Self-Management Concepts for Relational Database Systems

Dissertation

zur Erlangung des Doktorgrades
an der Fakultät für Mathematik, Informatik und Naturwissenschaften,
Fachbereich Informatik
der Universität Hamburg

vorgelegt von

Marc Holze

Hamburg, 2012

Erster Gutachter:	Prof. Dr.-Ing. Norbert Ritter
Zweiter Gutachter:	Prof. Dr.-Ing. Kai-Uwe Sattler

Tag der Disputation: 25.11.2011

Abstract

Self-managing (or autonomic) databases are intended to reduce the total cost of ownership for a DBS by automatically adapting to evolving workloads and environments. To reach this goal, commercial database management systems (DBMS) have recently been equipped with self-management functions, which for example support the database administrator (DBA) in identifying the appropriate indexes or in sizing the memory areas. However, existing techniques suffer from several problems: First, they are often implemented as off-line tools that have to be explicitly triggered by a DBA. Second, they strictly focus on automating one particular administration task, without considering possible side-effects on other components. Third, their execution causes additional overhead for the DBS. Fourth, they follow best-effort approaches, which cannot be controlled by high-level goals (e.g. response time, throughput).

This work presents an alternative solution to the problem of DBS self-management, which avoids the drawbacks of the existing self-management functions. Instead of extending a DBMS with a set of component-specific self-management functions, the developed solution is designed as one single self-management loop, which has a system-wide view on all configuration decisions. As long as the workload of the system does not change and the goals are met, this self-management loop only performs very light-weight monitoring operations on the workload, performance, and state information. For this purpose, several workload shift detection solutions are described and compared in this work. This work also comprises a workload classification solution, that groups similar workload events in order to further reduce the monitoring overhead. Furthermore, the workload information is analysed for cyclic patterns in order to predict upcoming workload shifts.

Only when the system-wide self-management solution detects a workload shift or a violation of the goals, it performs a heavy-weight reconfiguration analysis. Given the current workload and state of the DBS, this reconfiguration analysis has to derive a new set of configuration parameter values that meet the goals in the best possible way. For this purpose the system-wide self-management solution employs a system model, which quantitatively describes the behaviour of the DBS using mathematical models. As creating a complete quantitative description of existing DBMS in a system model is a complex task, a graphical modelling approach (using the SysML modelling language) that supports the evolutionary refinement of models is used. At runtime, the system model is evaluated by the self-management logic using multi-objective optimization techniques, where the goal values are represented as constraints. With this approach the system-wide reconfiguration analysis is performed in a single step, allowing the immediate judgement of the side-effects on other components.

Kurzfassung

Selbstverwaltende (oder autonome) Datenbanken sollen die Betriebskosten für Datenbanksysteme (DBS) reduzieren, indem sie sich automatisch an veränderliche Lastcharakteristiken und Umgebungsbedingungen anpassen. Um dieses Ziel zu erreichen haben die Hersteller kommerzieller Datenbankverwaltungssysteme (DBVS) begonnen, ihre Produkte mit Selbstverwaltungsfunktionen auszustatten, die den Datenbankadministrator beispielsweise bei der Bestimmung geeigneter Indizes oder bei der Festlegung der Größen verschiedener Hauptspeicherbereiche unterstützen. Doch die existierenden Ansätze leiden heute noch an zahlreichen Problemen: Zunächst sind diese oft als Offline-Werkzeuge konzipiert, die manuell durch den DBA gestartet werden müssen. Außerdem fokussieren sich ihre Analysen oft auf eine einzige administrative Aufgabe oder DBS-Komponente. Mögliche Seiteneffekte auf andere Komponenten werden nicht berücksichtigt. Auch der zusätzliche Aufwand, der bei kontinuierlicher Ausführung der Selbstverwaltungsfunktionen auf dem DBS entsteht, stellt ein Problem dar. Weiterhin kann keine der existierenden Selbstverwaltungsfunktionen Zielwerte für die Antwortzeit oder den Durchsatz berücksichtigen.

Diese Arbeit präsentiert eine alternative Lösung für die Selbstverwaltung von DBS, welche die Nachteile der existierenden Lösungen vermeidet. Anstatt das DBVS mit einer Vielzahl komponenten-spezifischer Selbstverwaltungsfunktionen auszustatten ist das entwickelte Rahmenwerk als eine einzige zentrale Selbstverwaltungslogik konzipiert, die über einen systemweiten Blick auf alle Konfigurationsentscheidungen verfügt. Solange sich die Lastcharakteristik des DBS nicht ändert und die Zielvorgaben eingehalten werden, führt diese Selbstverwaltungslogik nur sehr leichtgewichtige Überwachungsfunktionen aus. Im Rahmen der Arbeit werden zu diesem Zweck verschiedene Techniken zur Erkennung von Änderungen in der Lastcharakteristik eines DBS vorgestellt und miteinander verglichen. Weiterhin wird eine Technik für die Klassifikation von DBS-Anfragen beschrieben, die ähnliche Anfragen gruppiert und so den Überwachungsaufwand reduziert. Es wird außerdem gezeigt, wie zyklische Änderungen an der Lastcharakteristik erkannt und vorhergesagt werden können.

Nur wenn die systemweite Selbstverwaltungslogik eine Änderung der Lastcharakteristik oder eine Verletzung der Zielvorgaben erkennt führt diese eine schwergewichtige Rekonfigurationsanalyse durch. Ausgehend von der derzeitigen Nutzung des DBS und dessen aktuellem Zustand bestimmt diese Rekonfigurationsanalyse neue Werte für die DBS-Konfiguration, so dass dieses die Zielvorgaben so gut wie möglich erfüllt. Hierfür greift die systemweite Selbstverwaltungslogik auf ein Systemmodell zurück, welches das Verhalten des DBS in Abhängigkeit von dessen Konfigurationen mittels mathematischer Modelle quantitativ beschreibt. Die Erstellung

einer vollständigen quantitativen Beschreibung des DBS-Verhaltens ist jedoch eine komplexe Aufgabe. Daher wird in dieser Arbeit die Modellierungssprache (SysML) für die Definition der Systemmodelle eingesetzt, die auf Grund ihrer graphischen Darstellung eine evolutionäre Verfeinerung der Modelle erlaubt. Zur Laufzeit wird das Systemmodell von der systemweiten Selbstverwaltungslogik mittels mehrkriterieller Optimierungstechniken ausgewertet, wobei die Zielwerte als Randbedingungen definiert werden. Mit diesem Ansatz kann die Rekonfigurationsanalyse in einem einzigen Schritt durchgeführt werden, so dass mögliche Seiteneffekte einer Konfigurationsänderung unmittelbar berücksichtigt werden können.

Danksagung

Für die Unterstützung beim Verfassen dieser Arbeit danke ich:

meinem Doktorvater Professor Norbert Ritter für die wertvollen Diskussionen und Denkanstöße,

Professor Kai-Uwe Sattler für die Zweitbegutachtung,

Michael von Riegen für die großartige Zeit in F522 und die Unterstützung und notwendige Ablenkung,

allen VSIS-Mitarbeitern für die immer freundschaftliche Zusammenarbeit und

Ulrike Ranger für die vielen wichtigen Hinweise und das Aufrechterhalten meiner Motivation zur Anfertigung dieser Arbeit.

Contents

List of Figures

List of Tables

1 Introduction

At its core, a database system (DBS) is supposed to be a reliable, data independent storage system. With the advance of hardware and software development, the demands on these systems have increased significantly. For example, increasing data volumes have required support for sophisticated indexing techniques, which allow the adaptation of the physical design to the customer's needs. When later on Data Warehousing solutions were deployed, databases had to be enhanced by OLAP-specific features like multi-dimensional indexing and analytic operators. And even recently, databases have had to adapt to new evolving technologies and provide integrated support for e.g. multimedia content and XML data.

Database vendors have reacted to the changing requirements by adding new features to their systems with every release, using them as a selling pitch. As a consequence, today's database management systems (DBMS) have become overloaded with features. The result of this featurism is an increasingly complex architecture of commercial database systems. However, there has been considerably little attention on the development of administration interfaces and manageability of the new features and the overall database system. Today's enterprise-level DBMS like IBM DB2, Oracle or Microsoft SQL Server have hundreds of configuration parameters and many different physical design options, which have to be set-up carefully in each particular environment. For these reasons, the administration of database systems is a challenging task, which can only be performed by highly-skilled, scarce and thus expensive database administrators (DBAs). At the same time, there have been great improvements in disk capacity and processing power at drastically reduced prices. Hence, it has been noticed ([ACK+04], [DD06], [KLSW02]) that the total cost of ownership of database systems nowadays is no longer dominated by hardware costs. Instead, the costs for the required skilled application developers, DBAs and their training on new product versions are the driving factors.

As a way out of the increasing database maintenance costs, it is common sense in both database industry and research that the principles of autonomic computing need to be applied to database systems. The idea is to focus the research and development toward database systems that are able to maintain themselves to a large extent and to automatically adapt to changing usage patterns. In exceptional cases, where human interaction is required, they should provide concrete action recommendations to the DBA. However, currently only a few individual aspects of the DBS administration have been automated. Concepts for the coordination of the coordination of these individual fields of self-management function are missing. Furthermore, these functions typically do not consider the overall effects of their reconfiguration decision on the response time, availability or throughput of the DBS.

This work presents concepts and techniques for an integrated, system-wide view on DBS self-management. These concepts allow an optimization of the overall operation of the DBS according to high-level business goals. Section 1.1 first discusses the general characteristics of a self-managing DBS. Afterwards Section 1.2 introduces the principles of existing self-management technology in DBS and analyses their weaknesses. The contributions of this work are then outlined in Section 1.3. Section 1.4 describes the structure of the following chapters, which present the contributions of this work in detail.

1.1 DBS Self-Management

Currently there are several definitions in literature about the general characteristics that an autonomic systems should provide. This work agrees with [GC03] and considers the following properties to provide a good definition of the characteristics of autonomic systems:

Self-Configuration The notion of self-configuration describes the ability of a system to automatically adapt its configuration to changes in its hardware or software environment. All required configuration changes must not disrupt the operation of the system, i.e. changes of configuration parameters (*knobs*) have to be possible and take effect without shutting down the system. In addition, it is expected that a self-configuring system automatically recognizes its environment during installation and configures itself accordingly.

Self-Optimization In addition to the autonomic execution of configuration changes, which are enforced by changes in the system's environment, an autonomic system should also be able to internally optimize its performance. For this, the system has to continuously monitor its own performance and optimize it whenever necessary. In order to assess its current performance, the autonomic system should be able to consider high-level performance-goals defined by the end-user.

Self-Healing A system is self-healing if it is capable of recovering from failures. No matter whether the error was caused from internal or external reasons, the system must be able to detect it, analyse it and take appropriate recovery actions. For internal errors this means that the autonomic system must be able to identify malfunctioning system components, and either repair them or replace them with alternatives. Furthermore, advanced systems should also be able to predict upcoming errors from exceptional situations in individual components and trigger compensating actions, thus preventing a system-level failure. Again, all self-healing actions should be performed without or at least minimal disruption of the system operation.

Self-Protection Self-protection of autonomic systems refers to the ability to detect attacks from the outside world. Therefore, it is required to support authentication and authorization across all components and resources. Intrusions should be automatically detected and their

impacts minimized by encryption mechanisms. Of course, to avoid manual configuration overhead, the identity management of the autonomic system should seamlessly integrate with an overall enterprise IT landscape security management. A self-protecting system should therefore be able to import user identities and privileges from directory services, and to put high-level security policies into action.

When comparing existing database systems with the properties of autonomic systems, it becomes obvious that certain self-management aspects have been implemented for a long time. For example, the existing recovery algorithms based on logging and backup are an important self-healing feature. Crashes or media failures are automatically detected and the last transaction-consistent state is restored. DBS therefore already meet the self-healing requirements to a large extent. Furthermore, existing DBS typically allow the definition of fine-grained read/write privileges on the database objects. The rights for administrative changes to the DBS can also be restricted, and the user identification can be integrated with the operating system. Hence, most of the self-protection requirements are already met by DBS, too. In contrast, very little attention has been paid to self-configuration and self-optimization of the DBS in the past. Almost all performance-related configurations have to be set-up and adapted by the DBA. Even though the query optimization of DBS can be considered a self-optimization feature, the required optimizer statistics and the adequate optimizer level have to be configured manually, for example. Likewise, although the DBS automatically selects the most appropriate access paths for retrieving data, the DBA has to analyse the workload and identify the most appropriate indexes.

As self-protection and self-healing are already very well supported by existing DBS, the recent development of autonomic DBS technology has concentrated on self-configuration and self-optimization. Especially self-optimization of DBS is considered an important research area, because it promises higher DBS performance at lower resource and administration costs. For this reason most of the autonomic DBS functionality that has been developed recently is directed at this area. The self-management concepts presented in this work are also directed at the subject of self-optimization.

1.2 State of the Art

Over the past years database vendors have reacted to the high operation and maintenance costs by integrating autonomic functions into their products. When comparing the existing autonomic features, two general approaches can be distinguished:

Advisors Advisors are intended as administrative tools to provide the DBA with recommendations for performance improvements. Examples for advisors in commercial database systems are IBM DB2 Design Advisor [ZRL+04], Microsoft SQL Server Database Tuning Advisor [ACK+04]

and Oracle SQL Tuning Advisor [DDD+04]. The input to the advisors is the current workload of the database and a set of constraints, e.g. the maximum space for indexes and the maximum computation time. Using expert knowledge in their heavy-weight analysis algorithms, the advisors determine the set of indexes and materialized views that would provide the lowest statement execution cost from this input. Another example for an advisor is a configuration wizard [KLSW02] which helps the DBA to implement at setup-time for the configuration of the DBS at set-up time has been developed ([KLSW02]). The configuration advisor presents a sequence of questions about the future usage to the DBA and derives the appropriate setting of configuration parameters from best practice information.

From the perspective of a DBA, the advisors may be helpful tools in order to reduce the time required for DBS administration. Nevertheless, the advisors do not actually meet the requirements of autonomic computing. They do not automatically report the need for configuration changes, but the DBA has to trigger their execution when he suspects the possibility for a significant performance improvement. So the DBA still has to monitor the managed DBS carefully in order to detect the appropriate points in time for a reconfiguration analysis. In addition, the advisors are not able to consider high-level performance goals in their analysis but can only be controlled by constraints like the maximum disk space that may be occupied by additional indexes.

Feedback Control Loops The concept of feedback control loops origins from control theory [DHP+05]. It describes a design template for autonomic features, which is illustrated in Figure 1.1. In a feedback control loop, a *controller* continuously monitors the performance of a specific database component (the *managed resource*) via *sensors*. The sensors provide performance information about the managed resource according to predefined metrics. When this information indicates the need for re-tuning, the controller autonomically plans and executes actions to adapt the managed resource to the new situation by using the resource's *effectors*. The effectors resemble the administrative interface of the managed DBS component, e.g. its configuration parameters or maintenance functions. Whenever the controller changes the configuration of the managed resource via the effectors, it directly monitors the effects on the performance of the managed resource via the sensors again. This feedback on the effects of effector changes is considered in the following reconfiguration decision. Thus, the configuration is adapted in a loop with several small reconfiguration steps until a desired value for the performance of the managed resource has been reached.

The application of the feedback control loop pattern to database systems suffers from several problems: First, the latency of the effects of configuration changes can be large. If for instance a maintenance function is executed to improve the state of the component, it may take several hours before the positive effects become apparent. Not considering this feedback delay in the controller may cause *overreactions*. Second, the tuning logic in the control loops is highly specialized to a certain domain. Still its tuning decisions may have side-effects on other system components that are managed by other control loops. But currently there is no concept to

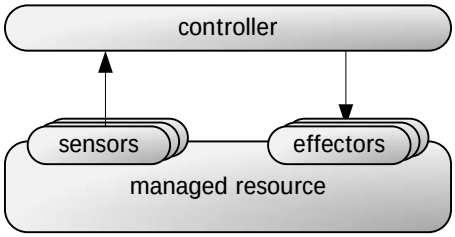


Figure 1.1: Feedback Control Loop Design Pattern

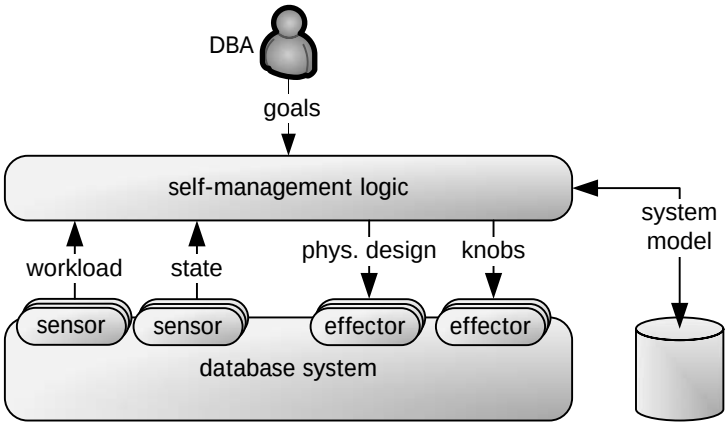


Figure 1.2: System-wide DBS Self-Management

manage the *interaction* of control loops. Third, all feedback control loops operate constantly, even if there is no need for a reconfiguration. With an increasing number of autonomic functions in a DBS, the required analysis effort may aggregate to a significant *overhead*. Fourth, to comply with the principles of autonomic computing [GC03], the reconfiguration actions in autonomic databases should be driven by the necessity to meet high-level business goals. But the feedback control loops have a limited view on the behaviour of the managed resource only. They therefore cannot take the effects of their reconfiguration decisions on the overall system performance into account (*goal-independency*).

1.3 Contributions

This work describes a novel approach towards DBS self-management, which avoids the problems of overreaction, interaction, overhead and goal-independency. An overview of the approach is given in Figure 1.2. Instead of running a set of independent feedback control loops, the approach is directed at maintaining a system-wide view on DBS self-management. It is based on a centralized *external DBS self-management logic*, which controls all reconfiguration decisions in the DBS.

There are two main factors that have to be monitored by the system-wide self-management logic: The *workload* provides information on how a DBS is used in its particular environment (e.g. in terms of CPU usage, the SQL trace, or page requests). It has major influence on many configuration decisions in a DBS, as e.g. the access paths must be selected so that they optimally

support the typical query structure, or the bufferpool sizes must be chosen according to it. The workload must furthermore be monitored continuously, because changes in the workload may lead to goal violations. Besides the workload, the self-management logic also has to consider other DBS-internal characteristics, like the average response time, the physical fragmentation, or the accuracy of optimizer statistics (*state*). The *self-management logic* must compare the current state of the DBS to the *goals* (e.g. response times, throughput, CPU or disk space usage, availability, operation cost) defined by a DBA, and start a reconfiguration analysis when there is a risk of missing the goals.

For the decision on which reconfigurations will meet the goals under the current workload and state, the self-management logic needs detailed knowledge about the DBS. This work refers to this knowledge as the *system model*. The system model contains quantitative information about the behaviour and performance of the DBS under specific configurations and usage scenarios. From the system model the self-management can therefore decide how the computing resources should be shared amongst the DBMS components and assess configuration alternatives (access paths, number of bufferpools, tablespace design, ...) in a specific environment. It is the task of a self-management logic to evaluate the knowledge stored in the system model and to decide which reconfigurations are necessary to meet the goals for the current workload and state. However, there will typically not only be a single set of goals in an enterprise, but different goal values for every application or user. The requests processed by a DBS are therefore assigned to service classes (e.g. by the DB2 Workload Manager [CCI+08]), where each service class may be subject of a separate set of goals. So the self-management logic of a DBS should also be able to consider different goal values for service classes.

With the system-wide view on DBS self-management the *interaction* of tuning actions can be avoided, because the self-management logic acts as a central point of analysis. So in contrast to the approach of independent feedback control loops, it can predict the effects of a reconfiguration on *all* components of the DBS. Undesired side-effects therefore can already be determined during the reconfiguration analysis processing stage. Furthermore, the system-wide view on DBS self-management also provides a solution for the problem of *goal-independency*: On the one hand the underlying system model enables the self-management logic to predict how well the high-level goals will be met under a certain configuration and under a certain system workload and state. On the other hand, all information about the current system workload and state of the entire DBS is available to the self-management logic, because there is no separation of duties as in independent feedback control loops. Thus, the self-management logic has all the required information to select the most appropriate action (or actions) to meet the goals and optimize the resource usage. Moreover, the quantitative information in the system model can be used to quantitatively predict the effects of a particular reconfiguration action. In contrast, the feedback control loops rely on the feedback of reconfigurations to adjust the managed resource to certain goal value in several small steps. Given that the system model is accurate, the solution therefore avoids *overeactions* even when the delay between the reconfiguration and the observable effects is long.

Finally, the centralized self-management approach can also reduce the *overhead* induced by self-management: With the set of feedback control loops in the DBS, all controller components simultaneously monitor and analyse the performance metrics provided by their managed resource, respectively. But in a typical enterprise scenario the usage of a DBS is almost constant over time, because the SQL queries are determined by the set of applications that access the database. These applications usually comprise a number of hard-coded SQL statements or statement templates and expose typical usage patterns. As long as the usage of a DBS does not change significantly, the usage of each of its components will most probably be constant, too. Thus, most of the analysis overhead for possible reconfiguration actions is dispensable while the workload of the overall DBS does not change. With the centralized self-management this unnecessary analysis overhead can be avoided by restricting the reconfiguration analysis to situations where either the workload changes significantly, or the state of a DBS component demands a maintenance operation, or there is a risk of missing user-defined goals.

The contributions of this work to the area of DBS self-management can be summarized as follows:

- *System-wide Self-Management:* The work describes novel concepts for DBS self-management based upon a central self-management logic with a system-wide view on all reconfiguration decisions.

- *Lightweight Workload Analysis:* The work develops a novel analysis framework for the workload of relational DBS. This framework allows the lightweight monitoring of the workload for significant changes. It therefore allows the quick adaptation of the DBS configuration to workload changes, while it restricts the analysis overhead for stable workloads to a minimum.

- *System Model Definition and Evaluation Framework:* The work proposes a framework (including tools and methods) for the development of systems models for existing DBS. It identifies the required information in the model and selects an appropriate modelling language (SysML) and environment. Furthermore, multi-objective optimization is identified as an appropriate method for the automatic deduction of optimal DBS configurations from the system model.

- *Coarse-grained DB2 System Model:* To illustrate the applicability of the self-management concepts, the work includes the definition of a coarse-grained system model for the IBM DB2. The DB2 model quantitatively estimates the response time of the DBS depending on its configuration and workload.

All contributions of this work are especially designed to be applied to existing DBMS. Consequently, the self-management logic is based on the existing administrative interface of DBMS, i.e. it does not require the re-implementation of DBMS components and does not have to be

integrated into the DBMS core. Likewise, the workload analysis is designed to operate on the standardized SQL interface of relational database systems.

Considering the system-wide self-management approach illustrated in Figure 1.2, there are also aspects which are not the subject of this work: First, the monitoring of the DBS-internal state information, i.e. of metrics like physical fragmentation or quality of optimizer statistics which may downgrade system performance without a change in the workload, is not examined. Second, a reliable guarantee for meeting the user-defined goals like response-time or throughput would require a sophisticated monitoring of the key performance indicators. Possible goal violations would have to be identified as early as possible, e.g. by performing a trend analysis. However, this work considers the goal value as a simple threshold, and triggers a reconfiguration analysis only when the threshold is exceeded. Third, only a coarse-grained system model for the IBM DB2 is developed. This model serves as a proof of concept, and therefore considers the most important stages of DBS statement processing only. Furthermore, it only comprises predictions of the response time, and it does not include the physical design alternatives. A more fine-grained DB2 model and a general evaluation of the most adequate granularity for system models will require additional extensive experimental evaluations and must be examined in future works.

1.4 Structure of Work

The following chapters describe the concepts and techniques developed for system-wide DBS self-management in detail. Chapter 2 first presents the existing approaches towards DBS self-management in detail. For this purpose it first introduces the general concepts of autonomic computing. Afterwards, it selects a set of representative off-line and on-line autonomic functions from commercial DBMS and describes their mode of operation. By comparing the functionality of the existing approaches to the goals of autonomic computing, the open challenges for a truly autonomic DBS are identified.

Chapter 3 surveys the possible solutions to the open challenges of autonomic DBS. From the alternatives, it identifies the centralized self-management logic with a system-wide view on the self-management decisions (as shown in Figure 1.2) as the most adequate approach. Afterwards, the requirements and basic design principles for the workload analysis, system models and self-management logic are introduced. Chapter 3 therefore serves as a solution overview, whereas details of the solution's conceptual and technical challenges are discussed in the following chapters 4, 5, and 6.

In order to support a lightweight continuous adaptation of the DBS to its current workload, this work follows the approach of a lightweight detection of significant shifts in the workload. The concepts developed for this workload shift detection are presented in Chapter 4. First, an appropriate processing model comprising the stages monitoring, preprocessing, classification and analysis is developed. For each of these stages the realization alternatives, algorithms and

related work are discussed in detail. Furthermore, an overview of the evaluation results is given.

Chapter 5 presents the concepts for the definition of system models. After discussing the necessary contents and usage requirements of the model, the graphical language SysML is identified as the appropriate modelling technique. Using this language, the chapter describes the exemplary coarse-grained system model for the IBM DB2. The required quantitative descriptions of the DB2 behaviour in the model are derived from experimental evaluations, whose results are also given in this chapter.

The evaluation of the system models in the self-management logic is the subject of Chapter 6. It describes how the required information can be extracted from the system model, and how goal functions can be constructed from it. Afterwards, the automatic deduction of appropriate DBS configurations using multi-objective optimization is described. An experimental evaluation finally illustrates the application of the self-management logic to the exemplary DB2 system model.

Chapter 7 concludes the work with a summary of the results and an outlook on future work.

2 DBS Self-Management

As discussed in Section 1.2, the problem of high maintenance costs has been recognized by both DBMS vendors and researchers. In order to decrease these costs, they follow the principles of autonomic computing, i.e. they develop self-management functions for DBS. The following sections summarize the mode of operation of important existing DBS self-management functions, where the focus is on self-management functions that are already available in commercial DBS. However, a complete survey of autonomic functions in today's DBS is not in the focus of this work (see [MRHA09], [MRHA08], [EPBM03], [WMHZ02] instead, for example). In order to provide an outlook on self-management functions that are likely to be integrated into commercial DBMS in the future, some promising research approaches are presented in this section, too. A comprehensive discussion of other current academic approaches is given in the related work sections of the chapters that describe the contributions of this work (Sections 4.7, 5.5, and 6.6).

In the following, Section 2.1 first introduces the basic characteristics of autonomic systems and architectural guidelines for building them. The off-line and on-line self-management functions of autonomic DBS are then presented in Sections 2.2 and 2.3. Section 2.4 compares the existing self-management functions to the goals of autonomic computing and thus identifies the open challenges.

2.1 Autonomic Computing

The research area of autonomic computing has been established in 2001 with the *Autonomic Computing Manifesto* [Hor01] published by IBM. This manifesto highlights the problem of the increasing complexity of IT infrastructures. It argues that the trend for increasing interconnection and integration between information systems has lead to IT infrastructures which are increasingly difficult to operate and maintain. As a consequence, many highly skilled and expensive administrators are required. The probability of administrative mistakes increases with the complexity of the IT infrastructure, too. [Hor01] argues that due to the IT complexity, the IT infrastructures may cause enormous costs and become un-manageable in the future. As a solution to this problem, IBM proposes the integration of self-management logic into the IT infrastructure. Like the *autonomic nervous system* of the human body, this self-management logic is intended to adjust the system to varying environments. Thus, the goal of autonomic computing is not to reduce the complexity of the IT systems, but to reduce the *perceived* complexity for administration and operation.

Autonomic computing in [Hor01] is labelled a *holistic vision*, which does not only refer to the local self-management of individual systems, but to the self-management of entire computer networks. In order to control the complex self-management decisions in the network, the interface between the human operator and the self-managing infrastructure is supposed to be as simple as possible. Hence, the autonomic computing manifesto demands the usage of high-level business policies (goals) for this purpose. In addition, it defines the characteristics of an autonomic system, which (besides others) comprise the self-configuration, self-optimization, self-healing, and self-protection (see Section 1.1) . These four properties have been identified as the most relevant properties of autonomic systems in later publications (e.g. [KC03], [GC03]), too, and are commonly referred to as the *self-** properties.

IBM has also published a set of guidelines and concepts for the realization of autonomic systems in the architectural blueprint [IBM05]. This blueprint organizes an autonomic system into several layers as illustrated in Figure 2.1. The lowest layer comprises the *managed resources* in the IT infrastructure that are controlled by the self-management logic, e.g. servers, databases, applications and services. Each of these resources may contain local self-management functionality (illustrated as a circled arrow in Figure 2.1), which may or may not be visible externally. On the second layer *touchpoints* provide a standardized interface to the manageability of the underlying managed resources. Thus, the resource-specific sensors and effectors (e.g. log files, APIs, commands, ...) can be accessed in a standardized way. Based upon these standardized interfaces, *touchpoint autonomic managers* perform the self-management for a single resource or a group of resources. They are responsible for the implementation of the self-* properties for the managed resources they control. Their decisions are controlled by goals or policies. In addition, touchpoint autonomic managers again expose a standardized touchpoint, which can be used to control their behaviour. These touchpoints are used by the *orchestrating autonomic managers* on the next higher layer in order to realize system-wide self-management. As an example for an orchestrating manager [IBM05] identifies a workload manager, which has to adapt all computing resources of the IT infrastructure to the current workload. The orchestrating managers again are controlled by the high-level goals defined by the IT infrastructure operator using *manual managers*.

In order to perform their self-management tasks, the autonomic managers require detailed knowledge about the system they manage. For example, they may require historical information about system events in order to identify the source for a system error, or rules which allow the prediction of the effects of configuration changes. Hence, the architectural blueprint [IBM05] designs *knowledge sources* that are available across all layers of the autonomic system. These knowledge sources are intended to hold three types knowledge: solution topology knowledge (describing the structure of the system), policy knowledge (goals and constraints that have to be met), and problem determination knowledge (historical information and reasoning rules).

Like [KC03], the blueprint [IBM05] proposes a four-staged processing for the implementation of autonomic managers, which is shown in Figure 2.2. In the first stage *monitor*, the autonomic manager collects information about the current state of the managed resources (its configu-

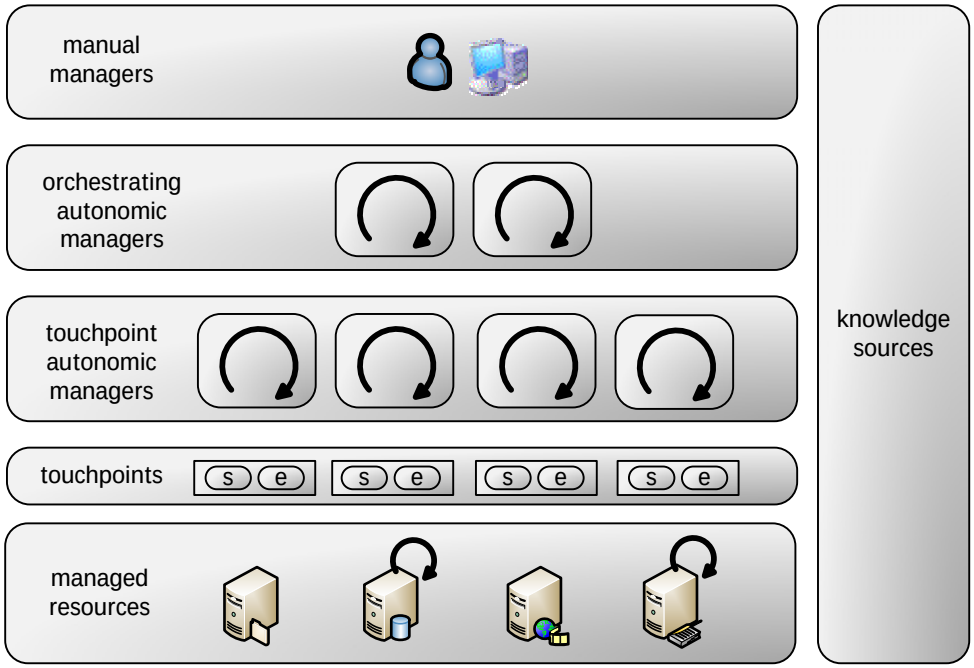


Figure 2.1: Autonomic Computing Architecture Blueprint [IBM05]

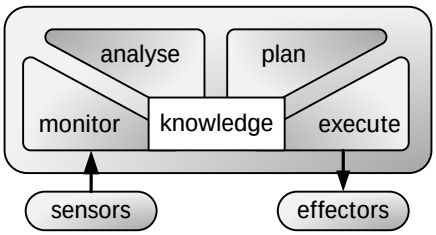


Figure 2.2: MAPE Loop [KC03]

ration, offered capacity, performance metrics and throughput) via touchpoints and correlates this information. The following stage *analyze* observes the sensor information and determines whether or not a change to the managed resources configuration should be made, e.g. because one of the goals is currently missed. In this case it passes the desired configuration change to the subsequent processing stage *plan*. This stage creates an execution plan for implementing the required configuration change in the managed resource, e.g. a simple command or a multi-step workflow. The scheduling of the configuration change plan and its actual execution are performed in the final stage *execute* of the autonomic function.

The autonomic computing concepts described above constitute the foundation of the development of self-managing DBS technology. Although the autonomic computing concepts are very general, the four stages of the autonomic function implementation, for example, have served as a blueprint for many autonomic functions in DBS. Likewise, the hierarchical structure of autonomic functions has often been referred to as a possible solution to undesired interaction problems between multiple independent DBS self-management functions.

2.2 DBS Off-line Self-Management Tools

Choosing the best set of indexes is a difficult problem for a DBA, because in-depth knowledge about the workload of the DBS is required for this purpose. Nevertheless, the available indexes have significant influence on the costs of query execution and therefore on the overall performance of the DBS. Thus, the first approaches towards DBS self-management have focused on the problem of index selection. Heavy-weight algorithms have been developed for this purpose, which today are available as index wizard tools in most commercial DBMS. The following Sections 2.2.1 and 2.2.2 describe two exemplary index selection tools (IBM DB2 Design Advisor and Microsoft Database Tuning Advisor). In addition, the Oracle self-management architecture is outlined in Section 2.2.3.

2.2.1 IBM D2 Design Advisor

Although often referred to as an autonomic feature, the first index advisor in IBM DB2 has been shipped with IBM DB2 V6.1 and described in 2000 in [VZZ+00], i.e. long before the autonomic computing manifesto has been published. The first academic approaches have even been published as early as 1988 in [FST88]. As illustrated in Figure 2.3, the *DB2 Index Advisor* is an administrative tool that operates outside the database engine. From the perspective of the DBA it operates as a black box, which takes a set of SQL statements as *workload* information and *constraints* as input parameters, and produces a set of *recommended indexes*, i.e. additional indexes that should be created by the DBA. The DB2 Index Advisor supports two types of constraints: the maximum disk space that may be used for additional indexes (e.g. 10 GB) and the maximum computation time of the index advisors (e.g. 10 minutes). It is the task of the DBA to actually implement the recommended indexes.

In contrast to previous approaches like [FST88], the DB2 Index Advisor utilizes the cost model of the DBS-internal optimizer in order to judge the benefit of an additional index. For this purpose it uses several extensions of the optimizer: In a first step (*determine costs*) it determines the execution costs of each of the SQL statements in the workload by using an *explain* mode of the optimizer. In the explain mode, the DB2 optimizer does not actually execute a given SQL statement, but only determines it's execution plan and execution costs. The second step *get candidates* then uses the *recommend* functionality of the DB2 optimizer, which for every statement returns the indexes which would optimally support the statement's execution. For this purpose the optimizer analyses the columns referenced in the predicates as well as the ordering or grouping requirements. It then uses the heuristic described in [VZZ+00] in order to determine a set of promising column combinations for indexes. Each of these column combinations is then added as a *virtual index*, i.e. as an index which only appears to exist for the optimizer, but is not actually physically present. In the third step *try candidates*, the index advisor triggers a re-computation of the execution plans for every statement using the *evaluate* mode of the optimizer. Like the explain mode, the evaluate mode does not actually execute the

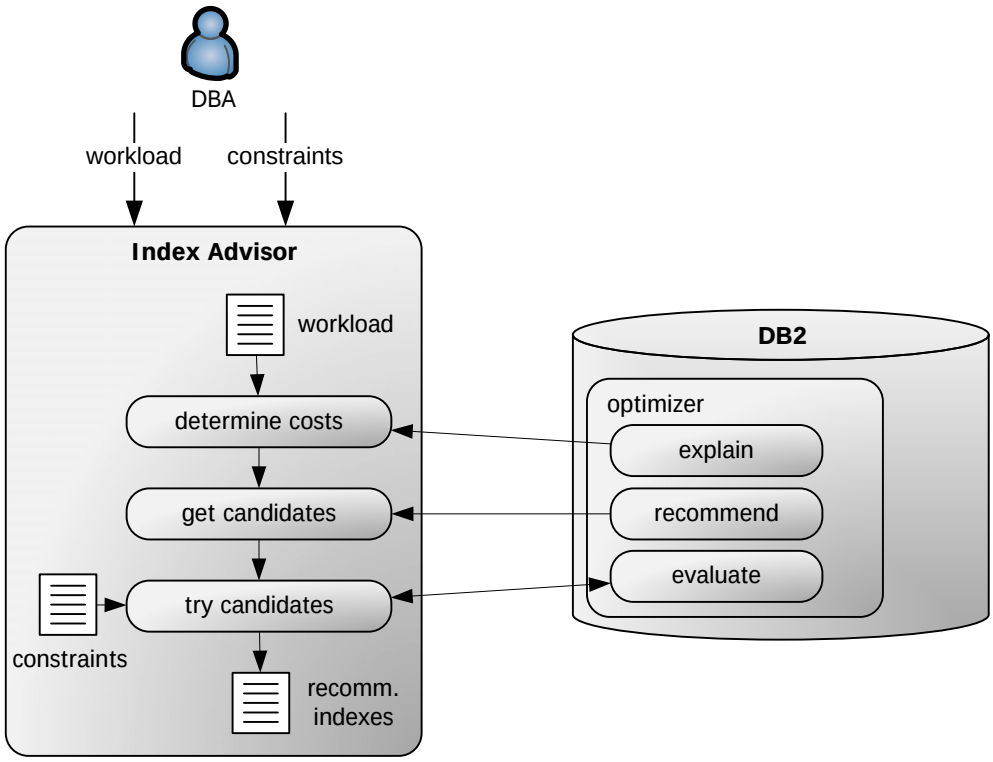


Figure 2.3: DB2 Index Advisor described in [VZZ+00]

SQL statements, and it additionally considers the virtual indexes. All virtual indexes that are used in the resulting execution plans are returned as the result to the index advisor together with the estimated execution costs. By comparing original execution costs with the costs using the virtual indexes, the index advisor selects the set of indexes that provides the highest overall benefit using a knapsack algorithm in the final step *select candidates*.

The algorithms in the DB2 Index Advisor have some limitations. Most importantly, they do not consider negative benefits from indexes in case of data modification operations, and they incorrectly assume independency between the selected indexes. For this reason the final processing step *select candidates* additionally performs random replacements in the set of candidate indexes and then re-computes the overall execution costs for the given workload until the time limit is reached. This heuristic is assumed to compensate the limitations of the algorithms.

The DB2 Index Advisor described in [VZZ+00] focuses on recommending a set of appropriate indexes only, whereas all other physical design options are ignored. Thus, IBM has developed the *DB2 Design Advisor*, which also considers materialized views (MQTs), multi-dimensional clustering (MDC) and partitioning. As for the indexes, for each of these techniques an independent self-management solution had been developed ([ZZL+04], [LB04], [RZML02]). The design advisor described in [ZRL+04] integrates these individual solutions to an overall physical design recommendation tool. For this purpose the design advisor first defines different levels of dependencies between the physical design options (strong, weak, none). For example, there is a strong dependency between indexes and MQTs, because the selection of an MQT can make an index useless, and because an MQT might require an index to be useful. In contrast, the

dependency between indexes and partitionings is considered as weak, because the partitioning should be selected in a way that minimizes intermediate result sizes and not by the available indexes. However, the available indexes can influence the join methods in the execution plan, and therefore affect the selection of the partitioning (for details see [ZRL+04]).

In order to determine the adequate physical design, i.e. the optimal set of indexes, MQTs, partitioning, and MDC decisions, the DB2 Design Advisor employs a hybrid approach: For strong dependencies new self-management components have been built, which cover the joint search space of the dependent physical design options at the same time (*integrative* approach). For dependencies of type "weak" or "none", an *iterative* solution is taken, i.e. each physical design option is optimized separately using the existing self-management techniques.

The implementation of the DB2 Design Advisor follows the approach of the DB2 Index Advisor. It extends the existing *recommend* and *evaluate* modes for indexes by corresponding self-management-logics for MQTs, MDCs and partitionings. In principle, all modes can be activated independently from each other. For example, the mode evaluate indexes can be combined with mode recommend partitioning. With this configuration, the DB2 Design Advisor recommends a partitioning while assuming that the indexes suggested from a previous iteration are actually present. Following its dependency definitions, there are three components in the DB2 Design Advisor: An *IM* component recommends Indexes and MQTs, *C* recommends MDCs and *P* partitionings. The implementation of these components corresponds to the solutions described in [ZZL+04], [LB04], and [RZML02]. A detailed description of the DB2 Design Advisor algorithm is given in [ZRL+04].

As stated in [ZRL+04], the optimization time in index advisors typically grows exponentially with the workload size. In order to keep the analysis overhead reasonable, the DB2 design advisor therefore comprises a workload compression functionality. The selected approach retains only the top k most expensive queries in the workload, whose total cost is smaller then $X\%$ of the original workload cost. When executing the DB2 design advisor, the DBA is then offered three different levels of compression: low ($X = 60\%$), medium ($X = 25\%$), and high ($X = 5\%$)

The experimental results given in [ZRL+04] show that for a TPC-H benchmark [Tra08] database with all 22 queries the design recommendations were retrieved after 10 minutes. The physical design analysis in this scenario had covered indexes, MQTs, MDCs, and a partitioning. After implementing the recommended design, the response time had improved by 84%. In addition, the experiments described in [ZRL+04] show that the medium workload compression level can reduce the analysis time by factors 2 to 10 (depending on the workload size).

2.2.2 Microsoft SQL Server Database Tuning Advisor

Like the DB2 Design Advisor, the Microsoft SQL Server Database Tuning Advisor is based on an index selection tool [CN97] built long before the autonomic computing manifesto has been published. The mode of operation of the original index selection tool is similar to the DB2 Index Advisor. Most importantly, it also uses the DBS-internal optimizer in a "what-if" mode,

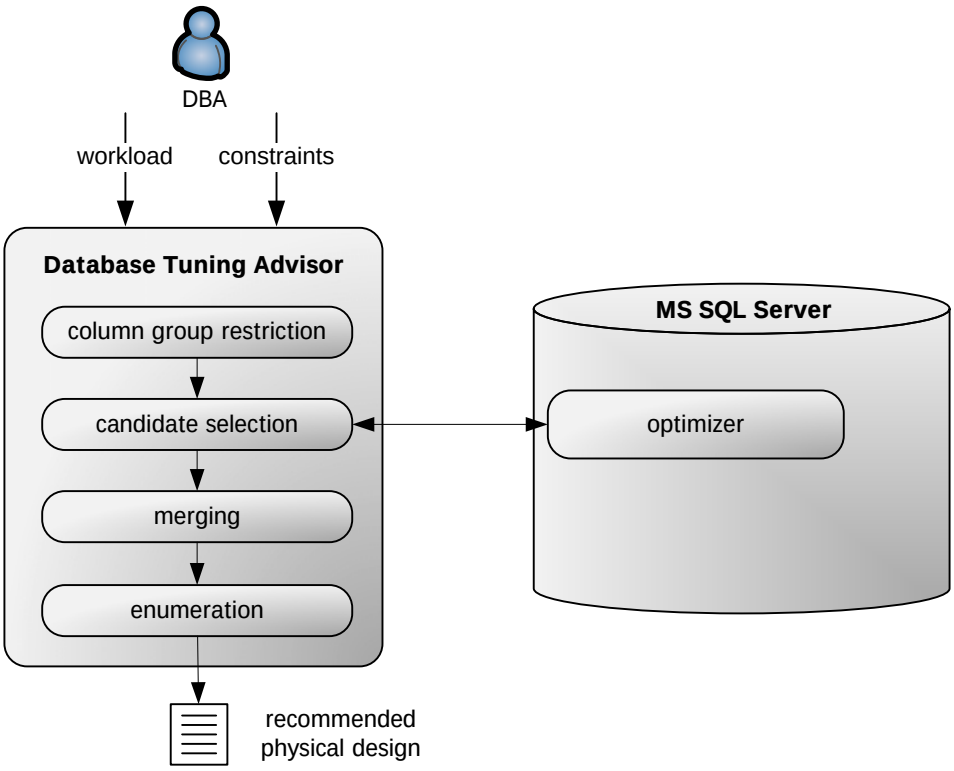


Figure 2.4: Microsoft SQL Server Database Tuning Advisor as described in [VZZ+00]

i.e. in a mode that determines the costs of query execution considering virtual indexes. [CN98] describes the "what-if" mode of the SQL Server optimizer in detail. The index advisor of the SQL Server afterwards has been extended by support for recommendations of materialized views [ACN00] and horizontal and vertical partitioning [ANY04].

The Database Tuning Advisor comprises the physical design recommendation techniques for indexes, materialized views and partitioning. Its architecture, which is described in [ACK+04] and [ABCN06], is illustrated in Figure 2.4. Like the DB2 Design Advisor, the Database Tuning Advisor takes a set of SQL statements as workload information and a set of user-defined constraints as parameters. The constraints may refer to the disk space, computation time, and the horizontal partitioning options. From this information the Database Tuning Advisor computes the recommended physical design in four processing stages:

In the first stage *column group restriction*, interesting *column-groups* are extracted from the workload. A column-group is a set of columns which is referenced by at least one of the queries in the workload. As the number of possible column groups grows exponentially with the number of tables and columns in the database, the Database Tuning Advisor performs a pruning on the column groups. For this pruning operation the column groups are first attributed with column group costs, which are defined as the fraction of the costs of all queries in the workload that reference the column group. All column groups whose column group costs do not exceed a given threshold are excluded from further processing. A detailed example for the computation of column group costs is given in [ANY04].

In the *candidate selection* stage, the interesting column-groups are used in order to determine

the partitioning, materialized views and indexes. This stage is executed on a per-query-basis, i.e. for every query the effectiveness of a partitioning, materialized view and index is evaluated independently. Only the interesting column groups are considered as the basis for the physical design options in this stage. In order to the estimate the benefits of a particular configuration option, the optimizer of the DBS is used in the "what-if" mode. As a result, the candidate selection stage produces a large set of configurations, which on the one hand would optimally support the queries in the workload, but on the other hand may be over-specialized for the DBS workload, may require too much disk space, and may slow down updates.

The task of the *merging* stage is the augmentation of the initial candidates with additional physical design options. These additional configurations have a more general characteristic than the initial configurations, and therefore can serve multiple queries in the workload. The merging technique used in the Database Tuning Advisor is described in [ANY04] in detail.

The final stage of the physical design recommendation (*enumeration*) chooses a set of physical design options from the candidate configurations created by the merging stage. For the choice of the candidates to be added to the final configuration, a simple heuristic is used: For a small fraction of the available disk space, the optimal set of physical design options is determined by building all possible subsets and computing the combination with the least execution costs for the given workload. Based on this "seed", the other configuration options are added incrementally, where in each step only one option is added (the one which provides the greatest benefit). Thus, the enumeration stage provides a trade-off between accuracy and performance.

Like the DB2 Design Advisor, the Database Tuning Advisor in the Microsoft SQL Server also provides a workload compression technique in order to scale to large workloads. However, a different approach is taken in this case: Instead of retaining the most expensive queries only, a clustering is performed in order to identify similar queries in the workload. The SQL statements in the given workload are compared to each other using a distance function for this purpose.

The mode of operation of the Database Tuning Advisor is similar to the DB2 Design Advisor. Although there are minor differences, e.g. the location of the candidate generation (DBS-internal optimizer vs. external advisor tool), the general processing steps are identical. However, it is important to note that – in contrast to the DB2 Design Advisor – the Database Tuning Advisor strictly follows an integrated approach for deriving the physical design options. Thus, as described in [ACK+04] and [ANY04], the dependencies between indexes, materialized views and partitioning are considered during candidate selection. This makes the approach less flexible with respect to the extension for further physical design options in the future.

As described in [ACK+04], the effectiveness of the Database Tuning Advisor has been evaluated for both real-world workloads and benchmark workloads. For the real-world workloads, the produced recommendations have caused a performance increase of up to 50% when compared to a manually-tuned DBS. For a previously un-tuned TPC-H [Tra08] database, the observed improvement in the benchmark execution time has even been 83%. The reported workload analysis times without workload compression are significant: The analysis of a workload with 15.000 statements took 35 minutes, whereas for a workload with 176.000 statements more than

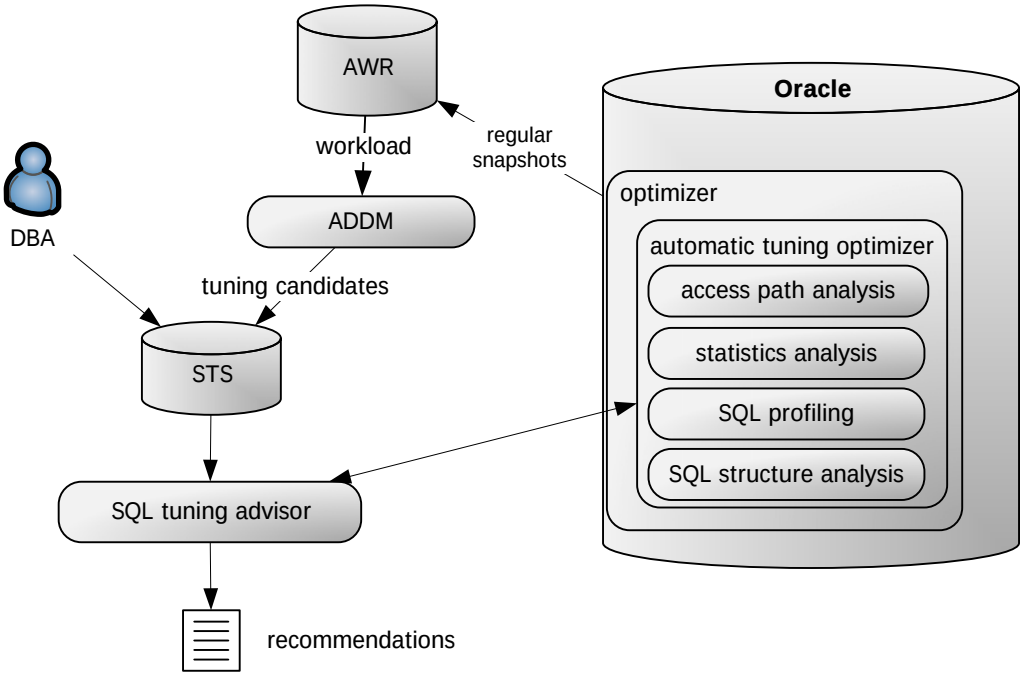


Figure 2.5: Oracle SQL Tuning Advisor as described in [DDD+04]

15 hours were required, for example. Using the workload compression, the efforts could be reduced by factors ranging from 0 to 43.

2.2.3 Oracle SQL Tuning Advisors

Compared to the solutions in IBM DB2 and Microsoft SQL Server, the SQL Tuning Advisor [DDD+04] in the Oracle DBMS takes a more general approach. Instead of focusing on the recommendation of a physical design only, the Oracle SQL Tuning Advisor also identifies missing statistics, sophisticated persistent execution plans (SQL profiles), and more efficient ways of query formulation.

Figure 2.5 illustrates how the Oracle SQL Tuning Advisor is embedded into the overall self-management architecture of the Oracle DBMS. As shown in the figure, the Oracle self-management framework takes regular *snapshots* of the DBMS and stores them in the *Automatic Workload Repository (AWR)*. The snapshot interval is usually set to one hour. These snapshots contain detailed statistical information about the time spent in particular database functions, information on the active sessions and system configuration data. A detailed discussion of the snapshot contents is given in [DRS+05].

Whenever a new snapshot has been taken and stored in the AWR, the *Automatic Database Diagnostic Monitor (ADDM)* investigates the snapshots for performance issues. For this purpose ADDM analyses the time consumption from two different points of view: On the one hand, it analyses the time spent at the various phases of statement processing, e.g. connection establishment, statement optimization, statement execution. On the other hand, it analyses the overall times spent waiting for system resources like CPU, I/O, or locks. As described

in [DD06], ADDM employs a graph that distinguishes symptoms from root causes in order to determine the reasons for performance problems. Depending on the cause, ADDM may execute additional advisors like the Memory Advisor, Segment Advisor, and the SQL Tuning Advisor.

If ADDM detects that particular SQL statements consume excessive resources, it recommends the execution of the SQL Tuning Advisor for these high-load statements. ADDM stores the corresponding statements in the *SQL Tuning Set (STS)* for this purpose, which comprises the SQL statement, its execution statistics (e.g. CPU time, disk reads, ...) and its execution context (e.g. compilation parameters, variable bind values, ...). The STS may not only be filled from the AWR contents, but the DBA may also provide the SQL statements manually.

