath T. V. Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Michael Dahlin, and Michael Walfish. Depot: Cloud Storage with Minimal Trust. *ACM Trans. Comput. Syst.*, 29(4):12:1–12:38, 2011.

[MSMO97] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3):67–82, 1997.

[MTK+11] Kunal Mukerjee, Tomas Talius, Ajay Kalhan, Nigel Ellis, and Conor Cunningham. SQL Azure as a Self-Managing Database Service: Lessons Learned and

Challenges Ahead. *IEEE Data Eng. Bull.*, 34(4):61–70, 2011.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[MW15] Nathan Marz and James Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., 2015.

[Mye85] Brad A Myers. The importance of percent-done progress indicators for computer-human interfaces. In *ACM SIGCHI Bulletin*, volume 16, pages 11–17. ACM, 1985.

[MyS17] MySQL Documentation. `https://dev.mysql.com/doc/`, 2017. (Accessed on 09/15/2017).

[Nag04] S. V. Nagaraj. *Web caching and its applications*, volume 772. Springer, 2004.

[New15] Sam Newman. *Building microservices - designing fine-grained systems, 1st Edition*. O'Reilly, 2015.

[NFG+13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *NSDI*, pages 385–398. USENIX Association, 2013.

[NGMB16] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, 2016.

[Nie94] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[NMMA16] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. " O'Reilly Media, Inc.", 2016.

[Not10] Mark Nottingham. RFC 5861 - HTTP Cache-Control Extensions for Stale Content. 2010.

[NSWW16] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. Non-volatile storage. *Commun. ACM*, 59(1):56–63, 2016.

[Nuo17] NuoDB: Emergent Architecture. `http://go.nuodb.com/rs/nuodb/images/Greenbook_Final.pdf`, 2017. (Accessed on 04/30/2017).

[NWG+09] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131. IEEE Computer Society, 2009.

[NWO88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. Comput. Syst.*, 6(1):134–154, 1988.

[OAE+11] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMCloud. *Commun. ACM*, 54(7):121–130, 2011.

[Oak14] Scott Oaks. *Java Performance - The Definitive Guide: Getting the Most Out of Your Code*. O'Reilly, 2014.

[Oda17] OData - open data protocol. `http://www.odata.org/`, 2017. (Accessed on 06/05/2017).

[ÖDV92] M. Tamer Özsu, Umeshwar Dayal, and Patrick Valduriez. An Introduction to Distributed Object Management. In M. Tamer Özsu, Umeshwar Dayal, and Patrick Valduriez, editors, *Distributed Object Management, Papers from the International Workshop on Distributed Object Management (IWDOM), Edmonton, Alberta, Canada, August 19-21, 1992*, pages 1–24. Morgan Kaufmann, 1992.

[Off17] Office 365 for Business. `https://products.office.com/en-us/business/office`, 2017. (Accessed on 06/05/2017).

[OL88] Brian M. Oki and Barbara Liskov. Viewstamped replication: A general primary copy. In Danny Dolev, editor, *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988*, pages 8–17. ACM, 1988.

[ON16] Kazuho Oku and Mark Nottingham. Cache Digests for HTTP/2. `https://tools.ietf.org/html/draft-ietf-httpbis-cache-digest-01`, 2016. (Accessed on 06/05/2017).

[Onl17] Salesforce Online CRM. `https://www.salesforce.com/en`, 2017. (Accessed on 06/05/2017).

[OO13] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. *Draft of October*, 7, 2013.

[Ope17] Open API Initiative. `https://www.openapis.org/`, 2017. (Accessed on 07/28/2017).

[Ora17] Oracle Result Cache. `https://docs.oracle.com/database/121/TGDBA/tune_result_cache.htm#TGDBA616`, 2017. (Accessed on 06/05/2017).

[ÖV11] M.T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer, 2011.

[ÖVU98] M Tamer Özsu, Kaladhar Voruganti, and Ronald C Unrau. An Asynchronous Avoidance-Based Cache Consistency Algorithm for Client Caching DBMSs. In *VLDB*, volume 98, pages 440–451. Citeseer, 1998.

[Par17] Parse Server. `http://parseplatform.github.io/docs/parse-server/guide/`, 2017. (Accessed on 07/28/2017).

[PB03] Stefan Podlipnig and László Böszörményi. A survey of Web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, 2003.