For the statements stored in the STS the Oracle SQL Tuning Advisor can perform four different types of analyses: *SQL profiling*, *access path analysis*, *statistics analysis*, and *SQL structure analysis*. These analysis functions are implemented in an extension of the Oracle optimizer which is referred to as the *automatic tuning optimizer*. The SQL tuning advisor acts as a front-end, which passes the SQL statements to the automatic tuning optimizer in the DBMS and presents the results to the user.

The subject of the *access path analysis* function is the identification of indexes that would minimize the execution time of a given SQL statement. This task comprises two steps in the automatic tuning optimizer: In the first step, candidate indexes are identified by analysing the equality predicates, range predicates, and ordering clauses in the SQL statements. The effectiveness of each index candidate is then evaluated by using the Oracle optimizer in a "what-if" mode, i.e. the estimated costs of executing the SQL statement are determined assuming that the candidate index would be present. Being a prerequisite for accurate optimizer results, the required table and column statistics are determined prior to the "what-if" analysis. Whenever the automatic tuning optimizer finds that the execution of the statement using one or more of the candidate indexes is by factors faster than without the candidate indexes, it suggests the creation of these indexes.

It is important to note that the access path analysis function in the Oracle SQL tuning advisor only considers one statement at a time. Thus, the overall effects of an index on the entire workload are not considered. As indexes may impose significant overhead in the case of frequent modifications to the indexed data, the recommended indexes have to be checked precisely by the DBA. In addition, the SQL Tuning Advisor recommends the execution of the SQL Access Advisor. This additional tool collects the recommendations for each individual statement and consolidates them into a global recommendation [DDD+04]. Furthermore, the SQL access advisor also recommends materialized views. Unfortunately, only the usage of the SQL access advisor is documented [Hob03], whereas its underlying concepts and implementation are not.

This section has outlined the Oracle SQL Tuning Advisor and illustrated how it is integrated into the overall self-management architecture of the Oracle DBMS. In particular, the mode of operation of the access path analysis has been explained. Details on the three other analysis functions (SQL profiling, statistics analysis, SQL structure analysis) are given in [DDD+04].

As a result it can be stated that the Oracle SQL Tuning Advisor – like the DB2 Design Advisor and the Microsoft SQL Server Database Tuning Advisor – is a maintenance tool that has to be executed and supervised by a DBA. To overcome this limitation, Oracle has enhanced its self-management architecture by a technique for the autonomic detection and installation of reconfigurations in its latest release (Oracle 11g). However, as described in [BDDY09], the automatic adaptation of the SQL Tuning Advisor analysis results is only possible for the SQL profiling functionality, whereas the three other analysis function results still require the validation and implementation by the DBA.

2.3 DBS On-line Self-Management

The off-line tools described in Section 2.2 are helpful to a DBA, because they reduce the time required to tune the DBS properly. However, the DBA still remains responsible for identifying the points in time when the execution of these off-line tools is advisable. Thus, these tools do not yet meet the goals of autonomic computing, which require an automatic adaptation of the system to changing environmental conditions (self-configuration, self-optimization). Current industrial and research approaches are therefore directed at creating on-line DBS self-management functions. The major challenge of creating on-line self-management functions is the design of light-weight analysis algorithms. Section 2.3.1 presents the on-line self-management functions for the database memory in IBM DB2. Two research approaches for on-line index self-management are afterwards presented in Section 2.3.2. Self-tuning optimizer statistics are introduced in Section 2.3.3.

2.3.1 On-line Memory Management

A DBS typically distinguishes multiple memory areas, each serving a specific purpose. For example, the system buffer caches the pages that have recently been accessed, because there often is a high probability that these pages will be requested again in the near future. Similarly, there are usually dedicated memory areas for sorting records, caching execution plans and storing the locks on database objects. In the past, it has been a challenging tuning task for the DBA to assign the available physical memory to these memory areas, because the optimal setting is highly workload-specific. The vendors have therefore equipped their DBMS products with autonomic memory management functions, which continuously adapt the memory area sizings to the current workload. Although apparently autonomic memory management has been realized in Oracle [LNK+03], IBM DB2 [SGAL+06] and Microsoft SQL Server [Cor10], only IBM has published the implementation details. The following paragraphs therefore describe the automatic memory management in DB2 only.

The Self-Tuning Memory Manager (STMM) in IBM DB2 continuously adapts the sizes of the system buffer segments (referred to as *bufferpools* in DB2), the sorting area, the compiled statement cache, and the lock list. Its decisions on which areas have to be extended or shrinked

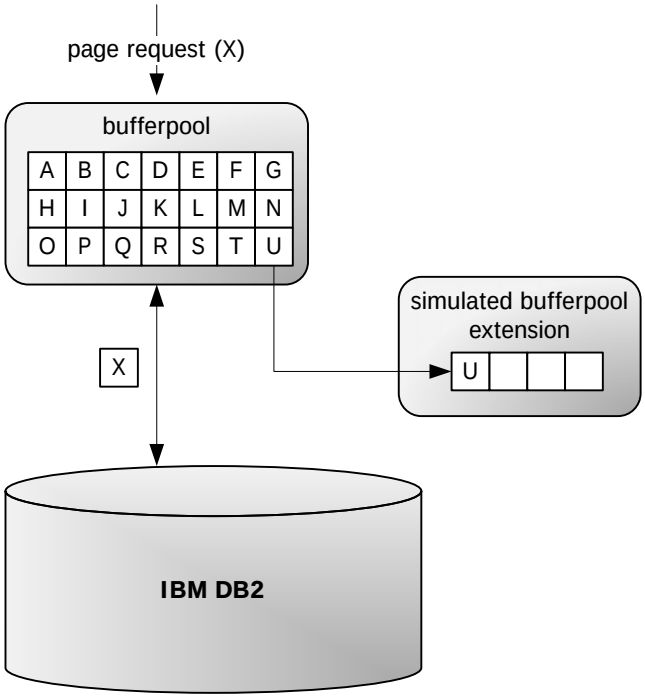


Figure 2.6: Benefit Estimation in the DB2 Self-Tuning Memeory Manager as described in [SGAL+06]

are based on a cost-benefit analysis for every memory area. In order to compare the benefits in the managed memory areas, the STMM defines the *saved system time per unit memory* as a common metric. Thus, the STMM can compare the benefit generated by the reduced CPU usage (larger statement cache) to the benefit of the reduced I/O requests (larger bufferpools).

The calculation of the costs and benefits in the STMM is specific to every memory area. However, only the benefit computation methods for bufferpools and for the compiled statement cache have been published [SGAL+06]. The benefit calculation for bufferpools is illustrated in Figure 2.6. In order to determine the system time that could be saved if the bufferpool was larger than it currently is, the STMM uses a *simulated bufferpool extension*. Whenever a page has to be removed from the actual bufferpool because the buffer pool is completely filled and a new page request arrives (page X in the example in Figure 2.6), the ID of the removed page (page U in Figure 2.6) is stored in the simulated bufferpool extension. For all page requests that are found in the simulated bufferpool extension, but not in the bufferpool itself, the time required for the removal of the old page and the reading of the requested page is tracked. The sum of all these page read times is considered as the cumulative saved time if the bufferpool was extended by the size of the simulated bufferpool extension. The cumulative saved time is then divided by the number of pages in the simulated bufferpool extension in order to determine the saved system time per unit memory (i.e. page) metric. The calculation of this metric for the compiled statement cache is given in [SGAL+06].

The cost/benefit-metrics of memory areas are evaluated by a multi-input multi-output controller. The controller uses a model, which for every memory area describes the expected

cost/benefit for the case that its size is increased or decreased. A regression technique (least squares) is employed in order to fit a curve through the 40 most recent size/benefit observations. After the model has been built, an accuracy test is performed on the model: First, a statistical test (F-Test; [Tri04]) is performed on the data in order to ensure the significance of the model for the past observations. Second, the slope of the model is checked to be negative. A positive slope would indicate that giving more memory to the consumer would increase the system time spent in this component, which is not a reasonable result.

If the model has passed the accuracy test, it is used by the controller to compute the memory area's target size. The STMM employs the integral control law for this purpose (for details see [SGAL+06]), which in control theory is usually used to adapt an observed output of a system to some predefined reference value. As for tuning the memory area sizes the benefit values of the optimal sizes are not known, an artificial output is computed as the difference between the average benefit of all memory consumers and the individual memory area's benefit.

After the target memory sizes have been computed, the STMM re-distributes the memory using a greedy algorithm. First the memory areas are divided into two groups: areas with an expected benefit greater than the average (group B) and areas with a benefit value smaller than the average (group C). Pages are then taken from the areas in group C (starting with the areas with the least costs) and donated to the areas in group B (starting with the areas with the largest expected benefit). This process is continued until no further memory can be transferred, either because there are no more pages to take away from the members in group C, or the areas in group B have reached their target sizes. To avoid misconfigurations, the STMM additionally places memory transfer limits on the areas: every area may be increased by at most 50% and decreased by at most 20% during a reconfiguration step. Thus, the strategy of the STMM is not to find an optimal setting of the memory areas in a single step, but to achieve convergence after several tuning steps.

The experimental results for the STMM reported in [SGAL+06] show that – compared to an out-of-the-box configuration – the STMM can improve the performance of a DBS for a standard benchmark workload by 300%. STMM has required about 90 minutes to achieve this result. Furthermore, it is reported that the resulting performance was within 1,4% of a hand-tuned DBS.

2.3.2 On-line Index Selection

Unlike memory management, the on-line index selection approaches have not yet been integrated into commercial DBMS products. The following paragraphs therefore outline three prominent research approaches: the QUIET framework, the COLT framework, and the approach by Bruno and Chaudhuri.

The QUIET framework [SGS03] has been the earliest on-line index selection solution. Its goal is the automatic creation and removal of indexes, while considering a DBA-defined space limit for the index data. QUIET runs outside of the DBS, where – as illustrated in Figure 2.7

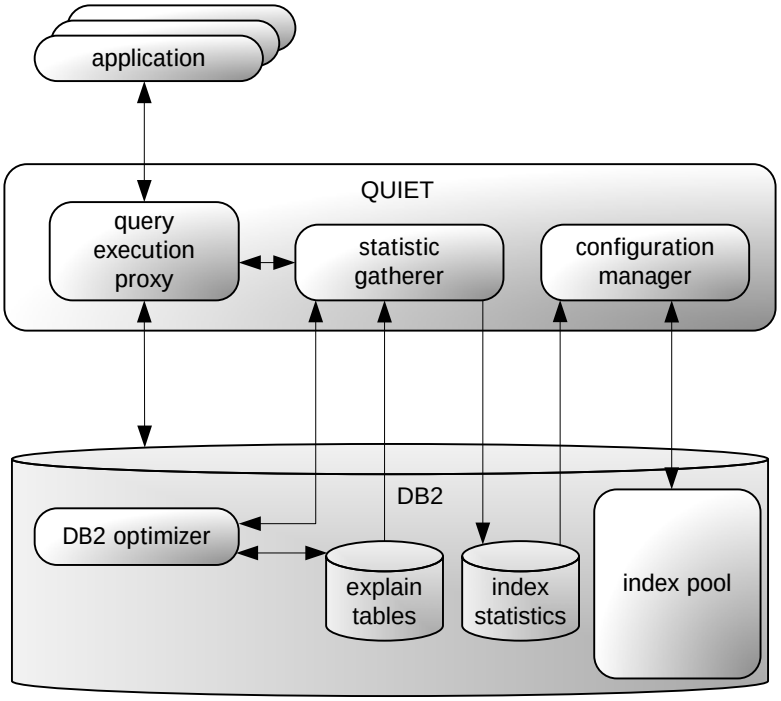


Figure 2.7: Illustration of the QUIET Framework described in [SGS03]

– it acts as a proxy intercepting the queries submitted to the DBS. The QUIET framework has been implemented on top of IBM DB2. Every single query that is executed by an application is intercepted by the *query execution proxy* and then forwarded to both the DBS and the *statistics gatherer* component of the QUIET framework. In order to determine the most adequate set of indexes, QUIET employs the *DB2 optimizer* extensions that have been developed for the DB2 Index Advisor (see Section 2.2.1): For each query the benefits of additional indexes are computed by first estimating the execution costs with the current set of indexes (explain mode), then deriving a set of useful additional index candidates (recommend mode), and finally re-estimating the query execution costs assuming that the additional indexes were present (evaluate mode). Details for how the index benefits are derived from the overall query execution time reduction are given in [SSG04].

The benefits that the statistics gatherer has computed for all indexes (i.e. both virtual and materialized indexes) are stored and maintained in the *index statistics* tables. In addition to continuously updating the index benefit statistics for the recent queries, the statistics gatherer applies an ageing strategy to older index statistics. Thus, it is ensured that the current workload of the DBS is more important for index selection than historic workload. The index statistics are evaluated by the *configuration manager* component of the QUIET framework. This component employs a greedy algorithm (for details see [SSG04]) to select and materialize those indexes that provide the highest overall benefit. In order to avoid thrashing, reconfigurations are performed only when the aggregate benefit of the index configuration changes exceeds a given threshold.

Due to the index recommendation for every query and the DBS-external implementation of QUIET, the overhead caused by this on-line index selection is significant. As reported in

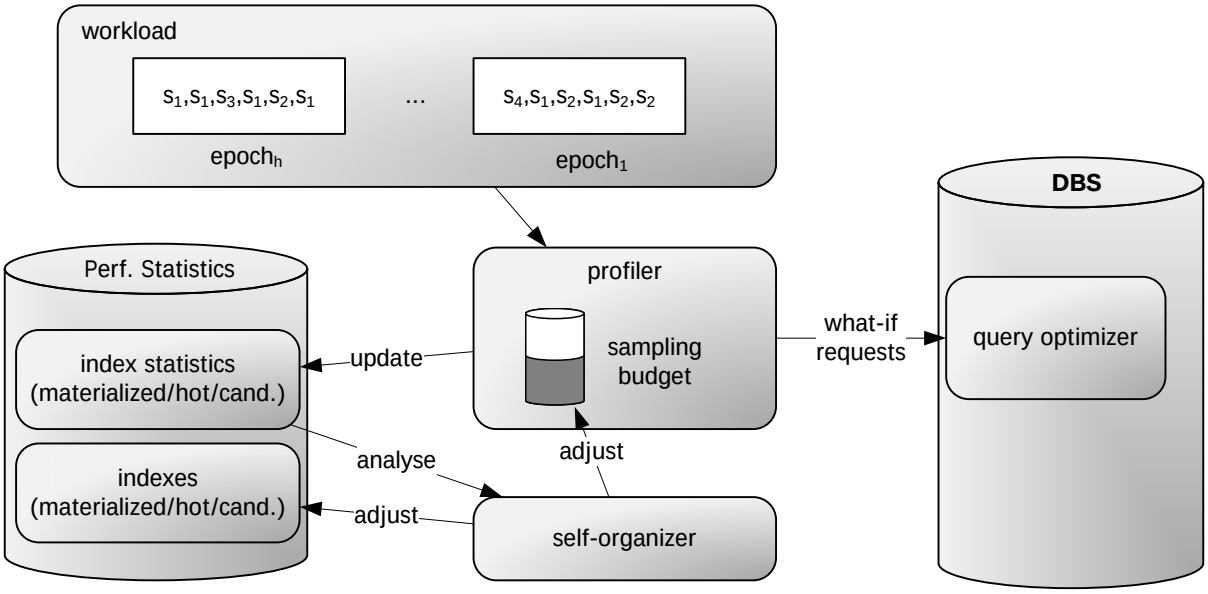


Figure 2.8: Illustration of the COLT Framework described in [SAMP07]

[SSG04], the overhead caused by the index selection solution is approximately one second per query. In order to reduce the overhead of the framework, it has been enhanced in two direction in [LSSS07]: On the one hand, the index analysis components have been integrated into the DBS-core of PostgreSQL, thus allowing a tight coupling with the optimizer. On the other hand, the overhead for building the indexes has been minimized by introducing the concept of *deferred indexes*. When deferred indexes are created they are empty and therefore not usable for query processing at first. Instead they are built up in an opportunistic way, i.e. only when a query that requires a tablescan to the base table of the index has to be executed upon user-request anyway. In this case, the tablescan operators in the execution plan are replaced with new plan operators that implement the required piggy-back index creation. In addition to the performance optimizations, [LSSS07] also introduces the concept of *soft indexes*. This type of index identifies all indexes that are managed by the on-line index selection framework. Thus, physical design decisions that have been made by a DBA are not affected by the self-tuning decisions.

Although the functionality of the COLT framework [SAMP07] is similar to QUIET to a large extent, the focus of the COLT design is the self-regulation of the induced overhead. Figure 2.8 provides an overview of the COLT framework. As can be seen from the figure, COLT also divides the workload of the DBS into equally-sized windows referred to as *epochs* and limits the number of epochs to a size h. For all statements in the considered windows COLT maintains a set of candidate indexes, which are derived from the predicates in the SQL statements. For all of these indexes the *profiler* continuously updates the statistics, i.e. the information about the benefits expected from materializing an index.

In order to keep the continuous analysis overhead small, COLT distinguishes three subsets of the candidate indexes: *candidate*, *hot*, and *materialized*. For the candidates, i.e. the indexes

which are not in the hot or materialized sets, only very coarse but cheap statistics are computed. These statistics express the difference in query execution costs when executing the queries either using a table scan or an index. Standard cost formulas are employed for this purpose, i.e. no "what-if" call to the optimizer is executed. In contrast, the execution costs with the indexes in the hot index set are determined by an appropriate "what-if" call to the optimizer. But in order to reduce the overhead, not all queries are selected for a "what-if" analysis. The profiler instead has a *sampling budget*, which defines the probability of a "what-if" analysis for a query. In addition, COLT performs a clustering of the incoming queries: all queries accessing the same tables, having the same join predicates, and the same selection predicates are assigned to one cluster. A "what-if" optimization call is then performed only once per cluster. The statistics for the materialized indexes are computed in same way as the statistics for the hot indexes. The key difference is of course, that the "what-if" analysis in the optimizer must pretend that the given index is *not* present.

At the end of each epoch the *self-organizer* is invoked, which performs a reorganization of the indexes and a re-budgeting of the sampling budget. The self-organizer analyses the statistics of the indexes and computes the benefit of materializing the indexes, assuming that the workload for the next h epochs remains the same as in the past h epochs. For indexes that are not in the materialized index set, it also takes into account the materialization costs. It is important to note that candidate indexes are never materialized directly. If the self-organizer determines that a candidate was helpful, it moves the index to the hot index set first, allowing the profiler to collect detailed statistics for it in the following epoch. During the re-budgeting phase, the self-organizer updates the sampling budget of the profiler. Its goal is to increase the sampling budget in case it suspects a change in workload and to decrease it when the workload has not changed. For this goal the self-organizer assumes a best-case scenario for the usefulness of the hot indexes and re-computes the new materialized indexes. If there are major differences between this "optimistic" materialized set and the "normal" materialized set, the self-organizer increases the sampling budget. Details on the self-organization rules are given in [SAMP07].

The evaluation results reported for COLT show that for a stable workload the query execution performance using the on-line tuning approach becomes almost the same as the performance of a DBS that has been optimized with an off-line tuning tool after some time. Of course, the on-line approach requires some time in the beginning to derive the correct physical design. For a changing workload, the performance of the DBS managed by COLT even exceeds the performance of the design resulting from the off-line tool design. The reason for this observation is that COLT can adapt the design to changes in the workload, whereas the off-line tool is executed only once for the entire workload and therefore has to choose an intermediate design. While the evaluation in [SAMP07] shows that the number of "what-if" calls is effectively reduced when the workload does not change, the overall relative overhead caused by COLT is not reported.

Bruno and Chaudhuri have described a solution [BC07] to the on-line index selection problem, which is fully integrated with the query optimizer of the Microsoft SQL Server. However, the article also describes a research prototype that has not (yet) been integrated into the SQL Server product. The focus of the described solution is to realize on-line index selection with as little overhead as possible. In particular, the goal of the authors is to avoid *any* additional calls to the optimizer, i.e. they do not use the "what-if" functionality of the optimizer. Instead, only the information that can be retrieved from the optimizer during normal query processing is analysed in order to identify the need for additional indexes. For this purpose the authors employ a technique developed in [BC06], which allows the tagging of the queries' execution plans with *index requests*. Index requests resemble cost estimations for the usage of alternative index structures, which were considered but not used by the optimizer when building the execution plan. Based on the actual execution plans and the taggings representing the alternatives, the on-line index selection solution can estimate the effect of alternative physical designs, e.g. the creation of an additional index. This approach allows the on-line indexing solution the generation of a locally-optimum execution plan. Obviously, the creation of globally-optimum execution plans, which also considers different join orders under a hypothetical physical design is not possible.

Based on the observations about the benefits of additional indexes that can be derived from the execution plans, the on-line index selection described by Bruno and Chaudhuri has to decide which indexes actually to create. The basic idea of their approach is to evaluate whether or not the cost of the index creation is smaller than the benefit from this index in the future. Of course, this question is impossible to decide for an on-line index selection, because the future workload is unknown. Hence, the solution described in [BC07] takes an approach where the index creation conceptually "lags behind" the current workload: An index is created only when the self-management logic discovers that – considering the past – the creation of the index would have been beneficial some time ago, because since then the benefit for the observed workload would have exceeded the index creation costs. Thus, the on-line index selection makes the assumption that the workload observed in the past remains stable, at least in the near future.

To account for index interactions, the on-line index selection assigns usefulness levels to every index candidate. The usefulness of an index I_1 with respect to another index I_2 is defined as follows: it takes the value -1 if the I_1 columns do not include any I_2 columns, the value 0 if the I_1 columns include I_2 columns, the value 1 if additionally the leading column of I_2 agrees with I_1's, and the value 2 if additionally I_2 is a prefix of I_1. Whenever the index selection algorithm decides to add or drop an index, it adapts the current benefit statistics for all other index candidates by computing and comparing their usefulness levels. In addition to index interactions, the on-line index selection algorithm also takes into account storage constraints and prevents the oscillation of physical design decision. The details of the implementation are given in [BC07].

The on-line index selection algorithm by Bruno and Chaudhuri makes some assumptions and approximations. Nevertheless, the reported evaluation results show that the costs of executing

a workload using the on-line algorithms are close to the costs when the entire workload is known in advance and the physical design is created accordingly. Compared to the usage of the off-line SQL Server Database Tuning Advisor in advance, the on-line solution of course exhibits longer execution times at the beginning of the workload. However, with time the execution costs become almost identical. In the presence of significant changes in the workload the on-line variant even achieves smaller execution times, because the physical design is adapted to the changes in the workload over time. The reported overhead caused by the on-line index selection is only 1.7%.

2.3.3 On-line Statistics Collection

In a DBS, the task of the optimizer is to create procedural execution plans for declarative queries. Hence, the optimizer has to determine the most efficient way of executing a query submitted by an end-user. To achieve this goal, the optimizer requires detailed statistics about the data, especially the cardinality of the base tables and the selectivity of predicates, because this information is necessary to estimate the size of intermediate results. In order to keep the overhead reasonable, optimizers typically make a number of assumptions, e.g. uniformly distributed data values, statistical independence of predicates, and the correctness of the statistics. If any of the assumptions is incorrect, sub-optimal execution plans will be created, causing large processing overheads. Traditionally, it has been the task of the DBA to provide the relevant and accurate statistics for the optimizer. As the selection of the correct statistics is a challenging task for the DBA, solutions for the autonomic maintenance of optimizer statistics have been developed. The following paragraphs introduce the LEarning Optmizer (LEO) [MLR03] developed as a research project by IBM. Another example for self-managing statistics are the self-tuning histograms, which are described in [ABCN06].

An overview of the LEO architecture is given in Figure 2.9. The left part of the figure illustrates the usual query processing stages in IBM DB2: An incoming query is first transformed into a query execution plan. A query execution plan is a tree structure, where the leafs represent base table access (e.g. table scans, index access) and the inner nodes represent operators on the data (e.g. selections, joins, groupings). The edges in the query execution plan are attributed with cardinality estimations of the data sets passed between the operators. LEO stores these *cardinality estimations* for every query execution plan that is produced by the optimizer. For this purpose the *code generator* has been extended by a component which stores the query execution plan information into a dedicated file. The code generator then produces an executable program, which is executed in the *runtime system*. The runtime system also has been extended for LEO in order to store the *actual cardinalities*, i.e. rows actually processed by the operators.

The main component of LEO is the *cardinality analyser*. It runs outside the DB2 optimizer as a separate process. For every query execution found in the actual cardinalities storage area, the analysis component first identifies the corresponding query execution plan. By comparing the estimated and the actual cardinalities it calculates *adjustments*, which are stored as additional

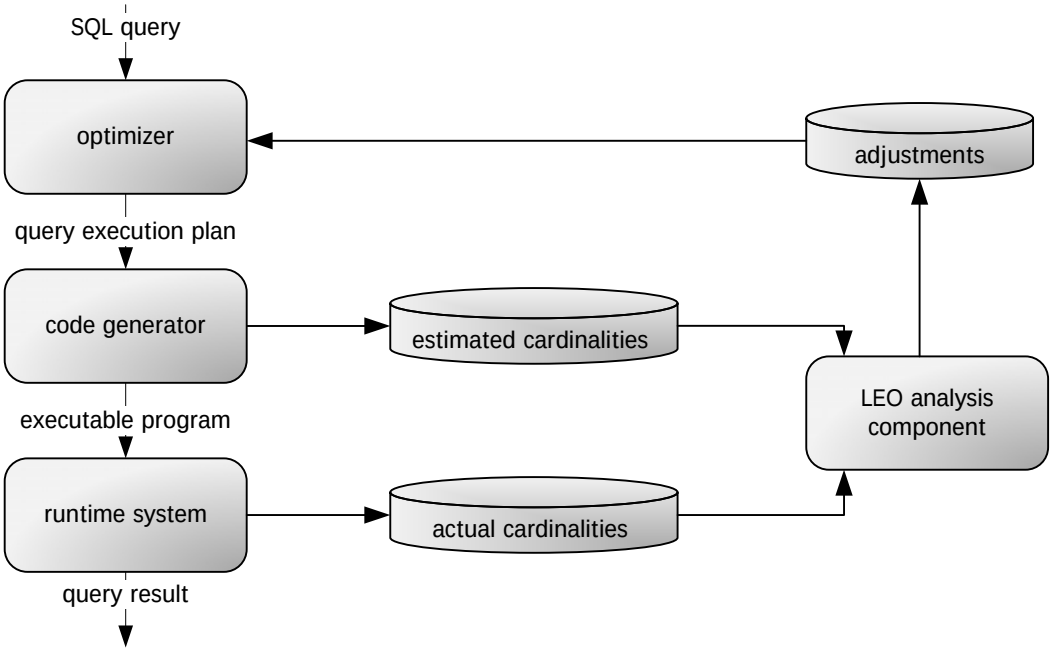


Figure 2.9: The LEarning Optmizer (LEO) as described in [SLMK01]

knowledge for the optimizer. To avoid a large number of insignificant adjustment factors, LEO only calculates the adjustments for cardinality deviations larger than 5%. Details on the adjustment factor calculation rules for different types of predicates (e.g. equality, join, start/stop key, ...) are given in [SLMK01].

It is important to note that these adjustment factors constitute an additional input to the optimizer, i.e. they are not used to modify the base statistics directly. For base table statistics, the optimizer for example applies this factor to the number of pages, which is usually used in order to estimate the I/O overhead of performing a tablescan. Furthermore, it computes corrected statistics for the indexes, i.e. the number of leaf/non-leaf pages. The rules for applying the adjustment factors to single-table predicates, join-predicates and correlated predicates are also given in [SLMK01].

As reported in [MLR03], the monitoring overhead of LEO is below 4% of the total query execution time. However, the experimental evaluation in [SLMK01] has shown that the usage of the adjustment factors in the optimizer has resulted in execution plans that were 14 times as fast as the original execution plans.

2.4 Open Challenges in Current Approaches

With the index advisors and the on-line memory management, the previous sections have presented the most important self-management features that currently exist in DBMS. In addition, other on-line self-management functions like an on-line index selection and statistics collection have been discussed, which are still in the status of research prototypes. These autonomic functions will probably be integrated into the commercial products in the future. However, when

comparing these selected examples of DBS self-management to the characteristics of truly autonomic systems (see Section 2.1), it becomes obvious that DBS today do not yet meet the goals of autonomic computing. The following paragraphs discuss the deficiencies of the existing self-management solutions with respect to the goals of autonomic computing in detail.

2.4.1 Self-Optimization

One of the central aspects of autonomic computing is the self-optimization of a system when changes in its environment occur. Considering database systems, this environment is especially determined by the workload of a DBS, because the workload determines how the out-of-the-box product is actually used.

The off-line index advisors described in Section 2.2, which are currently shipped with commercial DBMS products, fail to meet this requirement. They do not automatically adapt the physical design of the DBS to changes in the workload. Instead, their execution has to be triggered manually by the DBA. IBM DB2 and the Microsoft SQL Server do not even alert the DBA when their index advisors should be executed. The DBA has to continuously monitor the workload and performance of the DBS and manually trigger the execution of the advisor when he suspects it reasonable. Continuously executing these advisors is not possible because of their heavy-weight analysis algorithms. Furthermore, it is the task of the DBA to validate the results of the advisors and to implement the recommended changes. Using its Automatic Workload Repository to store regular performance snapshots, the Oracle self-management architecture in contrast can at least detect SQL statements that consume excessive resources. For these statements it then recommends the execution of the SQL Tuning Advisor. Although in its latest release the Oracle self-management architecture [BDDY09] even automatically executes the SQL Tuning Advisor for these statements, the results still have to be validated and applied by the DBA. The reason for this limitation is that the Oracle SQL Tuning Advisor does not consider the overall DBS workload in its analysis, but only individual statements. In Oracle this task is subject of the SQL Access Advisor, which also suffers from the problem of a heavy-weight analysis algorithm, i.e. it cannot be executed continuously.

The on-line index selection solutions that have been described in Section 2.3 are explicitly designed for a continuous operation. Each of the existing solutions focuses on one particular administration task that arises if the workload changes, e.g. index adaptation, memory management, and additional statistics collection. Considering that a commercial DBMS today typically offers hundreds of configuration parameters, it becomes obvious that the existing solutions can cover only a small fraction of the total DBS configuration. Furthermore, many autonomic functions for DBS only exist as research prototypes and have not yet been integrated into commercial products (like LEO for DB2 and the on-line index selection for the SQL Server). To keep the overhead reasonable, the algorithms that are used to detect the necessity of a reconfiguration are very specific for the particular administration task. The on-line index selection by Bruno and Chaudhuri described in Section 2.3.2, for example, only evaluates

whether or not creating an additional index would have reduced the execution costs. The result of this analysis cannot be used in order to judge whether or not there has been a general change in the workload. So the results of their analysis cannot be used to infer the necessity of other configuration changes (e.g. the optimizer level).

As a result it can be stated that meeting the goals of autonomic computing requires on-line self-management functions within DBS. Research and development in this area has started some years ago, but still only a small fraction of the overall configuration parameters is covered by self-management functions. Thus, the self-optimization of the overall DBS in case of workload changes is an open challenge.

2.4.2 Goal-Independency

Following the metaphor of the autonomic nervous system, every autonomic system should locally perform configuration changes whenever they are necessary without user interaction. However, the holistic vision expressed in [Hor01] requires *all* systems in an enterprise to automatically adapt to high-level goals and policies that are defined by an administrator.

None of the existing self-management solutions in current DBMS is capable of considering goals in their analysis. The off-line index advisors, for example, recommend indexes based on the reduction of the *overall* query execution costs. They are not able to consider different response time or throughput goals for specific user groups during their analysis. In the presence of space constraints, for example, the index advisors would dismiss an index frequently accessed by a high-priority user group in favour an index for a table accessed by a best-effort user group, if it found that this would reduce the overall costs. Likewise, the on-line memory functions also do not consider any goal definitions. For example, it could be required to size bufferpools according to the response time goals of the users that access their data. As an another example, it might be reasonable to reduce the size of the bufferpools (as long as the response time goals are met), and to increase the size of the memory area for locks to meet throughput goals, even though this might increase the overall query execution costs. Thus, considering high-level goals for the self-management decisions is an open challenge to autonomic functions in DBS.

2.4.3 Interdependency

The current solutions to DBS self-management are highly specific to a particular DBS component or administration task. The index advisors, for example, only considers the effects of creating an index, while the memory self-management functions only consider the memory areas. If now the DBA discovered a performance problem, he would probably start the design advisor to identify missing indexes. At the same time, the memory manager would probably increase the size of the affected system buffer, because a missing index requires the storage of many additional pages for performing scan operations. This example shows that two different reconfiguration actions may be taken for the same problem. The creation of the index would

have a (positive) side-effect on the system buffer, because less scan operations would have to be performed. Like the positive side effects of physical design decisions on the processing effort, there may as well be negative side-effects. For example, increasing the system buffer size will increase the local performance, i.e. the buffer hitratio, thus reducing the response time of the DBS. However, the increase of the buffer size might also have negative impact on the throughput, if the lock list size has to be reduced accordingly.

Above examples illustrate that the complexity of DBMSs makes it hard to even predict the system-wide effects of changing a single configuration parameter or a physical design decision. Hence, today's feedback control loops monitor and analyse the performance and workload related to one configuration parameter of one system component only. Neither do they consider possible side-effects on other components, nor are they able to coordinate their tuning decisions with other feedback control loops. Concepts for describing the interdependencies between configuration parameters and the expected behaviour of system components are currently missing.

2.4.4 Overhead

As discussed in Section 2.4.1, only on-line self-management functions can meet the goals of autonomic computing. Every on-line self-management function continuously monitors some particular sensor information, analyses this information and initiates appropriate reconfiguration actions when required. However, currently many aspects of DBS configuration and maintenance are not yet under the control of on-line self-management functions. Thus, both researchers and vendors are developing additional on-line self-management functions, which will be integrated into DBMS products in the future. While on the one hand these autonomic functions will reduce the maintenance overhead for the DBA, they will on the other hand contribute additional processing overhead to the DBS. The limitation of the overhead induced by on-line self-management functions therefore is an open challenge to DBS self-management.

2.4.5 Workload-pattern Unawareness

Current on-line self-management functions continuously analyse the appropriate sensor information from the DBS and perform reconfigurations when appropriate. The memory self-management, for example, extends the system buffer size when it discovers that during the past observation interval a greater system buffer would have been beneficial. Likewise, the existing on-line index selection algorithms (COLT, Bruno and Chaudhuri) create a new index when they discover that in the past the index would have decreased the overall query execution costs. Thus, the on-line self-management functions make their tuning decisions based on the assumption that the workload remains stable in the future. In many real-world scenarios, however, this assumption may be incorrect. For example, there may be major differences in the DBS during day-time, when the DBS is used in an on-line transaction processing (OLTP) mode, and night-time, when complex reports are generated for the following business day. If now a

self-management function decides to change the configuration shortly before the workload is about to change, then the resulting configuration might be inadequate for the following workload. This is especially a problem when the reconfiguration actions are expensive, e.g. changes to the physical design. Hence, the self-management functions should refrain from reconfigurations when a workload change is expected in the near future. However, there are no concepts for considering periodic workload changes in current DBS self-management solutions.

2.4.6 Overreaction

Some configuration actions within a DBS can take some time before they take effect. If the lock list size is increased, for example, the full effects on the throughput of the system can only be monitored after all transactions already running have completed and returned their (possibly escalated) locks. When these creation times are disregarded in the analysis algorithms, then there is a risk of overreaction, because the tuning parameter value may be further increased before the full effects have become obvious.

A second risk for overreaction are self-management functions that do not have precise knowledge of the quantitative effects of configuration parameter changes. While this is not the case for the on-line self-management functions presented in Section 2.3, many approaches implement rule-of-thumb algorithms (cf. [WMHZ02]). Changing a configuration parameter without being able to predict the effects of the change is of course a risk for the overall system performance. For this reason the configuration parameters are typically changed in very small steps, i.e. the self-management functions accept long adaptation intervals to avoid overreactions. This method is also employed by the DB2 self-tuning memory manager [SGAL+06], but only before it has learned a dependable model for the effects of memory area size changes. The challenge for the design of new self-management solutions therefore is the development of precise quantitative models, which allow the prediction of the (system-wide) effects of configuration parameter changes.

3 Goal-Driven System-Wide Self-Management

The review of the current DBS self-management technology in Section 2 has shown that the existing concepts do not yet meet the requirements towards truly autonomic systems. In particular, the DBS can not yet be controlled by high-level goals. Section 3.1 discusses the general approaches that might be followed towards meeting goals in DBS processing and identifies *system-wide self-management* as the most appropriate solution. The following sections then outline the general concepts that have been developed for the challenges faced when creating a system-wide self-management framework: Section 3.2 presents an approach for a lightweight workload analysis, Section 3.3 analyses the required contents of a knowledge base for DBS self-management, and Section 3.4 discusses the tasks of the self-management logic in detail. Conclusions are discussed in Section 3.5.

3.1 Goal-Driven Self-management

As discussed in Section 2.4, the existing solutions for DBS self-management focus on one particular administration task only. They do not consider side-effects caused by resource sharing or processing dependencies and cannot consider high-level goals in their analysis. The following paragraphs discuss several approaches to overcome these limitations.

RISC-style DBMS Architecture The implementation of database management systems has been dominated by *layered approaches* for a long time. Structuring the DBMS into layers provides several advantages, e.g. an easier implementation of higher layers because of a higher level of abstraction, and an increased robustness against implementation changes within the layers. Figure 3.1 illustrates the five-layer architecture of a DBMS proposed by Härder and Reuter in [HR83]. Weikum et al. have discussed in [WMHZ02] that the current DBMS architecture is not suitable for fully autonomic databases. Even in a strictly layered architecture, there are plenty of implicit interdependencies between the system components. On the one hand these dependencies are caused by resource sharing between the layers. On the other hand interdependencies are caused by interferences of the data processing mechanisms across the layers.

In order to reduce the complexity of self-management, Weikum et al. advocate a radical departure towards a DBS that is based on RISC-style components [WMHZ02]. Each component

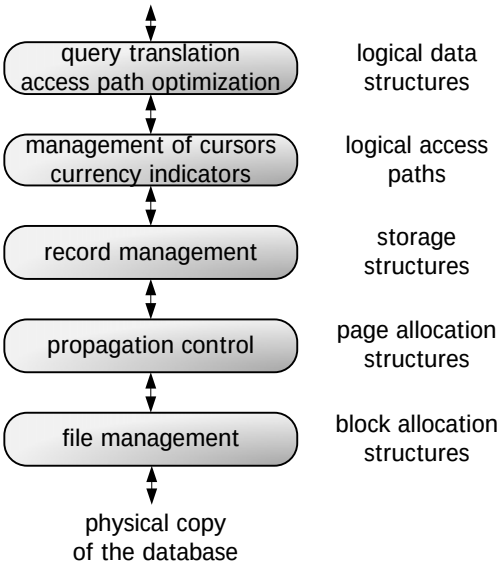


Figure 3.1: Layered DBMS proposed by [HR83]

is supposed to have a limited functionality, making it possible to predict the component's performance using *mathematical models*. [WMHZ02] proposes that the mathematical model is defined at the same time as the implementation of the RISC component itself, because all internal dependencies are known at implementation time. The resulting model is supposed to quantitatively predict the behaviour of the RISC component under all possible configurations and workload characteristics. By evaluating its mathematical model, every RISC-component could therefore perform a dependable self-optimization.

The proposed architecture uses rather coarse-grained components intended as building blocks for enterprise applications. In particular, the authors propose to create a select-project-join engine, which could either be used directly by an application, or upon which a multidimensional query engine could be plugged. To reduce the interaction complexity when multiple RISC components are combined, the interfaces of the RISC components should be designed as narrow as possible. To simplify self-configuration and -optimization, even SQL is proposed to be replaced by a narrow component API.

[WMHZ02] sketches the problem of meeting high-level business goals only very briefly. It demands that the mathematical performance models of the individual data processing components should be composed to an overall model. However, how this challenge could be solved is not described.

The RISC-style architecture demands a re-implementation of the entire DBMS. Moreover, even the DBS applications would have to be re-designed to operate on the RISC components with specific APIs instead of querying a single DBS via SQL. It can therefore only be considered as a long-term solution, which cannot be applied to the high number of existing DBS in enterprises.

Prioritization The RISC-style architecture requires a novel, component-oriented realization of DBMS, where the characteristics of each component have to be defined as a mathematical model. A solution that avoids such a radical departure from the existing DBS architecture is the prioritization of queries according to their service class (defined by e.g. the DB2 Workload Manager [CCI+08]). With this approach the DBMS could internally consider the goals of the service class at every stage of statement processing. Simple solutions can employ admission control techniques for this purpose. The framework described by Niu et al. [NMP+06] for instance intercepts all arriving queries before they are executed in the DBS. It then decides which of the intercepted queries are allowed to execute at a certain point in time based on the estimated costs for their execution. However, more sophisticated solutions could also assign resources to queries based on their service class, e.g. the amount of memory in the system buffer that they may occupy or the priority of disk reads.

Simple admission control techniques on the one hand have the advantage that they can be easily plugged on to existing DBMS. On the other hand they completely ignore the possibility of configuration changes in order to meet performance goals. For example, the admission control might prevent the execution of best-effort service class queries due to the risk of violating the response time goal for more important service classes, although one additional index could easily solve the performance problem for all classes. In contrast, considering the service classes' goals throughout the query processing chain would allow a very fine-grained assignment of computing resources to every individual query. This resource assignment would be much more specific than with the existing administration possibilities. On the downside, this prioritization would require a re-implementation of most of the DBS components, and it only covers performance goals like response time and throughput and is not usable for cost goals or availability goals. Furthermore, the prioritization is only reasonable if there actually is a number of distinct service classes with corresponding performance goals.

Context-Aware Autonomic Functions As discussed above, the RISC-style DBMS architecture and prioritization approaches both require a re-implementation of today's technically mature DBMS. For this is not a reasonable approach for existing DBMS, the vendors have instead designed a set of autonomic managers which all automatize a particular administration task or DBS component, and suffer from the problems discussed in Section 2.4.

To overcome these problems, the currently independent autonomic managers would have to become aware of their environment, i.e. of the context in which they operate. For all reconfiguration actions that may cause side-effects, they have to make agreements with those autonomic functions that may be affected by the reconfiguration. The autonomic computing blueprint [IBM05] proposes a hierarchical structure as shown in Figure 3.2 for this purpose. This hierarchical structure corresponds with the typical layered implementation of DBMS (see Figure 3.1). With the hierarchical structure every autonomic function is controlled by its immediate superordinate. For this purpose it must offer an appropriate touchpoint. This touchpoint has to provide a description of the autonomic function's constraints and the environmental resources

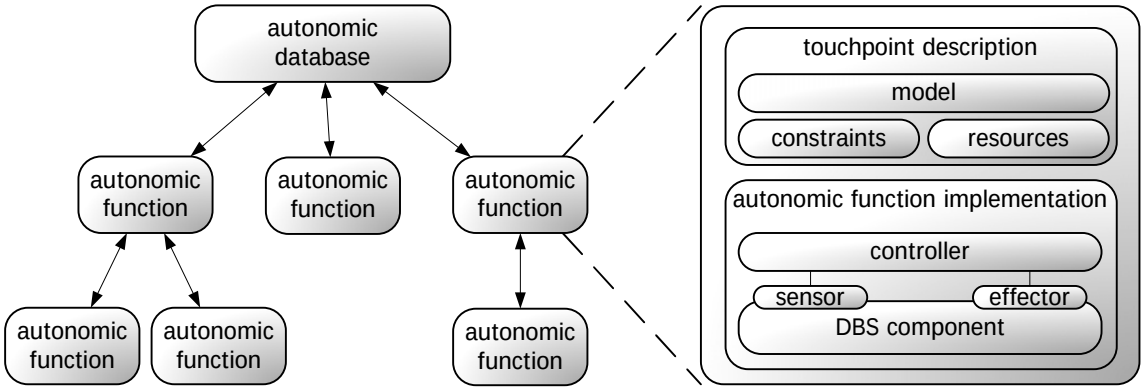


Figure 3.2: Hierarchy of Autonomic Managers

upon which it depends. Based on this information the description should contain a model, which quantitatively predicts the behaviour of the managed resource. The implementation of every autonomic function is required to strictly adhere to the assigned constraints. This way, the consequences of resource sharing amongst the DBS components will be predictable and controllable throughout the system hierarchy.

Although the autonomic computing blueprint recommends a hierarchical structure for the autonomic functions, the topology of the communication between the autonomic functions in principle can be organized randomly in general. For example, the autonomic functions could employ economic models to trade resource usage. Nevertheless, detailed knowledge about the system-wide effects of particular reconfiguration actions is required.

The advantage of the usage of a set of autonomic functions is that many of the existing autonomic functions could be re-used. New autonomic functions could be easily developed and integrated into the DBMS over time. Still, while it is widely accepted that a coordination of autonomic functions is necessary in order to avoid interaction problems (e.g. [WMHZ02], [IBM05], [Rab09]), no actual solutions have been developed towards this goal. One reason for this fact is that the composition of the separated autonomic function descriptions to an overall performance model is a task that is hard to solve in general. A second reason is that the quantitative models of the performance are limited to one particular DBS component. Thus, side-effects on other components (e.g. creating an index or changing the optimizer level) could be not be modelled intuitively.

System-Wide Self-Management The major challenge faced by the approach of context-aware autonomic function is the definition and resolution of dependencies between the autonomic managers. The approach of system-wide self-management [HR07] avoids these problems by designing a single centralized self-management logic, which has an integrated view on all self-management decisions in the DBS. Figure 3.3 illustrates this approach, where a single self-management logic monitors the sensor information provided by the DBS. The sensor information is compared to the goals defined by the DBA. Whenever the sensor information demands a reconfiguration of the DBS, the self-management logic performs a heavy-weight *re-*

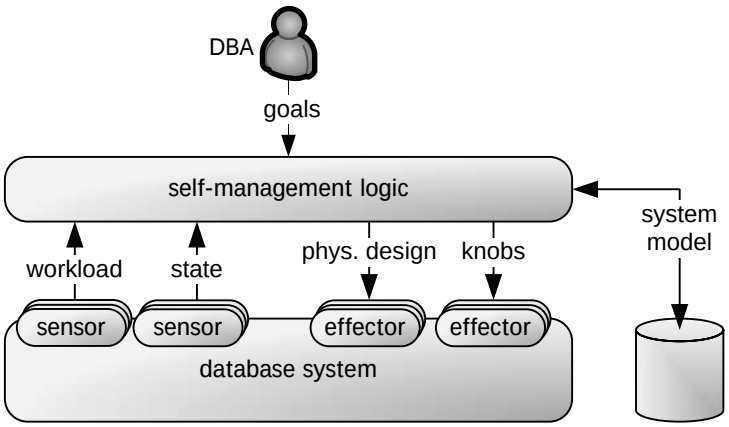


Figure 3.3: System-wide DBS Self-Management

configuration analysis, which determines the optimal effector settings – considering the current state, workload and goal definitions. During the reconfiguration analysis the self-management logic has to consider all dependencies between the possible configuration changes. Following the autonomic computing blueprint [IBM05], the information about the dependencies is assumed to be given in a knowledge base referred to as the system model.

Like the context-aware autonomic functions, the system-wide self-management can be built outside of the DBMS core, i.e. it is not required to re-implement or extend the existing DBMS components. Instead, the self-management logic can be based on the existing sensors and effectors of the DBS. Furthermore, the system-wide effects of reconfigurations do not have to be agreed upon by several independent autonomic functions, but the centralized self-management functions can judge the overall costs and benefits directly. As the self-management logic operates on a system-wide level, it can also connect the individual effector settings to the prediction of how well the high-level goals like response time, throughput, resource costs and availability will be met. So the problems of interaction and goal-independency can both be resolved with a centralized, system-wide self-management approach. In addition, the system model can be used to store quantitative, mathematical models of the behaviour of the DBS in order to avoid overreactions. So although the design of the system-wide self-management approach shown in Figure 3.3 resembles the general feedback control loop design, it does not operate as one. It is instead intended to find the adequate configuration settings in a single (yet heavy-weight) analysis run.

It is important to note that all of the self-management architectures discussed above are valid approaches towards the goal of a truly autonomic DBMS. However, as discussed above, the general design of the system-wide self-management approach inherently avoids the problems of interaction, overreaction and goal-independency. It has therefore been chosen as the guideline for the DBS self-management concepts developed in this work. But even with this guideline, many open challenges still have to be solved in order to create an autonomic DBS that meets the requirements towards truly autonomic systems:

- *Lightweight Workload Monitoring:* The workload has major influence on the required DBS configuration. To avoid latencies and unnecessary resource usage costs, it would be required to permanently check whether or not the current configuration is still appropriate for the current workload. However, with the heavy-weight reconfiguration analysis this permanent analysis would cause an unacceptable overhead. For this reason a lightweight solution for the adaptation of the DBS configuration to the workload is required.

- *Anticipatory State Monitoring:* Even when the workload of a DBS is constant, the internal state of some components may require administrative actions. For example, the fragmentation of the data over time may require a reorganization, the statistics of the optimizer may have to be updated, or old data may have to be archived because of disk capacity problems. The state will typically show a trend over time, which should be observed by the self-management logic. Upcoming maintenance operations should then be scheduled in time before problems actually appear and for time periods when little workload is expected on the DBS.

- *Reliable Service-Level Management:* The goals for the different service classes defined by the DBA should be met by the DBS under all conditions. Not only should the self-management logic therefore consider the goals as thresholds and start a reconfiguration analysis when the thresholds are exceeded, but it should try to detect possible goal violations as early as possible. Only then the reconfigurations can be triggered in time in order to avoid goal violations.

- *System Model Definition:* The system model is supposed to provide all the information about the performance of the DBS under all possible configurations. Furthermore the model has to consider the workload situation. In order to build such a model, the precise requirements about the contents of the model have to be identified. Furthermore, an approach for actually deriving the quantitative behavioural information from a particular DBS must be developed, and a solution for mapping the behaviour to overall goal values has to be found.

- *System Model Evaluation:* At runtime the system model has to be evaluated in order to derive DBS configurations that meet the goals defined by the DBA for the current workload and state. Ideally, the self-management logic should not just find any configuration that meets the goals, but the configuration that exceeds the goals as far as possible while minimizing the resource usage.

This work focuses on the challenges *Lightweight Workload Monitoring*, *System Model Definition*, and *System Model Evaluation*, because these impose the most interesting research questions. The requirements and the general design principles for these components are described in the following Sections 3.2, 3.3, and 3.4. The subject of state monitoring is excluded from this work, because the execution of maintenance functions is a regular task for a DBA, which

(unlike the workload) does not require a continuous monitoring activity. Goal definitions are seen as simple thresholds in this work, i.e. no anticipations about possible future goal violations are made. This approach allows the development of fully functional solutions for the system model definition and evaluation, whereas techniques for the reliable service-level-management can be examined in the future.

3.2 Workload Monitoring and Analysis

The DBS workload, which provides information on how a particular DBMS is actually used in its environment, has major influence on almost all reconfiguration decisions in a DBS. For example, the access paths should optimally support the typical query structure, and tablespaces of frequently accessed tables should be adequately partitioned across multiple physical devices. In addition to these physical design decisions, many values of configuration parameters also depend on the workload. The bufferpool size should match the working set of the application, the optimizer level should be set according to the application type (OLTP or online analytical processing (OLAP)), and the number of concurrent transactions must be limited to avoid lock contention for update-intensive workloads, for example. Additionally, maintenance operations like backups and statistic updates should be scheduled for periods when there are only few other activities. So there is a strong need for an autonomic database to continuously adapt the DBS configuration to the workload.

The straight-forward approach for the continuous adaptation to the workload is to continuously execute a reconfiguration analysis in the self-management logic for the current workload. Whenever the result of the reconfiguration analysis would mandate a configuration change, then the new configuration could be set-up immediately. Although possible, this straight-forward approach would cause an unreasonable overhead, because the reconfiguration analysis is a complex task that may take a significant time to complete. For instance, the approaches to index selection in today's DBMS ([ZZL+04], [DDD+04], [ACK+04]) are based on a workload analysis. The required analysis is so heavy-weight that they cannot be used for continuous analysis, but must be explicitly triggered as an offline-tool by the DBA when he expects it to be necessary.

This work therefore has developed a different approach for adapting the DBS configuration to the workload. It is based on the observation that the workload presented to an enterprise database is not random, but determined by the applications that are using this database in the company's IT landscape. Especially OLTP and reporting applications typically operate on a fixed set of static SQL or statement templates. Thus, the structures for DQL and DML operations are fixed and only parametrized with the actual attribute values at runtime. Although for OLAP applications this is not the case (because the SQL statements directly depend on the user navigation through the data cube) it is likely that users will prefer certain evaluation directions. Hence, in an enterprise database scenario, the database workload typically is composed of a large set of fixed SQL statements that only differ in the actual attribute values, and a small set

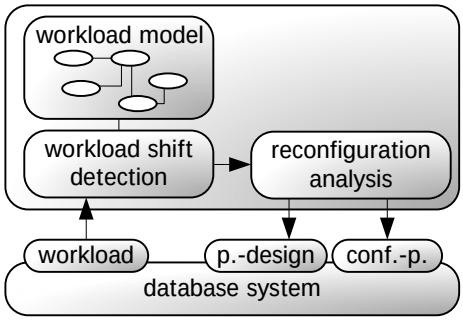


Figure 3.4: Two-staged workload analysis

of statements that are user-generated. As long as the composition of this workload is fix, there is no need for a database to perform extensive analysis for re-configuration possibilities. The situation where the fluctuations in the workload are small and do not require self-management activities are referred to as a *stable workload*. For a database that is well-tuned for its current workload, it is only important to know when there is a change in the workload. A significant change in the workload that could possibly require re-configuration is referred to as a *workload shift*.

The resulting two-staged workload analysis is illustrated in Figure 3.4. The first stage only performs a light-weight *workload shift detection*. For this purpose it maintains a *workload model*, which provides a description of the typical workload of the DBS. The workload model represents the workload that the current configuration has been determined for. Only when a shift actually is detected, a heavy-weight *reconfiguration analysis* actually needs to be performed to determine the benefit of possible re-configuration actions. After a workload shift has been detected, the workload model is replaced with a new model.

Ideally, the lightweight workload shift detection stage should detect exactly those usage change scenarios that actually *do require* a configuration change. However, in order to decide whether or not a configuration change is required the most appropriate configuration for the current workload would have to be determined and compared to the current configuration of the DBS. So the required analysis overhead would be the same as the overhead for the heavy-weight reconfiguration analysis itself. Hence, the first lightweight workload shift detection stage can only detect usage changes that *might require* a reconfiguration of the DBS.

In order to define the requirements of which usage changes should be detected as potential workload shifts, a pragmatic approach has been selected: the scenarios which should be detected are all those scenarios which would usually cause a DBA to check the configuration of the DBS. In particular, the installation of a *new client application*, the deployment of a *new application version* (with a changed set of SQL statements), and *obsolete applications* should trigger a workload shift alarm. Also a *usage change* of an application can impose a significant shift to the composition of the overall database workload, e.g. if the number of users for an application increases by a magnitude. In addition to these single event shifts, also *periodic shift scenarios* should be identified: in a Data Warehousing application for example, the workload during

daytime is typically made up of complex query statements, whereas at night it consists of mass update operations. Or, as an example for a longer-term periodic workload-shift, the end-of-month reporting of an OTLP application database will generate a totally different workload than the usual operation. Of course, the entire workload shift detection mechanism must be sufficiently robust against minor fluctuations in the workload to reduce the probability of false alarms. The concepts and techniques that have been developed for detecting these usage changes are described in Section 4 in detail.

3.3 DBS System Models

The system model is supposed to serve as a knowledge base for the system-wide self-management logic. On the one hand, this knowledge should include a *structural description* of the managed DBS, e.g. its available sensors and effectors and the way they can be accessed. Maintaining this structural information about the DBS in the system model has the advantage that the self-management knowledge itself does not have to be adapted with every new release of the DBMS that adds a relevant sensor or effector. On the other hand – and more importantly – the system model must allow to predict the performance of a DBS under a particular configuration and its current workload. Chaudhuri and Weikum state this problem as the ability to predict

$$workload \times config \rightarrow performance \tag{3.1}$$

in [CW06]. With the ability to predict the performance from a given workload and configuration, the self-management logic has the required knowledge to find an adequate configuration for a certain performance goal.

In order to provide solutions to the problem of type 3.1, quantitative *behavioural descriptions* of the DBS are required. Thus, the system model must comprise adequate mathematical models for this purpose. An example for this kind of model is given in [BK09]. To approximate the response time of a DBS, the authors predict the sum of I/O time caused by all cache misses. The number of cache misses is derived from the estimated hitratio of the DBS system buffer, which is modelled as:

$$hitratio_{buff}(x) = 1 - e^{char_{buff} \cdot x} \tag{3.2}$$

where x denotes the size of the system buffer and $char_{buff}$ refers to the characteristic of the system buffer. The characteristic of a bufferpool reflects the locality of the workload and can be determined by observing the hitratio for particular bufferpool sizes. The predicted hitratios for different values of $char_{buff}$ are illustrated in Figure 3.5. With the mathematical description in Equation 3.2 and the observed workload characteristic, a self-management logic could predict the hitratio (and therefore the performance) of the system buffer component for different sizes.

The above example has illustrated the use of a quantitative model to predict the performance of a DBS component. Still, in order to predict the performance of the overall DBS, many

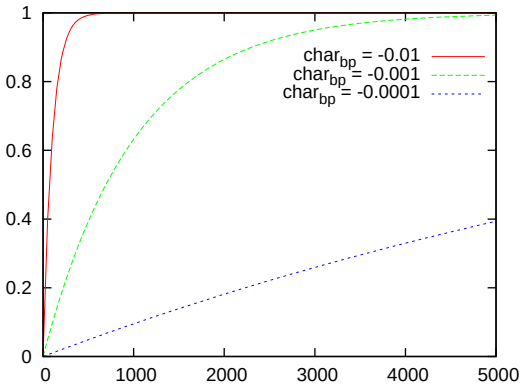


Figure 3.5: Expected Hitratios for different Bufferpool Characteristics

models are required. These models have to be appropriately interconnected to represent the interdependencies that exist in the DBS, e.g. caused by resource sharing. But due to the complexity of DBS architecture, an exact system model that precisely describes the performance of a DBS under all possible configurations and workload conditions will never be attainable. However, as noted in [WMHZ02], even an approximate, coarse-grained model would provide great benefits over the existing trial-and-error approaches of feedback control loops without models of their managed resource.

Because of the huge analysis effort for modelling the behaviour of a DBS, the selected approach towards creating a system model for autonomic DBS is to create a separate model outside of the self-management logic. The rationale behind this approach is that in a first step a very coarse-grained model can be created and tested. This model can be restricted to the most important components, a subset of their actual sensors and effectors, and an approximated quantification of their behaviour. Only after this model has proven to sufficiently well predict the system behaviour under different configurations and workloads, it can be refined incrementally without having to adapt the self-management logic itself. Thus, the system model approach provides an excellent basis for investigating the adequate level of detail knowledge that has to be available in the self-management logic. In order to support this incremental refinement of the system model a graphical modelling language (SysML) has been selected for defining the system models. Chapter 5 illustrates the usage of SysML for creating a system model for IBM DB2. It also proposes an approach for deriving a mathematical model using experimental evaluations.

3.4 Self-Management Logic

The behavioural descriptions in the system model describe the performance of the DBS under particular configurations and workloads. It is the task of the self-management logic to evaluate these descriptions in order to determine DBS configurations that meet high-level goals set-up by the DBA. A prerequisite for this task is that the information in the system model relates the sensors, effectors and behavioural descriptions to the expected goal values. This work therefore

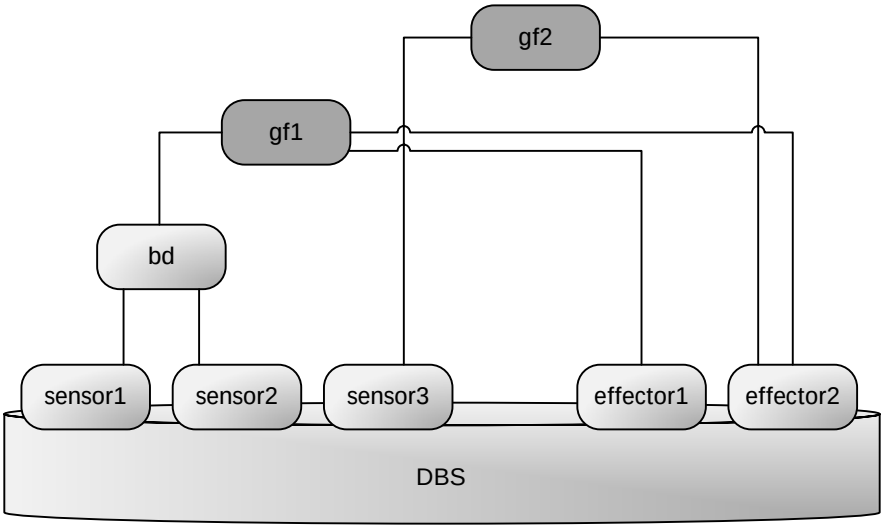


Figure 3.6: Defintion of Goal Functions in the System Model

defines the notion of *goal functions*, whose result values correspond with the high-level goals that may be defined, e.g. response time, throughput and costs. The variables in the body of the goal functions must be mapped to either terminal values, i.e. sensors and effectors, or to other behavioural descriptions that are finally mapped to terminal values again. Figure 3.6 illustrates this composition of goal functions as an abstract example with two goal functions *gf1* and *gf2*. As can bee seen from the figure, the values of *gf2* depends on the value of *sensor3* and *effector2*, whereas *gf1* depends on the result of the behavioural description *bd* (which again depends on *sensor1* and *sensor2*), and on both effectors. From the model shown in Figure 3.6 the self-management logic therefore can easily determine that chaning *effector2* would have effect on both goal functions.

At runtime, the self-management logic has to evaluate the goal functions by parametrizing them with the current values of the DBS sensors. For this purpose it has to extract all the required information about the (possibly composed) goal functions from the system model. Having composed and parametrized the goal functions, the self-management has to find values for the effectors referenced in the functions which meet the goal values. Moreover, to avoid unnecessary reconfiguration runs, the self-management logic should find effector settings which meet the goals in the best possible way. Thus, the problem of finding the appropriate DBS configuration is an optimization problem, where the function values have to be minimized (e.g. response time, operation cost) or maximized (throughput, availability). It is important to note that these goal functions may be opposing: using additional disks may reduce the response time, whereas it increases the operation cost, for example. Similarly, the execution of a reorganization maintenance function may increase the overall throughput, whereas it reduces the availability of the system. So the challenge for the self-management logic is to find an optimal set of effector values for several, possibly opposing, goal functions. Generally speaking, it has to determine an optimal configuration vector $\boldsymbol{x}$ for an objective vector $F(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots f_k(\boldsymbol{x}))$, where $f_1(\boldsymbol{x})$ to $f_k(\boldsymbol{x})$ each define one goal functions. In addition, it also has to take into account the

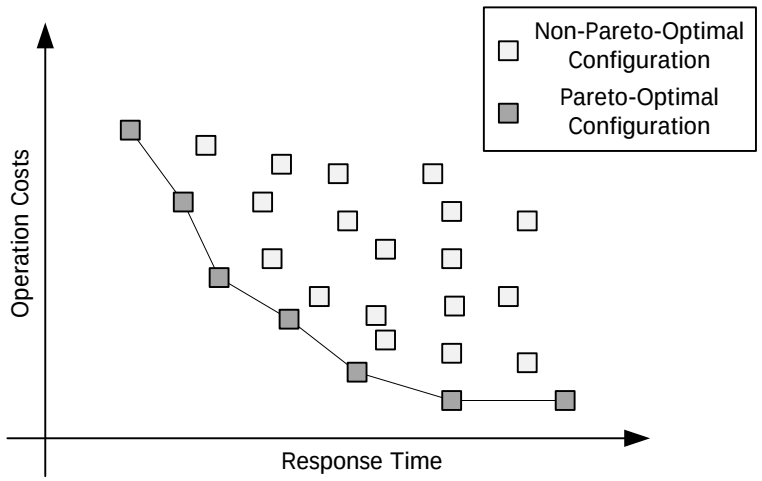


Figure 3.7: Parto-Optimal Configurations for a Minimization Problem with two Goal Functions

set of constraints $E(\boldsymbol{x}) = (e_1(\boldsymbol{x}), e_2(\boldsymbol{x}), \ldots e_k(\boldsymbol{x}))$, which may be defined on the effector values, i.e. on the allowed values for $\boldsymbol{x}$.

Finding a solution to this type of problem is the subject of multi-objective optimization (MOO; e.g. [CVL07]). Techniques of this research area typically determine a Pareto-Optimum P^*, which is a set of optimal trade-offs between the functions in the goal vector $F(\boldsymbol{x})$. A trade-off is considered as optimal when the value of one goal function cannot be enhanced without harming the value of at least one other goal function, i.e.

$$P^* = \{\boldsymbol{x} | \neg \exists \boldsymbol{x}' : F(\boldsymbol{x}') \leq F(\boldsymbol{x})\} \tag{3.3}$$

where $F(\boldsymbol{x}') \leq F(\boldsymbol{x})$ iff $\forall i \in 1..k : f_i(\boldsymbol{x}') \leq f_i(\boldsymbol{x})$ for all objective functions to be minimized and $\forall i \in 1..k : f_i(\boldsymbol{x}') \geq f_i(\boldsymbol{x})$ for all objective functions to be maximized (*dominance*). Figure 3.7 illustrates the Pareto-Optimum for two goal functions (operation costs and response time), where the values of both functions should be minimized. Each rectangle in the plot represents a possible configuration $\boldsymbol{x}$ of the DBS. The Pareto-Optimum is formed by the configurations shaded in dark-grey, because they are not dominated by any of the other configurations.

Chapter 6 discusses the application of multi-objective optimization to the problem of DBS self-management in detail. This includes an introduction to multi-objective optimization techniques, algorithms for extracting and composing goal functions from a system model, and the support for service classes, i.e. for multiple goal values for service classes.

3.5 Conclusions

On the one hand, the system-wide self-management approach followed in this work elegantly avoids the problems of interaction and overreaction by implementing a centralized reconfiguration logic. In addition, only configurations that meet the high-level goals set up by the DBA are derived by the self-management logic. Thus, it also provides a solution to the problem of

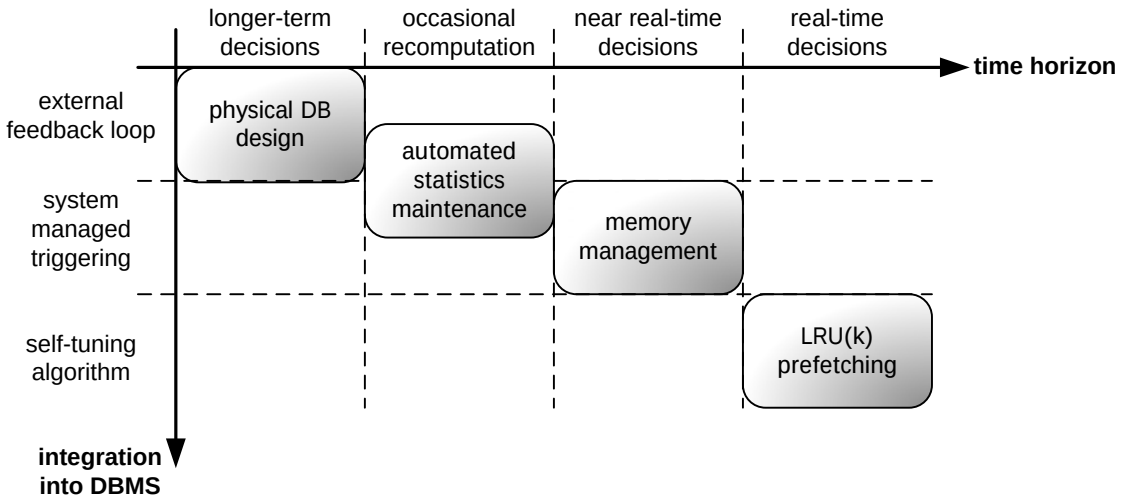


Figure 3.8: The Spectrum of Self-tuning as defined in [CW06]

goal-independency of the existing self-management solutions. On the other hand, the selected system-wide self-management approach comes at the cost of a heavy-weight MOO analysis in order to derive configuration changes for individual components. It is therefore not suitable for self-optimization tasks that require very fast reactions.

In order to asses the self-tuning problem domains for which the selected system-wide self-management approach is adequate it has been compared to the spectrum of self-tuning tasks described by Weikum and Chaudhuri in [CW06]. As shown in Figure 3.8, this spectrum classifies self-tuning tasks by two dimensions: In the time dimension, it distinguishes long-term decisions, occasional recomputations, near real-time decisions and real-time decisions. The second dimension is the required level integration of the self-tuning algorithms into the DBMS. Figure 3.8 also provides examples for the classification of existing self-tuning solutions according to this spectrum.

When comparing the system-wide self-management approach to the spectrum in Figure 3.8, it becomes obvious that in the dimension *integration into DBMS* self-tuning tasks that require a full integration into the DBMS, i.e. a self-tuning request processing algorithm, can not be covered. In contrast, all tasks that can be solved by heavy-weight external analysis algorithms (like physical design tuning), can of course be solved by the centralized self-management-logic, too. Considering the objective of goal-driven DBS self-management, the level *system managed triggering* in the spectrum becomes obsolete: Instead of performing uncoordinated reconfiguration actions to individual DBS components, the reconfiguration analysis should only be triggered when the self-management logic itself detects that goals are missed or that the workload or state has changed significantly.

Similar to the DBMS integration dimension, the level *real-time decisions* can be not be covered by the system-wide self-management approach in the time dimension. In order to assess the applicability of the selected approach for the other levels, the overhead caused by the reconfiguration analysis algorithms has to be known. As will be shown in Section 6.5, the

typical runtime of the MOO algorithms can be limited to one minute. Given an implementation of an anticipatory state monitoring and a reliable service-level management (cf. Section3.1), the system-wide self-management approach therefore can be applied to self-management tasks at all other levels of the time dimension, i.e. up to and including the level *near real-time decisions*.

Although the self-management logic in principle can derive configuration changes from the first three levels at the same time, it might be reasonable to distinguish these problem classes. The reason is that the long-term decisions in Figure 3.8 cover those reconfiguration decisions that cause large reconfiguration costs. The creation of an index, for example, typically is an expensive operation. Thus, when the self-management logic detects that its high-level goals are likely to be missed, it should perform those reconfiguration that imply the lowest configuration change costs. Only if the goals can not be reached, more expensive reconfiguration actions should be taken into account. A discussion of the possible solution selection strategies is given in Section 6.4.

4 Workload Monitoring and Analysis

As discussed in Chapter 3, the DBS workload has major influence on many configuration decisions for a DBS. Following the goal of a self-managing database, the DBS configuration must therefore be adapted automatically to significant changes in the workload. However, a continuous reconfiguration analysis for the currently observed workload would cause an unacceptable analysis overhead. For this reason the following sections present a solution for the lightweight identification of *shifts* in the DBS workload. Only when a significant change in the workload is detected, a heavyweight reconfiguration analysis must be executed.

Section 4.1 first outlines the overall processing model selected for the identification of workload shifts. The individual stages Monitoring, Classification, and Shift Detection of this processing model are then discussed in the following Sections 4.2, 4.3 and 4.4. Afterwards, Section 4.5 describes how the knowledge about the DBS workload gathered during workload shift detection can be exploited in order to predict future changes in the workload. All concepts are evaluated in Section 4.6 and related work is discussed in Section 4.7.

4.1 Workload Analysis Processing Model

The following sections provide an overview of the design of the lightweight workload monitoring and analysis solution. Section 4.1.1 first analyses the key challenges and derives the main processing stages from them. In order to the comprehensibility of the subsequent sections of this chapter, Section 4.1.2 then gives a high-level overview on the overall solution.

4.1.1 Processing Stages

The approach towards building a lightweight workload shift detection component is based on maintaining a *workload model* in the self-management logic (cf. 3.2). This model is intended to permanently store an abstract representation of the database workload. In order to avoid any effort for the DBA, the self-management logic has to create and maintain this model autonomically by monitoring the workload of the database.

For the construction and maintenance of the workload model a set of key challenges can be identified (see Figure 4.1): First, a *workload monitoring* source has to be selected, which provides the relevant DBS workload information. From this workload information, a workload model has to be built (*model creation*), which describes the workload that has usually been observed for the DBS in the near past (*typical workload*). Of course, the model must be as

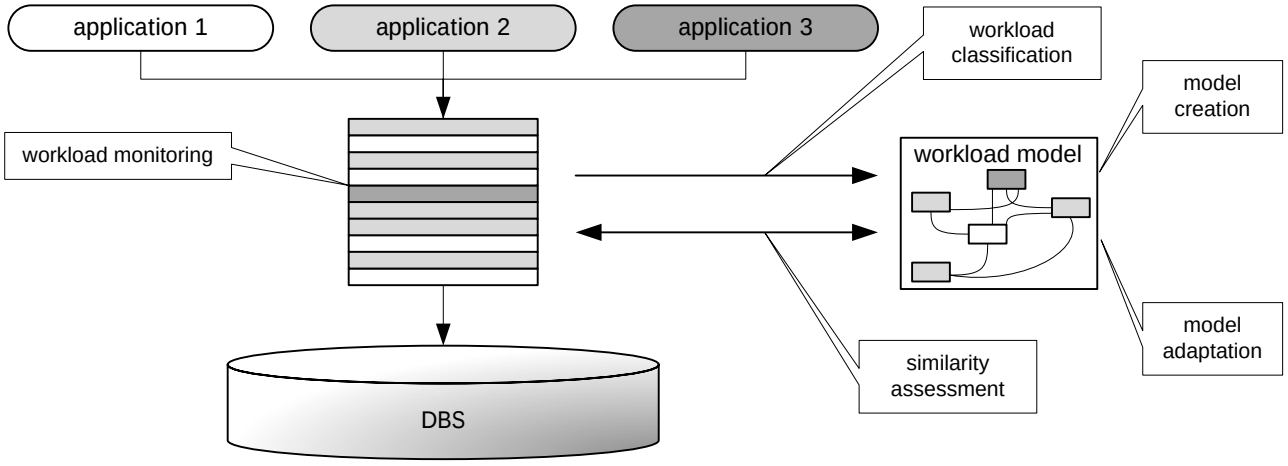


Figure 4.1: Key challenges of workload shift detection

compact as possible to enable a lightweight comparison with the current load, and it should abstract from specific details to provide robustness against minor fluctuations. Hence, an *workload abstraction* should be applied to the workload information before storing it as a workload model. After the model has been created, the current workload can be continuously compared to it in order to detect workload shifts. For this purpose, the self-management logic has to judge how well the currently observed workload matches the workload model (*similarity assessment*). Whenever significant deviations between the actual workload and the workload are detected, a *model adaptation* has to be performed in order to reflect the changes in the workload.

For finding solutions to the above key challenges, a survey of other research areas in computer science has been performed. As a result, the challenges in pattern recognition, especially in speech recognition, have shown to be comparable. Like workload shift detection, speech recognition also requires the assessment of measured data (spoken voice) with the help of (language) models. Hence, the applicability of the well-established processing model developed by this discipline has been investigated for the purpose of workload shift detection [HGR08]. Figure 4.2 provides an overview of the speech recognition processing model as described in [Fin08] and the adaptations that have been developed for DBS workload analysis in this work. The remainder of this section describes the individual stages of this processing model in detail.

The first stage of the speech recognition processing model is the *monitoring* of the raw data. In this stage the continuous input signal is sampled in regular intervals and passed on for further processing in terms of discrete measurands. For workload shift detection, there are many possibilities for monitoring the DBS workload, e.g. operating system metrics or DBS-internal load characteristics. Thus, the first task for realizing a workload shift detection is the selection of an appropriate workload information source. This aspect is discussed in Section 4.2.

In the speech recognition processing model, the monitored information is then preprocessed, e.g. filtered and described in terms of multi-dimensional feature vectors. *Preprocessing* the DBS workload is also reasonable for workload shift detection in order to filter certain types of SQL

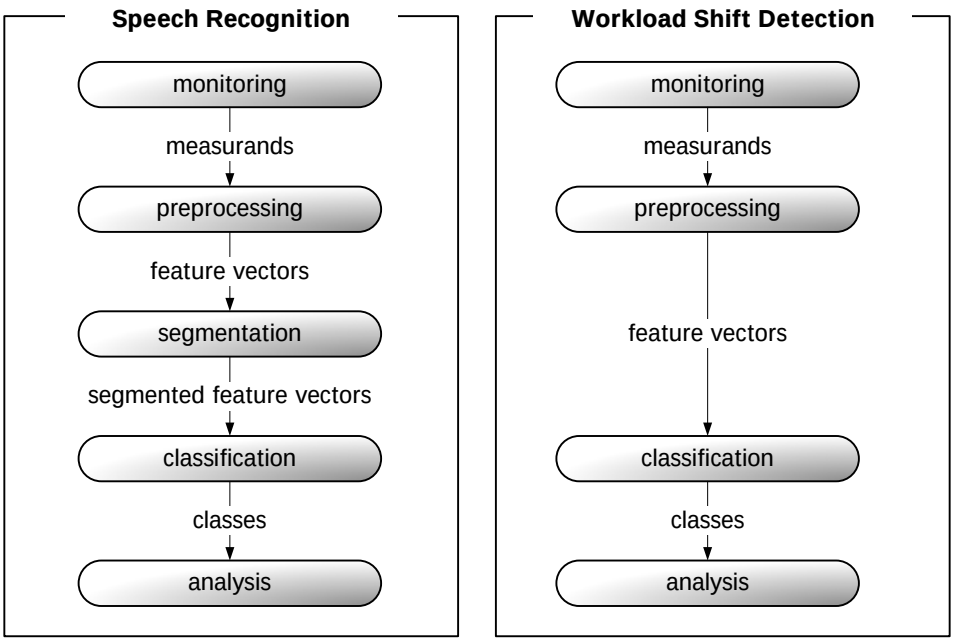


Figure 4.2: Processing Models in Speech Recognition and DBS Workload Analysis

which should not be considered in the model, e.g. DDL statements. The selection of appropriate features from the DBS workload is a more complex task, which is discussed in Section 4.3.

On the stream of preprocessed feature vectors the speech recognition processing model next performs a *segmentation* into meaningful sections, i.e. splitting at the word boundaries. However, for DBS workload the feature vectors already represent the meaningful units of observation, so the segmentation stage can be omitted.

Classification refers to the placing of individual observations into groups according to certain characteristics. For speech, this stage maps the property vector segments to a symbolic representation of the spoken words. Only some characteristics of the recorded speech are evaluated for this purpose (thus allowing the same word to be pronounced/spoken in different ways). Classification is also required for DBS workload in order to fulfil the requirement of a compact workload model. Otherwise every distinct feature vector ever observed would have to be represented in the workload model. However, as classification always entails the loss of information, it is the task of the classification method to keep this loss as small as possible. A solution for the classification of DBS workload is presented in Section 4.3

The last stage of the processing model is the *analysis* of the classified information. In the context of speech recognition this typically is language understanding. For the detection of workload shifts this last stage comprises the comparison of the classified DBS workload with the workload model. Furthermore the entire lifecycle of the model, i.e its creation and adaptation, must be maintained in this stage. The workload modelling techniques and model management concepts that have been developed for this purpose are described in Section 4.4.

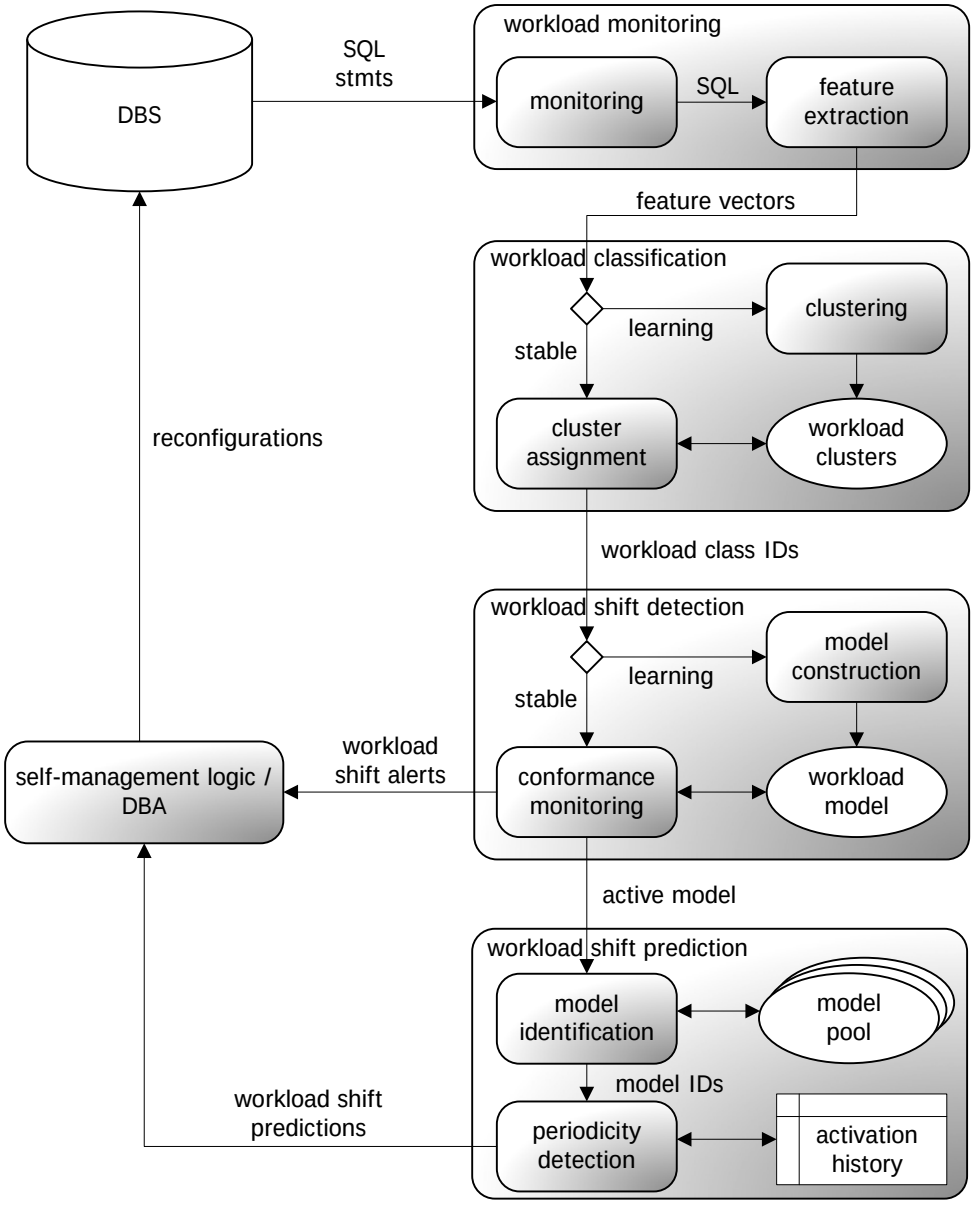


Figure 4.3: High-level Overview of the Workload Monitoring and Analysis Framework

4.1.2 Solution Overview

An overview of the overall workload monitoring and analysis framework is shown in Figure 4.3. Each of the design decisions that lead to the depicted architecture are discussed in detail in the subsequent sections of this chapter. Thus, this overview serves as high-level description of the data flow and is intended to increase the comprehensibility of the following sections.

As shown in the figure, a *monitoring* component of the *workload monitoring* stage continuously observes the workload from the DBS in terms of the SQL statements (see Section 4.2). The *feature extraction* then extracts a set of relevant features from the workload events, e.g. the operation type, the tables accessed, and the grouping requirements (see Section 4.3).

The resulting feature vectors are then passed to the *workload classification*, where similar workload events are grouped (see Section 4.3). *Clustering* techniques are employed in order to

derive these groups in a learning phase. Once the clusters do not change anymore, they are frozen and used for the classification of observed workload events (*cluster assignment*).

The workload class information is then passed to the *workload shift detection* component, which identifies significant changes in the workload (see Section 4.4). For this purpose, it creates a model from the workload classes observed in its learning phase (*model construction*). This model constitutes a statistical description of the typical workload composition. Section 4.4 investigates the applicability of several statistical techniques for this purpose, and it describes the usage of n-gram models and two-window models for this purpose in detail. After the workload model sufficiently well describes the typical workload, the currently observed workload of the DBS is compared to this model in a *conformance monitoring* stage. Whenever the current workload of the DBS does not match the model any more, the workload model is discarded and a new model is learned from the changed workload. In addition, a workload shift alert is raised, which triggers the self-management-logic (or the DBA) to identify a more appropriate DBS configuration for the changed workload.

After a new workload model has been learned, the new model is passed to the *workload shift prediction* component. This component analyses the workload of the DBS for periodic patterns (see Section 4.5). For this purpose it maintains a model pool, which assigns IDs to all workload models that have been observed in the past. Thus, recurring workload models can be identified, and a *periodicity detection* can be performed on the activation times of the models in the pool. The information about the periodicity of workload models is provided to the self-management-logic (or the DBA) of the DBS, where it can be exploited to optimize the configuration of the DBS. For example, the self-management logic or DBA could associate particular DBS configurations with periodic workloads and apply them to the DBS without any analysis effort when the expected workload change is observed.

4.2 Workload Monitoring

An analysis of the existing self-management functions for DBS has shown that various workload information sources are used: Some autonomic features like the IBM DB2 Utility Throttling [PRD+04] analyse the CPU usage, memory usage or I/O activity operating system metrics. Others monitor database-internal metrics of specific sub-components, like the buffer pool hit ratios or optimizer cost estimations. Autonomic memory management ([LNK+03], [SGAL+06]) for example observes the bufferpool hit ratios, and the autonomic statistics collection mechanisms [AHL+04] rely on optimizer metrics. A third type of DBS workload information is the set of SQL statements that is submitted to the DBS. This workload information is typically evaluated by automatic index selection self-management functions, e.g. [ZRL+04] and [ACK+04].

Considering the shift scenarios that have to be identified (e.g. installation of new application, obsolete applications, usage changes; cf. Section 3.2), monitoring at the statement level is

considered as most appropriate. At this level, the requests submitted by the applications can be observed directly. Whenever a new application is installed or an application is modified, the new statement templates will cause a new set of SQL statements. If instead internal DBS metrics from particular DBMS components were used, the presence of a usage change of the DBS would have to be inferred indirectly. Furthermore, the load of the DBMS-components directly depends on the set of observed SQL statements and the centralized monitoring at the statement level imposes less overhead than monitoring several independent sources. The same limitations would exist if the workload was monitored in terms of CPU, memory and I/O-usage. In addition, the analysis results in this case could further be biased by other application processes or maintenance operations running on the same server.

When monitoring the DBS workload at the statement level, every measurement represents one request to the DBS. So the available workload information on the one hand includes all the information that can be extracted from the observed SQL statement texts, e.g. tables accessed, search predicates, and sorting and grouping requirements. On the other hand, also the characteristics about the internal execution of the SQL statements could be considered, especially the information from the execution plans, like the table scan operations, index access, and cardinality estimations. The advantage of taking into account these internal load characteristics for classification is that the resulting workload classes would also depend on the internal state of the DBS. For example, different execution strategies chosen for an SQL query due to data skew or optimizer statistic changes could be distinguished. Furthermore, syntactically different but semantically identical SQL statements could be mapped to the same class by comparing the execution plans.

Despite the advantages, the usage of the internal execution characteristics as workload information is not appropriate for workload shift detection. Instead, only the information that can be observed at the SQL API of the DBS is considered. There are three reasons for this decision: First, the objective of workload shift detection is the identification of significant changes in the *usage* of the DBS. However, considering internal execution characteristics would result in two different workload events for an identical SQL statement, only because the optimizer has changed the access plan (e.g. table scan instead of index access because of updated distribution statistics). Although a changed execution plan definitely affects the effort within the DBS, it does not reflect any change in the usage of the DBS. For this reason the internal state of the DBS should be regarded separately from the workload. Second, in a real-world DBS the workload is not generated randomly. Instead, there typically is a fixed set of database applications, which work with a fixed set of statement templates or static SQL. Hence, the identification of semantically equivalent but syntactically different SQL statements is of minor relevance. Third, by monitoring at the SQL API only the observed workload information conforms to the standardized SQL language. The entire workload shift detection approach is therefore easily applicable to all relational DBS.

Despite the fact that monitoring at the SQL API is the appropriate choice for workload shift detection, the subsequent steps of the processing model are designed to also support other

internal characteristics. Thus, the workload monitoring and analysis approach can also be applied to scenarios where additional information about internal execution characteristics are required.

4.3 Feature Selection and Classification

The representation of all observed SQL statements with distinct statement texts in the workload model would cause an unacceptable model size. For a database with one single table of 100,000 customers that are accessed by their ID, the workload model would comprise 100,000 different SQL statements, for example. Those extensive model sizes on the one hand would thwart the goals of a lightweight workload monitoring and analysis. On the other hand, it would make the workload shift detection sensitive to minor fluctuations or evolutions in the workload, because every minor difference statement text (e.g. a query for a new customer ID) would be considered as a new workload event. Following the workload analysis procedure, all observed workload events are therefore classified, i.e. they are assigned to groups according to certain characteristics [HGR09].

This section describes an approach for the classification of statement-level DBS workload events. For this purpose, it first identifies the requirements to a workload classification in Section 4.3.1. By analysing these requirements, clustering techniques are identified as the most appropriate method for deriving the classification rules in Section 4.3.2. Section 4.3.3 then analyses the features that may be extracted from DBS statement-level workload information and selects an appropriate distance function for them. Using this distance function, Section 4.3.4 finally describes the usage of clustering techniques for the identification of classification rules and the required rule management algorithms.

4.3.1 Classification Requirements

The classification of DBS workload for workload-aware autonomic functions has to regard a number of requirements. Figure 4.4 gives an overview of these functional and non-functional requirements.

Classification The goal of workload classification is the reduction of distinct workload events by placing the events into groups according to certain characteristics. For this purpose it must define *classification rules* for SQL statements, which assign every statement to exactly one class. To hide the diversity of events in the observed DBS workload from the workload shift detection, only the class information must be passed on. A reasonable label should be chosen for the classes, which characterizes the SQL statements assigned to it. This reduction of DBS workload diversity is valuable not only for workload shift detection, but would lead to a significant reduction of the analysis effort for other workload-aware functions, too. For

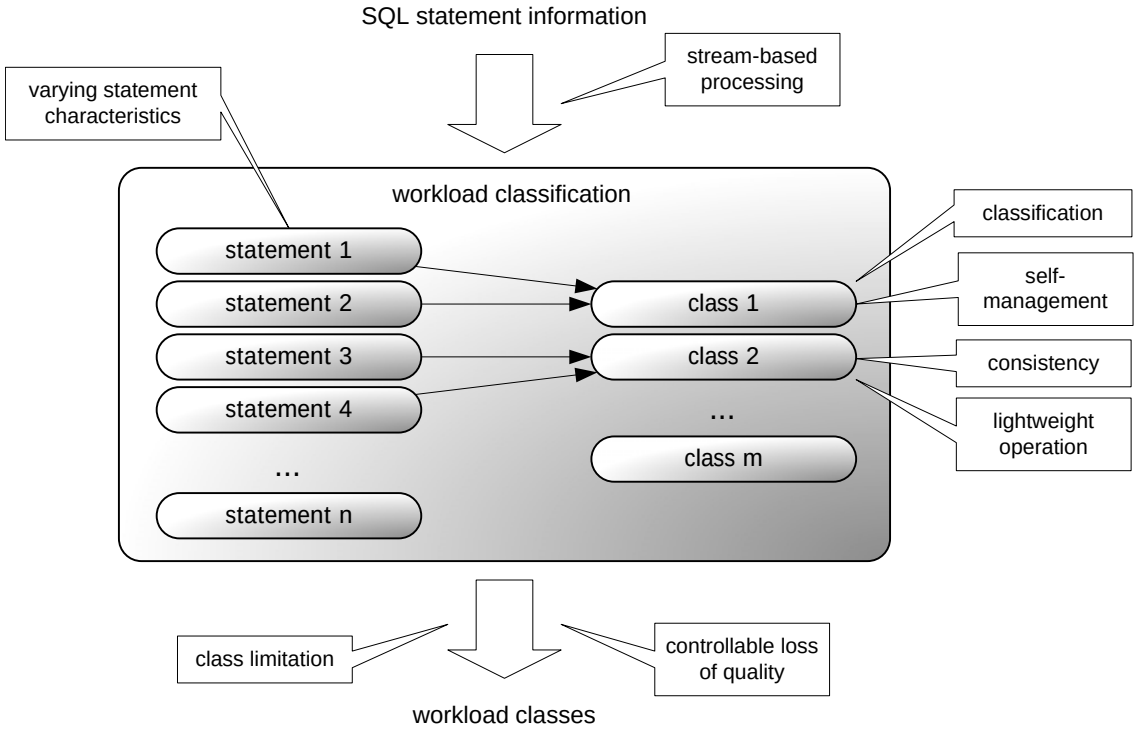


Figure 4.4: Overview of functional and non-functional workload classification requirements

example, the identification of the relevant statistics for the optimizer or the index structure, which support the typical queries in the best possible way, could be accelerated[1].

Varying Statement Characteristics In order to make the workload classification applicable for both internal and SQL-level characteristics, it must be possible to consider different characteristics of the statements and their processing details. Hence, the workload classification must support all types of features that may be selected from this information.

Controllable Loss of Quality Classifying the workload refers to placing the observed SQL statements into groups by considering some characteristics but ignoring others. The information on the ignored characteristics is not available to the subsequent workload analysis function. This missing information necessarily affects the quality of the workload analysis and may even prevent the detection of workload shifts. So to assure that the functionality of the analysis function is not harmed, the acceptable loss of quality must be definable. A measure for the loss of quality induced by the workload classification is therefore required. The acceptable loss of quality should be exploited as far as possible to reduce the analysis overhead.

Class Limitation The workload shift detection requires an upper limit for the number of classes to restrict the analysis overhead. Obviously, the class limitation is relevant only when

[1] In fact, there has been related work on the compression of SQL workload [CGN02] by Chaudhuri et al., which strictly focuses on the purpose of index selection; see Section 4.7 for a detailed discussion.

the loss of quality goal demands more classes than the class limit. Otherwise as little classes as necessary to meet the quality loss goal should be used.

Self-Management Workload classification reduces the diversity of the workload to ensure a lightweight analysis for workload shifts. As such, it is part of a self-management functionality for DBS, which is intended to reduce the maintenance overhead for DBS. So the workload classification may not impose additional effort for the DBAs again, but must itself be completely self-managing.

Stream-based Processing The workload of a DBS is created continuously by the database applications. In order to quickly adapt the DBS configuration to a workload shift, this workload information has to be observed and analysed continuously. So from a workload classification perspective, the workload can be seen as a stream of incoming events, which have to be assigned to classes and passed on to the subsequent workload analysis function. However, the workload shift detection and other autonomic functions typically are executed in parallel to DBS query processing. However, there are no real-time processing requirements so that minor delays (up to several minutes) are acceptable.

Consistency As the workload classification hides the diversity of the original workload, the subsequent workload analysis has no information of the original statements which have been mapped to a class. All analysis is based on the class information only. For this reason, the workload classification must assure the consistency of classification results, i.e. statements of the same type must be mapped to the same class throughout the overall lifetime of the classification. In particular, all changes to the classification rules must be agreed upon with the workload analysis function. Otherwise false workload shifts could be detected, for example.

Lightweight Classification The assignment of observed DBS workload to a class must be lightweight, because it is performed continuously for every SQL statement processed by the DBS. The additional effort caused by the classification should be small in comparison to the effort in the workload analysis function. So most importantly the computation of the classification rules must be feasible for large workloads in reasonable time.

4.3.2 Design

The basic task of the workload classification is to reduce the workload diversity by assigning similar DBS requests to the same class. One option for performing this classification would be the assignment of statements to classes based on the load they cause on the DBS (*DBS-load classification*). For example, all statements resulting in a tablescan operation on a particular table could be assigned to one class. However, as discussed in Section 4.2, the internal load characteristics would mix the usage characteristics of the DBS with its internal state. The

DBS-load classification is therefore not appropriate for the detection of significant changes in the usage of the DBS. Considering the workload shift scenarios that have to be identified (application modifications, usage changes), a more semantic view should be applied instead: similar actions in the application programs should be classified into the same class (*semantic classification*). Unfortunately, an automatic derivation of the required classification rules is not possible, but the DBA would have to manually define the rules instead. The resulting effort would contradict the *Self-Management* requirement for workload classification. For this reason a *statement text classification* has been developed, which assigns statements to classes based on the similarity of the characteristics available from the SQL statement texts.

For realizing the statement text classification, a number of classification techniques can be found in the literature which support the automatic deduction of classification rules, like Bayes classifiers or decision trees ([HK06]). These techniques always require a set of classified training data, from which the classification rules can be derived. To provide the training data, the DBA would have to define workload classes and assign them to a representative set of sample statements manually. The usual classification techniques therefore would entail additional effort for DBAs and are therefore not appropriate for workload classification. In order to support the requirement of self-management, the workload classification rules must be deduced automatically by analysing the available statement information.

In order to classify the workload events based only on the statement texts two approaches have been developed. The first approach, *signature classification*, classifies the SQL statements by replacing all concrete parameter values in the statement text with a wildcard-character. If a class that represents the resulting un-parametrized string already exists, then the SQL statement is assigned to this class. Otherwise, a new class is dynamically created. This simple signature classification has the advantages of being inexpensive and not requiring any DBA configuration effort. On the downside, the number of classes and therefore the resulting workload analysis overhead cannot be known or limited in advance.

In contrast to signature classification, the second approach is directed at allowing a strict limitation of the number of workload classes. As this approach is based on clustering the events based on selected features for identifying the workload classes, it is referred to as *feature classification*. Clustering techniques place similar objects into groups by calculating a distance measure between the data objects. The resulting groups depend on the (previously unknown) structure of the observed data only. The identification of the clusters can be performed automatically. Furthermore, the number of clusters can be limited in advance.

Accoding to the the general discussion in [JMF99], the following challenges have to be taken into account for clustering DBS workload:

- *Data Representation:* Definition of the number, type and scale of the properties (*features*) of the observed statements.

- *Distance Measure:* Definition of a distance function, which quantifies the similarity between two statements based on the vectors of their relevant features.

- *Clustering:* Selection of a clustering algorithm, which assigns statements to classes according to their similarities.
- *Data Abstraction:* Determination of a compact description for the statements assigned to each class, i.e. definition of class labels.
- *Cluster Assessment:* Assessment of the clustering result by an application-specific quality criterion.

From the above steps, two major challenges can be identified: the definition of a distance function for SQL statements, and the design of a stream-oriented clustering solution. In order to quantify the distance between two SQL statements, the distance function cannot apply usual distance metrics like the euclidean distance. The reason is that many features of SQL statements have non-numerical values (like table names, column names, ...). But for the design of the distance function it is important to consider all features, which can possibly be derived from SQL statements. Otherwise the workload classification would not be applicable for a wide variety of workload-aware autonomic functions. Furthermore, not all features may be equally relevant for the distance calculation and so the distance function should allow the definition of weights for the individual features. For example, the tables accessed will be more relevant for index selection than the projection columns.

With an appropriate distance function, the clustering approaches from the literature can be used to identify clusters in the set of SQL statements. These clustering approaches typically expect the total number of clusters (i.e. classes) as an input parameter. Considering the *Controllable Loss of Quality* requirement, this approach is not feasible for DBS workload classification. Instead, the appropriate number of classes must be automatically chosen so that the loss of quality matches the requirement of the subsequent workload analysis function, and the *Class Limitation* is regarded. Furthermore, the usual clustering techniques expect the entire set of data objects as input in order to determine the clusters. However, for DBS workload classification, it is necessary to support stream-oriented processing, which continuously assigns the observed SQL statements to classes. In particular, the total set of data objects is not known completely at any time for workload classification, because the DBS workload may evolve. As long as a previously unobserved statement is similar to an existing class, it must be assigned to that class. But for autonomic functions it is especially interesting to recognize new workload types, because this new workload might require reconfigurations. So the clustering must also be able to create new classes, which represent statements that are significantly different from the statements in existing classes. Despite the additional classes the clustering must assure the *Consistency* of the classification.

As the realization of a signature classification is straight-forward, the following sections focus on solutions to the challenges faced by the feature classification approach only: A distance function for assessing the similarity of SQL statements is developed in Section 4.3.3. Section 4.3.4 afterwards presents the concepts developed for stream-oriented classification of DBS workload.

Table 4.1: Feature Types Applicable to DBS Statement-Level Workload Information

Feature Type	Description	Feature Type Value	Examples
nominal-qualitative	values can be distinguished, but not ordered; arithmetic operations are not applicable	atomic: string value	statement type (SELECT, INSERT, UPDATE, DELETE)
		complex: set of string values	relation/column names referenced in a query
ordinal-qualitative	values can be distinguished and ordered; arithmetic operations are not applicable	atomic: string values	isolation level (RC, RU, ...)
		complex: interval of (ordered) string values	table locks acquired (S, IX, X)
discrete-quantitative	discrete numerical values; arithmetic operations are permitted	atomic: numeric	no. of page requests, no. of predicates
		complex: closed interval	lower/upper bound of range queries
continuous-quantitative	continuous numerical values; arithmetic operations are permitted	atomic: numeric	CPU usage
		interval: closed interval	statement execution period

4.3.3 Distance Function

In order to quantify the similarity of SQL statements, a distance metric for their feature vectors is required. Section 4.3.3.1 first analyses the feature types that can be derived from DBS statements. Section 4.3.3.2 then describes a general distance function which is applicable for these types.

4.3.3.1 Feature Types

A general-purpose workload classification must support all different features that may be extracted from DBS workload events. In particular, it must be possible to use non-numerical, i.e. nominal and ordinal, feature types, which are present in SQL statement texts, for example. Table 4.1 provides an overview of the different feature types that are distinguished in [IY94]. For every feature type it additionally gives a short description and an example. As these feature types may not only occur as atomic values, the column *feature type value* defines the types of their complex values.

As discussed in [JMF99], the selection of features heavily depends on the analysis goal and should be subject of an experimental evaluation. Hence, a complete list of all possible features for DBS workload events cannot be given. Instead, the examples in Table 4.1 illustrate that all these feature types must be considered in the distance metric, because there *might* be features of these types. An overview of various characteristics that might be selected as features for SQL statements is given in [YCHL92]. In the following the selection of workload event features

Table 4.2: Feature Selection for Workload Shift Detection

Feature	F.-Type	Value	Weight
relation names	nom.-qual.	set	0.50
selection columns	nom.-qual.	set	0.25
projection columns	nom.-qual.	set	0.13
statement type	nom.-qual.	atomic	0.06
no. of subqueries	disc.-quant.	atomic	0.03
aggregate functions	nom.-qual.	set	0.01
grouping	nominal-qual.	set	0.02

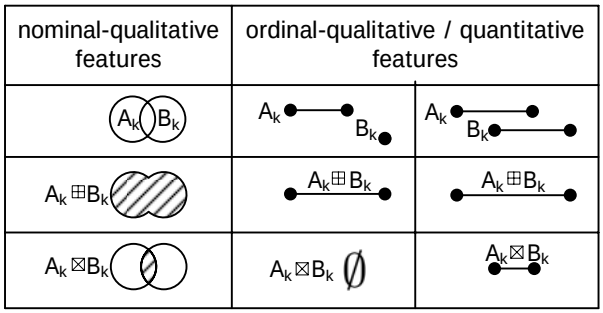


Figure 4.5: Illustration of Join and Meet Operators

for the workload shift detection is discussed. An overview of the SQL statement text features that have provided good results in the experimental evaluation of the workload shift detection solution (see Section 4.6.2) is given in Table 4.2.

4.3.3.2 Distance Metric

Usual distance metrics like the Euclidean distance cannot be used on the nominal and ordinal feature types of the DBS statement-level workload information, because arithmetic operations cannot be applied. Hence, the *generalized Minkowski metric* by Ichino and Yaguchi [IY94] is used instead. It is based on the *cartesian space model* $(U^{(d)}, \boxplus, \boxtimes)$, where $U^{(d)}$ defines a d-dimensional feature space for all feature types described in Table 4.1. Nominal-qualitative feature type values are generally represented as sets, whereas the other feature types values are intervals.

The distance metric on the events in $U^{(d)}$ is based on the two operators $\boxplus$ ("join") and $\boxtimes$ ("meet"). For two events A and B, the result of the join $A_k \boxplus B_k$ of their feature values depends on the type of the features: For quantitative and ordinal-qualitative feature types, it computes the closed interval $A_k \boxplus B_k = [min(A_{k_l}, B_{k_l}); max(A_{k_u}, B_{k_u})]$, where A_{k_l} and B_{k_l} are the lower bounds of the intervals and A_{k_u} and B_{k_u} their upper bounds. For nominal-qualitative feature types, it computes the union of the feature values, i.e. $A_k \boxplus B_k = A_k \cup B_k$. The $\boxtimes$ operator calculates the intersection of the feature values, i.e. $A_k \boxtimes B_k = A_k \cap B_k$ for all feature types. Figure 4.5 illustrates the semantics of the $\boxplus$ and $\boxtimes$ operators by giving examples for the feature types.

Based on the operators $\boxplus$ and $\boxtimes$, [IY94] defines the distance ϕ between two nominal-qualitative features A_k and B_k as

$$\phi(A_k, B_k) = |A_k \boxplus B_k| - |A_k \boxtimes B_k| \tag{4.1}$$

where $|A_k \boxplus B_k|$ and $|A_k \boxtimes B_k|$ denote the number of elements in the sets. The value of $\phi(A_k, B_k)$ therefore is the number of elements, which A_k and B_k do *not* have in common. For quantitative and ordinal-qualitative features, the distance definition in Equation 4.1 would only consider the outer distance of intervals. Hence, the distance for these feature types is defined as

$$\begin{aligned}\phi(A_k, B_k) =& |A_k \boxplus B_k| - |A_k \boxtimes B_k| + \\ & \gamma(2|A_k \boxtimes B_k| - |A_k| - |B_k|)\end{aligned} \tag{4.2}$$

where γ is the weight for the inner distance of intervals in the distance and may be chosen between 0 and 0.5.