[PB07] Al-Mukaddim Khan Pathan and Rajkumar Buyya. A taxonomy and survey of content delivery networks. *Grid Computing and Distributed Systems Laboratory, University of Melbourne, Technical Report*, page 4, 2007.

[PB08] Mukaddim Pathan and Rajkumar Buyya. A Taxonomy of CDNs. In Rajkumar Buyya, Mukaddim Pathan, and Athena Vakali, editors, *Content Delivery Networks*, volume 9 of *Lecture Notes Electrical Engineering*, pages 33–77. Springer Berlin Heidelberg, 2008.

[PD10] Daniel Peng and Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*, volume 10, pages 1–15, 2010.

[PF00] Meikel Pöss and Chris Floyd. New TPC benchmarks for decision support and web commerce. *SIGMOD Record*, 29(4):64–71, 2000.

[PG12] Dan RK Ports and Kevin Grittner. Serializable snapshot isolation in PostgreSQL. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, 2012.

[PH03] Sunil Patro and Y. Charlie Hu. Transparent Query Caching in Peer-to-Peer Overlay Networks. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, page 32. IEEE Computer Society, 2003.

[PH09] Sang-Min Park and Marty Humphrey. Self-Tuning Virtual Machines for Predictable eScience. In Franck Cappello, Cho-Li Wang, and Rajkumar Buyya, editors, *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid 2009, Shanghai, China, 18-21 May 2009*, pages 356–363. IEEE Computer Society, 2009.

[PHS+09] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009.

[Pla13] Hasso Plattner. *A course in in-memory data management*. Springer, 2013.

[PLZ+16] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. FairRide: near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, 2016.

[PM96] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve World Wide Web latency. *Computer Communication Review*, 26(3):22–36, 1996.

[PN09] Radu Prodan and Vlad Nae. Prediction-based real-time resource provisioning for massively multiplayer online games. *Future Generation Comp. Syst.*, 25(7):785–793, 2009.

[Pop14] Raluca Ada Popa. *Building practical systems that compute on encrypted data*. PhD thesis, Massachusetts Institute of Technology, 2014.

[Por09] Ely Porat. An Optimal Bloom Filter Replacement Based on Matrix Solving. In Anna E. Frid, Andrey Morozov, Andrey Rybalchenko, and Klaus W. Wagner, editors, *Computer Science - Theory and Applications, Fourth International Computer Science Symposium in Russia, CSR 2009, Novosibirsk, Russia, August 18-23, 2009. Proceedings*, volume 5675 of *Lecture Notes in Computer Science*, pages 263–273. Springer, 2009.

[Pos81] Jon Postel. Transmission control protocol. 1981.

[Pos17] PostgreSQL: Documentation: 9.6: High Availability, Load Balancing, and Replication. `https://www.postgresql.org/docs/9.6/static/high-availability.html`, 2017. (Accessed on 07/28/2017).

[PPR05] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal Bloom filter replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 823–829. SIAM, 2005.

[PPR+11] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In Jeffrey S. Chase and Amr El Abbadi, editors, *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, page 9. ACM, 2011.

[Pri08] Dan Pritchett. BASE: An Acid Alternative. *Queue*, 6(3):48–55, May 2008.

[PRZB11] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 2011. 00095.

[PSS09] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[PSV+14] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building Web Applications on Top of Encrypted Data Using Mylar. In Ratul Mahajan and Ion Stoica, editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 157–172. USENIX Association, 2014.

[PSZ+07] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors, *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pages 289–302. ACM, 2007.

[PV97] Dhiraj K. Pradhan and Nitin H. Vaidya. Roll-Forward and Rollback Recovery: Performance-Reliability Trade-Off. *IEEE Trans. Computers*, 46(3):372–378, 1997.

[PZ13] Raluca A. Popa and Nickolai Zeldovich. Multi-Key Searchable Encryption. *IACR Cryptology ePrint Archive*, 2013:508, 2013.

[QSD+13] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradar, and others. On brewing fresh espresso: LinkedIn's distributed data serving platform. In *Proceedings of the 2013 international conference on Management of data*, pages 1135–1146. ACM, 2013.

[Rah88] Erhard Rahm. Optimistische Synchronisationskonzepte in zentralisierten und verteilten Datenbanksystemen/Concepts for optimistic concurrency control in centralized and distributed database systems. *it-Information Technology*, 30(1):28–47, 1988.