Using $\phi(A_k, B_k)$, [IY94] defines the generalized Minkowski distance d_p for two events A and B as

$$d_p(A, B) = \left[\sum_{k=1}^{d} \left(c_k \left(\frac{\phi(A_k, B_k)}{|U_k|}\right)\right)^p\right]^{\frac{1}{p}} \tag{4.3}$$

where $|U_k|$ denotes the number of possible values of the feature k (the interval length for continuous-quantitative features). Dividing the feature distance $\phi(A_k, B_k)$ by $|U_k|$ normalizes the feature values. This avoids an implicit weighting of the features by choosing different dimensions (e.g. hours instead of seconds). Thus, the feature distances are normalized to the interval $[0; 1]$. The normalized, dimensionless feature distances may instead be weighted by the weight coefficient c_k, where $\sum_{k=1}^{d} c_k = 1$. As for the usual Minkowski metric, parameter p defines the order of the metric.

The generalized Minkowski metric for mixed feature-types builds a sound basis for the quantification of DBS workload event similarity, because it supports all relevant feature types from Table 4.1. It therefore fulfils the requirement of *Varying Workload Characteristics*. However, in order to be used for workload classification, it is also necessary to meet the other requirements defined in Section 4.3.1: *Self-Management*, *Stream-Based Processing*, *Consistency*.

The *Self-Management* requirement demands that all configuration parameters of the distance metric must be either set automatically or to a fixed value. All parameters, i.e. the feature dimensions $k = 1..d$, the feature weights w_k, and the Minkowski order p depend on the goal of the workload analysis only, but not on a particular DBS environment. Hence, they can be set to fixed values. In Section 4.6 a concrete parametrization for workload shift detection is proposed.

To avoid implicit weighting, the feature distance $\phi(A_k, B_k)$ is divided by the size of the feature domain $|U_k|$ in Equation 4.3. However, due to the *Stream-based Processing*, the size of the feature domain may vary over time. The domain size of a feature representing the names

of the relations, for example, is affected by new relations. Hence, the distance calculation could determine different distance values for SQL statements in this case and so could violate the *Consistency*.

A simple solution to ensure *Consistency* while supporting *Stream-Based Processing* would be to store the distance values between known feature vectors. The domain size could then be changed without affecting the consistency of previous distance calculations. But as an enlarged feature domain size will lead to smaller normalized distance values for previously unobserved feature vectors, the validity of the triangle inequality for the distance metric could not be ensured with this solution. For this reason, the value of the domain sizes $|U_k|$ may not be changed during workload classification. A simple lifecycle model for the workload classification has been designed for this reason, which freezes all $|U_k|$ after a learning phase (see Section 4.3.4.2). However, an evolving workload may still may cause feature distances $\phi(A_k, B_k)$ greater than the original U_k. So the dilemma of conflicting requirements for consistency and stream-based processing enforces a weakened definition of the normalization constraint, which usually assures distance values in the interval $[0; 1]$. For stream-based workload classification these distance values can only be assured during the learning phase, and may be exceeded afterwards. However, the increased distance values will actually favour the identification of new workload classes. Hence, the increased distances for new SQL statements may be considered as an advantage rather than a drawback.

4.3.4 Classification

Using the distance function described in Sec 4.3.3, clustering techniques can be employed to perform self-managing workload classification. Section 4.3.4.1 describes the clustering algorithm selected, and the quality criterion for the adequate number of classes. The management of the classification rules for stream-based operation is the subject of Section 4.3.4.2.

4.3.4.1 Classification Rules

Partitional clustering algorithms are best suited for DBS workload events, because they can be efficiently implemented and identify isotropic clusters (cf. [JMF99]). In contrast, hierarchical clustering techniques also identify chain-like or concentric clusters and can be used to identify streets or structures in an image, for example. Density-based clustering requires a partitioning of the feature space, which is not possible for the nominal feature types of DBS workload events.

Partitional clustering techniques determine the *centroid* of a predefined number of clusters. A centroid is the representative of a cluster, which is computed as the arithmetic mean of all data items within the cluster. But the mixed-type feature vectors of DBS workload events do not allow the computation of a centroid, because arithmetic operations are not possible on all feature types. Instead, an alternative representative, the *medoid*, has to be chosen from the events in the cluster. A medoid is the event with the lowest distance to all other elements in the cluster.

Algorithm 4.1: Classification Rule Learning

Algorithm: LloydCluster

Input: Workload W, Number of Classes k ($1 \leq k \leq |W|$)

Output: Classes C

```
1  M_current ← selectRandomVectors (W, k);
2  repeat
3      forall the vectors v_i ∈ W do
4          m_nearest ← getNearestMedoid (v_i, M_current);
5          assign (v_i, m_nearest);
6      end
7      M_old ← M_current;
8      M_current ← ∅;
9      forall the medoids m_i ∈ M_old do
10         V_i ← getAssignedVectors (m_i);
11         m_i ← recalculateMedoid (V_i);
12         addMedoid (M_current, m_i);
13     end
14 until medoids = oldMedoids for two consecutive loops;
15 return C ← M_current;
```

Because of its attractive time and space complexity, the k-Means algorithm is the most popular partitional clustering algorithm. The original k-means algorithm by MacQueen [Mac67] selects the first k events as cluster centroids. Each following event is assigned to the nearest cluster, and afterwards the cluster centroid is recalculated. Despite its efficiency (it passes over the events only once) this approach cannot be used for workload classification, because it may violate the *Consistency* requirement: Due to the recalculation of centroids, a workload event after some time may be assigned to a different workload class than before. The iterative variant of the k-Means algorithm described by Lloyd [Llo82] for workload classification has been chosen instead. An overview of the Lloyd-algorithm applied to DBS workload feature vectors is given in Algorithm 4.1. It randomly selects k events from the workload W and uses them as the initial medoids of clusters (**1**). Every event is then assigned to the medoid with the shortest distance (**3-6**), where the distance is calculated according to Equation 4.3. Afterwards, the medoids are recalculated for every cluster (**9-13**). The latter two steps are repeated until the medoids are stable (**14**).

The clusters derived by Algorithm 4.1 constitute the workload classes. Every workload class is characterized by the feature vector of its medoid. All incoming feature vectors can be classified by determining the closest medoid, i.e. the closest class. Together with the distance function, the resulting medoids therefore unambiguously define classification rules.

Like every partitional clustering algorithm, the algorithm by Lloyd requires the number of classes k as a parameter. Several constraints apply to the selection of k: First, the number of classes must be derived automatically (*Self-Management*). Second, the decision on the class

Algorithm 4.2: Number of Workload Classes

Algorithm: Classification

Input: Workload W, Maximum Quality Loss δ $(0 \leq \delta \leq 1)$, Class Limit k_{max}
Output: Classes C

```
1  if |W| ≥ COMPR_RATIO * k_max then
2  |   C ← LloydCluster(W, k_max);
3  |   if q_norm(W, C ) ≥ δ then                          /* see Equ.4.5 */
4  |   |   return C;
5  |
6
7  k_min ← 1;
8  k_max ← min(k_max, |W|);
9  while k_min < k_max do
10 |   k ← (k_min + k_max)/2;
11 |   C ← LloydCluster(W, k);
12 |   if q_norm(W, C ) ≤ δ then                          /* see Equ.4.5 */
13 |   |   k_max ← k − 1;
14 |   else  k_min ← k + 1;
15 end
16 return C
```

number must consider the acceptable *Loss of Quality*. Third, the *Class Limit* must be regarded.

The prerequisite for choosing the right number of classes is a measure for the quality loss caused by the classification. This loss of quality is caused by assigning the workload events to clusters, where the medoids then represent all events in the clusters. So the loss of quality depends on the dissimilarity between the classified event v_i and its medoid m_i, which can be quantified by the distance function d_p in Equ. 4.3. Hence, the overall loss of quality ql_{mse}, which is caused by the classification C of a workload W using k classes, can be computed as the mean squared error

$$ql_{mse}\left(C\left(W,k\right)\right) = \frac{1}{n}\sum_{i=1}^{n} d_p\left(v_i, m_i\right)^2 . \tag{4.4}$$

The quality loss measure ql_{mse} could be used to define a threshold δ for the maximum quality loss allowed. Due to the definition of d_p, the range of values for ql_{mse} depends on the order p chosen for the Minkowski metric: During the learning phase the normalized feature distances take values in the interval $[0;1]$, and the sum of the feature weights is also 1. Hence, assuming the maximum normalized distance value 1 for every feature, the maximum distance for two events is $(\sum_{k=1}^{d} c_k^p)^{1/p}$. For example, the maximum event distance for order 2 and ten equally weighted features is $\approx 0,32$. So the quality loss measure is normalized to the maximum event distance

$$ql_{norm}\left(C\left(W,k\right)\right) = \frac{ql_{mse}}{(\sum_{k=1}^{d} c_k^p)^{1/p}} . \tag{4.5}$$

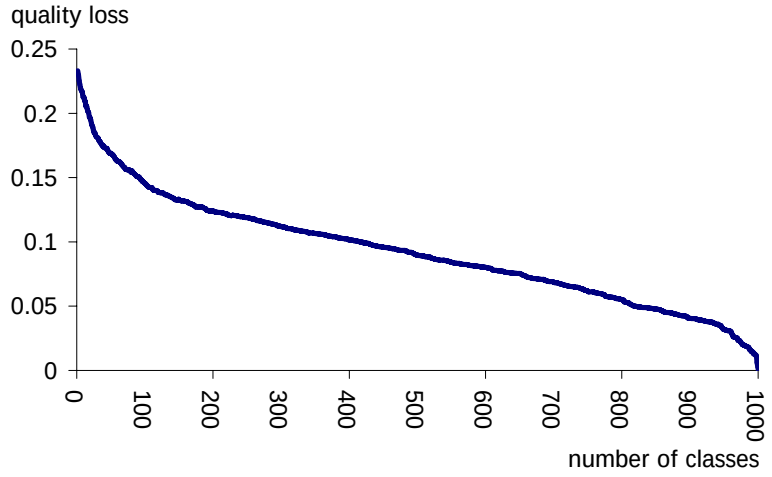


Figure 4.6: Quality loss caused by the classification of 1000 distinct feature vectors

Using ql_{norm}, the acceptable quality loss δ for a workload analysis function can be defined independently from the order of the Minkowski metric. The workload classification then has to find the minimum class number k such that $ql_{norm}(C(W,k)) \leq \delta$. For this task the approach described in [CGN02] is employed by performing the binary search described in lines **7-15** of Algorithm 4.2. It uses the heuristic that the quality loss decreases monotonously with an increasing class number. Figure 4.6 shows a sample plot for the normalized quality loss induced by the workload classification depending on the number of classes (for the classification of 1000 distinct feature vectors). A cluster analysis is performed for the current number of classes in every loop (**11**). Afterwards, the quality loss ql_{norm} for the clustering is compared to δ, and the class number is increased or decreased accordingly. For large workloads it is likely that the quality loss is exceeded even when the class limit is reached. To avoid the effort for a binary search in this case, the algorithm has been extended with an optimization for large workloads (**1-4**). When the workload is by factors (`COMPR_RATIO`) larger than the class limit, then a clustering is performed with k_{max} first.

4.3.4.2 Classification Management

The algorithms in Section 4.3.4.1 expect a fixed list of workload events as parameters. They make no provisions for a self-managed, stream-based workload classification with consistent results. To achieve this goal, a lifecycle for the workload classification has been designed: It is initialized in state *learning*, where the classification rules are learned from the observed workload. Afterwards, the workload classification switches to state *stable*. In this state no changes are made to the classes anymore, i.e. the medoids are frozen. Each element from the continuous stream of events is assigned to the cluster (class) with the nearest medoid. In order to ensure consistency, the workload classification must not switch back from state *stable* to state *learning* on its own account.

Rule Learning

The task of the learning phase is to find a classification for the DBS workload. This includes the decision on the adequate number of classes, and the identification of the cluster medoids, which represent the workload classes. Furthermore, the sizes of the feature domains (cf. Section 4.3.3.2) must be determined. During the learning phase, the observed workload information is not passed on to the subsequent workload analysis function. The major challenge regarding the learning phase is the detection of its end. The goal is to freeze the classification rules as soon as the classification reflects the DBS workload sufficiently well. The obvious solution would be to execute the cluster analysis in short intervals, ending the learning phase when there are no or only little changes in the results. This would cause significant overhead, because each run would require the binary search for the class number.

The approach developed in this work for the detection of the end of the learning phase (see Algorithm 4.3) is based on workload characteristics that can be obtained with less overhead. First (lines **1-5**), the workload is read in regular check intervals ci and buffered for the duration t, where t can be used to define a minimum duration for the learning period. In every interval, the total number of distinct feature vectors observed by then is stored (**4**). After the minimum learning interval t, the algorithm regularly checks the ratio of new feature vectors, which have been added to the observed workload during the last period t (**6-13**). For this purpose, it compares the most current number of known distinct feature vectors (**12**) with the oldest element in the ring buffer (**11**), which holds the number of vectors known time t ago. If the value is above a threshold β, e.g. 5%, the workload observation continues until the ratio drops below β (**7**). Otherwise, the learning phase of the classification rules ends, and the buffered workload is analysed to settle the feature domain sizes. Afterwards, the classification rules are derived from the observed workload using Algorithm 4.2 (**15**), and the classified events are passed on. Thus, the classification rules must be computed only once.

Medoid Distance Classification

After the classification rules have been determined in the learning phase, the workload classification switches to the stable state. In this state, all incoming feature vectors are classified by assigning them to the cluster with the nearest medoid. The classification is performed until the workload classification receives an external order to start a new learning phase because of a workload shift. An overview of the classification of feature vectors is given in Algorithm 4.4. It shows that the class of an observed feature vector v_{obs} is determined as the medoid with the shortest distance in the set of medoids C (**3**). Only the medoid information is passed on to the subsequent workload analysis function (**17**), whereas the original feature vector is discarded.

Even in the stable phase new feature vectors, which are significantly dissimilar from the existing classes, may be observed because of an evolving workload. Assigning these dissimilar vectors to existing classes would bias the classification result, because they would simply cause a more frequent appearance of the existing classes. But the workload shift detection has a

Algorithm 4.3: Learning Phase

Algorithm: LearningPhase

Input: Minimum Learning Period t, Check Interval ci, New Statement Threshold β $(0 \leq \beta)$

```
1  l ← t / ci ;                                          /* ring-buffer length */
2  for i = 0 to l − 1 do
3  |   W ← W ∪ observeWorkload(ci);
4  |   buffer [i ] ← countDistinctFeatVectors(W);
5  end
6  n1 ← buffer [0]; n2 ← buffer [i ];
7  while n2 > (n1 · (1 + β)) do
8  |   W ← W ∪ observeWorkload(ci);
9  |   i ++;
10 |   buffer [i mod l] = countDistinctFeatVectors(W);
11 |   n1 ← buffer [(i + 1) mod l];
12 |   n2 ← buffer [i mod l];
13 end
14 settleFeatureDomainSizes (W);
15 Classification(W, δ, kmax) ;                          /* see Alg.4.2 */
```

particular interest on new workload types to detect changes (see Section 4.4). So even in the stable phase, it must be possible to add classes in order to reflect new workload event types.

When creating new classes, the requirements of *Consistency* and *Class Limit* must be considered. To obey the class limit, a parameter k_{inc} has been introduced that defines the limit for additional classes in the stable phase (**5**). Ensuring the consistency of the classification is more complex, as the cluster limits are not described explicitly, but only by the existing medoids. For new classes it must be ensured that the feature vectors of other clusters will not suddenly be assigned to the new class (*stolen*), because the distance to the new medoid is smaller. In other words, it must be assured that the distance of all events assigned to the existing clusters is smaller than to the medoid of the new cluster.

In order to guarantee the consistency of the classification results, the maximum distance of all feature vectors that have been assigned to the medoid in the past has been stored as the *radius* of each medoid in a first solution. Given the triangle inequality property of the generalized Minkowski metric, the consistency of classification results can be guaranteed when the distance between the new class and all other classes is at least twice as large as the other classes' radiuses. However, the experiments have shown that this condition is too restrictive, because it fails whenever a new class *might* steal vectors from the exiting classes. Figure 4.7 illustrates this behaviour: No new class can be added for the vector v_1 in this case, although it would neither steal vectors from A nor B. The reason is that the distance to the medoid of class A is not twice as large as A's radius.

To overcome this problem the solution shown in Algorithm 4.4 has been developed. It focuses

Algorithm 4.4: Classification using Medoid Distances

Algorithm: DistanceClassification

Input: Statement v_{obs}, Classification Rules C, Class Limit k_{max}, Additional Classes Allowed k_{inc}

Output: ID of Statement Class

```
1  if m_nearest ← lookup (v_obs) then
2  |  return m_nearest
3
4  m_nearest ← getMedoid(v_obs, C) ;
5  dist ← d_p(v_obs, m_nearest) ;
6  if numberOfClasses(C ) < (k_max + k_inc) then
7  |  if 0 < getRadius(m_nearest ) < dist then
8  |  |  if  ∀ m_i ∈ C :
9  |  |       ∀ v_i ∈ m_i :
10 |  |         d_p(v_i, lookup (v_obs)) < d_p(v_i, v_obs ) then
11 |  |  |  m_new ← addClass(C, v_obs) ;
12 |  |  |  updateRadius(m_new, 0) ;
13 |  |  |  storeStmt(v_obs, m_new);
14 |  |  |  return getId(m_new);
15 |  |
16 |
17
18 if  getRadius(m_nearest ) < dist then
19 |  updateRadius(m_nearest, dist)
20 storeStmt(v_obs, m_new);
21 return getId(m_nearest) ;
```

on the identification of situations where a new class actually *would* steal vectors from other classes. To make this decision, it is required to store all classified vectors and the medoid that they are assigned to (**13**, **20**). Before adding new classes it is then required to check whether the new class would steal a previously classified vector from its medoid. As validating the distance of all previously classified feature vectors for every new feature vector may cause significant overhead, the radius of each class is also stored (**12**, **19**). The prerequisite for creating a new class is that the distance to the medoid, which the observed event would be assigned to, must be at least as large as the radius of the *nearest* medoid (**7**). Only if this condition holds, the distance to all other events is validated (**8-10**). If it passes the validation, the observed vector is added as a new class (**11-14**). Hence, in the example in Figure 4.7, only the distance between v_1 and class B's medoid would be compared to class B's radius. As the validation then would show that no vectors would be stolen from other classes, a new class could be created for vector v_1. Although this solution on the one hand comes at the cost of storing all classified vectors, it on the other hand allows a quick classification of known incoming vectors by performing a quick lookup against a hash map (**1-2**).

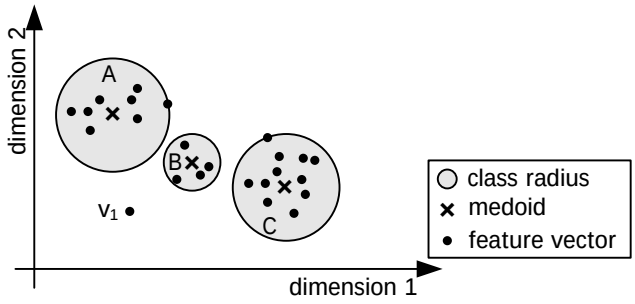


Figure 4.7: Medoid Distance Classes

Bounding Box Classification

In cases where a small memory footprint is required, the medoid distance classification described above may not be an adequate solution, because it requires the storage of all classified feature vectors in order to guarantee consistency. Hence, a second classification algorithm has been developed, which on the one hand requires more maintenance overhead, but on the other hand does not require the storage of all classified vectors. This algorithm is referred to as the *bounding box classification*, because it does not assign feature vectors to classes based on the distance to the closest medoid, but based on the bounding box that encloses the vector. In cases where the vector is not enclosed by a bounding box, the bounding box of the nearest class either is extended or a new class is created.

The bounding boxes of the classes are initially computed at the end of the classification learning phase. However, settling the bounding boxes by determining the maximum and minimum values along every dimension for the vectors in a class is not possible for DBS workload events, because nominal feature type values cannot be ordered. Instead, the bounding box of a class k is defined as a set of maximum distance values $b_{k_i}, i = 1..d$ from the medoid along every dimension. The distance values for every dimension can be calculated as the weighted, normalized feature distances

$$b_{k_i} = \max_{v \in V_k} \left(c_k \frac{\phi(v, m_k)}{|U_k|} \right) \tag{4.6}$$

where m_k denotes the medoid of class k, V_k is the set of vectors assigned to m_k during clustering, and ϕ is computed according to Equ. 4.1. So due to nominal feature types, only the maximum distance from the medoid can be calculated, but not the direction. This is illustrated in Figure 4.8, where the bounding box size is symmetric for every dimension.

Knowing the bounding boxes of the classes, the classification algorithm can assign the observed feature vectors to classes. Unfortunately, the bounding boxes of the initial classes identified in the learning phase may overlap in some cases. An example for this scenario is illustrated in Figure 4.8, where the two initial classes A and B overlap. However, in these cases the classes of the feature vectors are still unambiguously defined by the distances to the closest medoid. So in cases where a vector is enclosed by more than one bounding box, the distances to the medoids also have to be computed.

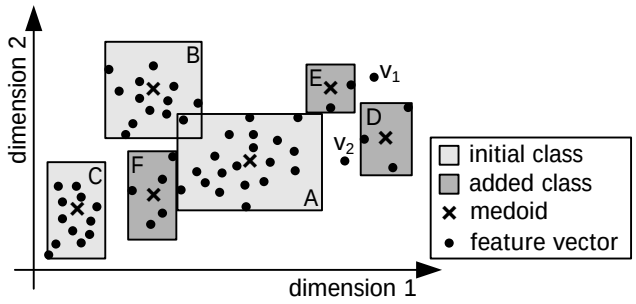


Figure 4.8: Original and Additional Box Classes

Algorithm 4.5: Classification using Bounding Boxes

Algorithm: BoxClassification

Input: Statement v_{obs}, Classification Rules C_{orig}, Add. Classification Rules C_{add}, Class Limit k_{max}, Add. Classes Allowed k_{inc}, Bounds Increase Allowed b_{inc}, Blacklist B

Output: ID of Statement Class

```
1  if size(B) >= size(C_orig) + size(C_add) then
2  |   m_new ← addClass(C_add, v_obs, avgBnds());
3  |   return getId(m_new);
4
5  m_nearest ← getClosestBoxMedoid(v_obs, C_orig, B);
6  if exceedsBounds(v_obs, C_orig ) then
7  |   m_nearest ← getClosestBoxMedoid(v_obs, C_add, B);
8
9  if exceedsBounds(v_obs, m_nearest ) then
10 |   if numberOfClasses(C_orig ) < (k_max + k_inc) and
11 |     growthLimitExceeded(v_obs, m_nearest, b_inc) then
12 |   |   add (B, m_nearest);
13 |   |   return BoxClassification (v_obs, C_orig, C_add, k_max, k_inc, b_inc, B);
14 |
15 |   if overlapsIfAdded(m_nearest, v_obs, C_add) then
16 |   |   add (B, m_nearest);
17 |   |   return BoxClassification (v_obs, C_orig, C_add, k_max, k_inc, b_inc, B);
18 |
19 |   updateBounds(v_obs, m_nearest);
20
21 return getId(m_nearest);
```

By using bounding boxes, ensuring consistency is easier than using the medoid distance classification: As long as the classes do not overlap each other, stealing of vectors is impossible. Thus, the consistency can be guaranteed by two constraints: new classes must not overlap any other class, and extended classes must not overlap any class added in the stable phase.

An overview of the bounding box classification algorithm is given in Algorithm 4.5. For every vector v_{obs}, the closest bounding box from the set of the original classes C_{orig} is determined first

(**5**). If this class does not enclose v_{obs}, then also the classes added in the stable phase C_{add} are searched (**6**-**7**). If the vector is enclosed by any of the classes' bounding boxes, then this class is returned as the result (**21**). Otherwise, it has to be checked whether an existing class must be extended or a new class must be created. A new class is created when the class limit has not yet been reached (**10**), and none of the existing classes may be extended far enough to enclose v_{obs} (**11**)[2]. To check the latter condition, the classification algorithm is called recursively (**13**) with the current class added to the blacklist B (**12**). When all classes have been marked as unsuitable, a new class is created (**1**-**3**).

In case the bounding box of $m_{nearest}$ can be extended to cover v_{obs}, it must be checked that the extension would not cause an overlapping with any of the additional classes C_{add} (**15**). An example for this situation is illustrated in Figure 4.8: the closest class for vector v_1 is E, but an extension of the class would cause an overlapping with D. In this case, E would be added to the blacklist and D (the second-closest class) could be successfully extended in the next iteration. In some rare cases, none of the existing classes may be extended in order to enclose v_{obs}. This scenario is also illustrated in Figure 4.8: neither D nor E can be extended to cover v_2, and also A cannot be extended because it would overlap F. For these rare cases the algorithm has to add a new class (**1**-**3**), even though this decision might violate the class limit. However, it is essential in order to obey the consistency requirement.

4.4 Workload Shift Detection

Using the workload classification concepts described in Section 4.3 the diversity of the DBS workload events (i.e. SQL statements) can be effectively reduced. The reduced diversity enables the creation of a workload model, which describes the typical composition of the DBS workload. Thus, workload shifts can be detected by comparing the currently observed workload to the model.

This section describes concepts for the realization of a workload shift detection component. For this purpose it first discusses the functional and non-functional requirements towards a workload shift detection component in detail (Section 4.4.1). Afterwards, existing techniques from various research areas are surveyed for their applicablility to DBS workload shift detection in Section 4.4.2. The application of the two most promising techniques (n-gram models and two-window methods) to the problem of workload shift detection is then discussed in Sections 4.4.3 and 4.4.4.

4.4.1 Workload Shift Detection Requirements

An overview of the requirements towards the detection of workload shifts for the purpose of self-managing databases is illustrated in Figure 4.4. The following paragraphs discuss these requirements in detail:

[2]The growth limit b_{inc} defines the maximum increase of the classes' original bounds (e.g. 20%).

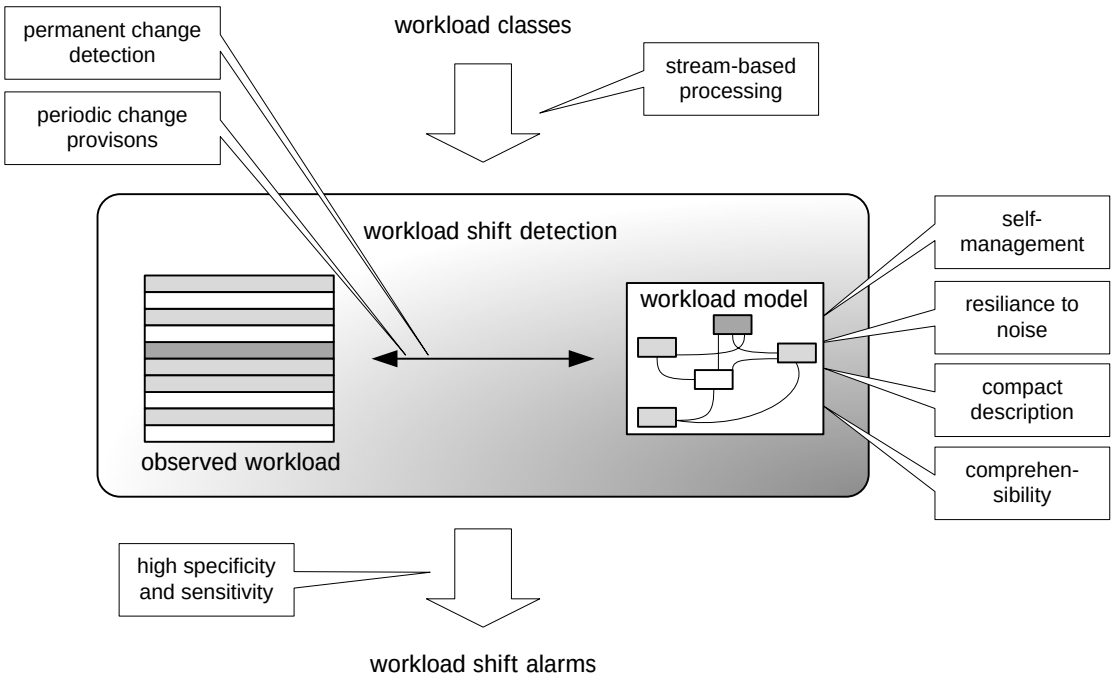


Figure 4.9: Overview of functional and non-functional workload shift detection requirements

Permanent Change Detection As discussed in Section 3.2, the decision of whether or not a particular change in the workload *does* require a change in the DBS configuration is equivalent to computing the required configuration change itself. As this computation is by far too expensive to be performed continuously, the objective for workload shift detection has to be weakened: the shift detection mechanism may identify all usage changes which *might* require a DBS reconfiguration. In particular, the usage changes that should be detected are all those scenarios, which usually require a DBA to re-analyse the configuration of a DBS. So the following scenarios must be detected as permanent workload shifts, because they lead to a permanent change in the database usage:

- *New Applications:* When new database applications are deployed, they will cause additional workload on the DBS. To meet the performance requirements of all applications it may be necessary to perform DBS reconfigurations. For example, it may be necessary to choose a different set of indexes or to adapt the sizes of bufferpools.

- *Obsolete Applications:* A workload that does no longer occur may also require DBS reconfigurations. If for example one database application was undeployed, some physical access structures or bufferpools could become obsolete. Also, the system resources can probably be re-distributed to better support the remaining applications.

- *Modified Applications:* With the installation of new releases, the functionality of enterprise applications usually evolves over time. The resulting workload may change significantly in every release.

- *Application Usage Changes:* Even with a constant number and type of database applications, the DBS workload may change due to the user behaviour. These changes can be caused by the way the database applications are used by the end users. If for example the number of users of a reporting tool doubled, the composition of the overall DBS workload could shift significantly.

Periodic change provisions In addition to permanent shift scenarios, databases often also face periodic workload shifts. Periodic shifts occur when a certain type of database usage occurs in regular intervals. For the workload shift detection, it is important to distinguish between two types of periodic shifts: *Short-term patterns* refer to periodic workload shifts where the interval between the usage changes is small, e.g. batch updates in an 15 minutes interval or hourly reports. In this case, the periodic reconfiguration analysis and execution effort is likely to exceed the resulting benefit. Hence, short-term patterns must not be detected as a workload shift. In contrast, the period length of *long-term patterns* justifies the reconfiguration analysis effort. An example is a Data Warehouse (DWH) scenario, where the DBS is loaded at night, and queried at daytime. These two distinct workloads require completely different DBS configurations. Every change of the workload should therefore be detected as a workload shift.

Stream-based processing The input to the workload shift detection is a continuous stream of workload classes which is produced by the workload classification component. In order to keep the DBS configuration up-to-date when the workload changes, this stream of workload classes should be analysed for shifts near-real-time. However, as for the workload classification there are no strict real-time constraints, because the workload analysis can be performed independently from the actual query processing.

Self-Management The workload shift detection is supposed to reduce the monitoring effort for DBAs by automatically triggering a reconfiguration analysis when the workload of the DBS changes. Hence, the workload shift detection must of course not impose additional administration overhead for the DBAs again, but must itself be completely self-managing.

Resilience to Noise The workload of a DBS can never be expected to be completely uniform. In practice, there will always be minor fluctuations in the workload over time, e.g. because of some exceptional maintenance operations. However, as the workload model holds valuable knowledge about the typical workload over a potentially long period of time, these fluctuations must not immediately cause the workload model to be replaced by a new model. The workload model management must instead provide an adequate resilience to noise.

Compact Description For a lightweight shift detection, the typical workload on the one hand should be represented in a *compact model*. It should therefore employ approximations

and abstractions to reduce the model size, and not simply maintain a complete history of all workload classes ever observed.

Comprehensibility One of the lessons learned from the first autonomic features in commercial DBMSs was that users do not trust in autonomic functionality [LLH+06]. For this reason the workload model should be easily comprehensible to a DBA.

Adaptability The duration and intensity of fluctuations may vary for every DBS environment. It must therefore be possible to adapt the threshold for workload shift alarms to a particular environment. Ideally, the workload shift detection thresholds should be adapted automatically.

4.4.2 Design

For workload shift detection, the workload class instances are compared to a model of the typical DBS workload. The following list summarizes the key aspects that have to be addressed for workload shift detection.

- *Model Learning:* A model describing the typical workload has to be automatically learned. An appropriate, comprehensible modelling technique has to be chosen for this purpose.
- *Workload Assessment:* The actual DBS workload must be continuously compared against the learned model. Metrics must be found that can be used to describe how well the observed workload matches the model. These metrics have to be assessed in a way that on the one hand reliably detects the usage change scenarios, and on the other hand assures resilience to noise.
- *Model Adaptation:* Whenever a significant workload shift has been detected, an alarm should be raised which triggers the DBS reconfiguration. Afterwards, the workload model must be automatically adapted in order to represent the changed workload.

Considering the shift scenarios that have to be detected, it is not required to recognize a single new statement in the workload. Instead, significant changes in the overall composition of statements must be recognized efficiently. Hence, statistical methods are applicable for this task. In order to identify appropriate modelling and shift identification techniques from this area, two distinct viewpoints can be taken: On the one hand, the workload events can be considered as statistically independent, i.e. only the relative frequency of the individual workload events is considered. The design of a solution from this viewpoint is discussed in Section 4.4.2.1. On the other hand, also the context of workload events can be considered (Section 4.4.2.2). The workload events are seen as statistically dependent in this case. This viewpoint takes into account the *behaviour* of the applications, e.g. the typical order of SQL statements in the transactions.

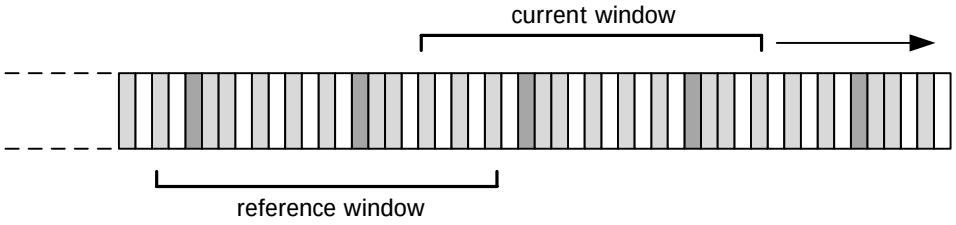


Figure 4.10: Illustration of two-window approach

The statistical approaches outlined above are based on assessing the probability of individual workload events. *Concept shift* detection techniques in contrast are based on characterizing the typical features of events that belong to a data stream. The applicability of this technique for DBS workload shift detection is discussed in Section 4.4.2.3.

4.4.2.1 Frequency Modelling

When considering the workload classes as a sequence of statistically independent events $X = (x_1, x_2, ..., x_n)$, the problem of detecting workload shifts can be generalized to the problem of *change point detection*. This well-known statistical problem is defined as the task of identifying the point in time t_1, when the distribution p_θ of a sequence of independent variables $X = (x_1, x_2, ..., x_n)$ changes from $\theta = 0$ to $\theta = 1$ and θ is a parametrization of the probability distribution p [BN93]. By observing the occurrences of workload classes, knowledge about the relative frequency of the individual workload classes can be gained (p_1). Thus, the historic workload information can be seen as a a probability distribution function of the database workload. In order to detect workload shifts, this distribution can therefore be compared to some reference probability distribution p_0 describing the typical usage of the DBS.

A prerequisite for the detection of workload shifts using the distribution comparison is the definition of the reference probability. Simply using the entire workload history for this purpose on the one hand would cause overhead and on the other hand would not account for possible workload evolutions over time. Instead, the usage of the two-window approach is appropriate, which has successfully been employed in other change point detection scenarios (e.g. [SG07] and [LS08]). With this approach, only the distributions within two sub-sequences of the workload history are compared (see Figure 4.10). The workload in the first window (*reference window*) describes the typical workload of the DBS, i.e. the workload model. Its position on the workload information remains fixed until a workload shift is detected. In contrast, the *current window* is constantly moved over the workload as new workload events occur. Both windows must have the same size. From each of these windows a probability distribution can be computed by counting the occurrences of the workload events.

Using the two-window technique, a workload shift can be detected by comparing the probability distribution in the current workload window with the probability distribution in the reference window. In the statistical literature, a number of techniques for testing observed data against reference distributions can be found [HEK09]. These tests typically compute a

test statistic, which quantifies the similarity of the distributions. However, many of these tests are designed for comparing an observation against a standard distribution like the binomial distribution or normal distribution (e.g. binomial test, χ^2 goodness-of-fit test, t-Test). Others, which support the comparison of arbitrary distributions, require continuously-scaled or ordinal variables and are therefore not appropriate for the nominal workload classes (e.g. the Wilcoxon-Mann-Whitney-Test). From the popular tests, only the χ^2 *homogeneity test* allows the comparison of arbitrary distributions for nominally-scaled variables. The usage of the χ^2 homogeneity test for DBS workload shift detection is discussed in Section 4.4.4.

Other measures for the similarity of two distributions can be found in the area of information theory. These measures are based on the entropy $H(X)$ of an information source X, which defines the average information content of the generated events. The *mutual information* $I(X;Y)$, for instance, is used to quantify the dependence between two information sources X and Y. It is usually employed in order to represent the information obtained through a possibly noisy channel by observing the output [HAH01], and is defined as

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= \sum_X \sum_Y p(x_i, y_i) log_2 \frac{p(x_i, y_i)}{p(x_i)p(y_i)} \ . \end{aligned} \tag{4.7}$$

The term $p(x_i, y_i)$ in Equation 4.7 refers to the joint distribution function of X and Y, which is computed as $p(x_i, y_i) = p(y_i|x_i)p(x_i)$.

If for the purpose of workload shift detection the workload model (first window) was considered as X and the currently observed workload (second window) as Y, the mutual information could be used to quantify their difference. A value of $H(X)$ for the mutual information would then indicate that the probability distributions are identical, whereas a value of 0 would indicate independence between X and Y. Unfortunately, the calculation of the joint probability function $p(x_i, y_i)$ requires knowledge about the conditional probability $p(y_i|x_i)$. This probability can neither be observed nor computed for the DBS workload information and the workload model, because the occurrence of an individual event can in no way be related to an event in the reference window.

For the reasons given above, the mutual information cannot be used for workload shift detection. But with the *Kullback-Leibler divergence* and the *cross-entropy*, information theory provides two other distance measures. In contrast to the mutual information, these distance measures are intended to compare two probability distributions p and q of only *one* information source X. The cross entropy measures the average number of bits required for encoding an event if the coding scheme is based on a distribution q instead of the true distribution p [HAH01]. It is defined as

$$H(p;q) = H(p) + D_{KL}(p||q) \ , \tag{4.8}$$

where $D_{KL}(p||q)$ refers to the Kullback-Leibler divergence [KL51], which in turn is defined as

$$D_{KL}(p||q) = \sum_{X} p(x_i) log_2 \frac{p(x_i)}{q(x_i)} \ . \tag{4.9}$$

Thus, the cross entropy and Kullback-Leibler divergence differ only by the entropy $H(p)$, which is added to $D_{KL}(p||q)$ in Equation 4.8. For the purpose of workload shift detection, p represents the "true" distribution of the data, i.e. the observations of the current DBS workload, whereas q reflects the workload model. So the entropy $H(p)$ does not provide any additional information on the similarity of the observations to the workload model and can be omitted. Hence, Section 4.4.4 focuses on the usage of the Kullback-Leibler divergence for workload shift detection only.

4.4.2.2 Behaviour Modelling

The approaches described in Section 4.4.2.1 assume that the workload events are statistically independent. However, in real-world enterprise scenarios, the SQL statements are issued by a fixed set of database applications. These applications implement a number of database transactions, which are composed from a fixed number of statement templates or static SQL. The probability of a workload event therefore may depend upon the events that have occurred before. So the context in which a statement appears can also be taken into account for the description of the typical DBS workload, i.e., the workload events can be considered as statistically dependent. Considering conditional probabilities in the approaches selected for frequency modelling in Section 4.4.2.1 could be seen as one solution for this goal. But the problem that arises in this case is that the probabilities of events might potentially depend on the entire history of previous events. As finding the exactly the correct history length is a challenging task, existing techniques for this purpose have been surveyed. It has been found that two distinct general approaches are commonly taken for solving this problem: On the one hand, time series analysis (Section 4.4.2.2) offers techniques for identifying models of a complete history of events by finding correlations between its values. On the other hand, there are techniques which make simplifying assumptions and limit the history of events. This approach is often used for language models (Section 4.4.2.2), which create approximate statistical descriptions of events with conditional probabilities.

Time Series Analysis

A time series is an ordered sequence of data points measured at discrete points in time. Usually these data points represent the values of a single observed data source, e.g. temperature data observed by a sensor in regular intervals, stock quotes or company earnings. These types of time series are characterized by a correlation between adjacent data points, i.e., their values are not statistically independent. The primary objective of a time series analysis is the identification of mathematical models that describe the observed time series data [SS00]. These models serve

the purpose of identifying trends in the data, and periodic or permanent changes in the data over time. In particular, this knowledge is used in order to predict the future values of the time series.

Several analysis techniques have been developed for the purpose of time series analysis. These techniques can be distinguished into techniques that can be applied to the time-domain representation of the time series, and into techniques for the frequency-domain representation. The time-domain techniques describe the correlation between data points by identifying the dependency of current values on past values in the time series. For stationary time series (where the mean value function is constant and the auto-covariance does not depend on the absolute time), the auto-correlation function exhibits linear correlations at specific time lags. Plots of this autocorrelation function can be used to build auto-regression (AR) models and moving average (MA) models of the time series. AR models express the dependency of the current value on the past values, whereas MA models describe the effects of white noise (i.e. values of a gaussian random variable with mean 0 and a constant variance) observed in the past. In order to apply these analysis techniques to non-stationary time series additional transformations like de-trending or differencing are required [SS00]. The combination of AR and MA models with differencing are referred to as ARIMA models [BJR08].

Time series analysis in the frequency-domain does not analyse the source data directly, but its decomposition into a linear combination of sine and cosine functions. The amplitudes of the frequencies of the sine and cosine functions can be computed using Discrete Fourier Transforms (DFT). Typically, a spectral analysis is performed on the resulting power spectrum in order to identify the dominant frequencies. An analysis in the frequency-domain is especially suited to examine the periodic nature of time series. However, as discussed in [SS00], similar results can be retrieved both in the time-domain and in the frequency domain.

From a self-management point of view, mathematical models describing the workload of the DBS are of course desirable. The typical workload then could be described in a very compact way using AR and MA models and a trend function, for example. These models would not only allow the detection of changes in the workload, but even the prediction of the future workload. Unfortunately, the established time series analysis techniques described above cannot be used for workload shift detection. The reason is that the existing time series analysis techniques all require interval-scaled data items in the time series. They cannot be applied to the stream of DBS workload events, because the DBS workload information is categorical data. The workload classes (or the set of SQL statements that they resemble) can not even be ordered: for instance, workload class 3 cannot said to be “smaller” or “greater” than class 10.

In addition to the classical time-series analysis for interval-scaled data, there has also been research on categorical time series analysis; an overview is given in [Wei09]. However, most of the proposed approaches are adaptations of the classical ARMA model, which require expert knowledge for the identification and parametrization of the models. Their use would therefore violate the self-management requirement of workload shift detection. [Wei09] also proposes the use of markov models for categorical time series, which are subject of the following section.

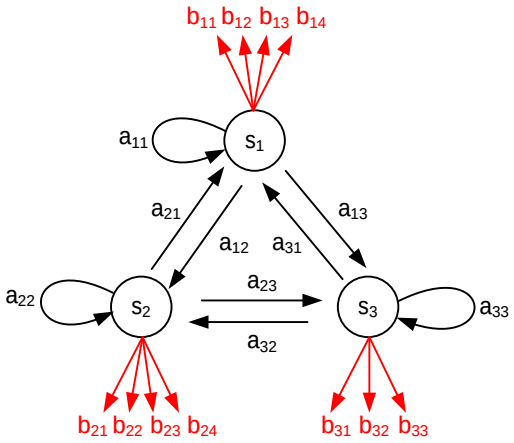


Figure 4.11: Illustration of a discrete hidden markov model

Language Models

The behaviour modelling approach intends to model the workload events issued by the DBS applications and the context in which they appear. This approach could also be considered as modelling the "language" that the applications "speak" towards the DBS. Hence, the concepts available in the area of speech recognition and understanding have been surveyed. In this area models are used in order to recognize and understand spoken voice. One modelling technique commonly used for speech recognition are *hidden markov models* [Fin08]. A hidden markov model, which is illustrated in Figure 4.11, consists of two layers: The first layer is a set of states $S = \{s_i | i \leq N\}$, for which transition probabilities $A = \{a_{ij} = P(s_j|s_i)\}$ are known. Hence, this layer models a stochastic process, where the behaviour of the process depends on the previous state only. In a second layer, the hidden markov models describe emissions $O = \{o_k\}$ that may be generated in each of the states. The emission probabilities depend on the current state only and not on previous states or emissions. Thus, for discrete emissions, the emission probabilities can be described as a set $B = \{b_{jk} = P(o_k|s_j)\}$.

Hidden markov models can be used to model stochastic processes which emit events according to individual probabilities. For a given hidden markov model, the probability of a sequence of observations O can be calculated. In addition, it is possible to infer the internal state changes in the stochastic process. This fact is exploited in speech recognition to perform a classification and segmentation at the same time, because it allows the identification of phoneme and word boundaries. For this reason hidden markov models are commonly used to define acoustic models, i.e. models which allow the identification of phonemes and words from a sequence of measured feature vectors.

If the hidden markov models were used for the workload model, the probability of creating a sequence of workload events from the model could be calculated. In other words, it could be judged how likely it is to see the observed workload if the applications behaved as described in the model. Thus, deviations from the typical behaviour (workload shifts) could be detected. However, there is currently no algorithm that can automatically infer a hidden markov model

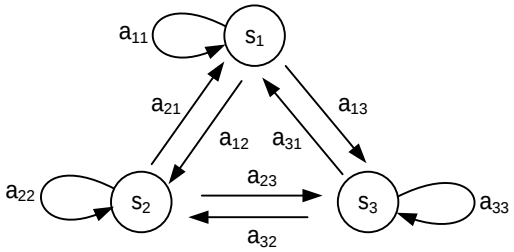


Figure 4.12: Illustration of a markov chain model of order 1

from a training probe [Fin08]. Hence, a basic model describing the typical behaviour of the applications would have to be created by a DBA manually. This manual effort of course violates the requirement of self-management and prevents the usage of hidden markov models. Furthermore, the unique ability of performing segmentation and classification at the same time is not required for DBS workload analysis, because the workload information already has a symbolic representation in terms of workload classes.

Instead of hidden markov models, the usage of *n-gram models* appears much more appropriate for creating a workload model. In speech recognition, n-gram models are used to create language models, i.e. models describing the grammar of a language. The n-gram models are based on markov chains, which can be used to model the behaviour of an event source behaving according to a stochastic process. It is assumed that this stochastic process generates a sequence of events $X = (x_1, x_2, ..., x_t : t \in T)$ in time space T, where each event X takes a value of the state space $S = (s_1, s_2, ..., s_n : n \in \mathbb{N})$. For the stochastic process to be a markov chain it must fulfil the markov property:

$$\begin{aligned} &P(x_i = s_i \mid x_1 = s_1, ..., x_{i-1} = s_{i-1}) = \\ &P(x_i = s_i \mid x_{i-m} = s_{i-m}, ..., x_{i-1} = s_{i-1}) \ . \end{aligned} \tag{4.10}$$

So the markov property defines that the probability of events does not depend on the entire history of previous events, but only on a limited history of length m, which is referred to as the *order* of the markov chain. Markov chains of order 1 are often illustrated as a graph, where the nodes represent the states S and the edges are attributed with the transition probabilities between the states (see Figure 4.12).

The modelling of an exact markov chain model for a DBS workload model would of course require thorough knowledge about the internal processing within the applications. For this reason n-gram models consider the markov property only as an approximation, i.e.

$$\begin{aligned} &P(x_i = s_i \mid x_1 = s_1, ..., x_{i-1} = s_{i-1}) \approx \\ &P(x_i = s_i \mid x_{i-m} = s_{i-m}, ..., x_{i-1} = s_{i-1}) \ . \end{aligned} \tag{4.11}$$

With n-gram models, the order of the markov chains in the n-gram models determines how

well the model approximates the underlying process. In contrast to hidden markov models and markov chain models, n-gram models are therefore automatically learnable and adaptable, i.e., they are adequate for meeting the self-management requirement of DBS workload shift detection. Section 4.4.3 investigates the usage of n-gram models for workload shift detection in detail.

4.4.2.3 Concept Modelling

As discussed in [Rei09], the detection of concept changes [YWZ06] is a research area in machine learning and data mining, which has similar goals as workload shift detection: The detection of changes in a stream of data. For this purpose a *concept* of the data is learned from a set of training data. The training data consists of a set of data items, which are classified into two classes "fitting" and "unfitting". From this training data, the concept identifies selected feature values which cause the data item to belong to one of these classes. For instance, all data items with the features `temperature = 'high'` $\wedge$ `clouds = 'low'` might be classified as fitting. Thus, the concept is a classifier that assigns the incoming data items to exactly two classes based on their feature values.

In order to detect changes in incoming data, most concept change detection approaches examine the ratio of misclassified data items. For this purpose the class of the data item suggested by the learned concept is compared to the real class of the data item. As long as the concept of the data has not changed, the number of misclassifications will be small, and it will increase when there are significant changes in the stream of data items. The approach presented in [NY07], for example, uses statistical tests on the misclassifications to detect concept changes. In contrast, [FHWY04] does not detect misclassifications continuously, but only observes the average number of data items per class over time. Only when there are changes the real class of the classified data items has to be determined.

Applying the approach of concept shift detection to the detection of workload shifts requires the learning of a corresponding concept in the first step. But providing training data with appropriate examples for "fitting" and "unfitting" workload events would impose a huge overhead for the DBA. Hence, the concept can only be learned from the observable workload, which resembles positive examples only. Afterwards, the learned concept can be used in order to classify the actual workload of the DBS and to compute the misclassification rate. However, to compute the misclassification rate it is required to know the correct class of each workload event. For DBS workload shift detection this means that every workload event has to be labelled as either fitting or unfitting *before* the classification. As this is not possible for DBS workload events, the concept shift detection techniques based on misclassification rate analysis are not appropriate for workload shift detection.

In addition to the analysis of the misclassification approaches there are also some concept shift detection approaches which operate on time windows. Their goal is to select a subset of the history of data items that represents the current concept of the data stream as precisely

as possible. For this purpose they select weighted examples (e.g. [MM00]) or time windows of varying size [KJ00]. These history subsets are constantly adapted to the evolving data stream. Obviously, this behaviour is not adequate for workload shift detection, where it is the goal to detect significant deviations from a reference model (not a continuous adaptation of the model).

4.4.3 n-gram Workload Models

This section details on the detection of workload shifts using n-gram models [HR08], which have been identified as a promising technique in Section 4.4.2. It first describes the creation of DBS workload models using n-gram models and afterwards describes the required lifecycle management for the workload model.

4.4.3.1 Workload Modelling

From the perspective of the workload shift detection, the database applications can be considered as a stochastic process generating a workload W. The workload consists of a sequence of workload classes $(w_1, w_2, ..., w_T)$ (cf. Section 4.3). As described in Section 4.4.2, n-gram models model events x_t generated by a stochastic process, which takes values from a state space $S = (s_1, s_2, ..., s_n : n \in \mathbb{N})$. Considering the goals of workload shift detection, these states s_i can be seen as the representation of the internal state of the application when generating the corresponding SQL statement at time t. For the mapping between W and the events X (whose values are represented in the workload model) two possible approaches have been identified:

- *Inference of Transactions:* All workload events that were caused in the same type of transaction are mapped to the same model state, i.e. each model state represents one transaction type in the application programs. The transition probabilities between the states express the user behaviour. This approach requires that the transaction-IDs are passed on together with the workload class information by all previous processing steps.

- *Statement Dependencies:* In this approach each workload event is directly mapped to a model state. The transition probabilities between workload events are computed according to their order in W. Thus, they mainly express the dependency of SQL statements within transactions (this depends on the markov chain length and the transaction lengths).

The inference of transactions has the advantage of resulting in a compact workload model with a small number of states. However, there is usually no information about the type of transaction in the workload W itself. It is instead necessary to identify the different transaction types by analysing the transaction-IDs in historical workload traces. While this is straight-forward for all transactions that consist of a fixed sequence of statements, it is a complex task for procedurally controlled transactions, where the number and type of SQL statements varies depending on loop- and branch-conditions. Yao et al. discuss the challenges of this problem and propose a solution using heavyweight algorithms in [YAH05]. The resulting overhead for transaction

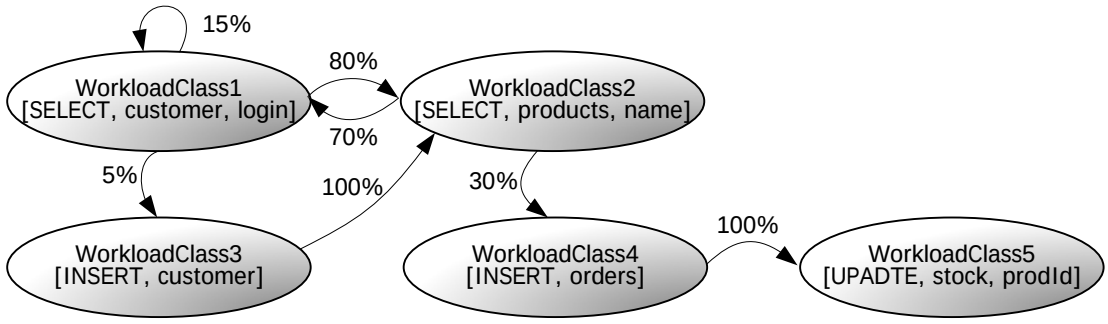


Figure 4.13: Example for DBS-Workload modelled as Markov Chain of Order 1

inference would violate the goal of a lightweight workload monitoring solution that can be executed continuously. Furthermore, the changes in the sequences and probabilities of the transactions that could be additionally detected with this approach are of minor relevance for workload shift detection. More important are the probabilities of SQL statement occurrences, which cause the actual load on a DBS. For these reasons the advantages of mapping the workload events to model states directly prevail, although this approach results in models with more states. Hence, W and X are considered as identical in the following and the workload W directly takes the values of the model states S. Figure 4.13 shows an illustration of such a markov chain workload model using the statement dependencies mapping. In the example shown a markov chain length of 1 is assumed, i.e. the probability of an event depends on its immediate predecessor only. Each state represents a workload class, which is identified by a unique number. The feature vector that each workload class stands for is illustrated, too.