[RAR13] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs: Services for a Changing World*. " O'Reilly Media, Inc.", 2013.

[Rea17] React - A JavaScript library for building user interfaces. `https://facebook.github.io/react/`, 2017. (Accessed on 05/26/2017).

[Ree08] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[Res17] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 (Draft). `https://tools.ietf.org/html/draft-ietf-tls-tls13-21`, 2017. (Accessed on 07/29/2017).

[RGA+12] Muntasir Raihan Rahman, Wojciech M. Golab, Alvin AuYoung, Kimberly Keeton, and Jay J. Wylie. Toward a Principled Framework for Benchmarking Consistency. *CoRR*, abs/1211.4290, 2012.

[Ria17] Riak. `http://basho.com/products/`, 2017. (Accessed on 05/25/2017).

[Rig17] RightScale Cloud Management. `http://www.rightscale.com/`, 2017. (Accessed on 06/05/2017).

[RL04] Lakshmish Ramaswamy and Ling Liu. An Expiration Age-Based Document Placement Scheme for Cooperative Web Caching. *IEEE Trans. Knowl. Data Eng.*, 16(5):585–600, 2004.

[RLZ06] Lakshmish Ramaswamy, Ling Liu, and Jianjun Zhang. Efficient Formation of Edge Cache Groups for Dynamic Content Delivery. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006), 4-7 July 2006, Lisboa, Portugal*, page 43. IEEE Computer Society, 2006.

[Rob16] Mike Roberts. Serverless Architectures. `https://martinfowler.com/articles/serverless.html`, 2016. (Accessed on 07/28/2017).

[Rom97] Steven Roman. *Introduction to coding and information theory*. Undergraduate texts in mathematics. Springer, 1997.

[RRND15] Pethuru Raj, Anupama Raman, Dhivya Nagaraj, and Siddhartha Duggirala. *High-Performance Big-Data Analytics - Computing Systems and Approaches*. Computer Communications and Networks. Springer, 2015.

[RRS+13] Ian Rae, Eric Rollins, Jeff Shute, Sukhdeep Sodhi, and Radek Vingralek. Online, asynchronous schema change in F1. *Proceedings of the VLDB Endowment*, 6(11):1045–1056, 2013.

[RS03] M. Rabinovich and O. Spatscheck. Web caching and replication. *SIGMOD Record*, 32(4):107, 2003.

[Rus03a] C Russell. Java data objects (jdo) specification jsr-12. *Sun Microsystems*, 2003.

[Rus03b] C. Russell. JSR 12: Java Data Objects (JDO) specification. *Sun Microsystems*, 2003.

[RXDK03] Michael Rabinovich, Zhen Xiao, Fred Douglis, and Charles R. Kalmanek. Moving Edge-Side Includes to the Real Edge - the Clients. In Steven D. Gribble, editor, *4th USENIX Symposium on Internet Technologies and Systems, USITS'03, Seattle, Washington, USA, March 26-28, 2003*. USENIX, 2003.

[Sak14] Sherif Sakr. Cloud-hosted databases: technologies, challenges and opportunities. *Cluster Computing*, 17(2):487–502, 2014.

[Sak17] Kunihiko Sakamoto. Time to First Meaningful Paint: a layout-based approach. `https://docs.google.com/document/d/1BR94tJdZLsin5poeet0XoTW60M0SjvOJQttKT-JK8HI/`, 2017. (Accessed on 07/16/2017).

[San17] Salvatore Sanfilippo. Redis. `http://redis.io/`, 2017. (Accessed on 07/16/2017).

[SB02] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*, volume 1. Prentice Hall Upper Saddle River, 2002.

[SBCD09] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[Sca17] Scalr: Enterprise-Grade Cloud Management Platform. `https://www.scalr.com/`, 2017. (Accessed on 06/05/2017).

[Sch96] Bruce Schneier. *Applied cryptography - protocols, algorithms, and source code in C, 2nd Edition*. Wiley, 1996.

[Sch16] Peter Schuller. Manhattan, our real-time, multi-tenant distributed database for Twitter scale. *Twitter Blog*, 2016.

[Sch17] Julian Schenkemeyer. Integration der Baqend-as-a-Service APIs in IOS. Masterarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, 12 2017.