As described in Section 4.4.2, n-gram models consider the markov property as an approximation only. Being aware of the approximation, n-gram models can automatically be learned from training data by counting the occurrences of events:

$$P(x_t \mid x_{t-n}, ..., x_{t-1}) = \frac{count(x_{t-n}, ..., x_{t-1}, x_t)}{count(x_{t-n}, ..., x_{t-1})} \tag{4.12}$$

For the creation and analysis of n-gram models, only sub-sequences with the length n of the events X are considered (n-gram models result in markov chains of order $n-1$). The quality of the model is mainly determined by the choice of n for the n-gram size, which is a trade-off between efficiency and accuracy.

As n-gram models are only an approximation of the real world process, every analysis on the model faces the problem of *unseen events*. Unseen events are events that have not been observed before, either because the event x_t has not occurred yet at all, or because it has not been seen with the history $(x_{t-n}, ..., x_{t-1})$ yet. To prevent these unseen events from being evaluated to a probability of 0, there are a number of techniques described in the literature [Fin08]. For the purpose of workload shift detection a combination of *absolute discounting* [Kat87] and *backing off* [JM80] has been chosen, because it can be efficiently implemented and has been successfully applied to other problem domains before (e.g. speech recognition [Fin99]).

4.4.3.2 Workload Model Assessment

For the detection of DBS workload shifts it is essential to assess how well the observed DBS workload is described by the workload model. In statistics, the metric commonly used to assess the quality of a probability model is the *perplexity*. For an observed workload W_{obs}, the perplexity PP of an DBS workload n-gram model can be computed as:

$$\begin{aligned} PP(W_{obs}) &= (P(w_{obs_1}, ..., w_{obs_T}))^{-\frac{1}{T}} \\ &= \left(\prod_{t=1}^{T} P(w_{obs_t} \mid w_{obs_{t-n}} ... w_{obs_{t-1}}) \right)^{-\frac{1}{T}} \end{aligned} \tag{4.13}$$

For the computation of the perplexity, the probabilities of the individual SQL statements $P(W_{obs_t})$ are taken from the workload model. Informally, the perplexity describes how "surprised" the model is by the observed workload. Low values indicate that the observed workload matches the model well, whereas high values are retrieved when there is a significant deviation. Hence, the perplexity is suitable as a *conformance indicator*, which quantifies the similarity between the workload model and the actual workload. Section 4.6 shows the perplexity values for DBS workload models in various experimental results.

It is important to note that the perplexity definition only implicitly considers obsolete workload: The perplexity of a workload has the same result, even if a considerable number of statements no longer occurs in the workload. However, removing the obsolete states from the model and re-calculating the perplexity could lead to significantly smaller perplexity values. The workload shift detection mechanism described in the following therefore has to make special provisions for this purpose.

4.4.3.3 Workload Shift Detection

Having introduced the basic concepts of n-gram model creation and assessment, now the management of the model for the purpose of workload shift detection is described. The workload shift detection manages the workload model according to the lifecycle illustrated in Figure 4.14. As the goal of autonomic databases is the reduction of the maintenance costs, the approach of creating an initial workload model from a set of manually created training data is not suitable. Instead, the workload model must be learned automatically. The shift detection logic therefore always creates a new workload model in state "learning", which indicates that the model must still be trained in order to correctly describe the typical DBS workload. After the model has learned the typical DBS workload it is switched to state "stable", where it stays as long as the fluctuations in the DBS workload are small. When a significant change is detected the workload model switches to state "adapting". In this state, it cannot be used for workload shift detections, but must be trained for the changed workload.

The management of the model's lifecycle is in the responsibility of the workload shift detection logic. Considering the power of the selected n-gram modelling approach, there are three main

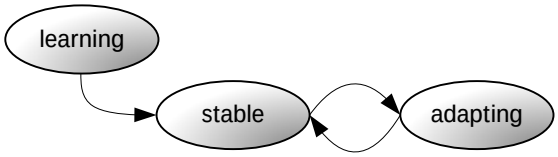


Figure 4.14: Workload Model Lifecycle for DBS Workload Shift Detection

challenges that must be addressed in this logic:

- A noise-resilient workload shift detection technique based on the perplexity measure.
- The decision on the end of the learning/adaptation phase.
- The detection of outdated states and transitions in the model.

The solutions that have been developed for each of these challenges are discussed in the following paragraphs.

Perplexity Monitoring for Workload Shift Detection

Using the perplexity metric, the similarity between the current workload of the DBS and the workload model can be quantified. In order to decide when a workload shift actually is reported, the perplexity value has to be monitored and analysed for significant changes continuously. The first approach that has been evaluated for this purpose is a simple *threshold-based* analysis technique. It exploits the fact that the value of the perplexity is meaningful with respect to the number of events (i.e. workload classes) represented in the model (cf. Equation 4.13): If the order of events is completely fixed according to the model and the observed workload adheres to this fixed order, then the perplexity value is 0. If the events are completely random and this randomness is perfectly reflected in the model, then the perplexity takes a value that is equal to the number of events in the model. In the case of previously unseen events or event histories the perplexity value exceeds this value significantly. Hence, the detection of workload changes can be achieved by defining a threshold for the maximum allowed perplexity value.

The challenge faced by the threshold-based workload shift detection is to choose an adequate threshold value. For a stable DBS workload that is created from the execution of a fixed number of transactions, the perplexity values might be assumed to be close to 0, because the sequence of statements within the transactions typically allows only little variation. However, this assumption is only valid when there are only small numbers of users or large think times between the transactions. When there are many concurrent transactions running on the DBS, the sequence of events in the workload may become more random, i.e., the perplexity values tend to take higher values. In addition, the events that are observed during the learning phase more probably do not reflect some possible event histories. Figure 4.15 plots the perplexity values that have been observed for the same stable DBS workload for different levels of transaction concurrency. The workload has been created from a pool of 100 structurally distinct SQL

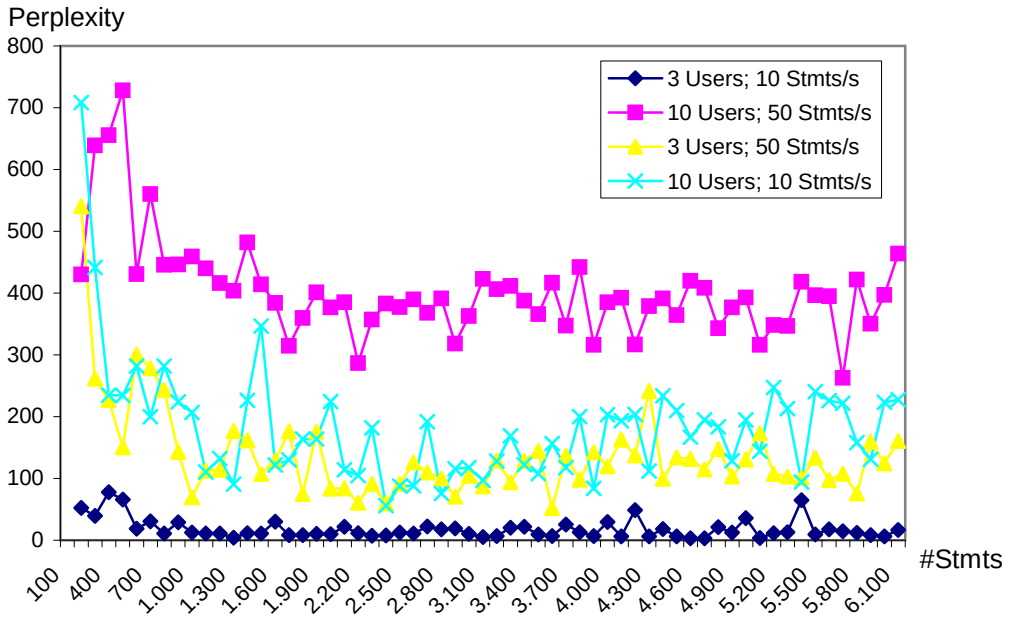


Figure 4.15: Perplexity Values for a Stable Workload depending on Transaction Concurrency

statements (mapped to 65 classes), which have been composed to a set of transactions with 1 up to 10 statementsper transaction. These transactions have then been executed by 3 and 10 users in parallel, where each user has created a load of 10 and 50 statements per second. It can be seen that the average perplexity value increases with the number of parallel users and with the number of statements per second.

By choosing a threshold for the perplexity, deviations from the usual perplexity values can be detected. A threshold value of 1.5 times the number of workload classes has proven suitable in the experimental evaluation (see Section 4.6.2), for example. However, as can be seen from Figure 4.15, the perplexity curves typically exhibit spikes due to minor fluctuations in the workload. Considering the *resilience to noise* requirement, a single value exceeding the threshold should not immediately cause a workload shift alert. For this reason a workload shift is reported only when a certain ratio (e.g. 30%) of perplexity values has exceeded the threshold. The period considered to calculate this ratio is the duration of the short-term patterns.

While the threshold-based shift detection approach allows a simple and fast detection of changes to the perplexity values, it depends on the characteristics of the typical DBS workload: The higher the degree of randomness in the workload, the higher the threshold has to be chosen. In order to avoid this effort for the DBA an alternative solution has been developed, which is based on comparing the distribution $S_{reference}$ of the perplexity values that have been observed during the learning phase to the current perplexity value distribution $S_{current}$. The length of these two periods again is determined by the length of the short-term pattern interval.

For the identification of significant deviations between these two distributions statistical tests are an adequate measure. In the literature [Tri04], several tests for comparing distributions

can be found, e.g. the t-Test, the Welch-Test, the F-Test, and the Wilcoxon-Mann-Whitney-Test. However – except the latter – all of these tests are parametric tests, i.e., they require the two distributions to be normally distributed. For the purpose of workload shift detection they are therefore not suitable, because a normal distribution cannot be assumed for these two distributions in general. Especially the current sample, which may exhibit a clear trend away from the previous mean value, may violate this assumption. The Wilcoxon-Mann-Whitney-Test ([Wil45], [MW47]) in contrast is a non-parametric test, i.e., it makes no assumptions about the underlying probability distributions of the compared samples. For this reason, it has been chosen in order to detect workload shifts from a history of perplexity values.

The null hypothesis of the Wilcoxon-Mann-Whitney-Test is that the two samples have the same distribution, whereas the alternative is that they do not. In order to decide whether or not the null hypothesis can be rejected, a test statistic U is computed from the rank sums of the two samples. For this purpose the perplexity values from both of the samples are then sorted according to their values, and each value is annotated with its rank in the merged list. Afterwards, the rank sums $R_{reference}$ and $R_{current}$ for both samples are computed by summing up all ranks of their perplexity values. From the rank sums, the test statistic U is determined as

$$U = min(U_1, U_2) \tag{4.14}$$

where U_1 and U_2 are computed as

$$U_i = n_1 * n_2 + ((n_i * (n_i + 1))/2) - R_i \tag{4.15}$$

and n_1 refers to the number of perplexity values in the $S_{reference}$, n_2 to the number of values in $S_{current}$, and R_1 and R_2 refer to $R_{reference}$ and $R_{current}$, respectively. For sufficiently large sample sizes ($n_1+n_2 >= 20$) the U test statistic is known to approximatively have a normal distribution with mean $\mu = \frac{n_1 n_2}{2}$ and standard deviation $\sigma = \sqrt{n_1 n_2 \frac{n_1+n_2+1}{12}}$. Given a significance level α, a critical value c can be easily determined from the cumulative distribution function of the normal distribution. If the absolute value of U exceeds the critical value c, the difference in the distribution of $S_{reference}$ and $S_{current}$ is considered significant. Hence, a workload shift alert is raised in this case. A smoothing of the perplexity values as in the threshold-based approach is not required in this case, because the compared samples already cover an entire short-term pattern interval.

End of Learning Phase Detection

The decision on the end of the "learning" phase for a workload model can also be made by monitoring the perplexity values. One possible solution is again based on comparing the perplexity values against a threshold. If the perplexity does not exceeded the threshold for a certain period of time during the learning phase, the model can assumed to be stable. Like before, the length of this period is determined by the maximum length of short-term patterns: As according to the

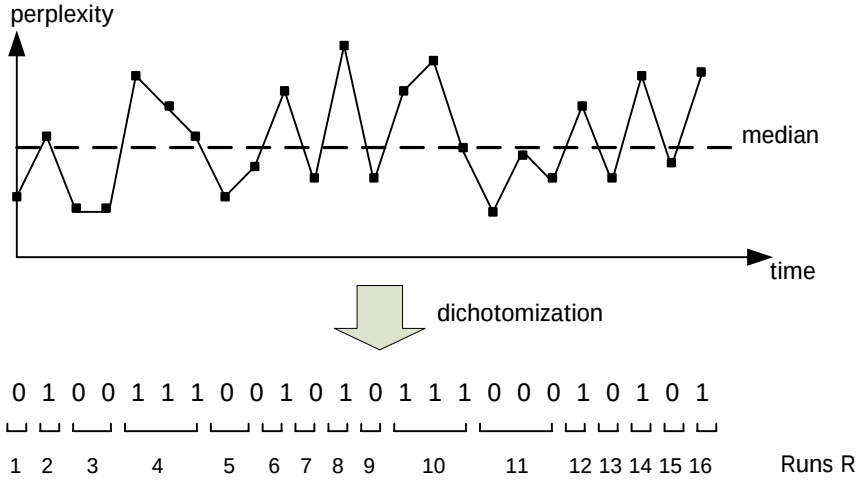


Figure 4.16: Computing the Number of Runs for a Perplexity Time Series

requirements identified in Section 4.4.1 these patterns must not be detected as workload shifts, they must be represented in the model. So after the workload has matched the model for this timespan, short-term patterns are part of the model. Like in the stable phase, the perplexity values in the learning phase may of course also show spikes caused by natural fluctuations in the workload. So to avoid unnecessary long learning periods, it is considered to be sufficient if only a certain ratio (e.g. 70%) of perplexity values does not exceed the threshold.

The threshold-based end-of-learning-phase detection has the disadvantage that the adequate threshold value has to be chosen depending on the workload characteristics of a particular DBS. A solution which meets the *self-management* requirements should instead automatically identify the perplexity level which represents the "stable" workload. In other words, it should be automatically detected when the perplexity time series data becomes stationary, i.e., when there is no trend in the perplexity data any more. For the purpose of identifying stationary time series data the Runs-Test [WW40] is applicable. Like the Wilcoxon-Mann-Whitney-Test the Runs-Test is a non-parametric test. It maps the problem of detecting a stationary time series to the problem of detecting randomness in dichotomous data. Hence, the perplexity data has to be dichotomised in a first step. This task is performed by computing the median of all perplexity values in the analysis interval. All values greater than the median are labelled as 1, whereas all smaller values are labelled as 0. Afterwards, the number of *runs* R is determined, where a run is defined as a sequence of data points with equal values (either 0 or 1). Figure 4.16 illustrates the computation of the number of runs.

The null hypothesis of the Runs-Test is that the data is randomly distributed around the median value, i.e. the time series is stationary. For this assumption to be valid, the number of runs must not be small or too high. In order to determine the critical number of runs which lead to a rejection of the null hypothesis, again an approximation of the R test statistic to a normal distribution is used. The R test statistic is approximately normally distributed with mean $\mu = \frac{2c_0c_1}{c_0+c_1}$ and standard deviation $\sigma = \sqrt{\frac{2c_0c_1(2c_0c_1-c)}{c^2(c-1)}}$ [Tri04], where c_0 denotes the number

of values labelled with 0, c_1 the number of values labelled with 1 and c the total number of values. So the critical value for the rejection of the null hypothesis can be computed using the cumulative probability function of the normal distribution with the given parameters μ and σ and the significance level α. Examples for the detection of the learning phase using the Runs-Test are given in Section 4.6.2.

Detection of Outdated Model Elements

While the perplexity metric provides a good indicator for new events or a different event composition, it is not meaningful in case of obsolete events (see Section 4.4.3.2). Hence, the following ageing mechanism has been incorporated into the workload model management: Every transition in the model is attributed with a timestamp, which indicates the most recent observation of the transition. When the age of a transition has exceeded a certain age, its information is removed from the model. For a consistent assessment of deviations from the workload changes, again the perplexity metric is utilized: an artificial sample probe is generated from the workload model before the removal of model elements, and then the perplexity for this sample probe is computed against the new, reduced model. If the perplexity value exceeds the threshold, a workload shift due to obsolete workload has occurred.

Workload Shift Detection Algorithm

Algorithm 4.6 summarises the overall threshold-based workload shift detection logic using n-gram models. The test-based workload shift detection is performed analogously. In a first step, the shift detection logic converts the observed workload W_{obs} to n-grams (**1**). Every n-gram consists of the observed event (workload class) and the history of $n-1$ preceding events, where $n-1$ is given as a predefined value `CHAIN_LENGTH`. Based on these n-grams, it computes the perplexity for the current model according to Equation 4.13 (**2**). The subsequent actions depend on the state of the model: If the model is in state "learning" or "adapting", then the transition probabilities in the model are recalculated according to Equation 4.12 (**27**), and new states are added to the model for previously unknown statements in this step. Afterwards the workload shift detection logic checks whether the model may be switched to state "stable". For this purpose it evaluates whether or not the workload has been observed for at least the duration of the short-term pattern `STP` and the ratio of perplexity values that have exceeded the threshold `THRESH` (**19-20**). If the ratio is below the configured stability factor `STAB_FACTOR`, the model is set to state "stable" (**23**). An alarm is raised in this case only if the model has previously been in state "adapting", because now the DBS workload is assumed to have reached a stability level again, and a DBS reconfiguration analysis is reasonable (**22**).

In contrast, if the model is in state "stable", then the model is not changed at all. That is, even if there are minor differences between the observed workload and the model, the transition probabilities and states are not updated. The workload shift detection logic solely checks the ratio of perplexity values that have exceeded the threshold during the last short-term pattern

Algorithm 4.6: Workload Shift Detection with n-Gram Models

```
Algorithm: detectShift
Input: observed workload W_obs, workload model model
1  nGrams ← convertToNGrams (W_obs, CHAIN_LENGTH);
2  pp ← model.computePerplexity (nGrams);
3  ppHistory.add (pp);
4  if model.state = "stable" then
5      if ppHistory.getLatest(STP).getRatioAbove(THRESH) > STAB_FACTOR then
6          model.state ← "adapting";
7          model.clear ();
8      else
9          if model.hasOutdatedStatesAndTransitions (2 · STP) then
10             probeNGrams ← model.generateProbe ();
11             model.removeOutdatedStatesAndTransitions (STP);
12             if model.computePerplexity(probeNGrams) > THRESH then
13                 raiseAlert ("DBS Workload Shift. Analysis required.");
14
15
16
17
18 if model.state = "learning" or model.state = "adapting" then
19     if ppHistory.length() > STP then
20         if ppHistory.getLatest(STP).getRatioAbove(THRESH) < STAB_FACTOR then
21             if model.state = "adapting" then
22                 raiseAlert ("DBS Workload Shift. Analysis required.");
23             model.state ← "stable";
24             return;
25
26
27     model.learnFrom (nGrams);
28     if model.state = "adapting" then
29         model.removeOutdatedStatesAndTransitions (STP);
30
31
```

interval, and changes the model state to "adapting" if necessary (**5-7**). Afterwards, the ageing mechanism described above is applied to the model (**9-13**). For a good estimation of the effects of the deletion of model elements, it is essential to always remove all elements of a past time period in one step. In order to be consistent with the existing concepts, the length of this past time period is also set to STP. If the changes performed in this step have been considered as significant, an alarm must immediately be raised to trigger appropriate reconfigurations.

Advanced Workload Modelling Concepts

The presented concepts for workload shift detection using n-gram models can be easily extended to suit different requirements: The workload management concepts of IBM DB2 [CCI+08], for example, allow to distinguish between a set of workload event origins (users, applications). So if a DBA requires the identification of changes in the behaviour of a single application, then the model can be restricted to consider this particular set of statements. In the same manner it is of course be possible to build multiple application-specific or user-specific workload models. This allows a more fine-grained detection of workload changes. However, a self-management logic controlled by high-level goals always has to consider the system-wide effects of possible reconfiguration actions. A single overall workload model therefore is considered sufficient for this purpose, because even if the change can be restricted to a single application or user, a-system wide analysis has to be performed anyway.

Like the application-specific workload models, it is also possible to build timeslot-specific models. If for instance the DBA knows that the workload of the DBS differs significantly between daytime and nighttime, separate models can be learned for them. But as this approach would require a deep understanding of the DBS workload by the DBA, an automated detection of these periodic workload scenarios is more appropriate. A solution to this problem is described in Section 4.5.

4.4.4 Two-Window Approaches

The n-gram modelling technique described in Section 4.4.3 creates approximate statistical descriptions of the dependence between workload events. On the one hand this approach represents the behaviour of the application programs in the model, but on the other hand it also causes long learning intervals and spikes in the conformance indicator due to previously unseen event histories. Section 4.4.2 therefore has identified two-window approaches, which consider the events as statistically independent, as an alternative solution. In the following, Section 4.4.4.1 first details the basic two-window analysis algorithm, before Section 4.4.4.2 presents two different similarity measures for the distributions in the windows.

4.4.4.1 Two-Window Workload Shift Detection

As introduced in Section 4.4.2, the two-window approaches are based on maintaining two windows on the observed workload data. The first window (*reference window*) defines the model of the workload, i.e. the section of the workload data which defines the typical workload of the DBS. This window remains fixed while the workload is stable. The second window (*current window*) defines the current workload of the system and is therefore shifted over the data as new workload information arrives (cf. Figure 4.10). Both windows have the same size.

In order to meet the self-management requirement, the reference window must of course be learned automatically (and not be provided by a DBA). For this reason the two-window

Algorithm 4.7: Workload Shift Detection with Two-Window Models

Algorithm: detectShiftTW

Input: observed workload W_{obs}, the current model state *state*

```
1  currentWindow.appendTrailing (W_obs);
2  currentWindow.removeLeading (W_obs.size ());
3  currentDistribution ← currentWindow.countAbsClassFreq ();
4  ci ← currentDistribution.compare (referenceDistribution);
5  ciHistory.add (ci);
6  if state = "stable" then
7      if ciHistory.getLatest(STP).getRatioAbove(THRESH) > STABILITY_FACTOR then
8          state ← "adapting";
9          currentWindow.clear ();
10         referenceWindow.clear ();
11     
12 
13 if state = "learning" or state = "adapting" then
14     if ciHistory.length() > STP then
15         if ciHistory.getLatest(STP).getRatioAbove(THRESH) < STABILITY_FACTOR then
16             if state = "adapting" then
17                 raiseAlert ("DBS Workload Shift. Analysis required.");
18             state ← "stable";
19             return;
20         
21 
22     referenceWindow.appendTrailing (W_obs);
23     referenceDistribution ← referenceWindow.countAbsClassFreq ();
24 
```

approaches require a similar lifecycle management as the n-gram models: The shift detection logic creates a new two-window model in state “learning”. In this phase the observed workload is recorded and added to the reference window. After the reference window sufficiently well describes the workload, the two-window model is switched to the state “stable”. In this state the reference window remains fixed and the current window starts to slide over the incoming workload information. These two windows are then compared in regular intervals. If there is a significant deviation between the event distributions, the workload model is changed to state “adapting”, where a new reference window is built up and a workload shift is reported.

The overall workload shift detection algorithm for two-window models is given in pseudo code in Algorithm 4.7. For every incoming probe of workload information W_{obs} the workload events are appended to the current window. At the same time the corresponding number of oldest, i.e. leading, elements is removed from the current window (**1-2**). Afterwards the probability distribution of the current window is determined by counting the occurrences of the events in this window. The algorithm computes a conformance indicator *ci* from the resulting

distribution by comparing it to the distribution of the events in the reference window (**3-4**). The two comparison techniques that have been evaluated for this purpose are described in the following Section 4.4.4.2.

The subsequent processing steps depend on the state of the model: If the model is in state "learning" or "adapting", the reference window is also extended by the observed workload information and its distribution information is adapted accordingly (**22-23**). In addition, the algorithm checks whether or not the learning phase may be ended (**13-19**). Algorithm 4.7 for this purpose employs the same threshold-based check as used by Algorithm 4.6 for the n-gram models. However, also the Runs-Test on the history of conformance indicators $ciHistory$ could be used. If the model is in state "stable", then the history of conformance indicators has to be analysed for a workload shift. Again, either the threshold-based approach used in Algorithm 4.7 (**7-10**) or the Wilcoxon-Mann-Whitney-Test can be applied for this purpose.

4.4.4.2 Similarity Metrics

The two-window workload shift detection requires a metric for the similarity of the probability distributions of the events in the reference window and the current window. Section 4.4.2 has identified two adequate techniques for this purpose: the χ^2 homogeneity test and the Kullback-Leibler-Divergence. The usage of these techniques for workload shift detection is detailed in the following paragraphs.

χ^2 homogeneity test

The χ^2 homogeneity test calculates a test statistic, which quantifies the probability that several samples of event observations adhere to the same probability distribution. In contrast to other tests the χ^2 homogeneity test is especially suited for DBS workload shift detection, because it does not require the comparison to a theoretical distribution but can directly compare two arbitrary distributions given by samples. In the case of two-window models, the samples that have to be compared are the reference window and the current window. The null hypothesis is that both windows adhere to the same distribution. By comparing the resulting test statistic of the χ^2 homogeneity test with the χ^2 distribution, the conformance indicator required by the two-window workload shift detection algorithm can be easily computed. As the χ^2 homogeneity test is suitable for categorical, i.e. nominally-scaled, data, it is applicable to the workload class information that constitutes the input stream for workload shift detection.

The calculation rule for the test statistic of the χ^2 homogeneity test is defined as

$$\chi^2 = \sum_{j=1}^{k} \sum_{i=1}^{m} \frac{(n_{ij} - E_{ij})^2}{E_{ij}} \tag{4.16}$$

where j denotes the sample, i identifies the event type, n_{ij} refers to the number of observations of event i in sample j, and E_{ij} denotes the expected number of observations given that the null

Table 4.3: Illustration of Marginal Totals Computation

	Class 1	Class 2	Class 3	Totals
$j = 1$ (Reference Window)	$n_{11} = 5$	$n_{21} = 1$	$n_{31} = 3$	$n_{*1} = 9$
$j = 2$ (Current Window)	$n_{12} = 4$	$n_{22} = 2$	$n_{32} = 3$	$n_{*2} = 9$
Totals	$n_{1*} = 9$	$n_{2*} = 3$	$n_{3*} = 6$	$n_{**} = 18$

hypothesis holds. The expected values are calculated as

$$E_{ij} = \frac{n_{i*} n_{*j}}{n_{**}} \tag{4.17}$$

where n_{i*}, n_{*j}, and n_{**} denote the marginal totals of the observations.

Table 4.3 shows an example which illustrates the calculation of the values described above for the two-window workload shift detection scenario. In the example, only three workload classes are distinguished. For every class the table cells report the absolute number of observations in the samples. It is important to note that the number of samples which have to be compared is always exactly two, i.e. $j \in \{1; 2\}$, where $j = 1$ refers to the reference window and $j = 2$ to the current window. Furthermore, the sizes of the two windows are identical, so that $n_{*1} = n_{*2}$. Both n_{*1} and n_{*2} obviously take the value of the window size (and the value of n_{**} is always twice the window size). Considering this equality of n_{*j} for all j and the calculation rules for the expected values E_{ij} in Equation 4.17, the values of E_{ij} do not depend on j anymore but only on i. Thus, Equation 4.16 can be simplified to

$$\chi^2 = \sum_{i=1}^{m} \frac{(n_{i1} - E_i)^2}{E_i} + \sum_{i=1}^{m} \frac{(n_{i2} - E_i)^2}{E_i} \tag{4.18}$$

$$= \sum_{i=1}^{m} \frac{(n_{i1} - E_i)^2 + (n_{i2} - E_i)^2}{E_i} \tag{4.19}$$

for the two-window scenario.

For sufficiently large samples ($n > 30$), the derived test statistic χ^2 is known to have a χ^2-distribution with $(k-1)(m-1)$ degrees of freedom. So in order to decide whether or not the samples adhere to the same distribution, χ^2 is usually directly compared to the critical value of the χ^2-distribution with $(k-1)(m-1)$ degrees of freedom and the given significance level α. The critical value is the point where –according to the χ^2-distribution – $100(1-\alpha)\%$ of the events are expected to have a smaller value than the test statistic. If the value of the test statistic exceeds the critical value, the null hypothesis is usually immediately rejected.

In order to increase the resilience to noise and to fit the requirements of the two-window model, the χ^2 homogeneity test methodology has been modified for the purpose of workload shift detection. Instead of directly comparing the test statistic against the critical value, the cumulative probability function of the χ^2-distribution with $(k-1)(m-1)$ degrees of freedom is evaluated at the position of the test statistic value. The result quantifies the probability

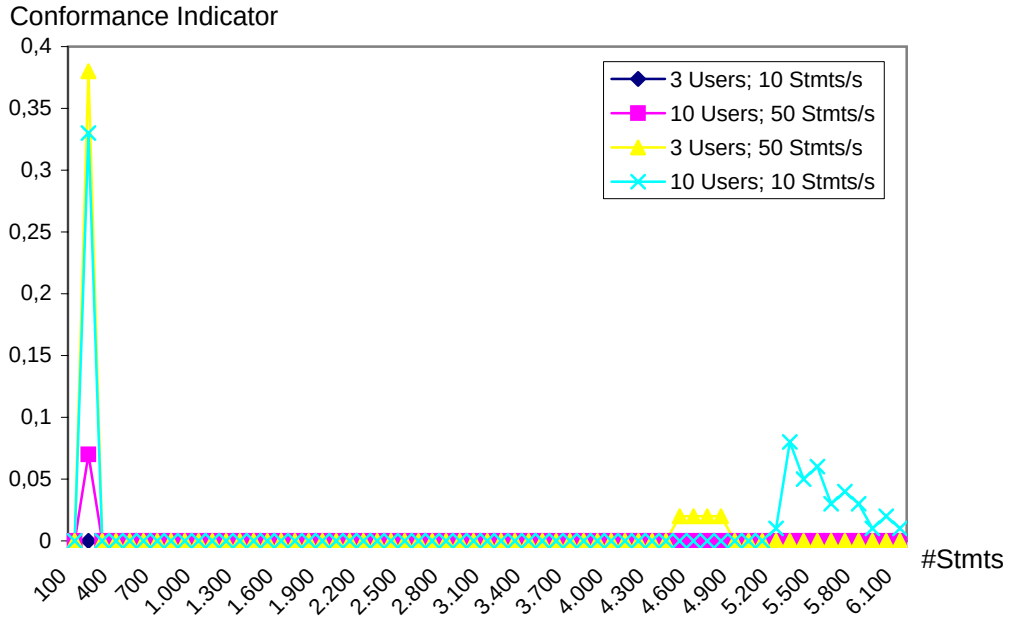


Figure 4.17: χ^2 Conformance Indicator Values for a Stable Workload at different Transaction Concurrency Levels

that the χ^2-distribution with $(k-1)(m-1)$ degrees of freedom takes a value smaller than the test statistic. The smaller this value is, the more similar are the compared samples, i.e. the reference window and the current window.

As illustrated in Figure 4.17, the conformance indicator remains at a low level (with minor disturbances) for a stable workload. The number and amplitude of spikes and fluctuations are much smaller than those of the n-gram models. Except the beginning of the learning phase, the level of the conformance indicator values of the χ^2 homogeneity test does not depend on the concurrency of the transaction executions. Hence, a threshold can be defined much more easily than with the n-gram workload shift detection. Considering the semantics of the conformance indicator, the threshold expresses the probability for the incorrect rejection of the null hypothesis (type I error). In other words, the threshold defines the probability of incorrectly assuming that the reference window and the current window do not adhere to the same distribution, while in fact they do. The experimental results shown in Section 4.6.2 illustrate that the commonly used significance level of 5%, i.e. a threshold of 0.95 yields a robust workload shift detection in most scenarios.

Kullback-Leibler Divergence

In addition to the χ^2 homogeneity test statistic Section 4.4.2 has identified the Kullback-Leibler divergence [KL51] as appropriate for comparing the two probability distributions that can be observed in the workload windows. Being a measure from information theory, the Kullback-Leibler divergence computes the expected number of additional bits which are required to encode events when a code based on q instead of the true distribution p. It is usually cal-

culated according to Equation 4.9. The larger the value of the Kullback-Leibler divergence, the less similar are the probability distributions. In contrast to the χ^2 homogeneity test, the Kullback-Leibler divergence is limited to the comparison of exactly two probability distributions. However, for detecting workload shifts with the two-window approach this limitation is not relevant, because there are only exactly two probability distributions which have to be compared (the one obtained from the reference window, the other one from the current window).

The definition of the Kullback-Leibler divergence in Equation 4.9 is not symmetric, i.e., its value changes when the two probability distributions p and q are swapped. A symmetric form of the Kullback-Leibler divergence, the Jenson-Shannon divergence [Lin91], is therefore used instead. The Jenson-Shannon divergence is computed as

$$D_{JS}(p||q) = \frac{1}{2}\left(\sum_X p(x_i) log_2 \frac{p(x_i)}{q(x_i)} + \sum_X q(x_i) log_2 \frac{q(x_i)}{p(x_i)}\right) . \tag{4.20}$$

The Jenson-Shannon divergence $D_{JS}(p||q)$ therefore computes the average value of the two Kullback-Leibler divergences $d(p||q)$ and $d(q||p)$.

For DBS workload shift detection the probability distributions p and q are derived from the relative frequencies of the workload events in the reference window and current window. As the windows only cover a subset of the entire workload generated by the DBS applications, there is the possibility of events which can be observed in one of the two windows only. Hence, its corresponding relative frequency in the other window would be computed to 0. Considering the definition of the Jenson-Shannon divergence in Equation 4.20, these cases can cause the following anomalies: If an event x_i has been observed in the current window but not in the reference window, i.e. $p(x_i) = 0$, then the fraction $\frac{q(x_i)}{p(x_i)}$ in the second summand is undefined, whereas the value of the first summand is 0. If an event x_i has been observed in the reference window but not in the current window, i.e. $q(x_i) = 0$, then the fraction $\frac{p(x_i)}{q(x_i)}$ in the first summand is undefined, whereas the value of the second summand is 0. So in both cases the Jenson-Shannon divergence is undefined. Even assuming a value of 0 for the undefined summands in these cases does not help, because then the overall contribution of event x_i to $D_{JS}(p||q)$ is 0. Hence, the probability distributions p and q would appear to be more similar, although in fact they are not. For these reason p and q are not computed as the simple relative frequencies, but according to the definition of Krichevsky and Trofimov [KT81]:

$$p(x_i) = \frac{N_{rw}(x_i) + 0.5}{n + |X|/2} , q(x_i) = \frac{N_{cw}(x_i) + 0.5}{n + |X|/2} , \tag{4.21}$$

where $N_{rw}(x_i)$ refers to the number of observations of x_i in the reference window, $N_{cw}(x_i)$ to its observations in the current window, n to the window size and $|X|$ to the number of distinct events. The definition of Krichevsky and Trofimov thus causes a redistribution of probability mass, which serves the purpose of avoiding undefined or 0-valued summands for the divergence calculation.

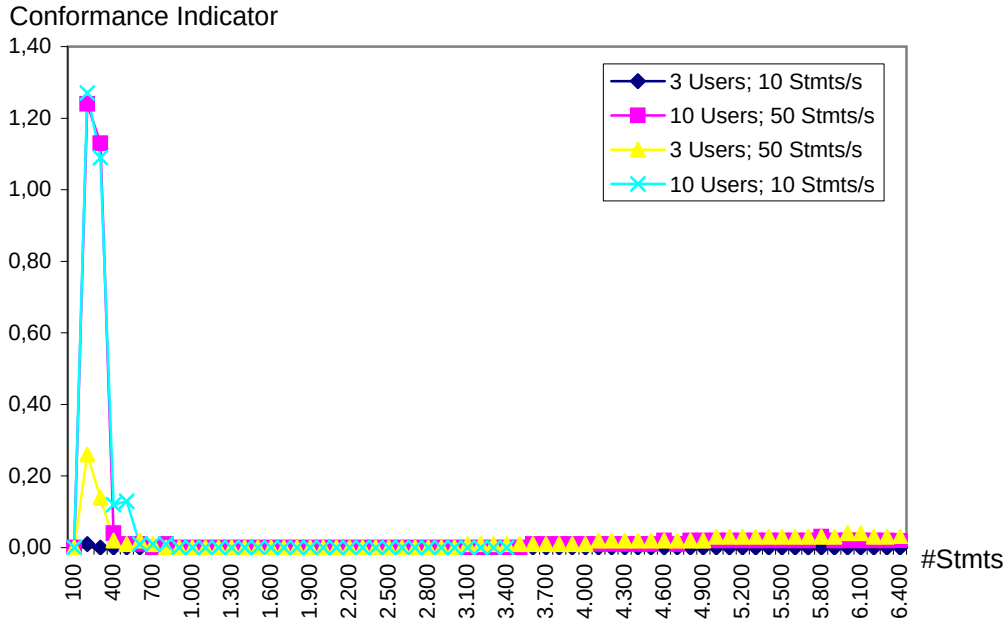


Figure 4.18: Kullback-Leibler Divergence Conformance Indicator Values for a Stable Workload at different Transaction Concurrency Levels

With the corrections by Jenson and Shannon and by Krichevsky and Trofimov the Kullback-Leibler divergence can be directly used as a conformance indicator for the two-window model workload shift detection. Figure 4.18 illustrates that this conformance indicator remains at a low level for a stable workload. Like the χ^2 conformance indicator, its level does not depend on the concurrency of the transaction executions. However, the values of the conformance depend on the number $|X|$ of distinct events in the windows. The larger the number of distinct events, the larger the value of the Kullback-Leibler divergence. A threshold-based workload shift detection therefore has to consider $|X|$ when deciding on a value for the shift detection threshold. Experiments have shown that a threshold of

$$t_{KLD} = 0.1log(|X|) \tag{4.22}$$

ensures a reliable detection of workload shifts and provides sufficient resilience to minor fluctuations in a stable workload. The experimental results for various evaluation scenarios are given in Section 4.6.2.

4.5 Workload Shift Prediction

The workload shift detection concepts described so far focus on recognizing permanent changes in the DBS workload, e.g. because of new DBS applications, or changes in the number or behaviour of the application users. In addition to these irregular, permanent changes to the workload, DBS often also face (long-term) periodic changes to the workload. For example, the workload in a DBS used for data warehousing significantly differs between daytime, when

complex queries are processed, and nighttime, when bulk-updates are executed. In this case, the DBS performance would significantly benefit from two distinct configurations: indexes, large sort areas, and a high optimization level at daytime, and no indexes, utility function heap space and a low number of parallel users at nighttime. Of course, the periodic patterns may also be more complex. For example, there may be OLTP workloads in the morning and afternoon hours, whereas reporting queries are executed at noon and in the evening, and batch updates at night. Furthermore, there may be long-term patterns in the workload like monthly or quarterly reports.

The concepts for workload shift detection discussed so far are perfectly suitable to detect these changes in the workload. Every periodic workload change is reported as a change to a new, previously unknown workload which requires an expensive reconfiguration analysis to detect possible reconfiguration actions. However, if the new workload is almost identical to a previously observed workload, the overhead for the reconfiguration analysis can be avoided. Instead of re-computing the appropriate DBS configuration after every periodic change, the appropriate DBS configuration for a particular workload profile can be stored and immediately applied when the workload reappears. Thus, it is possible to set-up a near-optimal DBS configuration without any reconfiguration analysis overhead immediately after a workload change. Moreover, the appropriate DBS configuration could even be applied pro-actively when a periodic workload change is due.

Existing DBS self-management functions do not currently consider periodic workload changes explicitly. Off-line tools like index advisors [DDD+04],[ZZL+04] or configuration advisors [KLS+03] do not consider changes in the workload at all, but the DBA has to manually re-execute them when he suspects a benefit. In contrast, on-line self-management functions automatically adapt the DBS to usage changes. Many of these functions (e.g. automated memory managers [DZ02], [SGAL+06]) learn a model of the observed system behaviour. As this model is valid for a specific workload only, it produces incorrect results when the workload changes. Hence, several hours may be required to learn a new model and to adapt the DBS to it. By then, the workload may already have changed back to the old pattern again. Although the adaptation to workload changes in other on-line self-management functions may be faster (e.g. index selection [BC07]), they do not consider workload periodicity explicitly. Hence, they may for example trigger the costly creation of a new index, although the current workload will most probably be replaced soon.

The following sections therefore describe the design of a novel analysis framework [HHR10] that identifies periodic workloads based on the workload shift detection concepts. Section 4.5.1 first identifies the key requirements. Section 4.5.2 then describes how recurring workloads can be identified, before Section 4.5.3 discusses the identification of periodicities. The prediction of future workload changes is discussed in Section 4.5.4.

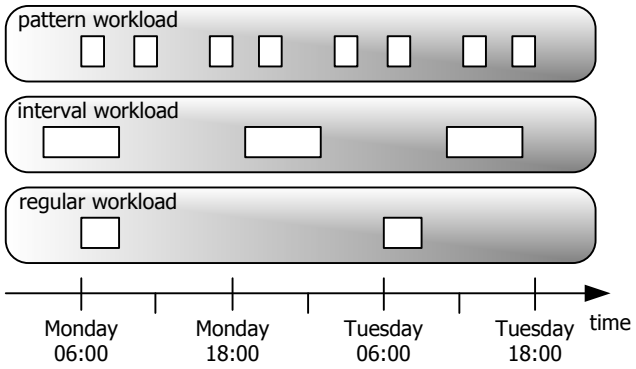


Figure 4.19: Types of DBS workload periodicity

4.5.1 Workload Shift Prediction Requirements

The goal of the identification and prediction of periodic changes or patterns in the DBS is the fast reactive or even pro-active adaptation of the DBS configuration to these changes. For this purpose, it must meet the following requirements:

Recurring Workload Detection – A prerequisite for the detection of periodicities in the workload is the identification of recurring workloads. Hence, it is necessary to store historic information that characterizes the workloads that have been observed in the past. Workloads that are sufficiently similar to workloads in the history must be identified as the same workloads.

Periodic Pattern Detection – From the workload history all types of periodicity in the workloads must be identified. Figure 4.19 illustrates the types of periodicity that can be distinguished: *Regular workloads* are workloads that can be seen at specific timestamps, e.g. every morning from 8 AM to 10 AM. *Interval workloads* in contrast appear in regular intervals, e.g. every 10 hours. Finally, *pattern workloads* refer to workloads whose appearance follows a certain pattern, e.g. repeatedly every 2 and 4 hours.

Pattern Adaptation – As the workload and the periodicities might evolve over time, the accuracy of the predictions must be continuously validated and adapted.

Dependability – Incorrect configurations for a DBS may cause a high processing overhead and – due to unavailability – even business losses. The predictions made about future workload changes must therefore be highly dependable. Hence, it must be possible for the DBA to define a minimum number of repetitions of the periodic workload, before periodicity is assumed.

Robustness – In real-world DBS, periodic workloads will be subject to fluctuations. On the one hand, the starting time and the duration of workloads may vary slightly for every appearance of the workload. On the other hand, there may be exceptions to the usual periodicity, e.g. due to server downtimes. The workload shift prediction must provide robustness to these fluctuations. However, the limits for the accepted fluctuations should be definable.

Self-Management – Like the other processing stages of the workload analysis, the workload shift prediction should not impose any additional configuration overhead on the DBA.

As described in the *Recurring Workload Detection* requirement, the prerequisite for the prediction of periodic workloads is the identification of workloads that have been observed in the

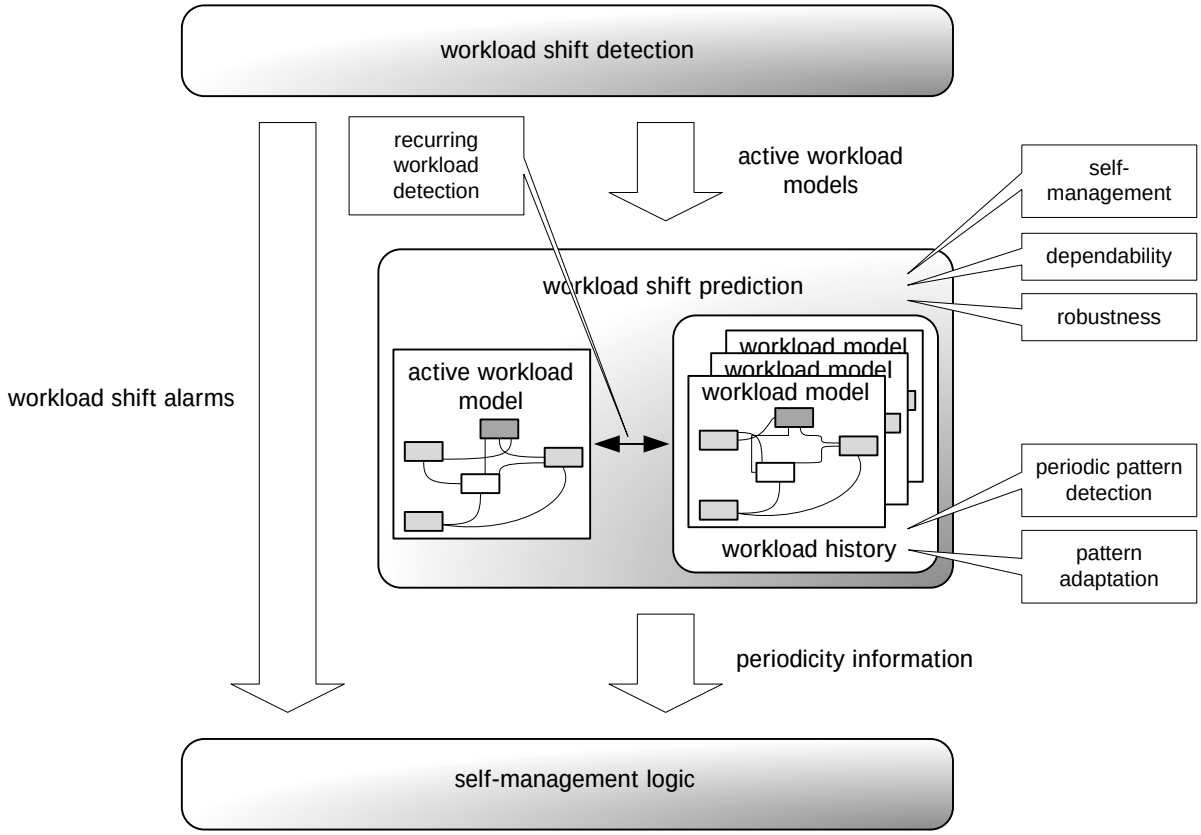


Figure 4.20: Workload Shift Prediction Requirements Overview

past. Hence, a precise characterization of observed workloads must be recorded and stored in a workload history. This characterization is already provided by the workload models that serve as a description of the typical workload of the DBS for workload shift detection. Thus, the information about the currently active workload model has to be passed to the workload shift prediction analysis component whenever a new workload model has been learned. Figure 4.20 illustrates this relationship of the workload shift prediction to the workload shift detection and the requirements discussed above.

With the periodicity analysis of DBS workloads based upon the workload models created in the workload shift detection component, the major challenges for predicting periodic workload changes can be summarized as follows: A concept for the comparison of workload models to identify recurring workloads (Section 4.5.2), a periodicity detection based on the activation timestamps of workload models (Section 4.5.3), and the adaptation of the periodic patterns at runtime (Section 4.5.4).

4.5.2 Identification of Recurring Workloads

As described in Section 4.5.1, the detection of recurring workloads is a prerequisite for the detection of periodicities. So instead of simply discarding the outdated workload model in case of a workload shift, the framework must store the workload model for future reference. Hence, a set of historic workload models must be maintained by the workload shift prediction solution.

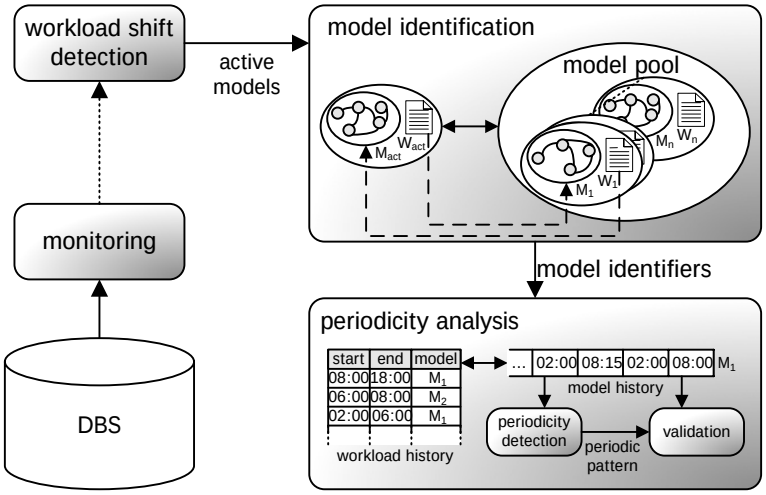


Figure 4.21: Recurring Workload Model Identification and Periodicity Detection

These historic workload models are illustrated as a model pool in Figure 4.21.

Whenever a workload shift occurs and a workload model M_{act} is activated, the model must be compared to the set of existing models M_1 to M_n. If a similar workload has not been observed before, M_{act} is added to the pool with a new model identifier M_{n+1}. If there already is a similar model M_i in the pool, then M_{act} may be discarded. In either case only the identifier of the model (M_i or M_{n+1}) is necessary for the subsequent periodicity detection stage (see Section 4.5.3).

The appropriate technique for comparing workload models depends on the types of the workload models. For the two-window approach described in Section 4.4.4, a workload model is defined by the reference window. Hence, the models can easily be compared to each other by employing the selected distance metric (Kullback-Leibler divergence or χ^2 homogeneity test statistic), because these metrics are symmetric. Every workload model M_i (i.e. reference window) from the model history is then compared to the currently activated model M_{act} (which then resembles the current window). If the resulting conformance indicator is below the threshold, then the models are considered identical. As this approach requires the compared windows to have an equal size, the larger window may have to be truncated.

If instead of a two-window model an n-gram model is used, the identification of similar models is more complex. A straight-forward approach for the comparison of n-gram models can exploit the fact that the underlying n-gram models of the workload models are approximations of markov chains. Hence, each workload model can be represented as a matrix, where the matrix values describe the transition probabilities between the workload classes. Comparing the workload models thus could be performed by comparing the matrixes. However, this approach suffers from two drawbacks: First, the size of the matrixes increases exponentially with the length of the markov chains. For the tri-gram models used in workload models (which resemble a chain length 2), the matrix size is $n^2 * n$, i.e., it grows cubically with the number of workload classes. Second, even almost identical DBS workloads will lead to minor differences in the

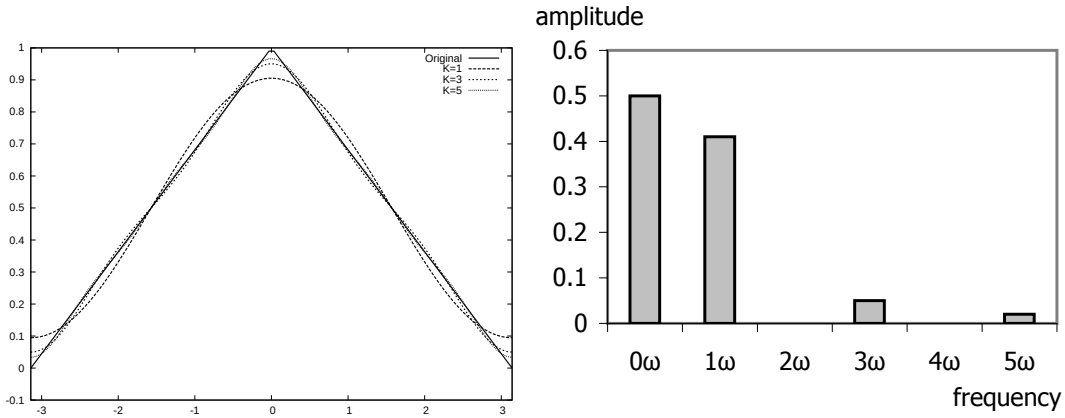
models because of natural fluctuations in the application users' behaviour. Hence, an exact comparison of the matrix elements is not appropriate, but a new measure for the "similarity" of two matrixes is required.

For the above reasons a different approach for the comparison of workload models has been realized, which integrates seamlessly with the n-gram workload shift detection concepts. As described in Section 4.4.3.2, the existing decision on whether or not an observed DBS workload matches the current workload model is based on the perplexity computation. This approach can also be generalized for the comparison of two workload models M_{act} and M_i. For this purpose, the perplexity of the workload W_{act}, that has been used to learn M_{act}, can be compared to model M_i. If the perplexity value is below the threshold used by the workload shift detection, then the model M_{act} can be seen as similar to M_i. However, this similarity measure is not symmetric: if the model M_i comprises a large number of workload classes, but the model M_{act} contains only a small subset, then the perplexity values may still be small. Hence, also the workload W_i, that has been used to create M_i, must be compared to model M_{act}. This bi-directional comparison is also illustrated in Figure 4.21.