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2012.

[SGDY16] Michael Schaarschmidt, Felix Gessert, Valentin Dalibard, and Eiko Yoneki. Learning Runtime Parameters in Computer Systems with Delayed Experience Injection. In *Deep Reinforcement Learning Workshop, NIPS 2016*, 2016.

[SGGS12] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 9. Addison-Wesley Reading, 2012.

[SGR15] Michael Schaarschmidt, Felix Gessert, and Norbert Ritter. Towards Automated Polyglot Persistence. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"*, 2015.

[Shi11] Rada Shirkova. Materialized Views. *Foundations and Trends® in Databases*, 4(4):295–405, 2011.

[SHKS15] Uta Störl, Thomas Hauf, Meike Klettke, and Stefanie Scherzinger. Schemaless NoSQL Data Stores - Object-NoSQL Mappers to the Rescue? In Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath, editors, *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, volume 241 of *LNI*, pages 579–599. GI, 2015.

[Sim08] Bojan Simic. The performance of web applications: Customers are won or lost in one second. *Aberdeen Group*, 2008.

[SKD17] Johannes Schildgen, Yannick Krück, and Stefan Deßloch. Transformations on Graph Databases for Polyglot Persistence with NotaQL. In Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*, volume P-265 of *LNI*, pages 83–102. GI, 2017.

[SKM08] Aameek Singh, Madhukar R. Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*, page 53. IEEE/ACM, 2008.

[Sky17] Skytap. `https://www.skytap.com/`, 2017. (Accessed on 06/05/2017).

[SL12] Sherif Sakr and Anna Liu. SLA-Based and Consumer-centric Dynamic Provisioning for Cloud Databases. In Rong Chang, editor, *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*, pages 360–367. IEEE Computer Society, 2012.

[Sla17] Slack. `https://slack.com/`, 2017. (Accessed on 06/05/2017).

[SLG+09] Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. Dynamic Resource Allocation for Database Servers Running on Virtual Storage. In Margo I. Seltzer and Richard Wheeler, editors, *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings*, pages 71–84. USENIX, 2009.

[SMA+07] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160, 2007.

[Sof17] SoftLayer | Cloud Servers, Storage, Big Data, & More IAAS Solutions. `http://www.softlayer.com/`, 2017. (Accessed on 06/05/2017).

[Spa17] Bruce Spang. Building a Fast and Reliable Purging System. `https://www.fastly.com/blog/building-fast-and-reliable-purging-system/`, 02 2017. (Accessed on 07/30/2017).

[SPAL11] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[SPBZ11] M. Shapiro, N. Pregui\cca, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.

[SRH97] Aman Singla, Umakishore Ramachandran, and Jessica K. Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *SPAA*, pages 211–220, 1997.

[SS83] Dale Skeen and Michael Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Trans. Software Eng.*, 9(3):219–228, 1983.

[SS94] Mukesh Singhal and Niranjan G Shivaratri. *Advanced concepts in operating systems*. McGraw-Hill, Inc., 1994.

[SSGW11] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In Jeffrey S.

Chase and Amr El Abbadi, editors, *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, page 5. ACM, 2011.

[SSMS15] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. Take me to your leader! Online Optimization of Distributed Storage Configurations. *PVLDB*, 8(12):1490–1501, 2015.

[SSS+08] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, volume 8, 2008.

[SSS15] S. Sippu and E. Soisalon-Soininen. *Transaction Processing: Management of the Logical Database and its Underlying Physical Structure*. Data-Centric Systems and Applications. Springer International Publishing, 2015.

[STR+15] Dharma Shukla, Shireesh Thota, Karthik Raman, et al. Schema-agnostic indexing with Azure DocumentDB. *PVLDB*, 8(12), 2015.

[SVS+13] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013. 00004.

[SW13] Michael Stonebraker and Ariel Weisberg. The voltdb main memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[SW14] Ivan Stojmenovic and Sheng Wen. The Fog Computing Paradigm: Scenarios and Security Issues. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, September 7-10, 2014.*, pages 1–8, 2014.

[Swa17] Docker Swarm. `https://www.docker.com/products/docker-swarm`, 2017. (Accessed on 05/20/2017).