While the described model comparison techniques integrate seamlessly with the workload shift detection techniques, there are also two limitations which apply to their usage in the overall workload monitoring and analysis framework: First, the comparison of the models can only be based on a threshold for the compliance indicator or perplexity value. A usage of the Wilcoxon-Mann-Whitney-Test is not possible, because only a single conformance value is determined. Second, the comparison techniques assume that the semantics of the workload classes do not change. However, when there is a workload shift and the *feature classification* is activated, then a new clustering will be performed on the observed workload information. The clustering identifies new medoids as workload classes so that the workload models cannot be compared to each other. For this reason the described model comparison techniques can only be used for the *signature classification* introduced in Section 4.3.2. As the signature classification exhibits some limitations compared to the feature classification (e.g. the lacking support for a strict class limit), the model comparison technique in the future should be extended by a possibility to preserve class identities when a re-clustering is performed. However, even with the signature classification the periodicity detection significantly reduces the reconfiguration analysis overhead in enterprise scenarios with periodic workloads.

4.5.3 Periodicity Detection

As illustrated in Figure 4.21, the workload history comprises the model identifiers along with their begin and end timestamps. To detect periodicities in this type of histories Fourier transforms are typically used (Section 4.5.3.1). However, as this technique does not meet all of the requirements of a workload shift prediction solution, an alternative custom solution is described in Section 4.5.3.2.



(a) Approximation of a periodic triangular signal with Fourier transforms

(b) Amplitude spectrum of the fourier transform with K=5

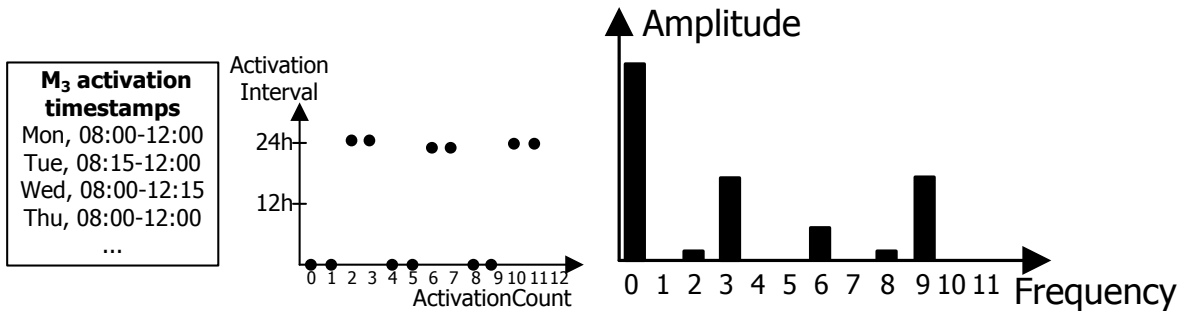
Figure 4.22: Power Spectrum of a Fourier Transform

4.5.3.1 Discrete Fourier Transform

Fourier series describe periodic signals as a linear combination of sine and cosine functions, whose frequencies are multiples of a fundamental frequency. Hence, a time-based periodic signal can be described as an amplitude spectrum, which denotes the amplitudes of the combined trigonometric functions. As outlined in Figures 4.22a and 4.22b, only multiples of a fundamental frequency will show amplitudes. Using a Discrete Fourier Transform (DFT), any discrete (non-)periodic signal can be represented in terms of an amplitude spectrum with a real and an imaginary component. To avoid having to distinguish both of these components, usually the power spectrum is computed (which is defined as the squared magnitude of the complex value; see [But06]). If the signal is periodic, this power spectrum shows energy at integral multiples of some fundamental frequency only. This characteristic is exploited by many existing periodicity detection approaches (e.g. [PN93] , [Buc09], [GRCK07]).

In order to apply the DFT to a history of DBS workload models, the model history must be converted into a discrete signal. The first step towards this goal is to separate the models by their identifiers, leading to individual *model histories*. Each model history defines when a particular model has been active (see Figure 4.21). Thus, the workload periodicity detection is simplified to detecting periodicities for single models. In the second step, each model history is converted into a time-discrete signal. For this purpose several approaches have been investigated, the most appropriate of which has been to add a data point to the signal for every activation of the model, where the value of the data point is the duration since its last activation (see Figure 4.23a). Two activations are always separated by a data point with the value 0. However, due to the characteristics of the DFT calculation, the transformation results are more accurate if every data point is inserted twice.

With the model history converted to a time-discrete signal, the DFT can be used to compute the power spectrum. If only the fundamental peak and its multiples have amplitudes, then



(a) Workload Signal (b) Power Spectrum

Figure 4.23: Representation of Model Histories

the signal is periodic. However, for DBS workload periodicity detection the robustness requirements demands that minor fluctuations must be allowed. In particular, a limit for the allowed deviations from the period must be definable. Hence, a periodicity measure p is defined, which describes the degree of periodicity of a workload model M_i as

$$p_{M_i} = \sum_{i=1}^{K} F_{i\omega} / \sum_{i} F_i \tag{4.23}$$

where ω denotes the fundamental frequency and $F_{i\omega}$ denotes the amplitude of $i\omega$. So as in [Buc09] and [PN93], p_{M_i} is computed as the ratio between the energy in the fundamental frequency and its multiples, and the total energy in the power spectrum. A value of 1 indicates perfect periodicity, and the value drops with increasing non-periodicity. In the example in Figure 4.23b almost all energy is at multiples of the fundamental frequency (i=3, i=6, i=9).

The detection of ω, which is a hard problem in general, can be solved in a straight-forward manner for the analysis of workload models: Due to the *dependability*, a minimum number of repetitions r_{min} must have been observed before a workload pattern is confirmed as being periodic. Whenever a new model is activated, the model history is checked for a pattern that shows exactly r_{min} repetitions. So p_{M_i} is calculated with $\omega = r_{min}$, and if the value is above a certain threshold the model is marked as periodic. It is important to note that this approach requires an immediate check for periodicity for all activated models M_i that have not yet been marked as periodic. Otherwise, it could be incorrectly identified as non-periodic just because there are *more* than r_{min} repetitions in the history.

Using DFT, the p_{M_i} measure provides an elegant solution to the problem of quantifying the periodicity of regular, interval and pattern workloads. However, as will be shown in Section 4.6.4, the responsiveness of p_{M_i} to non-periodicity depends on the period length. Defining a total threshold for the fluctuations of the model (e.g. 30 minutes) is not possible. Thus, a second solution *model interval analysis* has been designed for detecting DBS workload periodicity with absolute thresholds.

Algorithm 4.8: IntervalAnalysis

Input: IntervalList I_i, Min. Periods r_{min} ($r_{min} \geq 2$), Max. Fluctuation $maxFluct$
Output: Periodic pattern P_i

```
1  for k ← 1 to k * r_min > I_i.length do
       /* check range: [0;k * r_min − 1]                                  */
2      for i ← 1 to r_min do
3          for j ← 0 to k − 1 do
4              sl[j].add(I_i[((i − 1) * k)) + j]);
5          end
6      end
7      for j ← 0 to k − 1 do
8          if max(sl[j]) − min(sl[j]) > maxFluct then
9              range is aperiodic; continue outer loop;
10         
11     end
12     for j ← 0 to k − 1 do
13         P_i[j] ← avg(sl[j]);
14     end
15     k ← k + 1;
16 end
17 return P_i
```

4.5.3.2 Model Interval Analysis

Like the Fourier-based approach, the model interval analysis considers only the activation timestamps of workload models. In contrast, the duration of the model activations is not considered, because it is only relevant for the self-management function and for the detection of periodic workload changes. Furthermore, it also analyses the history of each workload model separately and is designed to operate continuously.

The model interval analysis "manually" analyses the intervals between the activations of a workload model M_i, which are stored in an activation interval list I_i. Each activation interval I_{i_j} in this list is computed as the difference between two subsequent activations $A_{i_{j-1}}$ and A_{i_j} of M_i. From I_i, the model interval analysis then has to detect the three types of periodic patterns (periodic, interval and pattern workloads). The detection of interval workloads is straight-forward: all intervals in I_i must be identical. The detection of periodic and pattern workloads in contrast requires a more complex analysis. If, for example, a periodic workload is activated daily at 08:00 and 14:00, then this results in the following activation intervals: [6h;18h;6h;18h;...]. Hence, the patterns in I_i may comprise several intervals ($6h; 18h$ in this case).

To detect all required patterns, the model interval analysis algorithm (Algorithm 4.8) analyses the activation interval list from the most recent interval $I_i[0]$ to the past. In iteration k, the interval list is analysed from the most recent time $I_i[0]$ to $I_i[k * r_{min} - 1]$. The rationale of this

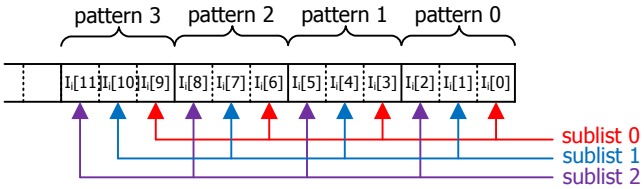


Figure 4.24: Illustration of Algorithm 4.8 for $p = 4$ in iteration $k = 3$

approach is that for $k = 1$ the interval list is analysed for r_{min} identical subsequent intervals, for $k = 2$ it is analysed for r_{min} pattern occurrences consisting of two intervals, for $k = 3$ for r_{min} pattern occurrences consisting of three intervals, and so on. Figure 4.24 illustrates the analysis of an interval list for $k = 3$. The interval values are assigned to k sub-lists sl, where the j-th interval value of each chunk is assigned to the same sub-list $sl[j]$ (**2-6**). Thus, the periodicity of the interval values can be judged by comparing the entries within each sub-list: if they are identical, then the interval values are periodic with pattern length k. The intervals between the workload model activations are computed as the average of all the observed values in each sub-list (**11-14**). As can be seen from the pseudo-code in Algorithm 4.8, the analysis of the interval list does not stop when the first periodic pattern has been found. Instead, it is continued until the entire activation interval list has been covered, because even longer patterns might be found in the interval list. Comparing the interval values for equality in the sub-lists is too strict because it does not not allow for any fluctuations (*robustness*). For this reason the decision on whether or not a given list of interval values is periodic is controlled by a parameter $maxFluct$ (e.g. 30 minutes). Only if the difference of the maximum and minimum interval values of all sub-lists is smaller than $maxFluct$, the range is considered to be periodic (**7-10**).

The model interval analysis described in Algorithm 4.8 reliably detects the periodic, interval and pattern workloads. In contrast to the Fourier-based approach described in Section 4.5.3.1, it allows the definition of an absolute value for fluctuations.

4.5.4 Adaptation of Periodic Workload Patterns

As illustrated in Figure 4.21, the task of the *validation* step is to evaluate the periodic patterns identified by the periodicity detection step, estimate future workload changes from these, and validate whether the actual workload changes match the predictions. As in the periodicity detection step, the appearances of each workload are evaluated separately for this purpose. Thus, a workload change is predicted whenever a workload model is likely to appear according to its periodic pattern. An example for a periodic pattern P_1 of a workload model M_1 is given in Figure 4.25. According to the approaches described in Section 4.5.3, this pattern consists of the intervals between the model activations only. So P_1 is a list with the values $[4h, 12h, 8h]$.

The future appearances of a model are easy to compute by evaluating the individual periodicity patterns provided as a result of the periodicity detection step. However, due to occasional irregularities in the workload, it may happen that the workload does not appear when it is ex-

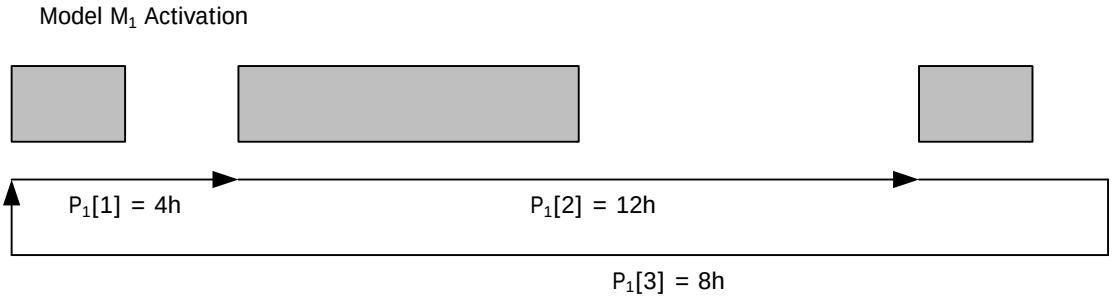


Figure 4.25: Example of a Periodic Pattern

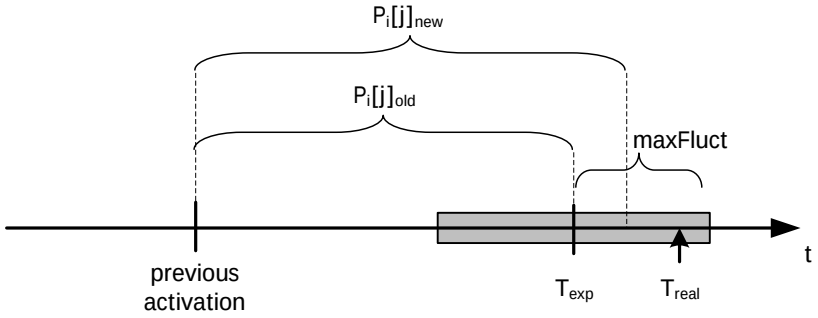


Figure 4.26: Adaptation of the Activation Intervals within Periodic Patterns

pected to. Furthermore, the intervals in the periodic pattern may evolve over time, or the entire periodic pattern may even become invalid. Thus, the actual appearances must continuously be compared to the expected appearances of the model.

Completely discarding the knowledge about periodic patterns as soon as there is a single exception to the expected appearance is not adequate (*robustness*). The knowledge about the periodicities instead has to be adapted over time, considering singular events as exceptions. The validation of the periodic patterns is described in Algorithm 4.9, which validates whether the activation of a workload model M_i at timestamp T_{real} conforms to the activation intervals defined in its periodic pattern P_i. It requires the starting time T_{start} of the current period and the counter j of previous model activations in the current period as parameters. From this information, the expected activation time T_{exp} of the workload model is computed by summing up the intervals (up to the current activation counter) (**1**-**4**). Afterwards, it is checked whether or not the model was activated within the expected time period (considering $maxFluct$ **5**). If so, then the interval knowledge in $P_i[j]$ is adapted to potential fluctuations by replacing it with the average of the expected interval and the actually observed interval (**6**-**8**). This adaptation of the pattern is also illustrated in Figure 4.26. If the activation is premature, then it is considered as an occasional singular event and is ignored (**15**). If the activation is late, a (persistent) failure counter *missing* is increased. If this counter exceeds a given threshold $maxFail$, the periodic pattern is discarded and a new pattern must be identified using the concepts described in Section 4.5.3 (**11**-**12**). Otherwise the workload model activation counter j is increased by 1, and the validation function is called recursively (**14**).

Algorithm 4.9: Pattern Validation and Adaptation

Algorithm: ActivationValidation

Input: Pattern P_i, Period Start T_{start}, Activation No. j $(0 \leq j < P_i.length)$, Activation T_{real}, Max. Fluctuation $maxFluct$, Max. Failures $maxFail$

Output: true/false

```
1  T_exp ← T_start;
2  for i ← 0 to j do
3  |  T_exp ← T_exp + P_i[i];
4  end
5  if |T_exp − T_real| < maxFluct then
6  |  I_obs ← T_real − (T_exp − P_i[j]);
7  |  P_i[j].set((P_i[j] + I_obs)/2);
8  |  return true;
9  else
10 |  if T_real > T_exp then
11 |  |  if ++missed > maxFail then
12 |  |  |  return false;
13 |  |  else
14 |  |  |  return ActivationValidation(P_i, T_start, j + 1, T_real, maxFluct, maxFail);
15 |  |
16 |  return true;
17
```

With the described approach it is possible to compare the activations of a workload model against the periodic patterns. So periodic patterns can be adapted to evolutions in the workload, and severe deviations may even cause the re-analysis of the workload model's history.

4.6 Evaluation

The following sections present the results of the experimental evaluation of the workload monitoring and analysis framework. All concepts developed for workload classification, workload shift detection and workload shift prediction have been evaluated with respect to their functional properties and the analysis overhead they cause. Section 4.6.1 introduces the workload generator used to evaluate the workload monitoring and analysis framework. Section 4.6.2 then compares the characteristics of the described workload shift detection techniques. The effects of the workload classification concepts on the workload shift detection are presented in Section 4.6.3. Section 4.6.4 describes the experimental evaluation of the workload shift detection approaches, before Section 4.6.5 gives a summary of the evaluation results.

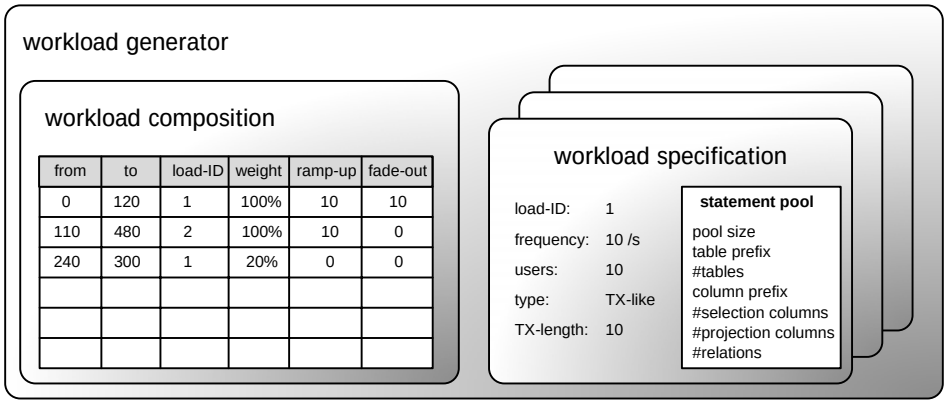


Figure 4.27: Load Specifications and Load Compositions in the Workload Generator

4.6.1 Workload Generator

The functional evaluation of the workload shift detection approaches must be based on the requirements defined in Section 4.4.1. For instance, it has to be evaluated whether or not the framework detects workload changes due to new applications, obsolete applications and usage changes. In order to create these evaluation scenarios, a *workload generator* has been implemented, which distinguishes *workload specifications* and *workload compositions*.

A *workload specification* simulates the load issued by a single DBS application. As illustrated in Figure 4.27, the workload specification comprises two parts: the description of the statement pool and the information on how the workload is generated from the statement pool. The statement pool represents the SQL statements that are issued by an application. It can either be retrieved from a trace file, or a synthetic pool can be generated. In the latter case, information on the characteristics of the statement pool must be provided (the *pool size*, the number *#tables* of relations accessed by the application, a *table prefix* and *column prefix* to distinguish the statements from other loads, and the ranges *#selection columns* and *#projection columns* restricting the number of columns in the select and where clauses).

The generated or predefined statement pool is used by the workload generator to compose a synthetic application workload. For this purpose it starts a number of threads, which independently select statements from the statement pool and thus compose an overall workload. The number of threads therefore represents the number of *users* using a DBS application. Each of the users accounts for a certain number of statements, which has to be given as the workloads *frequency* (in statements per second). For the selection of statements from the statement pool three options are available. First, the statements can be selected randomly. This option simulates a workload with no or very little correlation between a workload event and its history. Second, the statements can be selected in order. This option simulates a workload where every event is determined by its immediate predecessor. Third, a transaction-like workload can be generated. In this case the statement pool is segmented into a number of transactions, where every transaction comprises 1 to *TX-length* statements. The thread simulating the application user then randomly selects one of the transactions and executes the transaction in sequence.

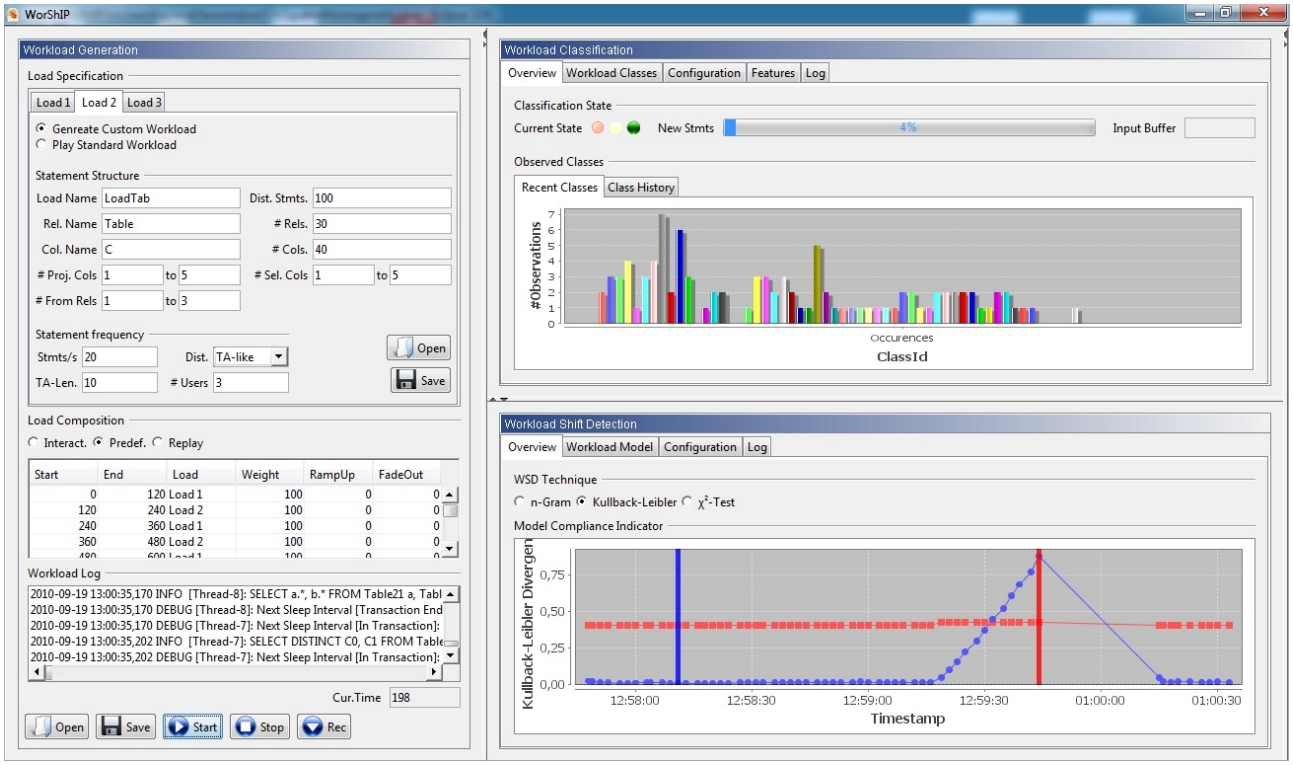


Figure 4.28: Screenshot of the graphical user interface designed for the workload generator, workload classification and workload shift detection

The think times between the selection of the statements are calculated from the *frequency* information. In case of a random selection of sequential processing, the think times are evenly distributed in order to meet the frequency requirement. For transaction-like processing, the think times between the statements within a transaction can be configured to be smaller than the average think time, e.g. only 5%. In this case the think time between the transactions is increased accordingly to meet the overall workload frequency requirements.

To simulate the workload shift scenarios required for the functional evaluation, the specified workloads can be composed to evaluation scenarios. These scenarios are defined using *workload compositions*. A workload composition allows the specification of start-times and end-times for workloads. As shown in the example illustrated in Figure 4.27, every workload may be used several times within a workload composition. Every usage of a load may furthermore be attributed with a weight, which applies to the frequency of the referenced workload. In the example in Figure 4.27, the third line in the workload composition refers to workload specification 1 with a weight of 20%. Thus, the frequency of the load is not the predefined 10 statements per second, but only 2 statements per second. To allow smooth or long-running transitions between the workloads, ramp-up and fade-out periods can additionally be defined. The left part of the screenshot in Figure 4.28 shows the graphical user interface that has been developed for controlling the parameters of the workload generator.

At runtime, the workload generator first creates or loads the statement pools defined by the workload specifications. For every time a workload specification is referenced in the work-

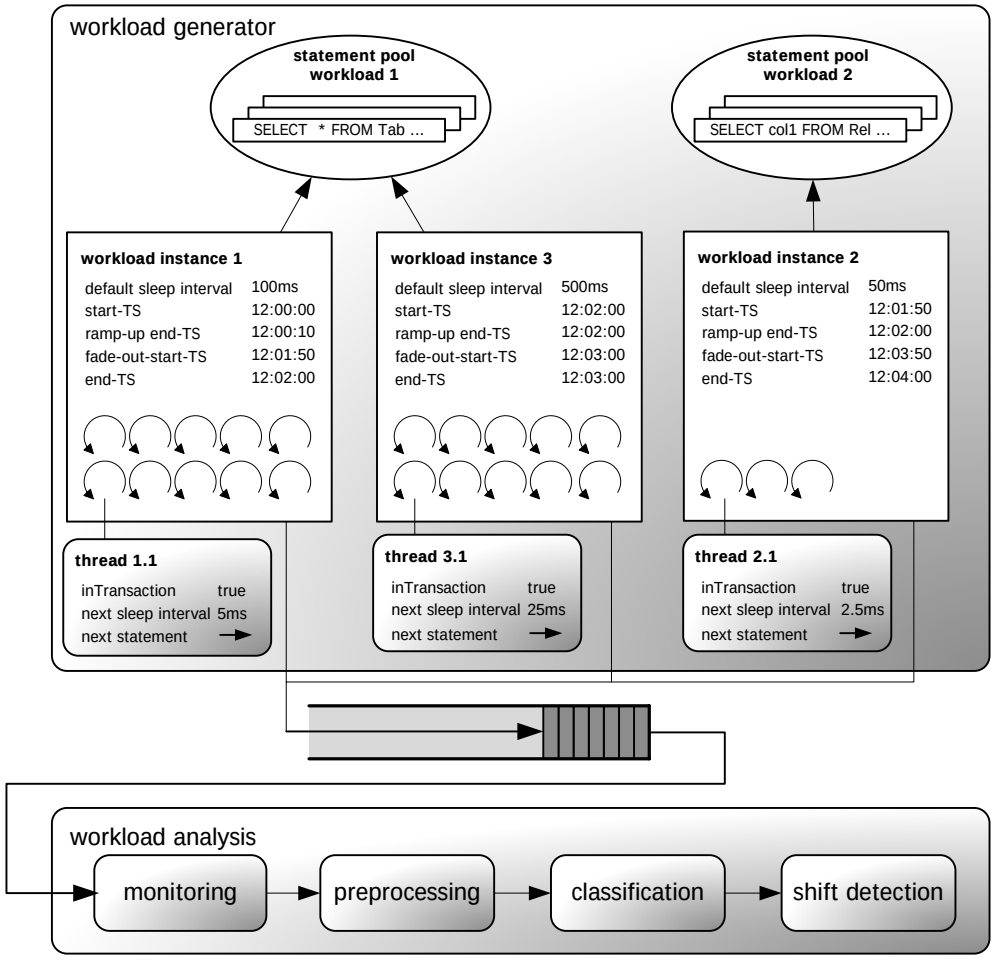


Figure 4.29: Load Generation Threads at Runtime

load composition, a separate *workload instance* is created. However, as shown in Figure 4.29, instances of the same workload specification use the same statement pool. Each workload instance creates the number of threads defined by the number of users in the workload specification. These threads share the same activation and deactivation timestamps, ramp-up and fade-out-information and the average think time between two generated workload events (*default sleep interval*). The default sleep interval is calculated from the frequency information in the workload specification and the particular weight definition in the workload composition. Every thread in a thread group acts as an individual user. Hence, it holds a pointer to the next statement and has a thread-specific sleep interval. For sequential and random workloads the thread-specific sleep interval matches the default sleep interval, whereas it changes for transaction-like workload generations depending on whether the user currently executes a transaction or not (*inTransaction*). All statements selected by the workload generator threads are added to one common workload queue. The workload monitoring and analysis framework polls the workload information from this queue in regular intervals.

It is important to note that the workload generator used for the implementation does not actually execute the statements on a DBS. Thus, problems with statement executions, e.g. due

to primary key or foreign key constraints, are avoided. This approach is valid because the usage of any internal execution information for the statements would not be reasonable for detecting *usage* changes of a DBS (cf. Section 4.2). If the statements in a statement pool are generated statements, then these statements furthermore do not actually match a particular schema. Instead, a hypothetical schema with *#tables* relations is assumed, and the structure of the SQL statements (number of projection columns, number of relations in from clause, number and type of predicates in the where clause, usage of sub-queries) is randomly composed. This simplification still fits the requirements for evaluating the described workload analysis concepts, because it is only important to distinguish *different* workload events, and this difference can be ensured by defining unique prefixes for the table names and column names in every workload specification.

4.6.2 Workload Shift Detection Evaluation

The following sections describe the functional evaluation results and the overhead induced by the workload shift techniques. Section 4.6.2.1 first discusses some efficiency aspects of the implementation of the n-gram models and two-window models. Section 4.6.2.2 afterwards describes the functional test scenarios and their corresponding results. Section 4.6.2.3 compares the efficiency of the different workload shift detection techniques.

4.6.2.1 Implementation Aspects

For the evaluation of the workload shift detection all three techniques presented in Section 4.4 have been implemented (n-gram models and two-window models with the χ^2 metric and the Kullback-Leibler divergence). In order to achieve an efficient implementation of the n-gram model in the shift detection component, the solution proposed in [Fin08] has been followed: Storing the transition probabilities in a transition matrix would require storage all events with all possible histories, even if they have never been observed. Instead, a suffix tree which only stores the events and histories actually seen has been used to store the transition probabilities. The root node of the tree holds a table with the relative probabilities for every observed statement type. Each child of the root node represents a certain history of the statement and stores the conditional probabilities for this particular history. Thus, the depth of the suffix tree is equal to the value n chosen for the n-gram models. A combination of the *absolute discounting* and *backing off* techniques [Kat87] prevents that the probability of an event takes a value of 0 for previously unseen events.

The two-window model implementation is based on two lists of workload events, where the first list represents the reference window and the second list the current window. To compute the Kullback-Leibler divergence between these windows Equation 4.20 must be applied to them. Every time the current window is moved because of the arrival of a new probe workload events, the Kullback-Leibler divergence has to be recalculated. However, if the size of the probe is small compared to the window size, the frequency of many workload event classes x_i does not

change, because they are neither present in the added leading part of the window nor in the trailing removed part of the window. Hence, the calculation of the Kullback-Leibler divergence can be optimized by restricting the recalculation of the summands to those workload classes x_i whose frequency is actually affected by the sliding of the current window.

A similar optimization as for the Kullback-Leibler divergence can be applied to the calculation of the χ^2 test statistic. This value depends on the expected value E_i (cf. Equation 4.19), which is calculated from the marginal totals n_{i*}, n_{*j}, and n_{**} as shown in Equation 4.17. For the values of n_{*j} and n_{**} are constants (n_{*j} is the window size, n_{**} is twice the window size), E_i only changes when n_{i*} changes. Furthermore, n_{i1} refers to the absolute frequency of class x_i in the reference window, and therefore does not change when the current window slides forwards. So the expected values E_i only depend on n_{i2}, i.e. the absolute frequency of class x_i in the current window. Hence, they only have to be adapted when the workload class x_i is present in either the added class observations or in the removed class observations.

4.6.2.2 Functional Evaluation

In order to evaluate the workload shift detection approaches against the requirements identified in Section 4.4.1, a number of testcases has been defined. These testcases cover the shift scenarios (New Applications, Obsolete Applications, Modified Applications, Application Usage Changes, Long-Term-Patterns) and the requirements of resilience to noise, short-term patterns and automatic learning/adapting. The design of all testcases is described in Table 4.4. For every testcase, the table lists the most important parameters of the workload specifications and the corresponding information on the workload composition. All workloads have been defined with a Transaction-like workload generation with a transaction length varying between 1 and 10 statements. The short-term pattern has been set to 20 probes for all testcases, and the probe size has been set to 100 workload events. Although the actual workload in real-world systems may of course be different, the testcases therefore resemble the *types* of workload changes that can be expected in real-world systems. The following paragraphs discuss the design of the testcases and their evaluation results in detail.

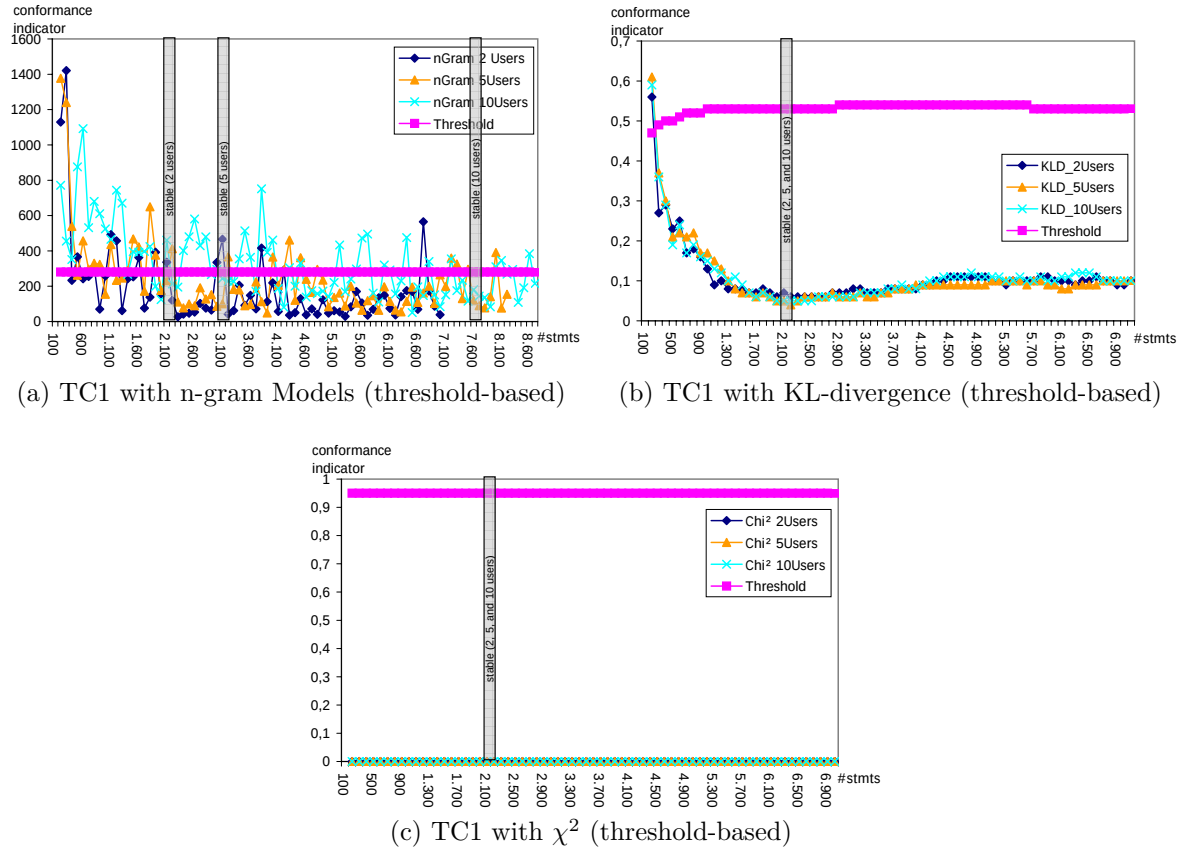
Testcase TC1 The subject of testcase TC1 is the validation of the automatic learning of a workload model from a stream of DBS workload information. Hence, TC1 generates a stable workload in a transaction-like manner, where the workload events are chosen from a pool of 1000 statements. The statements in the pool all are distinct with respect to their structure, i.e. they do not just differ in parameter values. In order to evaluate the effects of different levels of concurrency, the tests have been performed for 2 users (TC1.1), 5 users (TC1.2) and 10 users (TC1.3).

The experimental evaluation for TC1 has been performed with both the threshold-based stability detection and with the test-based stability detection. The experimental results for the threshold-based execution of TC1.1, TC1.2, and TC1.3 are illustrated in Figure 4.30. It can be

Testcase	Description	Composition			Load Specification			
		From	To	Weight	ID	Users	Freq.	Pool
TC1.1	Model Learning	0	480	100%	1	2	10	1000
TC1.2	Model Learning	0	480	100%	1	5	4	1000
TC1.3	Model Learning	0	480	100%	1	10	2	1000
TC2	Resilience to Noise	0	240	100%	1	3	10	100
		120	180	100%	2	1	5	100
TC3	Model Adaptation	0	120	100%	1	3	20	100
		120	240	100%	2	2	50	100
TC4	New Applications	0	240	100%	1	3	20	100
		120	240	100%	2	3	20	100
TC5	Obsolete Applications	0	240	100%	1	3	10	100
		0	120	100%	2	3	10	100
TC6	Modified Applications	0	120	100%	1	3	10	100
		0	360	100%	2	3	10	100
		120	360	80%	1	3	10	100
		120	360	20%	3	3	10	20
TC7	Usage Change	0	240	100%	1	3	10	100
		0	120	30%	2	3	10	100
		120	240	100%	2	3	10	100
TC8	Long-term Pattern	0	120	100%	1	3	20	100
		120	240	100%	2	3	20	100
		240	360	100%	1	3	20	100
		360	480	100%	2	3	20	100
		480	600	100%	1	3	20	100
		600	720	100%	2	3	20	100
TC9	Short-term Pattern	0	20	100%	1	3	10	100
		20	40	100%	2	3	10	100
		40	60	100%	1	3	10	100
		60	80	100%	2	3	10	100
		...	...	...	...	...	...	...
		200	220	100%	1	3	10	100
		220	240	100%	2	3	10	100

Table 4.4: Definition of Test Scenarios

seen that for all techniques and for all levels of concurrency a workload model is created and the stability of the workload is detected. However, the learning period of the n-gram model is heavily influenced by the level of concurrency: While for 2 users the model appropriately reflects the workload at the end of the short-term pattern – i.e. after the minimum learning interval of 2000 events – the learning period increases to 3100 events for 5 users and to 7600 events for 10 users. In contrast, the Kullback-Leibler divergence-based approach and the χ^2-based approach do not exhibit this behaviour. These two techniques are characterised by smooth runs of the conformance indicator values. Especially the χ^2 test statistic remains at a constant value of 0 in all scenarios. The Kullback-Leibler divergence shows a trend-like behaviour which is caused



(a) TC1 with n-gram Models (threshold-based)

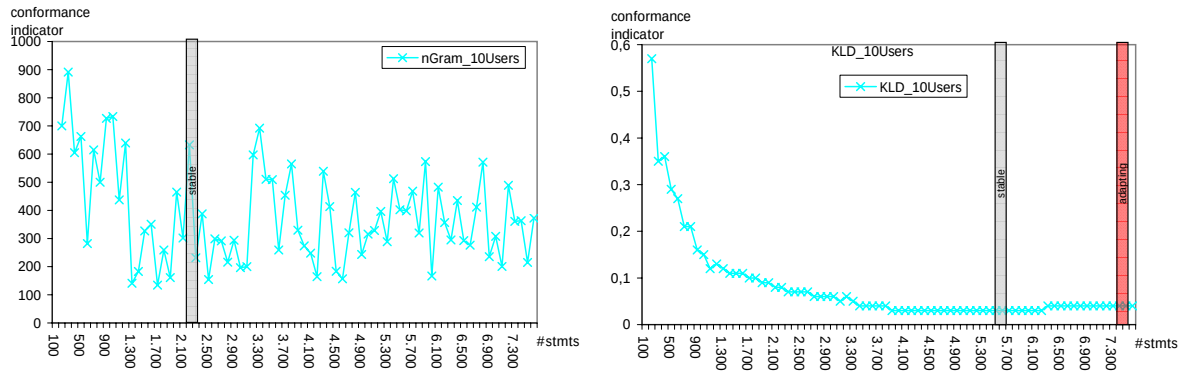
(b) TC1 with KL-divergence (threshold-based)

(c) TC1 with χ^2 (threshold-based)

Figure 4.30: TC1 Results (Model Learning): Threshold-based Shift Detection

by the large windows which are compared to each other. The perplexity of the n-gram models in contrast compares only the latest probe to the model, which results in random fluctuations around a mean value.

These observations on the characteristics of the conformance indicators have important implications on the usability of the test-based workload shift detection and end-of-learning phase detection techniques: The Runs-Test effectively tests for a random distribution of the values around a mean value. On the one hand it of course avoids the necessity of defining an absolute threshold value. On the other hand, it may cause unnecessarily long learning periods, because the values typically show a decreasing trend during the learning period. The Wilcoxon-Mann-Whitney-Test compares the rank sums of the values of the compared distributions. So even very small monotonous trends may cause the detection of workload shifts. Hence, it is not suitable for the smooth, long-running trends observed for the two-window models. Figure 4.31 illustrates these observations for the test-based workload shift detection. The Kullback-Leibler divergence in Figure 4.31b exhibits a long learning period for TC1.3, and furthermore a workload shift is detected due to a small monotonous increase in the stable phase. In contrast, the perplexity shows the characteristics of white noise and therefore no incorrect workload shift detection. For this reason the test-based workload shift detection is only evaluated for n-gram models but not for two-window models.



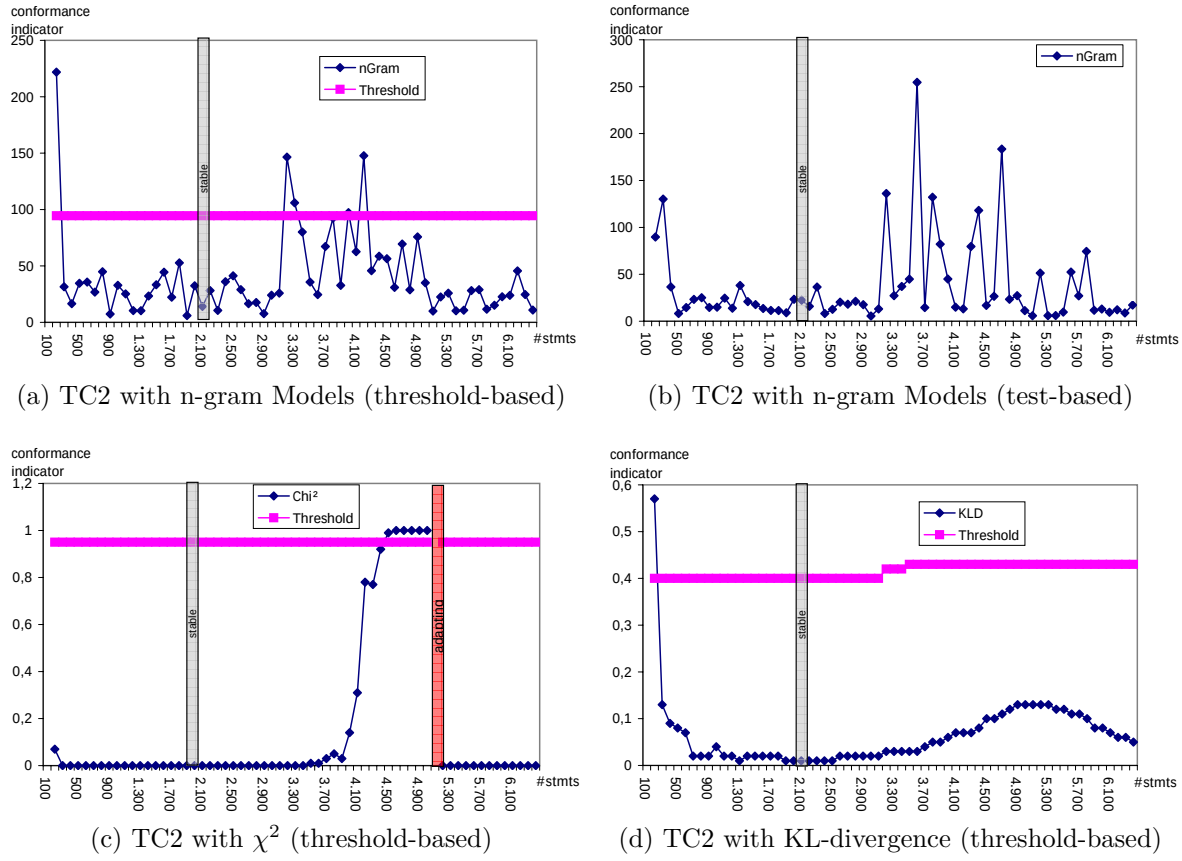
(a) TC1.3 with n-gram Models (test-based) (b) TC1.3 with KL-divergence (test-based)

Figure 4.31: TC1 Results (Model Learning): Test-based Shift Detection

Testcase TC2 The design of the second testcase TC2 evaluates the reaction of the workload shift detection techniques to noise. As shown in Table 4.4, the test scenario therefore on the one hand executes a stable load (ID 1) over the entire test period (with 3 users, 10 Stmts/sec each). In addition, a second load (ID 2) is added temporarily in the period $[120s; 180s]$, i.e., after the workload model has been learned, with only 1 user at 5 statements per second. As the second load is only a temporary load and accounts only for a small amount of the entire DBS workload, it can be considered as noise and should not cause the workload model to be rejected.

The results of the TC2 testcase for the different workload shift detection techniques are illustrated in Figure 4.32. The n-gram workload shift detection (Figures 4.32a and 4.32b) perplexity plots clearly show the influence of the noise in the interval $[3000; 5000]$. But neither the threshold-based nor the test-based shift detection report a shift for this scenario, because the deviations from the usual values are not significant. Likewise, the Kullback-Leibler divergence shift detection does not report a significant workload change. The conformance indicator remains clearly under the threshold in this case. In contrast, the χ^2 test statistic exhibits a heavy reaction to the temporary noise. All conformance indicator values after the additional workload exceed the threshold, causing a workload shift to be detected. The TC2 scenario therefore shows that the χ^2 test statistic provides little resilience to noise. Choosing a higher threshold to compensate this behaviour is hardly possible, because the conformance indicator already takes values very close to the maximum value 1.

Testcase TC3 TC3 evaluates the ability of the workload shift detection to adapt to a changed workload. For this purpose TC3 comprises two distinct workload specifications. The workload with ID1 is executed in the period [0s;120s] with 3 users at 20 statements per second. Hence, this workload exceeds the short term pattern length and allows a workload model to be learned for it. Afterwards, the second workload is executed in the period [120s;240s] with 2 users at 50 statements per second. Again, the execution period is longer than the defined short term pattern length, i.e., a workload shift should be detected and a new model should be learned. The conformance indicator values for TC3 are shown in Figure 4.33. It can be seen from



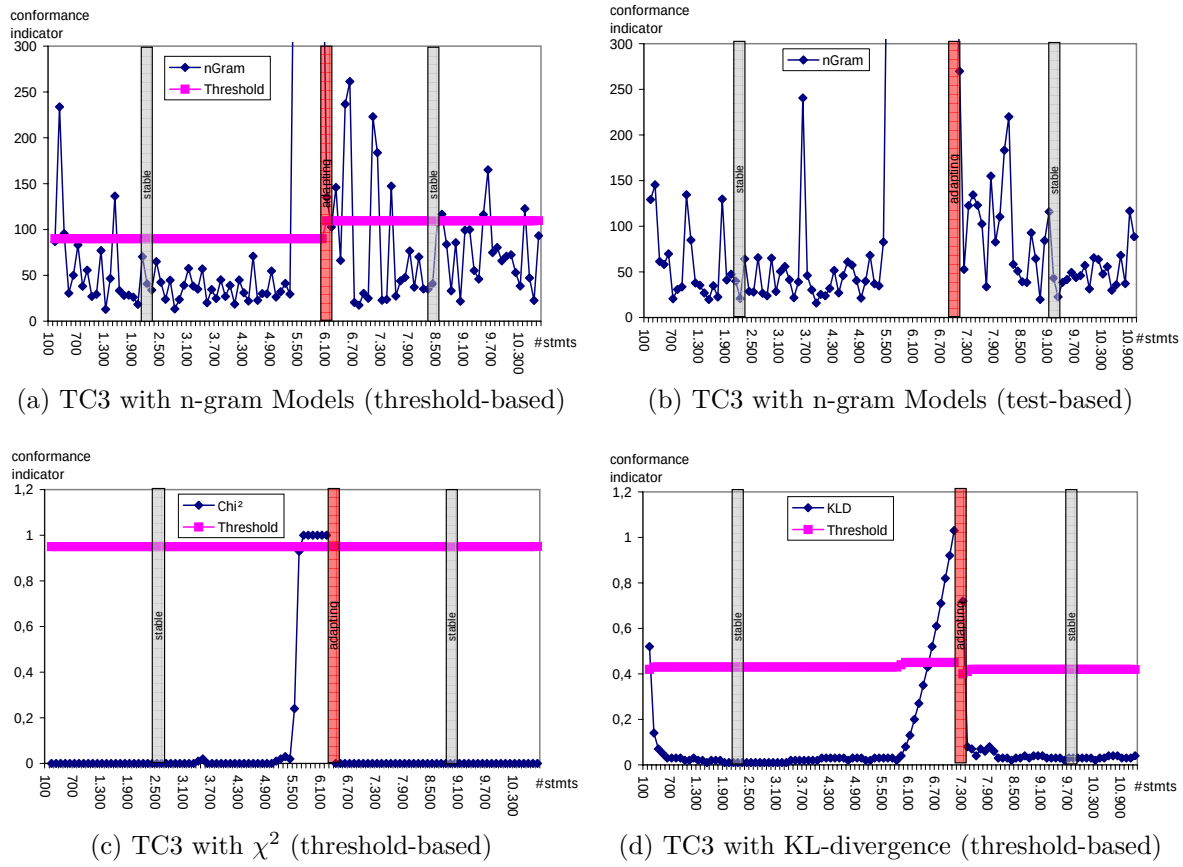
(a) TC2 with n-gram Models (threshold-based)

(b) TC2 with n-gram Models (test-based)

(c) TC2 with χ^2 (threshold-based)

(d) TC2 with KL-divergence (threshold-based)

Figure 4.32: TC2 Results (Resilience to Noise)



(a) TC3 with n-gram Models (threshold-based)

(b) TC3 with n-gram Models (test-based)

(c) TC3 with χ^2 (threshold-based)

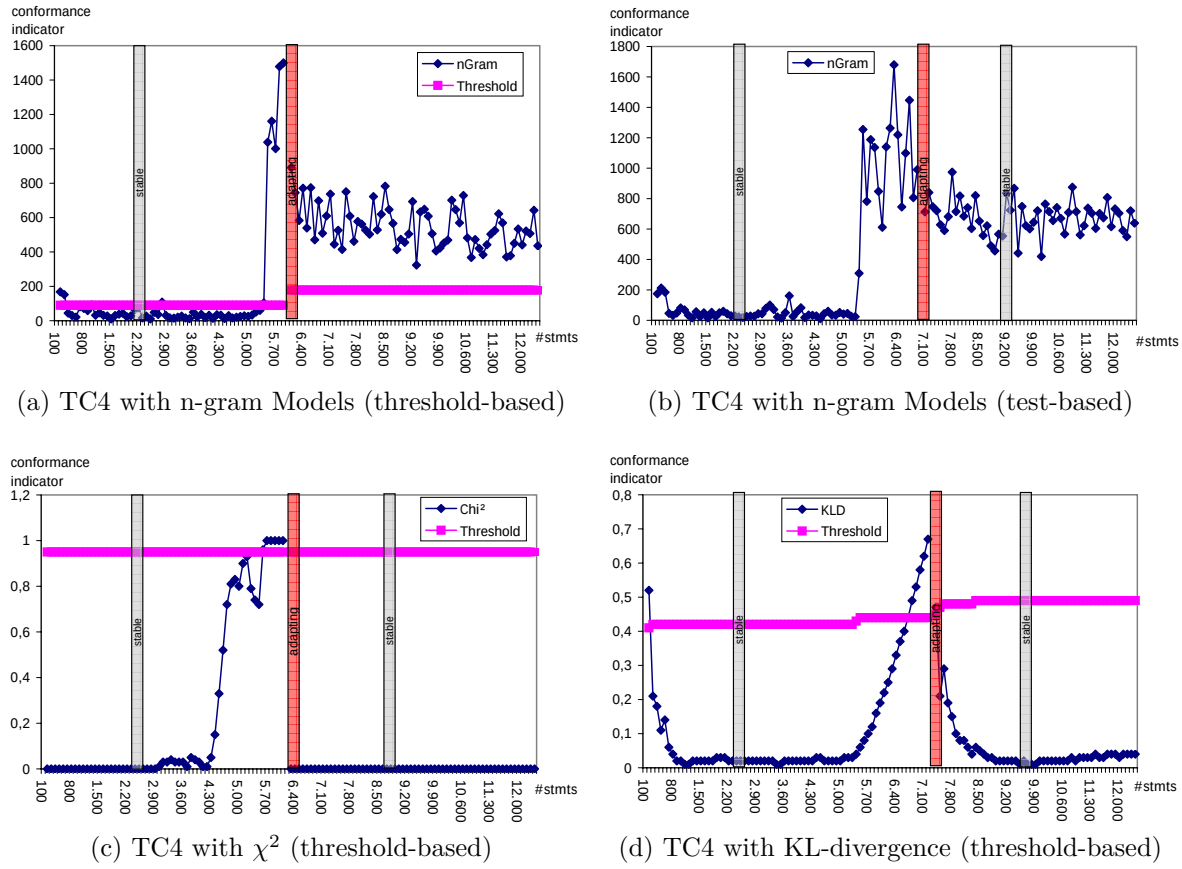
(d) TC3 with KL-divergence (threshold-based)

Figure 4.33: TC3 Results (Model Adaptation)

the plots that all workload shift detection techniques reliably detect the change between the workload. After the change has been detected, a new workload model is automatically learned from the changed workload and the state of the workload shift detection is changed back to stable.

Testcase TC4 As discussed in Section 4.4.1, one of the scenarios that have to be detected as a workload shift is the deployment of new applications. TC4 simulates this case by executing a stable workload (ID 1) over the entire test-interval. After the workload model has been learned from this workload and a stable state has been reached, a second workload representing the new application is added (cf. Table 4.4). The evaluation results for TC4 are plotted in Figure 4.34. As shown in the figures, all workload shift detection techniques recognize this scenario as a workload shift. While the Kullback-Leibler divergence and χ^2 based approach quickly learn the new model, this is not the case for the threshold-based n-gram approach. The test-based n-gram approach in contrast more quickly recognizes the stationariness of the perplexity time series and switches the model state back to the stable state.

Testcase TC5 TC5 evaluates the recognition of obsolete workloads. The test scenario for this purpose executes two distinct workloads in parallel in the interval $[0s; 120s]$. A model is



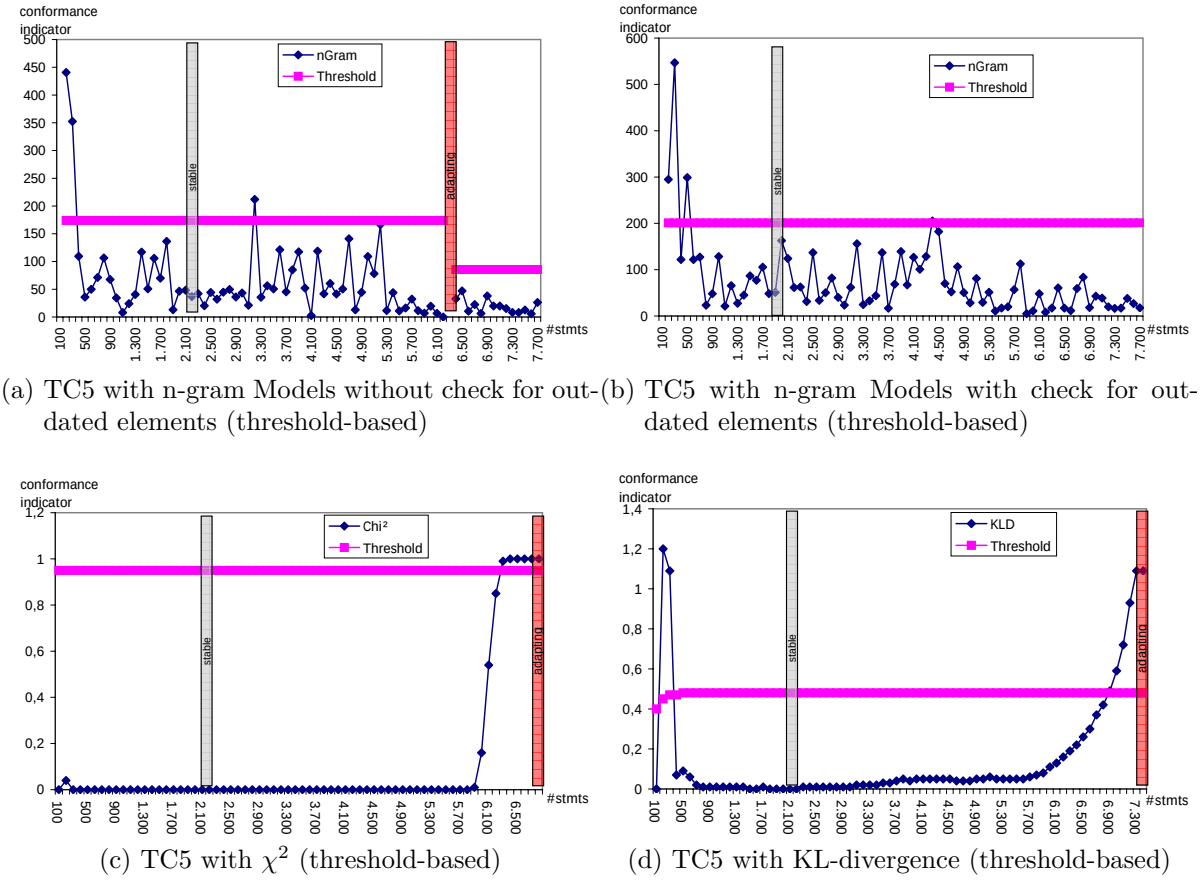
(a) TC4 with n-gram Models (threshold-based)

(b) TC4 with n-gram Models (test-based)

(c) TC4 with χ^2 (threshold-based)

(d) TC4 with KL-divergence (threshold-based)

Figure 4.34: TC4 Results (New Applications)



(a) TC5 with n-gram Models without check for outdated elements (threshold-based)

(b) TC5 with n-gram Models with check for outdated elements (threshold-based)

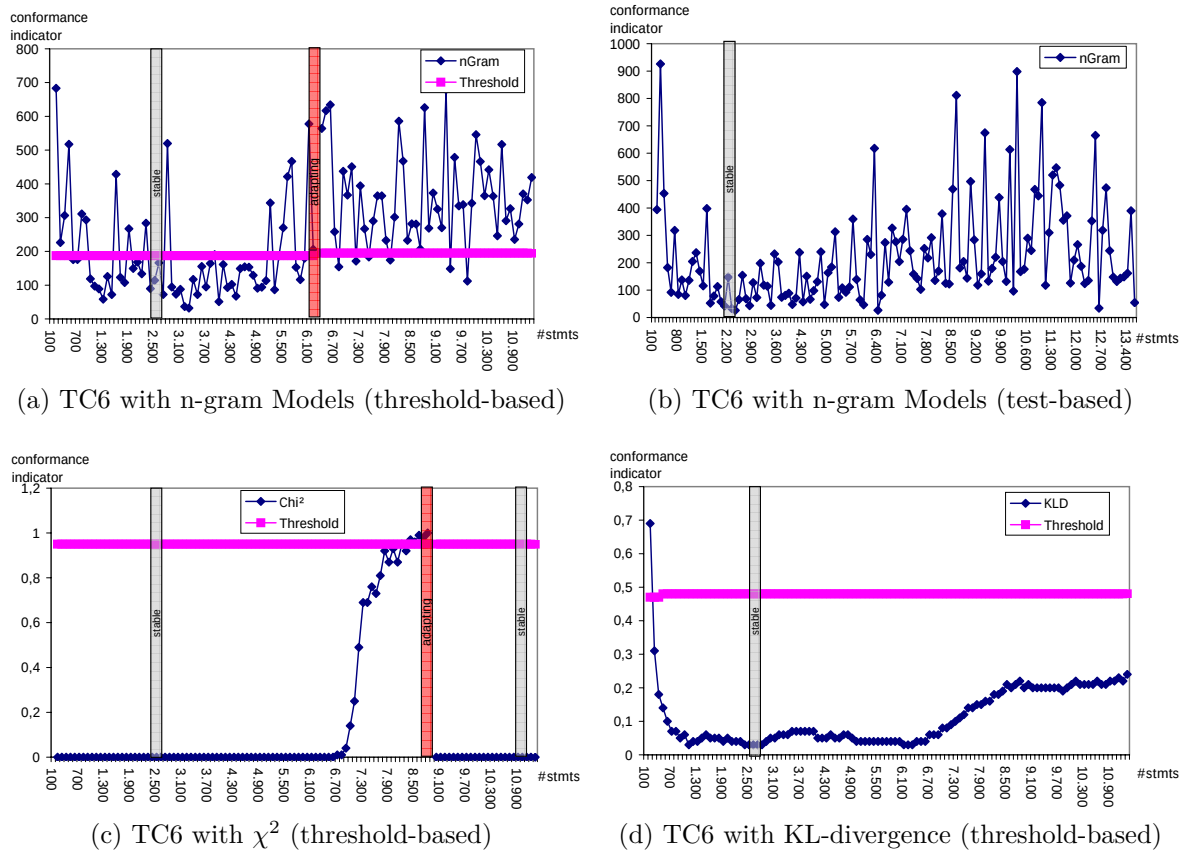
(c) TC5 with χ^2 (threshold-based)

(d) TC5 with KL-divergence (threshold-based)

Figure 4.35: TC5 Results (Obsolete Applications)

learned for these workloads. Afterwards, only one of these workloads is continued in the interval $[120s; 240s]$. Figure 4.35 shows that this scenario is successfully detected as a workload shift by the Kullback-Leibler divergence (Figure 4.35d) and χ^2 (Figure 4.35c) approaches. However, as discussed in Section 4.4.3.3, the n-gram workload shift detection does not detect obsolete workloads without an additional check for outdated model elements (Figure 4.35a). Only with this additional check the workload change is detected (Figure 4.35b).

Testcase TC6 In addition to new applications and obsolete applications also modifications or extensions to existing applications may require a reconfiguration of a DBS. For this reason also these changes should be detected as workload shifts. TC6 simulates a modified application by defining three different workloads: The workload with ID 1 represents the original workload of an application before it is extended with new functionality. It is executed with a weight of 100% in the interval $[0s; 120s]$. After this interval (and after the model has been learned) the extension of the workload causes an additional workload (ID 3) to be executed at a weight of 20%, whereas the workload with ID 1 is executed at 80% only. The workload with ID 2 simulates another DBS application which is not affected by the extension and therefore produces a stable workload over the entire evaluation interval $[0s; 240s]$. The results of the experiments for TC6 are shown in Figure 4.36. As can be seen from the plots, only the threshold-based workload



(a) TC6 with n-gram Models (threshold-based)

(b) TC6 with n-gram Models (test-based)

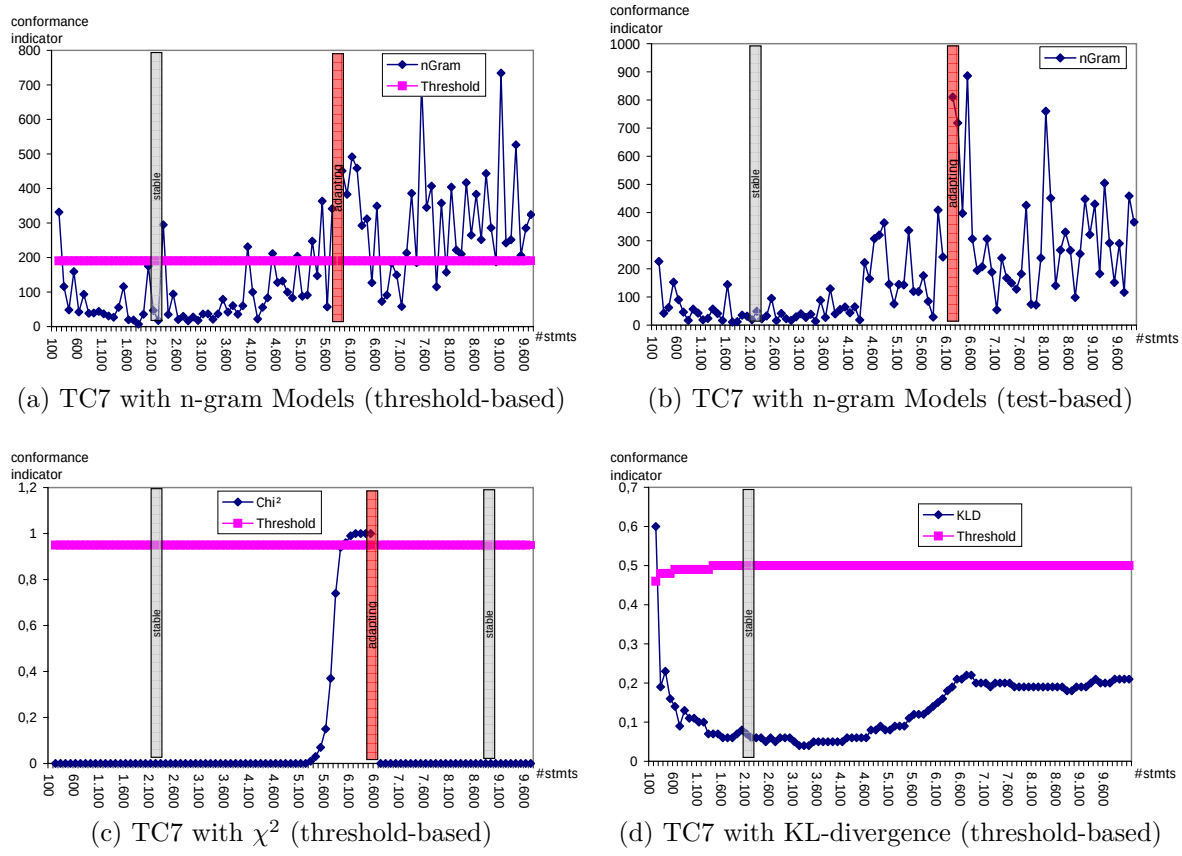
(c) TC6 with χ^2 (threshold-based)

(d) TC6 with KL-divergence (threshold-based)

Figure 4.36: TC6 Results (Modified Applications)

shift detection and the χ^2 test detect this test scenario as a workload shift. Although the Kullback-Leibler divergence shows an increase in the conformance indicator value, it does not reach the threshold. Likewise, the perplexity values of the test-based workload shift detection do not cause a significant deviation between the distributions in the learning interval and the most recent short-term pattern interval.

Testcase TC7 As discussed in Section 4.4.1, also usage changes of the DBS may cause a substantial change to the overall workload composition and therefore require reconfigurations. In particular, an increased weight of one particular application load may benefit from additional indexes or a different memory assignment. The testcase TC7 simulates a usage change in the following way: It specifies two distinct workloads, where the first workload (ID 1) represents a constant load over the entire evaluation period $[0s; 240s]$. In contrast, the weight of the second load (ID 2) is only 30% when the model is learned $[0s; 120s]$ and increased to 100% afterwards. The conformance indicators for testcase TC7 are plotted in Figure 4.37. The plots show that the n-gram model-based workload shift detection (both with thresholds and tests) and the χ^2 test detect this scenario as a workload shift. Only the Kullback-Leibler divergence conformance indicator does not exceed the defined threshold.



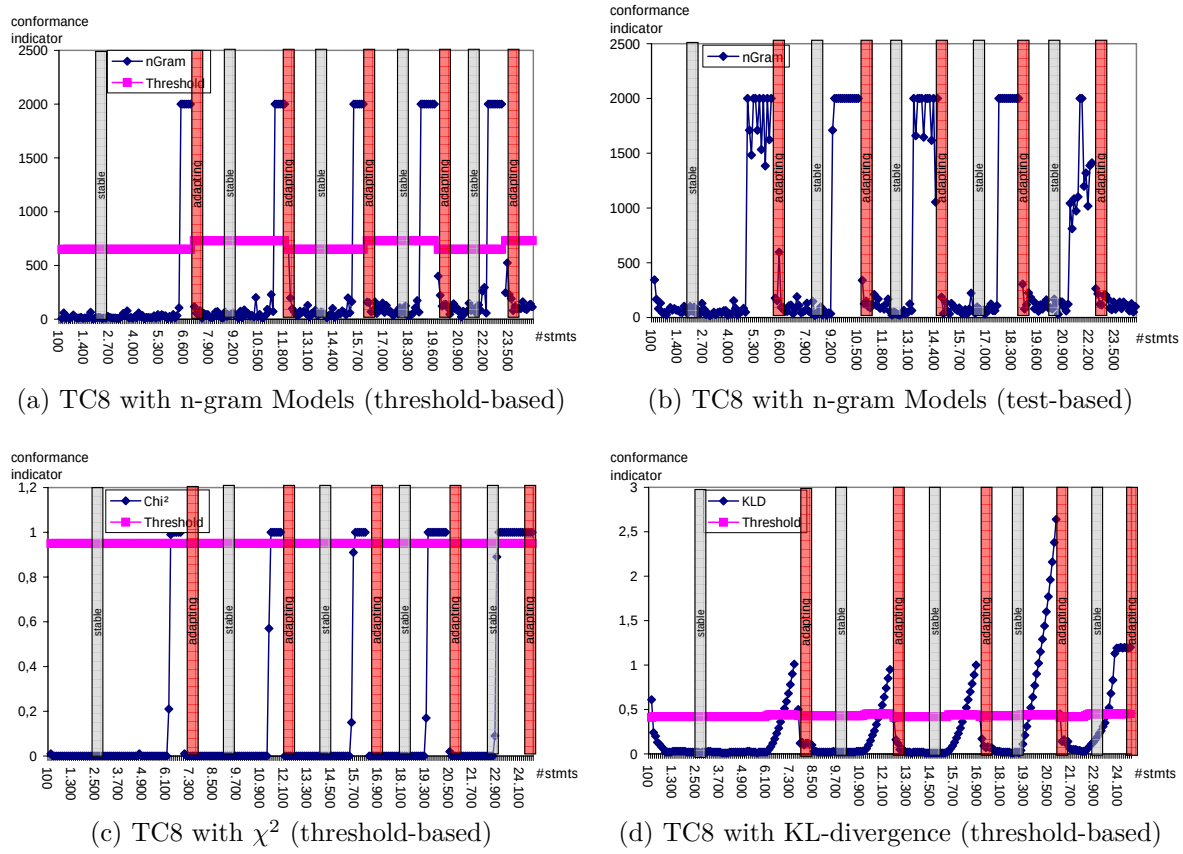
(a) TC7 with n-gram Models (threshold-based)

(b) TC7 with n-gram Models (test-based)

(c) TC7 with χ^2 (threshold-based)

(d) TC7 with KL-divergence (threshold-based)

Figure 4.37: TC7 Results (Usage Change)



(a) TC8 with n-gram Models (threshold-based)

(b) TC8 with n-gram Models (test-based)

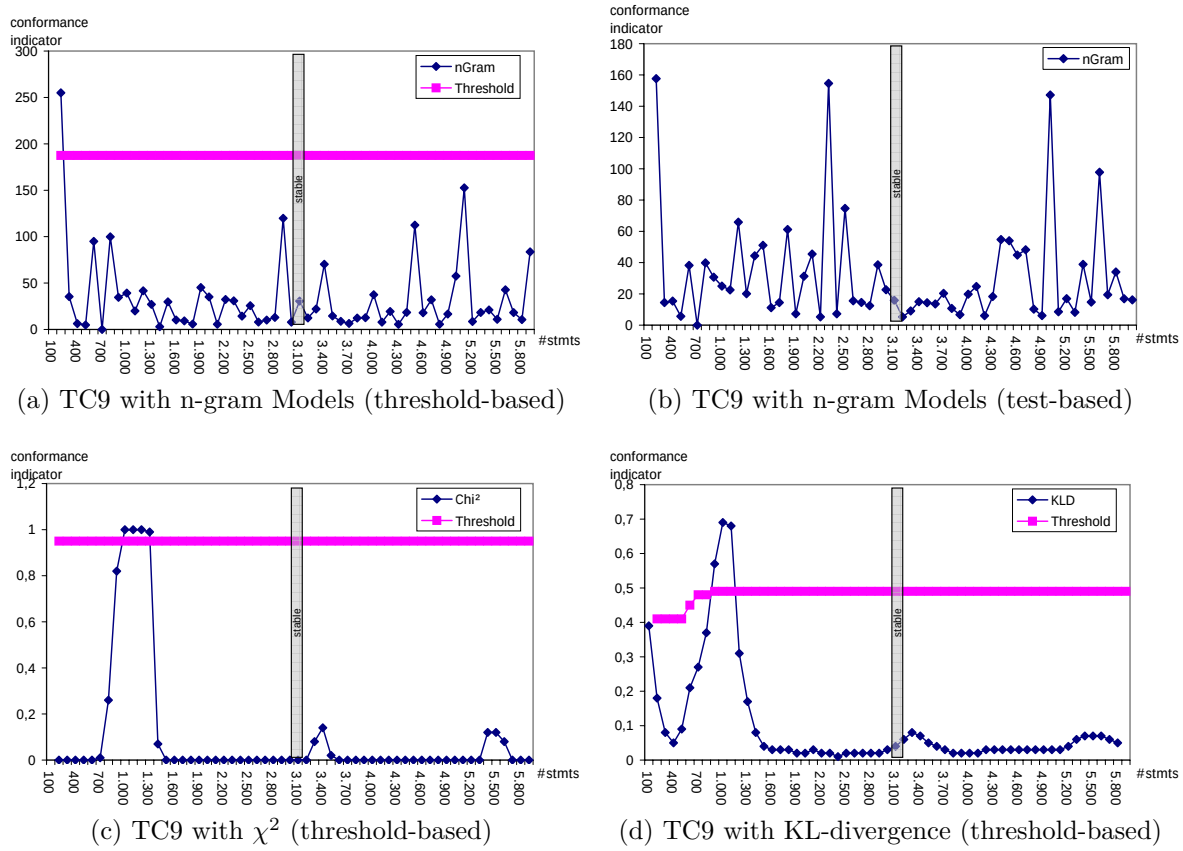
(c) TC8 with χ^2 (threshold-based)

(d) TC8 with KL-divergence (threshold-based)

Figure 4.38: TC8 Results (Long-term Pattern)

Testcase TC8 Testcase TC8 evaluates the reaction of the workload shift detection approaches to long-term patterns. It defines two distinct workloads which are executed in turns. The length of each execution period of the workloads exceeds the length of the short-term pattern. Thus, a workload shift is supposed to be detected for every change between these two workloads. Figure 4.38 illustrates that this actually is the case for all workload shift detection techniques. However, after every change the workload shift detection learns a new model, despite the fact that the same workload has been observed before (see Section 4.6.4 for the evaluation of the workload shift prediction).

Testcase TC9 The short-term pattern has been defined as the minimum time period where the overhead for the reconfiguration analysis and the implementation of the new configuration exceeds the expected benefits. Hence, periodic workload changes which are smaller than the short-term pattern length must not be identified as a workload shift. Instead, the periodic workload changes must be represented in the workload model. TC9 executes this kind of periodic workload with a short period length. As in TC8, its two workloads are executed in turns, but only for a period (20s) that is smaller than the short-term pattern, which has been extended to 30 probes (i.e. 3000 workload events) for this testcase. Considering the number of users (3) and their statement generation frequency (10 stmts/s), the short-term pattern



(a) TC9 with n-gram Models (threshold-based)

(b) TC9 with n-gram Models (test-based)

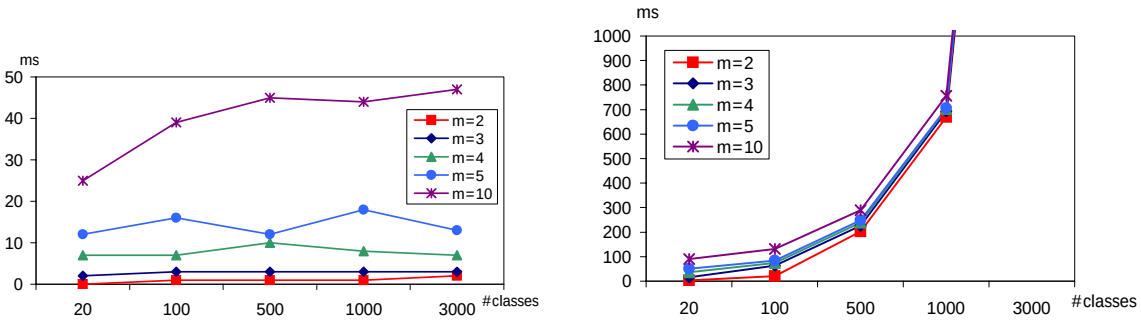
(c) TC9 with χ^2 (threshold-based)

(d) TC9 with KL-divergence (threshold-based)

Figure 4.39: TC9 Results (Short-term Pattern)

therefore spans a time period of 100 seconds. The results in Figure 4.39 show that all of the techniques correctly do not report a workload shift, because both application workloads are represented in their models.

Conclusions As a result from the testcases TC1-TC9, it can be recognized that the χ^2 test statistic reacts too sensitive to minor changes in the workload. This characteristic of the χ^2 test statistic has become obvious in TC2, where it has reported a workload shift for a temporary fluctuation. The evaluations for TC1 on the one hand have shown that the test-based workload shift detection is only appropriate for monitoring perplexity values of n-gram models. On the other hand, the evaluation of other testcases (TC4, TC6, TC7) has shown that the average perplexity value depends on the characteristics of the workload. Hence, choosing a threshold may be difficult for this approach. This makes a test-based detection of stability and workload shifts the more appropriate solution for n-gram models. The Kullback-Leibler divergence-based shift detection reacts to workload changes more smoothly and slowly than the perplexity for n-gram models. TC6 and TC7 are not detected as workload shifts by the Kullback-Leibler divergence, whereas they are by the n-gram model. The learning period for the Kullback-Leibler divergence shift detection is typically shorter than for n-gram models. So the choice of *the* adequate technique for workload shift detection depends on the specific DBS



(a) Perplexity Computation (b) Model Adaptation

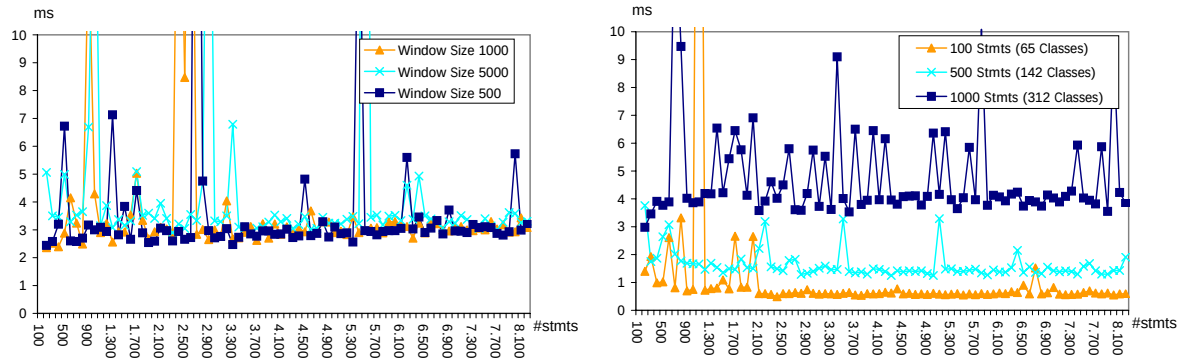
Figure 4.40: n-gram Computation Overhead

environment: If the workload is characterized by a high level of concurrency and fluctuations, then the Kullback-Leibler divergence is adequate. Otherwise, n-gram models more quickly identify changes in the workload and are more sensitive to usage changes.

4.6.2.3 Overhead Tests

The computation times for the perplexity of a probe and for the adaptation of a probe depend on the probe size, the number of statement classes and the markov chain length. Figure 4.40 illustrates the computation times on a desktop workstation (clock cycle: 3 GHz/ memory:1 GB) for different chain lengths m. As the probe size forms a scaling constant only (cf. Equation 4.13), it has been set to a fixed value of 100. The results show that the proposed workload shift detection is very efficient while the model is stable, because even with a long chain length $m = 10$ the computation of the perplexity is always less than $50ms$. The adaptation of a model in the "learning" or "adapting" states requires more overhead, but these adaptations will be necessary only very rarely.

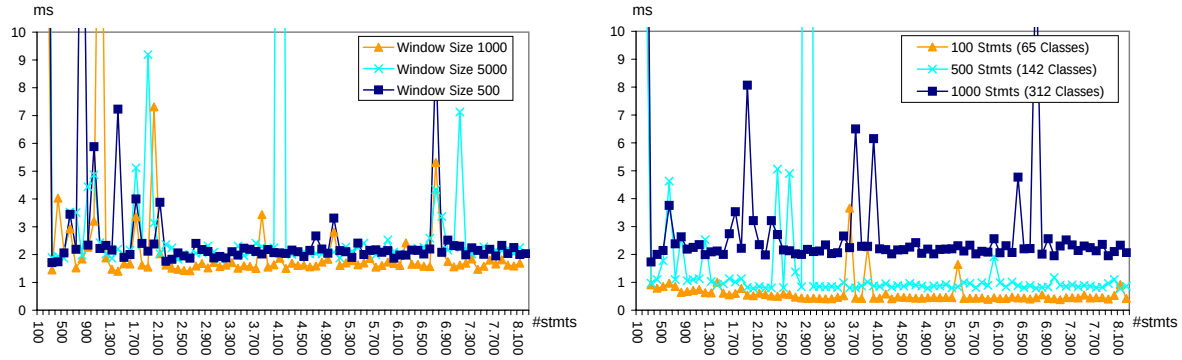
For the two-window models, the computation time for the conformance indicator in general depends on the window size. However, with the implementation described in Section 4.6.2.1, the Kullback-Leibler divergence can be computed with less effort: Its re-computation is not necessary for the entire windows, but only for those classes whose absolute frequency has actually changed by sliding the current window. For every changed class the corresponding summand of Equation 4.20 has to be recalculated. As the calculation effort for the summand is constant (given that the relative frequencies of the classes are managed in a hash-table), the overall overhead depends on the average number of workload classes that changes per workload probe. For this reason the overhead for computing the Kullback-Leibler divergence of a workload probe depends on the number of changed classes only, but not on the window size. The number of changed classes again depends on the probe size and the number of distinct workload classes. Of these, the probe size can be considered as a configurable constant (it only defines the interval when the computation effort will appear). The number of workload classes in contrast is an important configuration parameter for the workload shift detection, which requires a trade-off



(a) Kullback-Leibler Divergence Computation Effort Depending on the Window Size

(b) Kullback-Leibler Divergence Computation Effort Depending on the Number of Classes

Figure 4.41: Kullback-Leibler Divergence Computation Overhead



(a) χ^2 Test Statistic Computation Effort Depending on the Window Size

(b) χ^2 Test Statistic Computation Effort Depending on the Number of Classes

Figure 4.42: χ^2 Test Statistic Computation Overhead

between overhead and accuracy. The computation of the Kullback-Leibler divergence according to Equation 4.20 requires the aggregation of all class-specific difference values. Hence, there is a linear dependency between the computation overhead and the number of classes. Figure 4.41 illustrates the characteristics of the Kullback-Leibler divergence computation by giving the computation times per workload probe. All overhead characteristics have been derived using testcase TC1.2 and a probe size of 100 events. Figure 4.41a shows that the overhead of computing this conformance indicator for different window sizes does not change. Figure 4.41b shows the linear dependency between the number of workload classes and the computation overhead.

Like the Kullback-Leibler divergence, the overhead of computing the χ^2 test statistic depends on the number of classes but not on the window size. The reason is that the marginal totals and the expected values only have to be recomputed for those classes whose absolute frequency has actually changed. As can be seen from Equation 4.19, a linear dependency between the class number and the computation overhead exists. Figure 4.42 shows the observed computation overhead for the experimental evaluations. These overhead characteristics have also been derived using testcase TC1.2 and a probe size of 100 events.

Table 4.5: Workload Shift Detection Timestamps for different Quality Loss Thresholds

	Custom		TPC-C/H		TPC-W	
δ	#Classes	WSD	#Classes	WSD	#Classes	WSD
-	-	*4200*	-	*3100*	-	*7700*
0	200	3800	44	3100	38	7700
0.02	192	3800	40	3100	37	7700
0.04	186	3800	43	3100	31	7700
0.06	132	4000	28	3100	26	7700
0.08	123	3800	24	3100	13	8400
0.10	79	3800	14	3000	13	7800
0.12	48	3700	9	3600	7	7800
0.13	41	3700	9	3800	7	7900
0.14	23	3600	6	3800	7	7800
0.16	19	3600	3	-	6	7800
0.18	7	3400	3	-	2	-
0.20	4	-	2	-	2	-
0.22	4	-	2	-	2	-
0.24	1	-	2	-	2	-

4.6.3 Workload Classification Evaluation

The evaluation results of the workload shift detection techniques have shown that the overhead for computing the conformance indicators of the two-window models is affected by the number of workload classes. For the n-gram models, the model adaptation overhead in the learning phase is also influenced by this parameter. The workload classification therefore on the one hand reduces the required workload shift detection overhead. On the other hand, grouping similar workload events to classes also reduces the information available to the workload shift detection and may affect the accuracy of the detection results. Hence, the effects of workload classification on the workload shift detection are evaluated in Section 4.6.3.1. The overhead caused by the workload classification is determined in Section 4.6.3.2.

4.6.3.1 Functional Evaluation

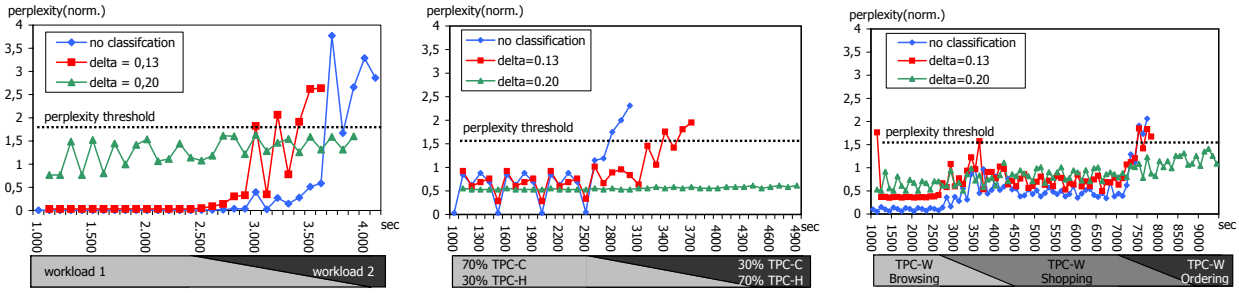
For the experimental evaluation, the workload classification prototype first converts the observed SQL statements into feature vectors by extracting the features described in Table 4.2 from the statement texts. Afterwards, the feature vectors are classified according to the concepts described in Section 4.3.4. Both the medoid distance classification and the bounding box classification have been implemented. All components are based upon an implementation of the distance function described in Section 4.3.3.2. After classification, the information on the observed workload classes is passed on to the workload shift detection module. For the evaluation the n-gram based workload shift technique has been selected, because it has proven suitable in Section 4.6.2.

The investigation of the effects of the workload configuration on the subsequent workload shift detection has been the focus of the functional evaluation. In particular, it has been examined which configurations of the classification still allow the reliable detection of workload shifts. For this purpose, various workload shift scenarios have been simulated: The *TPC-C/H* workload shift scenario starts with a workload that is composed of 70% TPC-C statements [Tra07] and 30% TPC-H statements [Tra08], and then slowly shifts to a different workload which is composed of 30% TPC-C statements and 70% TPC-H statements. The TPC-C workload [Tra07] is a workload from a standardized DBS benchmark, which simulates an OLTP workload on the DBS. In contrast, the TPC-H workload [Tra08] simulates an OLAP load on the DBS with long-running, complex queries. In a second scenario (*TPC-W*), the workload that starts with the SQL statements from the TPC-W [Tra02] Browsing Mix, then slowly shifts to the TPC-W Shopping Mix, and finally changes to the TPC-W Ordering Mix. TPC-W [Tra02] is a standard for the evaluation of database-backed web servers, which defines browsing, shopping, and ordering loads on the web server. In addition to the workload shift scenarios from standardized benchmarks, several *Custom* workloads with large numbers of distinct statements have been generated.

These workloads have been processed and evaluated with different workload classification configurations. The Tables 4.5 and 4.6 provide an overview of the results of the experiments. Table 4.5 reports when the workload shifts have been detected (*WSD*) and how many classes have been created (*#Classes*) for different quality loss thresholds δ. To ensure comparability, all results in Table 4.5 have been obtained using the medoid distance classification. The results show that for small δ values ($0.08 \leq \delta \leq 0.14$) the number of workload classes is effectively reduced, while the workload shifts are still detected. However, the timestamp of the shift detection may slightly deviate from the original shift detection.

Figure 4.43a-4.43c illustrate the perplexity values in the WSD component for the three workload shift scenarios *Custom*, *TPC-C/H*, and *TPC-W*. In the Figures 4.43a- 4.43c the perplexity values without classification, with a quality loss threshold of 0.13, and with a quality loss threshold of 0.20 are plotted. Figure 4.43c shows that for the *TPC-W* workload only the change towards the TPC-W Ordering Mix makes the perplexity exceed the shift detection threshold. The reason is that the DBS workloads, which result from the TPC-W Browsing and TPC-W Shopping load specifications, are very similar. However, by adjusting the perplexity threshold also this minor workload change could be detected.

In addition to the shift detection timestamps, it has been evaluated how many classes are added under different configurations of the workload classification. For this purpose a large test representing a smooth shift between two custom workloads has been processed, where each workload was composed by using statements from a distinct statement pool (1000 distinct statements in each pool). Table 4.6 describes for different values of δ, how many initial classes have been created (*#Cl.*), how many classes have been added using the bounding box classification (*#Add. Cl. Bounding Box*), and how many classes have been added using the medoid distance classification (*#Add. Cl. Medoid*). For the bounding box classification, the adding of classes



(a) Custom Workload (b) TPC-C/H Workload (c) TPC-W Workload

Figure 4.43: Perplexity Values for Test Scenarios

Table 4.6: Number of Classes Added for before WSD

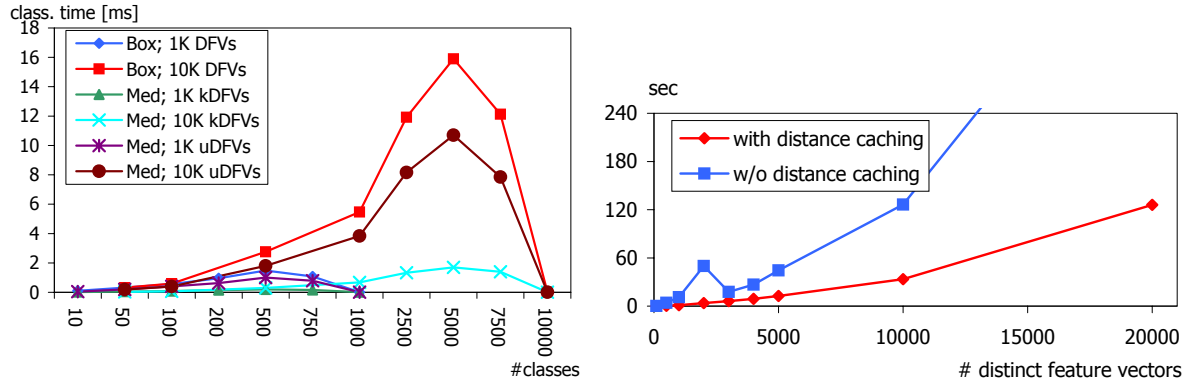
δ	#Cl.	#Add. Cl. Bounding Box					#Add. Cl. Medoid
		$d_{inc} = 0.0$	$d_{inc} = 0.3$	$d_{inc} = 0.5$	$d_{inc} = 1.0$	$d_{inc} = 2.0$	
0.05	478	101	93	25	2	0	110
0.08	249	35	23	23	0	0	63
0.10	126	32	20	13	0	0	32
0.13	27	19	5	0	0	0	7
0.20	9	5	3	0	0	0	3

also depends on the growth limit for the boxes d_{inc}. The results show that the total number of new classes that are added mainly depends on the size of the initial classes, and therefore on the acceptable quality loss δ. The influence of the growth limit also depends on δ: while for the many small classes at $\delta = 0.05$ the growth limit $d_{inc} = 0.5$ (i.e. the bounds may be extended by 50% of the original size in every dimension) does not affect the class creation, the same value entirely prevents the creation of classes for the few large classes at $\delta = 0.13$.

For the workload shift detection with n-gram models the following configuration settings have provided good results: The Minkowski order p has been set to the value 2, which resembles the Euclidean distance. The class limit has been set to 1000 ($k_{max} = 800$ and $k_{inc} = 200$). The classification learning phase ends when less than 5% new statements are observed ($\beta = 0.05$). The feature weights for the distance function have been set as shown in Table 4.2. For the quality loss threshold δ the value 0.13 has proven as an adequate value.

4.6.3.2 Overhead Tests

The effort analysis has shown that the assignment of classes in the stable phase is fast. As plotted in Figure 4.44a, the classification effort mainly depends on the number of classes. For a workload resulting in 1000 distinct feature vectors (*1K DFVs*) and 100 classes, the average classification time per vector was 0.09 ms for medoid distance classification (*Med*), and 0.73 ms for bounding box classification (*Box*), for example. For medoid distance classification, Figure 4.44a also distinguishes between cases where the distinct feature vectors have been classified before (*kDFVs*), and where only 50% of the DFVs have been classified before (*uDFVs*).



(a) Classification Time per Vector depending on the Number of Classes

(b) Computation Time of Cluster Analysis

Figure 4.44: Workload Classification Overhead

The results show that medoid distance classification is faster than bounding box classification, although of course it requires more memory to store all classified feature vectors. All tests were executed on a PC with 2.2 GHz and 2 GB memory.

Significant overhead is only caused by the cluster analysis at the end of the learning phase. Figure 4.44b illustrates the clustering effort depending on the number of different feature vectors. The effort reduction at 3000 feature vectors is caused by the optimization that avoids the binary search for the adequate class number for large workloads (see Section 4.3.4). As the distance calculation is called very often during the clustering, the clustering algorithm has been enhanced with a caching functionality for statement distances. Figure 4.44b shows that this optimization significantly reduces the computation time.

4.6.4 Workload Shift Prediction Evaluation

As for the shift detection and classification, the evaluation of the workload shift prediction is separated into functional evaluation (Section 4.6.4.1) and overhead analysis (Section 4.6.4.2).

4.6.4.1 Functional Evaluation

The subject of the functional evaluation has been the reaction of the periodicity measure p of the Fourier-based approach (see Section 4.5.3.1) to fluctuations of the workload history. For this purpose the workload scenario *S1* described in Figure 4.45 has been designed: Starting September 1st, the workload model M_1 in this scenario appears in regular intervals of six hours (the duration of the model is not relevant for the periodicity measure value). In order to evaluate the responsiveness of the periodicity measure to fluctuations in this model history, the start timestamp of the fourth appearance of the model has been iteratively increased by an offset x. For each offset x the periodicity measure p_{M_i} has been re-calculated for $r_{min} = 3$. As illustrated in Figure 4.45, the periodicity measure in this case constantly drops from a value of 1 at the offset 0 to below 0.8 for the offset 100.

scen.	M_1 start time
S1	01/09/09 06:00
	01/09/09 12:00
	01/09/09 18:00
	02/09/09 00:00 **+x**
S2	01/09/09 06:00
	08/09/09 12:00
	15/09/09 18:00
	22/09/09 00:00 **+x**

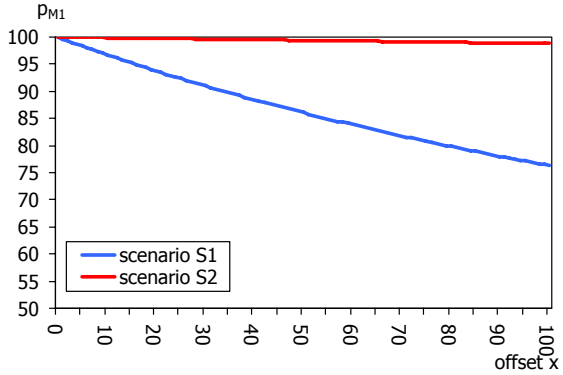


Figure 4.45: Effects of Fluctuations on p_{M_1}

In scenario *S2* the activation intervals of the model M_1 have then been increased to one week. The values p_{M_i} for different offset values of the fourth activation have then been re-calculated. Figure 4.45 shows that in this scenario p_{M_i} decreases far slower than in $S1$. So the value p_{M_i} is relative to the period length of workload pattern. Hence, the Fourier-based approach only supports the definition of fluctuation thresholds that are relative to the period length. While this may be the desired behaviour in some cases, it does not allow the definition of a fixed time limit for acceptable fluctuations. If this functionality is required, the model interval analysis approach (see Section 4.5.3.2) can be used instead, which reliably detects periodic patterns with absolute time limits.

4.6.4.2 Overhead Tests

To assess the overhead caused by the Fourier-based approach and by the model interval analysis, both of them have been applied to a sample workload history (length 250). The workload histories had to be analysed for five repetitions ($r_{min} = 5$) of a workload pattern of increasing length. Figure 4.46 illustrates the computation times of both approaches. It can be seen that although the overhead of both approaches increases with an increasing pattern length, the model interval analysis periodicity detection is always more efficient than the Fourier-based approach using the FFT algorithm [CT65]. Still, the absolute values of the periodicity are in both cases very small (less than 10 ms for a pattern with 40 model activations).

The overhead for the identification of recurring workloads is determined by the effort for the perplexity computation. The evaluation in Section 4.6.2.3 has shown that this value can be efficiently calculated: for a probe of 100 workload events and a n-gram length of 3, the computation takes less than 5ms on an average (independent from the model size). Due to the bi-directional comparison of workload models (cf. Section 4.5.2), the comparison of an outdated model M_{old} with a model from the pool would therefore take less than 10 ms for this probe size. Considering that this overhead would be caused only a few times a day (only when there is a workload change), this overhead is considered acceptable.

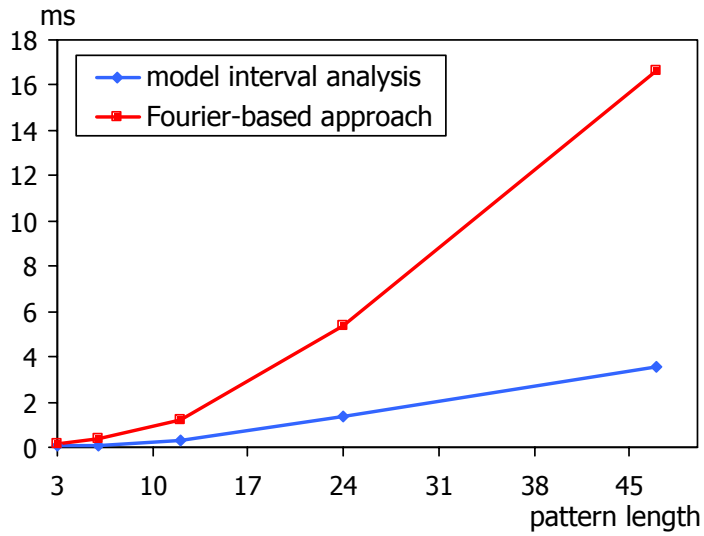


Figure 4.46: Periodicity Detection Overhead Analysis

4.6.5 Summary

The previous sections have presented the functional and overhead evaluation results for the workload monitoring and analysis concepts developed in this work. From the results of the functional evaluation of the workload shift detection it has become clear that not all combinations of the investigated concepts are reasonable: First, the small (but long-running) trends that can be observed in the conformance indicator values of the two-window solutions prevent the usage of the test-based shift detection techniques for them. So a two-window model has to be combined with a threshold-based shift detection instead. Second, the threshold-based shift detection is difficult for n-gram models, because the base-level of the perplexity value depends on the transaction concurrency. For n-gram models the shift detection therefore should be test-based. Third, a comparison of the χ^2 test statistic and the Kullback-Leibler divergence has shown that the Kullback-Leibler divergence is a better similarity measure for the two-window models, because the χ^2 test statistic is very sensitive to minor changes in the workload.

So as a result from the evaluations, two possible layouts for the workload shift detection can be identified: either an n-gram model with a test-based detection of the end-of-learning-phase and shift detection, or a two-window model with the Kullback-Leibler similarity metric and a threshold-based state change detection. The latter option on the one hand exhibits shorter learning periods (because it assumes statistical independence between the workload events), whereas on the other hand it reacts slowly to usage changes. The n-gram model option in contrast can detect usage changes more quickly. However, for workloads with a high number of distinct events and a high number of concurrent transactions it requires long learning intervals. So if the workload is characterized by a high level of concurrency and fluctuations, then the Kullback-Leibler divergence is adequate. Otherwise, n-gram models more quickly identify changes in the workload. As shown in Table 4.7, a different set of configuration parameters has to be set for these two techniques: For n-gram models the chain length n and the

Table 4.7: Parameters of the Workload Monitoring and Analysis Framework

Component	Technique	Parameter	Env. Spec.	Default Value
Monitoring	Signature Cl.	-		-
	Feature Cl.	Features		see Table 4.2
Classification	Signature Cl.	-		-
	Feature Cl.	Feature Weights		see Table 4.2
		Minkowski Order		2
		γ		0
		δ	X	0.13
		k_{max}		800
		k_{inc}		200
		b_{inc}		20%
		Check Interval		10sec
		Min. Learning	X	`STP`
Shift Detection	all	STP	X	10min
	n-gram	n		3
		α		2
	two-window	threshold		$0.1log(\|X\|)$
		stab. factor		0,08
Shift Prediction	all	threshold		like shift detection
		r_{min}	X	3
		Max. Fluct.	X	10 min
		Max. Errors	X	3

significance level α for the test-based shift detection have to be selected. Two-window models in contrast require the definition of a threshold and a stability factor (see Section 4.4.4). However, the values of these parameters are not environment-specific, but depend on the analysis goal. Hence, reasonable default values have been determined in the evaluation, which are also given in Table 4.7. The only parameter that is heavily environment specific is the short-term pattern length (STP), i.e. the time period that necessarily has to be represented in the workload model and therefore defines the minimum learning interval. This parameter should be selected by the DBA for both workload shift detection techniques.

The overhead evaluation results have shown that for both types of workload models the effort depends on the number of classes: For two-window models the computation of the similarity metric grows linearly with the number of classes. For n-gram models the perplexity computation overhead is constant for a certain history length, but the model adaptation overhead grows non-linearly with the number of classes in this case. So in order to limit the overhead for both model types, the number of distinct workload events should be restricted using the workload classification component with the feature classification technique. The functional evaluation of this component has shown that the quality loss induced by the quality loss can be conveniently controlled using a single configuration parameter (δ). As given in Table 4.7, this parameter is environment-specific and therefore should be set up by the DBA. For all other parameters of

the workload classification component (Feature Weights, Minkowski Order, γ, k_{max}, k_{inc}, b_{inc}, Check Interval) adequate default values have been identified during the evaluation. Although the minimum learning period is also environment specific in principle, it should simply be set to the same value as the STP parameter of the workload shift detection component.

The overhead tests show that classification is fast for previously observed events (which is the usual expected case). For 10,000 distinct feature vectors that are assigned to 1000 classes the classification overhead is 0.8ms per vector, for example. From the linear dependency of the Kullback-Leibler divergence computation overhead on the number of workload events the overhead per vector can be expected to be 1.3ms for 10,000 distinct vectors, whereas this overhead would be 0.13ms for 1,000 distinct vectors only. So for large workloads the classification overhead is justified, because it reduces workload shift detection overhead significantly. Due to the non-linear overhead increase for the model adaptation of n-gram models, the classification overhead here is reasonable even for comparatively small workloads.

After a new model has been learned from the DBS workload, the model information is passed to the workload shift prediction component. With the Fourier-based approach and the model interval analysis two techniques have been examined for this purpose. Their functional evaluation in Section 4.6.4 has shown that the the Fourier-based approach judges fluctuations in the periodicity relative to the period length. The model interval analysis in contrast allows the definition of absolute limits for the fluctuations. Considering the rare executions of the periodicity detection analysis functions, the analysis overheads of less than 20ms are neglectable in both cases. However, as described in Section 4.5.2 there currently is one important limitation: Due to the re-assignment of workload class IDs during the re-clustering of the workload events in the feature classification, the workload shift prediction can only be combined with signature classification. Otherwise recurring workload models could not be identified. In order to exploit the full power of the workload monitoring and analysis framework, this limitation should be eliminated in future work.

The workload shift prediction component requires the definition of a threshold value. As shown in Table 4.7, this parameter should be set to the same value as for the workload shift detection. In addition, the following parameters have to be set up adequately for a particular environment: the minimum number of repetitions r_{min}, the maximum allowed deviation from strict periodicity, and the maximum number of prediction mistakes that is accepted before the periodicity pattern knowledge is discarded.

4.7 Related Work

Before the conclusions from the design and evaluation results of the developed workload analysis concepts are summarized in Section 4.8, an overview of related work is given in the following first. In general, prior work on the analysis of database workloads can be classified into offline tools and online analysis. While the online analysis techniques are just currently emerging,

the offline tools have a longer history. The following sections discuss related online and offline approaches in the areas of workload shift detection (Section 4.7.1), workload classification (Section 4.7.2) and workload periodicity detection (Section 4.7.3).

4.7.1 Workload Models and Workload Shift Detection

An early work that has identified the need for a deeper understanding of the database workload is REDWAR [YCHL92]. Designed as an offline-tool, it provides a characterization of the SQL trace, like structural information on the processed statements and runtime statistics. The REDWAR analysis results are presented as a report, which can be used to build a benchmark workload or to plan the physical design. Hence, REDWAR is neither built for continuous operation, nor can it identify significant changes in the workload over time automatically. Still, it has identified the criteria of SQL statements that affect DBS behaviour and performance.

Like REDWAR, the physical design advisors shipped by the database vendors ([ACK+04], [DDD+04], [ZZL+04]) are offline tools. They recommend physical design structures for a given workload based on heavyweight algorithms (e.g. knapsack) and expert knowledge. The DBA has to provide a set of SQL statements that represents the typical DBS workload either manually or by recording an actual workload for a certain period of time. As the workload is considered as a "flat" set of statements, the advisors cannot exploit possible performance benefits from patterns in the workload as done by the n-gram workload shift detection. Furthermore, they do not consider periodic behaviour of the DBS workload.