[TAR99] Francisco J. Torres-Rojas, Mustaque Ahamad, and Michel Raynal. Timed Consistency for Shared Distributed Objects. In Brian A. Coan and Jennifer L. Welch, editors, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, PODC, '99Atlanta, Georgia, USA, May 3-6, 1999*, pages 163–172. ACM, 1999.

[TC03] Xueyan Tang and Samuel T. Chanson. Coordinated Management of Cascaded Caches for Efficient Content Distribution. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 37–48. IEEE Computer Society, 2003.

[TCB14] Adel Nadjaran Toosi, Rodrigo N Calheiros, and Rajkumar Buyya. Interconnected cloud computing environments: Challenges, taxonomy, and survey. *ACM Computing Surveys (CSUR)*, 47(1):7, 2014.

[TDW+12] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[Tec14] Akamai Technologies. 2014 Consumer Web Performance Expectations Survey. `https://www.akamai.com/us/en/multimedia/documents/content/akamai-performance-matters-key-consumer-insights-ebook.pdf`, 2014. (Accessed on 07/16/2017).

[Tes13] Claudio Tesoriero. *Getting Started with OrientDB*. Packt Publishing Ltd, 2013.

[TGPM17] Alexandre Torres, Renata Galante, Marcelo S Pimenta, and Alexandre Jonatan B Martins. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and Software Technology*, 82:1–18, 2017.

[The17] Django Web Framework. `https://www.djangoproject.com/`, 2017. (Accessed on 05/20/2017).

[Tho98] A. Thomasian. Concurrency control: methods, performance, and analysis. *ACM Computing Surveys (CSUR)*, 30(1):70–119, 1998. 00119.

[TJDB06] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, ICAC '06, pages 65–73, Washington, DC, USA, 2006. IEEE Computer Society.

[TM05] Francisco J. Torres-Rojas and Esteban Meneses. Convergence Through a Weak Consistency Model: Timed Causal Consistency. *CLEI Electron. J.*, 8(2), 2005.

[Tot09] Alexander Totok. *Modern Internet Services*. Alexander Totok, 2009.

[TPK+13] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 309–324. ACM, 2013.

[TRL12] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2012.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, Snowbird, Utah, USA, 2014. ACM.

[Tuk77] John W. Tukey. *Exploratory data analysis*. Addison-Wesley series in behavioral science : quantitative methods. Addison-Wesley, 1977.

[TV10] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[TvS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007.

[TW11] Andrew S. Tanenbaum and David Wetherall. *Computer networks, 5th Edition*. Pearson, 2011.

[Twe17] Kevin Twesten. Entwicklung einer deklarativen Offline-First Applikation unter dem Backend-as-a-Service Paradigma. Masterarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, 6 2017.

[TWJN01] Mark Tsimelzon, Bill Weihl, Larry Jacobs, and M Nottingham. ESI language specification 1.0. *Akamai Technologies, Inc. Cambridge, MA, USA, Oracle Corporation, Redwood City, CA, USA*, pages 1–0, 2001.

[Usa17] Usage Statistics of HTTP/2 for Websites, July 2017. `https://w3techs.com/technologies/details/ce-http2/all/all`, 2017. (Accessed on 07/29/2017).

[USC+08] Bhuvan Urgaonkar, Prashant J. Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier Internet applications. *TAAS*, 3(1):1:1–1:39, 2008.

[Use17] Apache Usergrid. `https://usergrid.apache.org/`, 2017. (Accessed on 07/16/2017).

[Vak06] Athena Vakali. *Web Data Management Practices: Emerging Techniques and Technologies: Emerging Techniques and Technologies*. IGI Global, 2006.

[VdV00] Aad W Van der Vaart. Asymptotic statistics (Cambridge series in statistical and probabilistic mathematics). 2000.

[Ver17] Versant Object-Oriented Database. `http://www.actian.com/products/operational-databases/versant/`, 2017. (Accessed on 06/05/2017).

[VGS+17] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz

Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1041–1052. ACM, 2017.

[VM14] Piet Van Mieghem. *Performance analysis of complex networks and systems*. Cambridge University Press, 2014.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[VPR07] Daniel A. Villela, Prashant Pradhan, and Dan Rubenstein. Provisioning servers in the application tier for e-commerce systems. *ACM Trans. Internet Techn.*, 7(1):7, 2007.

[Vue17] Vue.js. `https://vuejs.org/`, 2017. (Accessed on 05/26/2017).

[VV16] Paolo Viotti and Marko Vukolić. Consistency in Non-Transactional Distributed Storage Systems. *ACM Computing Surveys*, 49(1):1–34, June 2016.

[VW99] Paul Vixie and Duane Wessels. Hyper Text Caching Protocol (HTCP/0.0). Technical report, 1999.

[Wag17] Jeremy Wagner. *Web Performance in Action: Building Faster Web Pages*. Manning Publications, 2017.

[Wal14] Craig Walls. *Spring in Action: Covers Spring 4*. Manning Publications, 2014.

[Wan99] J. Wang. A survey of web caching schemes for the internet. *ACM SIGCOMM Computer Communication Review*, 29(5):36–46, 1999.

[Wan16] Mengyan Wang. Parse LiveQuery Protocol Specification. *GitHub: ParsePlatform/parse-server*, March 2016. Accessed on 12/14/2017.

[WAWB05] Adepele Williams, Martin Arlitt, Carey Williamson, and Ken Barker. Web workload characterization: Ten years later. In *Web content delivery*, pages 3–21. Springer, 2005.

[WB09] Craig D. Weissman and Steve Bobrowski. The design of the force.com multitenant internet application development platform. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 889–896. ACM, 2009.

[WBGsS13] Florian Wolf, Heiko Betz, Francis Gropengie\s ser, and Kai-Uwe Sattler. Hibernating in the Cloud-Implementation and Evaluation of Object-NoSQL-Mapping. In *BTW*, pages 327–341. Citeseer, 2013.

[WBP+13] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. *SPANStore*: cost-effective geo-replicated storage spanning multiple cloud services. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 292–308. ACM, 2013.

[WCB01] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In Keith Marzullo and M. Satyanarayanan, editors, *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*, pages 230–243. ACM, 2001.

[WDM01] Jörg Widmer, Robert Denda, and Martin Mauve. A survey on TCP-friendly congestion control. *IEEE network*, 15(3):28–37, 2001.

[Wes97] Duane Wessels. Application of internet cache protocol (ICP), version 2. 1997.

[Wes04] Duane Wessels. *Squid - the definitive guide: making the most of your internet*. O'Reilly, 2004.

[WF11] Patrick Wendell and Michael J. Freedman. Going viral: flash crowds in an open CDN. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 549–558. ACM, 2011.

[WFGR15] Wolfram Wingerath, Steffen Friedrich, Felix Gessert, and Norbert Ritter. Who Watches the Watchmen? On the Lack of Validation in NoSQL Benchmarking. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"*, 2015.

[WFZ+11] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 134–143. www.cidrdb.org, 2011.

[WGF+17] Wolfram Wingerath, Felix Gessert, Steffen Friedrich, Erik Witt, and Norbert Ritter. The Case For Change Notifications in Pull-Based Databases. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband, 2.-3. März 2017, Stuttgart, Germany*, 2017.

[WGFR16] Wolfram Wingerath, Felix Gessert, Steffen Friedrich, and Norbert Ritter. Real-time stream processing for Big Data. *it - Information Technology*, 58(4), January 2016.

[WGW+18] Wolfram Wingerath, Felix Gessert, Erik Witt, Steffen Friedrich, and Norbert Ritter. Real-Time Data Management for Big Data. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*. OpenProceedings.org, 2018.

[Whi15] Tom White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (4. ed., revised & updated)*. O'Reilly, 2015.

[Wie15] Lena Wiese. *Advanced data management: for SQL, NoSQL, cloud and distributed databases*. De Gruyter, Oldenbourg, Berlin ; Boston, 2015.

[Wil17] WildFly Homepage · WildFly. `http://wildfly.org/`, 2017. (Accessed on 05/20/2017).

[Wit16] Erik Witt. Distributed Cache-Aware Transactions for Polyglot Persistence. Masterarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, 8 2016.

[WJW15] Da Wang, Gauri Joshi, and Gregory Wornell. Using straggler replication to reduce latency in large-scale parallel computing. *ACM SIGMETRICS Performance Evaluation Review*, 43(3):7–11, 2015.

[WKW16] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. Speeding up web page loads with Shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 109–122, 2016.

[WM13] Zhe Wu and Harsha V. Madhyastha. Understanding the latency benefits of multi-cloud webservice deployments. *Computer Communication Review*, 43(2):13–20, 2013.

[WN90] W. Kevin Wilkinson and Marie-Anne Neimat. Maintaining Consistency of Client-Cached Data. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings.*, pages 122–133. Morgan Kaufmann, 1990.

[Wor94] Kurt Jeffery Worrell. Invalidation in Large Scale Network Object Caches. 1994.

[WP11] Erik Wilde and Cesare Pautasso. *REST: from research to practice*. Springer Science & Business Media, 2011.

[WPC12] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. CloudTPS: Scalable transactions for Web applications in the cloud. *Services Computing, IEEE Transactions on*, 5(4):525–539, 2012.

[WPR10] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in practice: Hypermedia and systems architecture*. " O'Reilly Media, Inc.", 2010.

[WV02] G. Weikum and G. Vossen. *Transactional information systems*. Series in Data Management Systems. Morgan Kaufmann Pub, 2002.

[XCZ+11] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. Hacigümü\cS. ActiveSLA: A profit-oriented admission control framework for database-as-a-service providers. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 15. ACM, 2011. 00019.

[XFJP14] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Characterizing Facebook's Memcached Workload. *IEEE Internet Computing*, 18(2):41–49, 2014.

[XRB12] Cheng-Zhong Xu, Jia Rao, and Xiangping Bu. URL: A unified reinforcement learning approach for autonomic cloud management. *J. Parallel Distrib. Comput.*, 72(2):95–105, 2012.

[XSL+15] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via modular concurrency control. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 279–294. ACM, 2015.

[XZF+07] Jing Xu, Ming Zhao, José A. B. Fortes, Robert Carpenter, and Mazin S. Yousif. On the Use of Fuzzy Modeling in Virtualized Data Center Management. In *Fourth International Conference on Autonomic Computing (ICAC'07), Jacksonville, Florida, USA, June 11-15, 2007*, page 25. IEEE Computer Society, 2007.

[YAD14] Yavuz Selim Yilmaz, Bahadir Ismail Aydin, and Murat Demirbas. Google cloud messaging (GCM): an evaluation. In *IEEE Global Communications Conference, GLOBECOM 2014, Austin, TX, USA, December 8-12, 2014*, pages 2807–2812. IEEE, 2014.

[YADL98] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of the 18th International Conference on Distributed Computing Systems, Amsterdam, The Netherlands, May 26-29, 1998*, pages 285–294. IEEE Computer Society, 1998.

[YADL99] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Volume Leases for Consistency in Large-Scale Systems. *IEEE Trans. Knowl. Data Eng.*, 11(4):563–576, 1999.

[YBDS08] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. IEEE, 2008.

[YG16] Shui Yu and Song Guo, editors. *Big Data Concepts, Theories, and Applications*. Springer, 1st ed. 2016 edition, 3 2016.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[ZCO+15] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, July 2015.

[ZCS07] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *Fourth International Conference on Autonomic Computing (ICAC'07), Jacksonville, Florida, USA, June 11-15, 2007*, page 27. IEEE Computer Society, 2007.

[ZNAE16] Victor Zakhary, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. DB-Risk: The Game of Global Database Placement. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2185–2188. ACM, 2016.

[ZS17] Albert Y. Zomaya and Sherif Sakr, editors. *Handbook of Big Data Technologies*. Springer, 2017.

[ZSLB14] Liang Zhao, Sherif Sakr, Anna Liu, and Athman Bouguettaya. *Cloud Data Management*. Springer, auflage: 2014 edition, 2014.

[ZSS+15] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 263–278. ACM, 2015.

List of Figures

List of Tables

Listings

Statutory Declaration / Eidesstattliche Erklärung

English: Statutory Declaration

I hereby declare, on oath, that I have written the present dissertation entitled

"*Low Latency for Cloud Data Management*"

by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from these sources are clearly marked as such.

This thesis was not submitted in the same or in a substantially similar version, not even partially, to any other authority to achieve an academic grading and was not published elsewhere.

I agree that a copy of this thesis may be made available in the Informatics Library of the University of Hamburg.

Felix Gessert Hamburg, September 24th, 2018

German: Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass ich die vorstehende Arbeit mit dem Titel

„*Low Latency for Cloud Data Management*“

selbständig und ohne fremde Hilfe angefertigt und mich anderer als der angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Felix Gessert Hamburg, 24. September 2018