The only work so far which has identified the possible benefits from exploiting the order of statements in the workload is [ACN06]. In contrast to the advisors' set-based computations, this approach may adapt the physical design of the database for every statement (or set of statements). The authors describe algorithms that compute the physical designs which impose the least execution costs. These costs include the estimated costs for both statement processing and access path creation. The proposed approach suffers from the problem that the physical design can only be determined if the entire workload is known in advance. Even if it is assumed that the database workload is cyclic and consists of only one sequence that repeats continuously, a minor shift in the workload would make the analysis results unusable.

An early work on online workload monitoring and analysis is COLT [SAMP07]. It continuously determines the possible benefit of all possible additional indexes by using the database optimizer in a "what-if-mode", i.e. the optimizer determines the statement execution costs assuming the indexes were present. To reduce its own overhead, COLT self-regulates the sampling rate with which it selects statements to be evaluated. The sampling rate is increased when a workload shift is assumed. In contrast to the workload shift detection technique described in this work, COLT does not maintain an explicit model of the workload. Instead, it stores a history of the processed statements to assess the current physical design. A shift is not identified by a significant deviation from a model, but from the estimated benefits from additional indexes. Hence, only workload changes that require changes to the physical design can be de-

tected, whereas all other possible configuration changes to the DBS are not considered. As it lacks an explicit workload model, COLT is furthermore not able to predict periodic workload shifts.

Like COLT, the online tuning approach [BC07] developed by Bruno and Chaudhuri continuously monitors the workload and adapts the physical design accordingly. This approach gathers information about the execution plan from the optimizer during query processing, and then uses a technique described in [BC06] to derive the execution costs for alternative physical designs from it. Thus, it reduces the overhead by not having to issue additional optimizer calls. Furthermore, it also takes into account index interactions and can avoid the problem of oscillation between physical designs. Compared to the workload analysis presented in this work, the online tuning proposed by Bruno and Chaudhuri is strictly focused on the efficient adaptation of the physical design. Other changes to the configuration of the DBS, which may also be caused by workload shifts, are not considered. And as it does not store a model of the workload, it is not able to identify patterns in the workload in order to predict workload shifts.

The Psychic-Sceptic Prediction Framework (PSP) [EM04] is a work by Elnaffar and Martin which has analysed the detection and prediction of shifts from decision support (DSS) to OLTP workloads. Analytical methods are applied to historical workload models offline in order to estimate the time interval for expected shifts. Only during this interval the framework actually performs an online-analysis of workload samples to detect the shift. The PSP does not build a model of the SQL workload, but maintains a list of "DSSness" indicators only. Thus, it is limited to the domain of OLTP to DSS shift detection. Later [MEW06], the authors have identified the general need for both exploratory models (models of the monitored workload) and confirmatory models (models used by the self-management logic). Martin et al. describe in [MEW06] that exploratory models should be used to provide a compact representation of the DBS workload. These models are supposed to be created by unsupervised learning, e.g. clustering, but concepts for actually building these models for DBS workload are not given. However, only the need for these models is motivated, whereas a general approach for building them is not given.

n-gram models have also been used in [YAH05] for creating a statistical model of database workload. This work is directed at the inference of user sessions from SQL statement traces. Heavyweight analysis algorithms are applied on these models in order to identify and cluster procedurally controlled sessions. The information about the user sessions is intended to be used for the prediction of queries based on the queries already submitted, e.g. to optimize the cache replacement strategies.

4.7.2 Workload Classification

Chaudhuri et al. present an approach for the compression of SQL workload in [CGN02]. Their goal is the reduction of workload anaylsis overhead in autonomic DBS functions. For this purpose, they also use clustering techniques and have developed the binary search algorithm that has been extended in Section 4.3.4.1. In contrast to the approach described in this work

Chaudhuri et al. do not give a general distance function for workload events, but strictly focus on the subjects index selection and approximate answering of aggregation queries. They furthermore assume that the entire workload is known in advance, i.e. there are no provisions for stream-oriented processing for on-line self-management functions. The approach also does not actually perform a classification of the observed workload before passing it on, but removes statements from the workload that are considered dispensable.

An approach for a stream-oriented compression of SQL workloads has been reported in [Kol08]. Like [CGN02], it does not use a general distance metric but strictly focuses on a distance function for index selection. Hence, it considers the selectivity of a query as the only possible feature. Though stream-oriented, the approach in [Kol08] does not guarantee consistency of classification results, and it does not support the definition of a class limit. The workload compression again is achieved by retaining only some of the SQL statements in the original workload, which makes the solution unsuitable for workload shift detection, for example.

Several approaches have been developed for the clustering of data streams. Some of these approaches, like [OMM+02] and [OO08], employ on-line versions of the k-Means algorithm. However, the existing on-line k-means approaches are not suitable for DBS self-management functions, because they continuously adapt the centroids to the observed events, and therefore violate the consistency requirement. More advanced approaches like [CT07] and [AHWY03] follow a two-phase scheme: While the current stream is mapped to a micro-clustering or grid in an on-line-component, an off-line component identifies clusters considering the evolution of the stream based on historical information. The tasks for the identification of the nearest clusters and the creation of new clusters in the on-line component are to some extent similar to the approach in this work. However, the described concepts cannot be used because they lack support for consistency, do not support the definition of a quality loss limit, and are not suited for nominal features.

Like clustering techniques, also classification techniques have been adapted to data streams in the past, e.g. [AHWY04] and [FTARS06]. But also these existing data stream classification solutions are not applicable, because they require the availability of a separate training stream containing already classified events, which is not appropriate for DBS workload events.

The approach described in [EMSL08] classifies database workload into decision support (DSS) and OLTP workload. The authors for this purpose take performance snapshots of the DBS and extract the relevant attribute values from this information (e.g. queries ratio, number of sorts, throughput). In order to assign one of the two classes DSS or OLTP to a workload, a decision tree is used as a classifier. The decision tree is learned from classified training data. In contrast to the workload classification described in this work, the classification in [EMSL08] is restricted to the two classes DSS and OLTP. As the approach is based on classification techniques it cannot be generalized to arbitrary workload classes, because this would require the DBA to manually identify these classes and provide the corresponding training data.

4.7.3 Workload Periodicity Anaylsis

[Buc09] identifies periodicities in the number of SQL statements submitted to the DBS. For this purpose the authors also identify the fundamental peaks and harmonics in the Fourier transforms and calculate a periodicity metric on this information. Unlike the periodicity analysis presented in this work, [Buc09] does not allow the identification of periodicities between different *types* of workloads. Instead, only periodic structures in the processing costs caused by the statements submitted to the DBS can be identified. So no conclusions about the appropriate DBS configuration from the workload type is possible. Furthermore the approach described in [Buc09] does not support the definition of a minimum number of pattern repetitions, or an absolute limit for the length of the acceptable fluctuations.

Today's commercial DBMS also offer the storage of historic workload information, e.g. the AWR [DRS+05] in Oracle or the Workload Manager and Event Monitor [CCI+08] in DB2. Although the statistical and usage information of these repositories is exploited for various self-management tasks, it is not yet analysed for recurring workload situations or periodicity.

Following the goal of server capacity management, [GRCK07] detects periodic patterns in CPU or memory demands of enterprise applications. The pattern analysis is performed using Fourier transforms and autoregression. The degree of periodicity is then judged by a combination of the deviation in the time domain and in the value domain. Although the described periodicity detection are in some parts similar, there are some major differences: [GRCK07] assumes that the workload information is a continuous signal, which is not the case for the discrete activation of DBS workload models. Furthermore, the acceptance of fluctuations in the value domain are not suitable for the periodicity detection scenario, and there are no concepts for providing strict limits for the acceptable fluctuations.

The prediction of the workload in a mainframe operating system is discussed in [BBR+07]. In contrast to the approach described in this work, the authors do not analyse the workload for periodicity explicitly, but train neural networks from historic workloads. The results show that this method is applicable for short-term predictions only. [WHYZnt] presents a technique for predicting the workload in terms of CPU utilization in grids. Like [BBR+07], this approach does not focus on identifying long-term periodicity, but predicts the expected short-term grid utilization using time series analysis.

4.8 Conclusions

The workload of a DBS has significant influence on the required DBS configuration, because the workload represents the way the DBS is used in its particular environment. For this reason the DBS configuration should continuously be adapted to the workload. However, as discussed in Section 3, the resulting analysis for a continuous reconfiguration analysis for the entire DBS would be far too expensive. For this reason this section has presented concepts for a lightweight detection of significant changes in a DBS workload. The developed approach is based on the

Table 4.8: Workload Classification Requirements

Requirement	Feature Cl.	Signature Cl.
Classification	✓	✓
Varying Statement Characteristics	✓	-
Controllable Loss of Quality	✓	-
Class Limitation	✓	-
Self-Management	(✓)	✓
Stream-based Processing	✓	✓
Consistency	✓	✓
Lightweight Classification	(✓)	✓

observation that a lightweight shift detection cannot detect situations that *definitely* require a reconfiguration, because this decision again would require a reconfiguration analysis. Instead, it detects situations where a reconfiguration *might be* required. This analysis can be performed with much less overhead.

The processing model for the workload shift detection has been derived from the speech recognition approach, because the key challenges are comparable. The processing model for workload shift detection therefore comprises the stages *monitoring*, *preprocessing*, *classification* and *analysis*. Considering shift scenarios that should be detected (new applications, obsolete applications, usage changes, modified applications, periodically changing workloads), the SQL statement text has been identified appropriate workload information observed in the monitoring stage. The preprocessing then filters the SQL statement texts and – depending on the classification type – may convert the statements to feature vectors. In order to reduce the workload diversity and therefore the workload analysis overhead two classification techniques (signature classification and feature classification) have been developed. Finally the classified workload information is analysed for significant workload shifts, using either the n-gram modelling technique or the two-window approach. But as these techniques alone are not able to detect periodic patterns in the workload, an additional periodicity detection for the workload model activation has been developed. The following paragraphs discuss which of the requirements are met by the workload classification, workload shift detection and periodicity detection techniques that have been developed in this work.

The requirements towards a workload classification solution have been identified and discussed in Section 4.3.1. Table 4.8 lists these requirements and shows whether or not they are met by the signature classification and feature classification solutions. Both approaches obviously perform a *classification*, i.e. they assign the incoming SQL statements to exactly one class. In contrast, the requirement for *varying statement characteristics* is only met by the feature classification, because it allows the selection of adequate features for similarity calculation. The usage of the statement structure for signature classification in contrast cannot be changed. Also, the *loss of quality* cannot be measured or configured for signature classification, and no *class limit* can be defined. In contrast, the correct number of classes for a given quality loss and

Table 4.9: Workload Shift Detection Requirements

Requirement	n-gram Models	Two-Window Models
Permanent Change Detection	✓	✓
Periodic Change Provisions	-	-
Stream-based Processing	✓	✓
Self-Management	✓	(✓)
Resilience to Noise	✓	✓
Compact Description	✓	✓
Comprehensibility	(✓)	✓
Adaptability	✓	(✓)

class limit is automatically derived by the feature classification. The *self-management* requirement is met by both solutions. However, although good default values could be determined for the feature classification configuration parameters in the evaluation (Section 4.6.3), there is a risk that they are sub-optimal in some other environments. Both approaches fully support a *stream-based processing* and the *consistency* of classification results over time. Furthermore, the evaluation has shown that the classification of the SQL statements is lightweight. However, the simple elimination of the actual parameter values of the SQL statements by the signature classification is of course faster.

Table 4.9 lists how well the workload shift detection requirements are met by the different workload shift detection approaches. It can be seen that both the n-gram models and the two-window model are suitable to *detect permanent changes* to the workload. Still, both solutions do not consider long-term *periodic changes* in the workload. Every long-term periodic is identified as a new permanent change. Short-term periodic patterns in contrast are represented in the model and therefore do not cause frequent workload shift alerts. By employing a simple lifecycle model with learning and stable phases, both solutions are fully capable of *stream-based processing*. The self-management requirement is also met by both the n-gram models and the two-window models. However, the n-gram models allow the usage of test-based workload shift detection, whereas the two-window models require the definition of a threshold. Both solutions also provide an appropriate r*esilience to noise*, and can be described and stored in a *compact* way. The two-window models, which are characterized as a set of relative frequencies of the workload classes in the model, are more easily *comprehensible* than the n-gram models (especially if $m \geq 2$). The *Adaptability* requirement demands that the thresholds are adapted to the particular environment automatically. This automatic adaptation to the environment is possible only for the test-based workload shift detection of n-gram models, whereas it has to be performed manually for the two-window models.

As shown in Table 4.10, the periodicity detection solution for workload model activations meets all the requirements identified in Section 4.5.1: *Recurring workloads* can be detected by exploiting the existing similarity measures (perplexity or conformance indicators), all *periodic patterns* are detected by either the Fourier-based approach or the model interval analysis

Table 4.10: Workload Periodicity Detection Requirements

Requirement	Status
Recurring Workload Detection	✓
Periodic Pattern Detection	✓
Pattern Adaptation	✓
Dependability	✓
Robustness	✓
Self-Management	(✓)

algorithm, and an algorithm for the adaptation of the periodic patterns over time has been developed. Moreover, the periodic pattern knowledge is managed in a *robust* way, which avoids discarding the pattern after a single exception to it. The *dependability* of the predicted workload changes depends on the configured minimum number of periodic cycles before periodicity is assumed. Finally, the solution works in a *self-managing* way. Except the setting of the minimum number of repetitions and the acceptable fluctuations no maintenance overhead is caused for the DBA.

As a summary it can be stated that almost all requirements are fully met by the described workload shift detection and prediction solution. However, there still are some parameters of the approach which should be controlled at set-up time by a DBA to ensure that the solution meets the characteristics of the particular DBS environment (especially the workload shift detection threshold, the short-term pattern length, and the minimum number of repetitions for periodic workload). Nevertheless, it is assumed that the overall benefit of an automatic information about significant changes in the workload prevails for a DBA: Most significantly, the information about changes in workload frees the DBA from having to observe the workload manually. Furthermore, the workload models that are created for the purpose of workload shift detection also provide a compact description of the typical workload of the DBS. This information is valuable to a DBA whenever manual reconfiguration decisions have to be made. By comparing the model against the models that have been active in the past, the workload prediction component builds a model history. In addition to the prediction of upcoming workload shifts, this information will also be useful to a DBA, as it for example can be used to schedule maintenance tasks or to optimize the capacity planning for the DBS server. Finally, the workload shift detection can of course be coupled with the autonomic, goal-driven DBS self-management logic described in the following section.

5 Quantitative System Models for DBS

Whenever a workload shift has been detected or there is a risk of missing the high-level goals for key performance indicators like response time, throughput and availability, a reconfiguration analysis has to be performed by the self-management logic. During the reconfiguration analysis, the most appropriate DBS configuration for the current workload, state and goal values has to be found. Hence, the self-management logic requires detailed knowledge about the behaviour of the overall DBS under different configurations.

As discussed in Section 3.3, this work follows the proposal in [IBM05] and stores the knowledge required for the reconfiguration analysis in an external knowledge base. This knowledge base is referred to as a system model. In contrast to implementing the knowledge about the system structure and behaviour directly in the self-management logic, placing it in a separate model has the advantage that it can be easily extended, refined and adapted.

The following sections describe the concepts and techniques that have been developed [HR09] for the definition of DBS system models. Section 5.1 first describes a running example, which will be used in the following sections to illustrate and validate the modelling concepts. From this example the requirements towards the contents of a system model are identified in Section 5.2. Section 5.3 afterwards selects an appropriate modelling technique that allows the straightforward representation of the required contents. This selected modelling technique is afterwards used to define a coarse-grained system model of the IBM DB2 in Section 5.4. Related work is discussed in Section 5.5.

5.1 Running Example

Ideally, an appropriate, ready-made system model of a DBS would be required in order to develop reasonable modelling concepts for DBS in general. If such a model existed, the required contents and therefore the expressiveness of the modelling technique could be derived from it. Furthermore, it could be used to validate the modelling concepts by re-building the model with the selected approach. Unfortunately, such a complete model does not currently exist for any DBMS. Hence, an appropriate substitute is required.

The substitute model has to meet a number of requirements in order to be applicable to derive the requirements and be used for the illustration and validation of the system modelling concepts: Determining mathematical models for DBMS components is a difficult task, which requires extensive experimental evaluations. Hence, the exemplary model on the one hand has

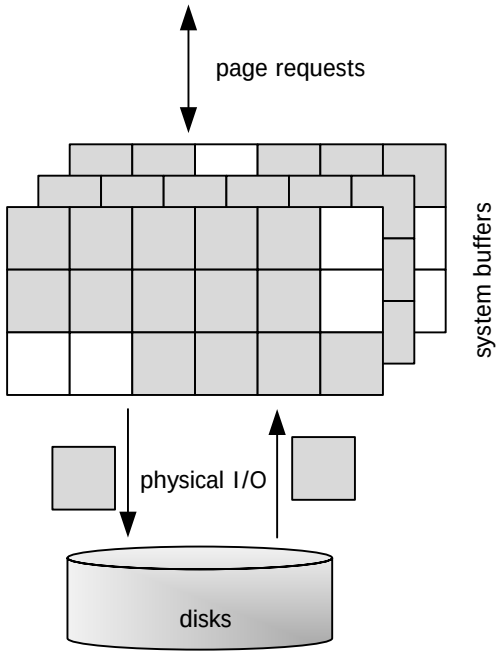


Figure 5.1: Multiple System Buffers in a DBS

to require mathematical models for a small part of the entire DBMS only, i.e. it has to be as compact as possible. On the other hand the model has to comprise all aspects of a complete system model, like sensors, effectors and behavioural descriptions. In particular, it must be possible to link at least one key performance indicator of the overall DBS to its mathematical model. In order to meet these requirements, a simple response time model for a DBS has been developed, which quantifies the response time depending on the configuration of the system buffer component only. Consequently, the model is based on a mathematical model of the system buffer component, whereas all other components of the DBMS are ignored.

Whenever a transaction program reads or updates a record in the database, the system buffer (see Figure 5.1) is responsible for mapping the page that the record is stored in to memory. Pages that already reside in memory are accessible to the transaction programs almost immediately. All other pages have to be read from the physical disks first, i.e., an I/O operation is required, which is by magnitudes more expensive than main memory access [HR83]. The ratio of page requests to the system buffer which can be served without an I/O operation is referred to as the *hitratio*. Obviously, a larger size of the system buffer increases the hitratio, and therefore decrease the average response time. In order to support different access patterns of the transaction programs the system buffer usually can be segmented or partitioned. The IBM DB2 DBMS for example allows the creation of a set of bufferpools [Int06], where for every bufferpool the data that is mapped to it can be defined.

The goal of the system buffer tuning is to minimize the overall data read costs of the DBS. Thus, the sizes of the individual system buffer segments have to be configured in a way that the overall data read costs are minimal. A prerequisite for the decision of how the available memory has to be distributed between the segments is the ability to predict the hitratio for

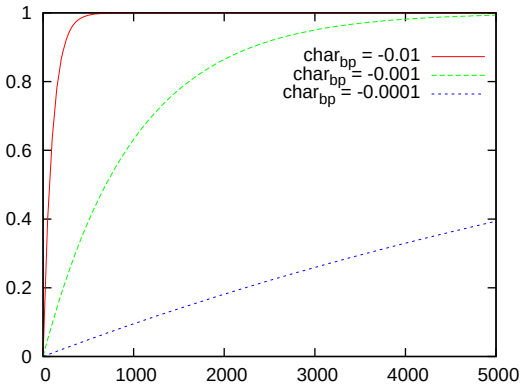


Figure 5.2: Expected Hitratios for different Segment Characteristics

particular sizes of the segments. However, depending on the access patterns of the transactions programs working on the data in the individual segments, the hitratio may react differently to buffer size modifications. In order to predict hitratio given a particular segment depending on its size, a mathematical model has been developed in [BK09]:

$$hitratio_{seg}(x_{seg}) = 1 - e^{char_{seg} \cdot x} \tag{5.1}$$

where $char_{seg}$ denotes the characteristic of the segment, which reflects the locality of the workload in that segment. The expected hitratios for different segment characteristics are given in Figure 5.2 (identical to Figure 3.5). The characteristic of a segment can only be determined by observing the hitratio for particular segment sizes over time.

With the mathematical model for the expected hitratio, it is also possible to approximate the overall data read costs. Using a strong simplification, [BK09] even estimates the average response time of the overall DBS from the hitratios of the system buffer segments as

$$RespT(\mathbf{x}) = \sum_{seg} (1 - hitratio_{seg}(x_{seg})) \cdot avgPageReq_{seg} \cdot syncIO_{seg} \tag{5.2}$$

where x_{seg} denotes the sizes of the system buffer segments seg, $avgPageReq_{seg}$ is the average number of page requests to a segment, and $syncIO_{seg}$ is the average I/O time for that segment (the I/O may vary as the segments may hold data from different disks). Thus, the response time is approximated as the sum of I/O time caused by all cache misses. The time for accessing pages that already reside in the system buffer and for all other statement execution stages (e.g. access plan generation, sorting, ...) is neglected.

Despite being a simplification, the response time model from [BK09] comprises many important aspects of a system model. The elements covered by the model are summarized in Table 5.1. First, the model of course requires a set of sensors, which provide the information about the workload and state of the DBS. These sensors include the information about the synchronous I/O time ($syncIOSensor[seg]$) and the average number of page requests per query ($avgPageRequestsSensor[seg]$) of a system buffer segment. Like the others,

these sensor types have to return values that are specific to a particular system buffer segment. As the number of these segments is not pre-defined, a sensor *segmentsSensor* returning the IDs of the system buffer segments is needed. In addition, the model requires a sensor $bufferCharacteristicSensor[seg]$ for the characteristic $char_{seg}$ of the system buffer segments. However, $char_{seg}$ is a parameter that is specific to the mathematical model for the expected hitratio given in Equation 5.1. Thus, $char_{seg}$ cannot be directly observed from the DBS, but has to be computed from the current hitratio and segment size by rewriting Equation 5.1 to

$$char_{seg} = \frac{1}{x} \cdot ln(1 - hitratio_{seg}) \ . \tag{5.3}$$

The characteristic of a system buffer segment therefore can be computed from the current size and the observed hitratio. As the hitratio again is defined as the ratio of logical reads and physical reads, two additional sensors $logicalReadsSensor[seg]$ and $physicalReadsSensor[seg]$ are required. In contrast to the sensors, only one type of effector is covered in the model: the $bufferSizeEffector[seg]$ controls the size of the system buffer segments. The mathematical model for the expected hitratio constitutes the only behavioural description $hitratio[seg, size]$ in the system model. The approximation in Equation 5.2 links this behavioural description to the high-level response time goal that may be defined by a DBA. Equation 5.2 therefore constitutes a goal function for the overall DBS.

With sensors, effectors, behavioural descriptions and goal functions the above model comprises all important structural components of a system model. However, there are two limitations that might downgrade its usefulness as a running example: First, there is only one type of effector and one DBMS component (the system buffer) which is described by a mathematical model. Nevertheless, by considering several independent system buffer segments, the dependencies between DBMS components still can be expressed with this model: With this approach increasing the size of one segment requires the reduction of the size of another segment, causing an interdependency between the configuration decisions. Second, the model comprises only one goal function that models the response time and therefore should be minimized. But as discussed in Section 3.4, a system model typically comprises several goal function with opposing objectives. As shown in Table 5.1, the model has therefore been extended with a second goal function $ResourceUsage$, which is intended to limit the memory usage and therefore the operation costs. The goal function is defined as

$$ResourceU(\mathbf{x}) = \sum_{seg} x_{seg} \tag{5.4}$$

where x_{seg} refers to the sizes of the system buffer segments. Extended in this way, the system buffer model comprises the important aspects of a system model and serves as a substitute for a complete system model in the following investigations.

Table 5.1: Model Elements

Model Element Type	Name	Description
Sensors	segmentsSensor	Returns the IDs of the different system buffer segments.
	syncIOSensor[seg]	Returns the average synchronous I/O-time for physical reads of the given system buffer segment *seg*.
	avgPageRequestsSensor[seg]	Returns the average number of logical reads to the system buffer segment *seg* per query.
	bufferCharacteristicSensor[seg]	Returns the current characteristic of the system buffer segment *seg*.
	logicalReadsSensor[seg]	Returns the total logical read operations to system buffer segment *seg*.
	physicalReadsSensor[seg]	Returns the total physical read operations caused by page requests to system buffer segment *seg*.
Effectors	bufferSizeEffector[seg]	Sets the size of the system buffer segment *seg*.
Behavioural Description	hitratio[seg, size]	Estimates the hitratio of system buffer segment *seg* for the new size *size* using Equation 5.1
Goal Functions	ResponseTime	Overall average response time for queries to the DBS.
	ResourceUsage	Overall memory usage by the system buffer.

5.2 System Model Requirements

Before a modelling technique for creating DBS system models is developed, the most important requirements for the model contents have to be identified. The following paragraphs discuss these requirements based on the system buffer model introduced in Section 5.1. An overview of the functional requirements towards the system model is given in Figure 5.3.

Refineability As discussed in Section 3.3, the definition of a complete DBS system model from scratch is prevented by the complexity of today's DBMS. Instead, a simplified model which focuses on a small set of components (like the running example) may be created in a first step only. Hence, the system modelling technique must allow the incremental refinement of the system model over time.

Hierarchical Structure When analysing the information in the system buffer model example, it becomes obvious that the model elements refer to different hierarchy levels of the DBS. While the goal functions are defined at the overall DBS-level, the *segmentsSensor* can be clearly as-

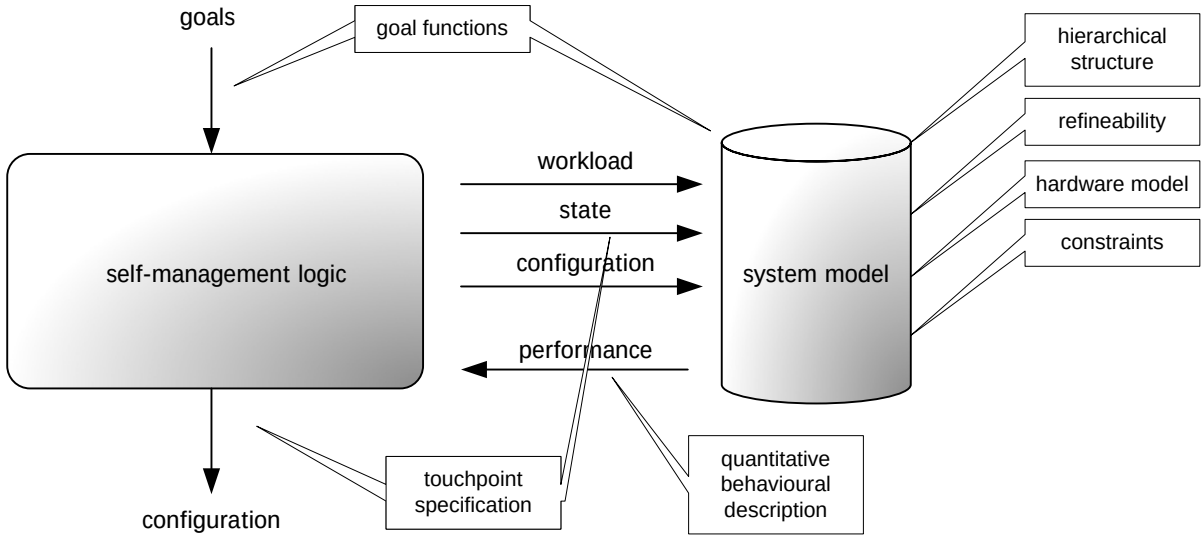


Figure 5.3: System Model Requirements Overview

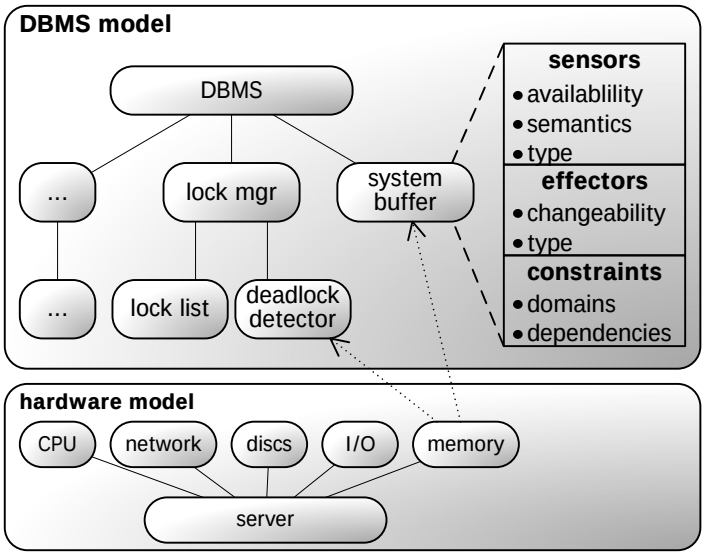


Figure 5.4: Required Information in a System Model

signed to the system buffer only. The $logicalReadsSensor[seg]$ and $physicalReadsSensor[seg]$, for example, reside at an even lower hierarchy level, because their values are specific to one system buffer segment. To allow a straight-forward development, maintenance and refinement of the system model, it is therefore reasonable to reflect this hierarchical structure in the system model. As shown in Figure 5.4, the hierarchical structural description of the DBMS can be built with the DBMS as the root node. Every level in the model then can refine the structural composition of the DBMS (Figure 5.4 is explained in greater detail in the following).

Touchpoint Specification The components that are modelled in the structural system model hierarchy will typically offer a set of configuration options and monitoring information about their performance, workload and state. In the system buffer model example, the size of each

system buffer segment can be configured and the number of logical and physical read operations can be monitored, for instance. Hence, the system model must allow the definition of touchpoints, i.e. sensors and effectors, at every level of the model hierarchy.

To allow an easy refinement and extension of the model, the touchpoint specifications has to contain precise information on the accessibility and meaning of the sensors and effectors. As shown in Figure 5.4, the model must describe how the *sensor* value can be retrieved (*availability*, e.g. from system catalogue or from an analysis function). Furthermore, the meaning of the sensor information (*semantic*, e.g. counter, high water mark, or current value), and its *type* (workload or state) need to be defined. For *effectors* it is necessary to describe their *type* (e.g. configuration parameter, physical design, maintenance function) and whether they can be manipulated online or offline (*changeability*). As in some cases reconfigurations may be associated with severe overhead, the changeability information must also provide information on the expected costs of using an effector, and the time expected until the effects of the reconfiguration are observable.

Constraints The effector values of DBMS components in many cases may not or must not be changed arbitrarily but are subject to *constraints*. For example, the size of the system buffer segments typically requires a minimum size, e.g. 100 pages. These simple constraints on the effector values also have to be stored in the system model and are referred to as *domains*. In addition to the domain restrictions, which are specific to a single effector, there may also be cross-effector *dependencies*. The sum of the sizes of the system buffer segments in the running example, for instance, are limited by the overall amount of available memory. Likewise, the administration manuals of DBMS often describe rules for the values between parameters of several effectors (e.g. "if parameter A is 0 then parameter B must be 1").

Behavioural Description The essential type of constraints for the automatic deduction of reconfiguration actions is the description of the expected *behaviour* of the component. The expected behaviour must be described in terms of a mathematical model of the component, which quantifies the performance of the component depending on its sensor and effector values. Only then it will be possible for the self-management logic to predict the effects of reconfigurations.

An important insight for creating the behavioural description is that only the *observable* behaviour of a component can influence the components behaviour. All other possible influences necessarily have to be neglected. The prediction of the hitratio of a system buffer segment in the running example only depends on the sensor and effector values, but not on any DBS-internal factors that can not be observed by the self-management logic. Thus, a behavioural description is defined as the prediction of the values of performance values based on sensor and effector values.

Goal Functions As illustrated in the running example, the sensors, effectors and behavioural descriptions all serve the purpose of predicting the overall DBS behaviour under different con-

figurations. Hence, the system model has to comprise goal functions which define the prediction rules for the high-level goal values that may be set by the DBA. These goal functions must link the goals to the behavioural descriptions, sensors and effectors of the system model.

Hardware Model Although the system catalogue of today's DBMSs usually also contains a description of the available hardware, there is a clear need to maintain a separate hardware model. The reason is that the information in the system catalogue does not provide information on the performance characteristics and costs of using a particular piece of hardware. This information will be required by the self-management logic in order to minimize the computation costs, while assuring the performance goals. Furthermore, a separate hardware model allows to explicitly represent resource competition between DBMS components. For example, the performance of both the lock list and the system buffer depend on the amount of memory assigned to it (illustrated as dotted arrows in Figure 5.4).

It is important to note that the description of the logical and physical design of a DBS is *not* part of the DBS system model. This information is instead available from the system catalogue. The logical and physical design are factors that influence the performance of the DBS. So from a system model point of view, the logical and physical design will be described as information available via a sensor (that will access the system catalogue), and can be adapted via appropriate effectors.

5.3 System Modelling

The greatest challenge for the development of system models is the complexity of today's DBMS. As a solution to this challenge the requirements analysis in Section 5.2 has identified the refineability and hierarchical structure as one of the key requirements to a system modelling approach. These requirements are most naturally supported by a visual modelling language, which allows the system modeller to only define the most important components at the top-level. More detailed descriptions can be viewed or added by stepping down into the components and sub-components. Thus, a system model may abstract from structural and behavioural details in a first step and add the missing information later where required.

Different visual modelling languages have been compared for the purpose of creating DBS system models in [Kar09]. In particular, the Common Information Model (CIM, [Dis08]) and the System Modelling Language (SysML, [Obj08]) have been investigated for their applicability according to general and use-case-specific criteria. The modelling language that has been selected for the definition of DBS system models is SysML. SysML has been designed to support the specification, analysis, design, and validation of a broad range of systems and systems-of-systems, including hardware and software aspects [Obj08]. As illustrated in Figure 5.5, SysML extends and redefines the UML2 language. To support a general *systems engineering*, the

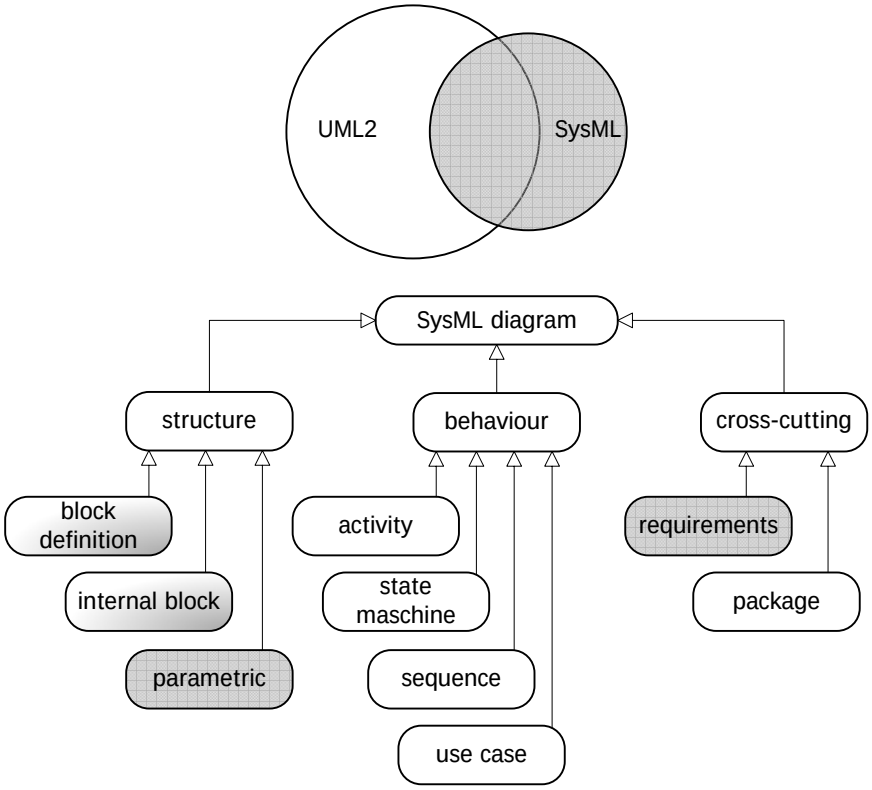


Figure 5.5: SysML Diagram Types [Wei08]

SysML for example redefines the UML class diagram as a *block diagram*. So the contents in a block definition diagram refer to the structural blocks of a system, and not to classes of an object-oriented application. Likewise, the UML composite structure diagram has been redefined as *internal block diagram* and is used to define the internal structure of a block, i.e. its parts and their connections.

The general modelling approach followed by SysML – a visual definition of hierarchically structured systems – matches the *hierarchical structure* and *refineability* requirements of DBS system models. In addition, there are special provisions for representing the available hardware in the block diagrams and for assigning the functional blocks of the system to hardware components (for details see [Wei08]). However, the most important reason for the selection of the SysML language is the new parametric diagram. The parametric diagram allows the connection of particular model elements via *constraints*. Constraints are defined as specialized blocks and comprise mathematical descriptions of system-wide invariants. In a parametric diagram, the parameters of constraints then can be connected to model elements at any level of the structural hierarchy. So by using constraint blocks for the definition of the behavioural descriptions of a DBS model, the system-wide dependencies in a DBS can be modelled in a visual and intuitive way. In the following, the usage of SysML for DBS system models is described in detail. The running example is modelled with SysML for this purpose.

The basis for all descriptions in the system model is a structural description of the DBMS and the available hardware. As the running example only covers the buffer pool, the most important

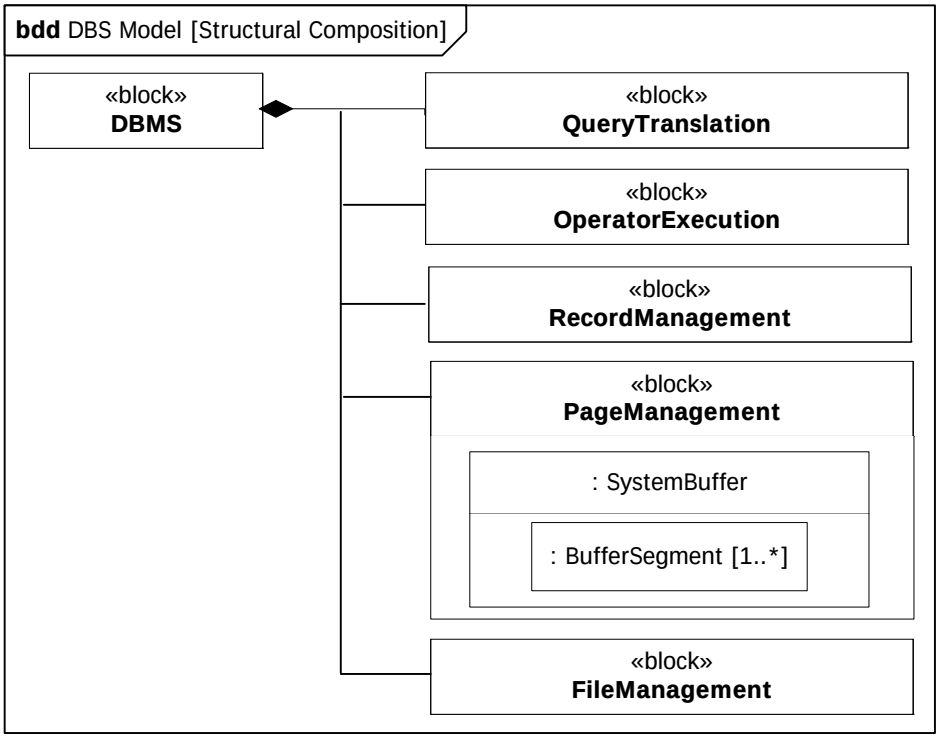


Figure 5.6: Running Example: DBMS Structure Definition

components of the overall DBS are modelled according to the general layered DBS architecture described in [HR83] (see Figure 3.1). As shown in Figure 5.6, a SysML block definition diagram is used to model the DBMS according to the layered architecture. Aggregation associations are used to express the composition of an overall DBMS from the layers defined in [HR83]. The page management layer, whose structure is more precisely defined by the running example, is extended with an internal block diagram description. It shows that the page management layer contains a system buffer, which again contains one or more system buffer segments.

The block definition diagram in Figure 5.6 is well suited to depict the structural composition of a DBMS. Every block in this diagram can furthermore be extended with compartments for the definition of the corresponding sensors and effectors. However, to increase readability when more blocks are added, also an additional block definition diagram can be created for this purpose. Figure 5.7 illustrates this separate definition of the sensors and effectors from the running example: The block *buffer segment* is extended with all segment-specific sensors from Table 5.1. In addition, the *bufferSizeEffector* is assigned to this block. The *segmentsSensor*, which returns the IDs of the available segments, in contrast must be a part of the overall system buffer. The goal functions in the running example refer to the overall DBS response time and resource usage. Hence, appropriate sensors (*ResponseTimeSensor*, *ResourceUsageSensor*) at the DBS-level are required which allow the monitoring of these values for possible goal violations.

It is important to note that in Figure 5.7, the sensors and effectors in the structural definition diagrams all refer to specific types. Hence, every sensor and effector has to be represented by

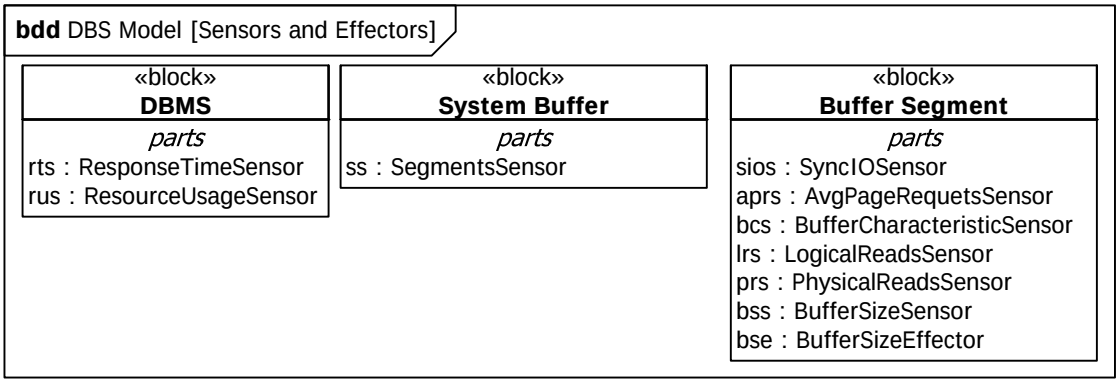


Figure 5.7: Running Example: Sensor and Effector Definitions

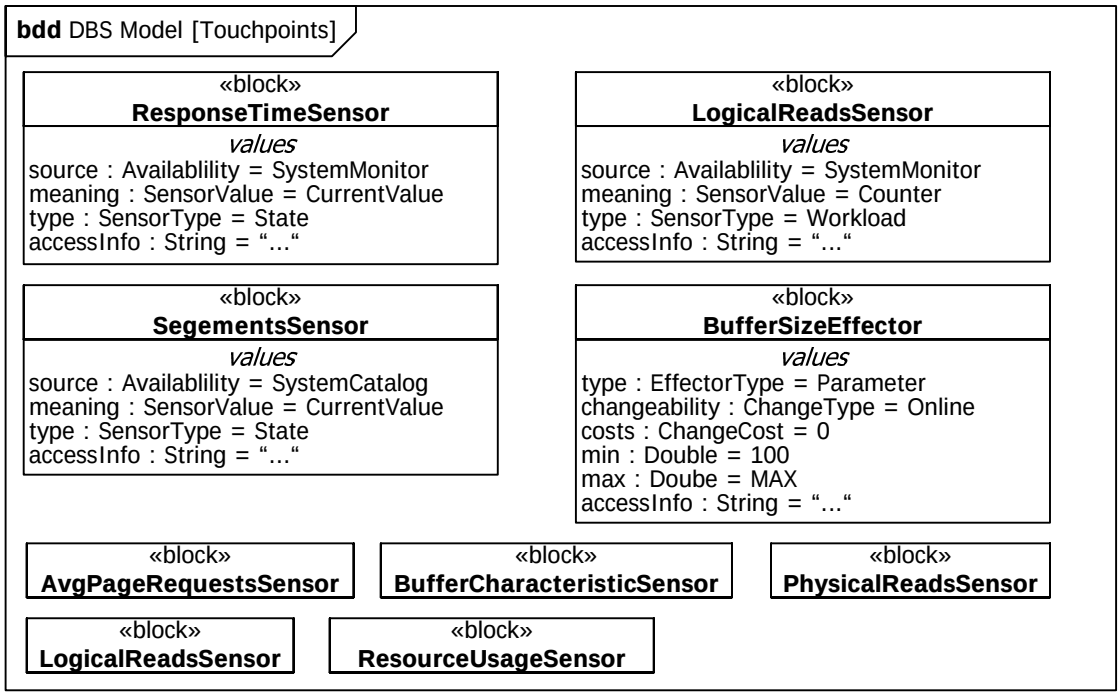


Figure 5.8: Running Example: Touchpoint Specifications

a separate `Block` in the model, as specific characteristics must be stored in order to provide the necessary *touchpoint specification*. The required touchpoint information is illustrated in Figure 5.8 (the full information is illustrated only for the first four types). For every sensor its source, meaning, and type has to be defined. Furthermore, specific access information like a command string or parameter value may be required, and minimum and maximum values for effectors. In order to assure that the self-management logic can interpret the parameters at runtime, the source, and meaning parameter values must be chosen from a predefined list (e.g. *Counter*, *HighWaterMark*, and *CurrentValue* for meaning).

In addition to the structural information about DBMS components, the DBMS model has to describe the effectors' dependencies and the components' behaviours. For these purposes the `ConstraintBlock` element is used to define standard mathematical expressions as `ConstraintRules`. Figure 5.9 illustrates the representation of the behavioural description *hitratio[seg,size]*

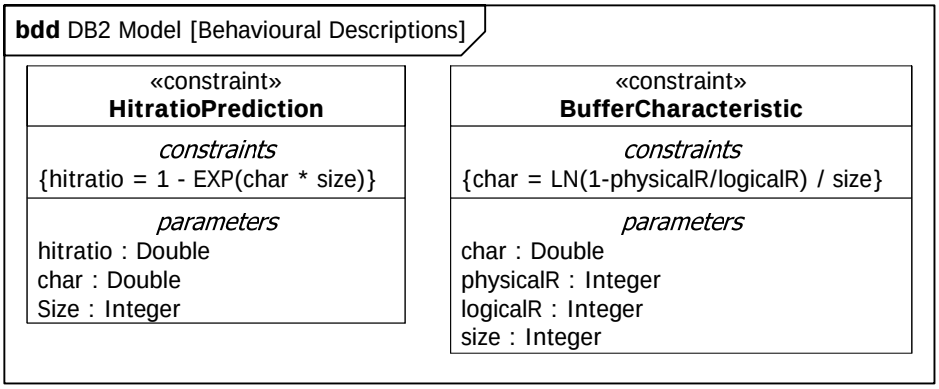


Figure 5.9: Running Example: Constraints

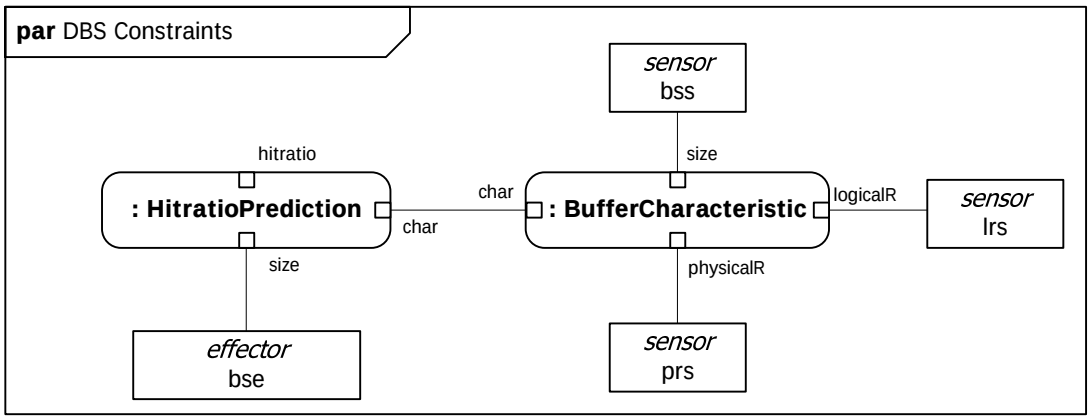


Figure 5.10: Running Example: Parameter Specifications for the Constraints

as defined in the running example in a SysML `ConstraintBlock` *HitratioPrediction*. The `ConstraintBlock` defines both the mathematical expression and the types of its parameters. The calculation rule for the bufferpool characteristic (see Equation 5.3) is also defined in a `ConstraintBlock` in Figure 5.9. As described above, the SysML parametric diagram is used to link the parameters of the constraints to the sensors and effectors of the structural DBMS model. The parametric diagram for the hitratio prediction constraint is given in Figure 5.10. The rounded rectangles in this diagram refer to instances of the constraints defined in Figure 5.9. The parameters of the constraints are referred to as `ConstraintProperties` by the SysML standard and depicted as little white squares within the constraint instances. For every parameter the constraint instances either define a link to a sensor or effector of the model, or they connect it to the result parameter of another constraint instance (as shown for the buffer characteristic *char* in the example). It is important to note that the size of the buffer segments is used in two ways in the example: on the one hand it is used as a sensor value by the *BufferCharacteristic* constraint instance, while on the other hand it is used as an effector by the hitratio prediction constraint. This distinction is important because effectors mark variables whose values may be optimized by the self-management logic. As in case of the *BufferCharacteristic* constraint its value must not be modified, but only the current value has to be read, it has to be modelled as a sensor.

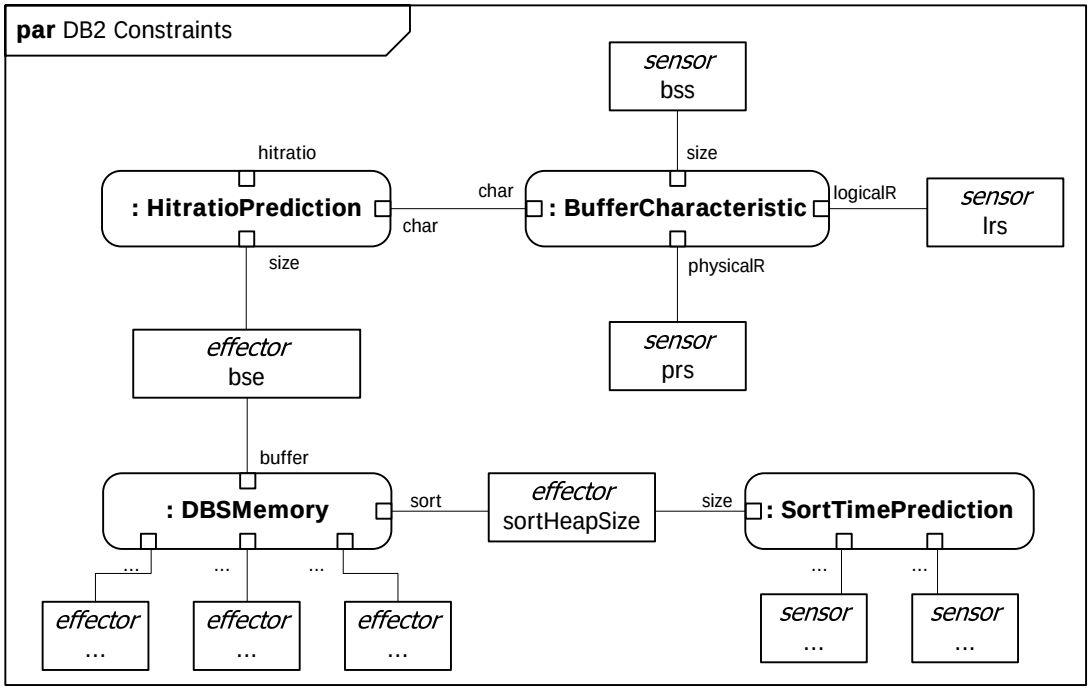


Figure 5.11: Modelling Dependencies in Parameteric Diagrams

Using the parametric diagram, dependencies between the different DBS components can be modelled in a straight-forward manner: The side effects of effectors can be easily defined by connecting the effectors to all relevant constraint instances. An extension to the running example which illustrates the definition of dependencies is shown in Figure 5.11. In this example the buffer size effector (*bse*) has additionally been linked to a constraint instance which defines resources competing for DBS memory. Besides others, also the sort area requires DBS memory. A self-management logic evaluating the exemplary model therefore can determine that by increasing the buffer size the hitratio can be increased. However, it can also see that the sort area size has to be reduced at the same time, which would increase the predicted sort time.

As discussed in Section 5.2, the description of the system behaviour must be related to the goal definitions in order to decide whether or not the goals defined by a DBA will be met. Hence, the system model is extended with *goal functions*. These are modelled as additional `ConstraintBlocks` and represent the high-level goals that may be set by a DBA. Typically these goals will refer to properties such as response time, throughput, resource usage, availability and operation costs. Considering the visual modelling approach, other goal functions may be added on demand, of course. Each of the goal functions must quantitatively describe how its value depends on the DBS configuration. Two examples for quantitative goal functions have been given for the response time and resource usage in the running example in Equations 5.2 and 5.4. Figure 5.12 illustrates how these functions can be defined in SysML using `ContraintBlock` model elements in a block definition diagram.

Unlike the previous constraint definitions, the goal functions in Figure 5.12 refer to a *set* of DBMS components: the number of system buffer segments is not known at modelling time and

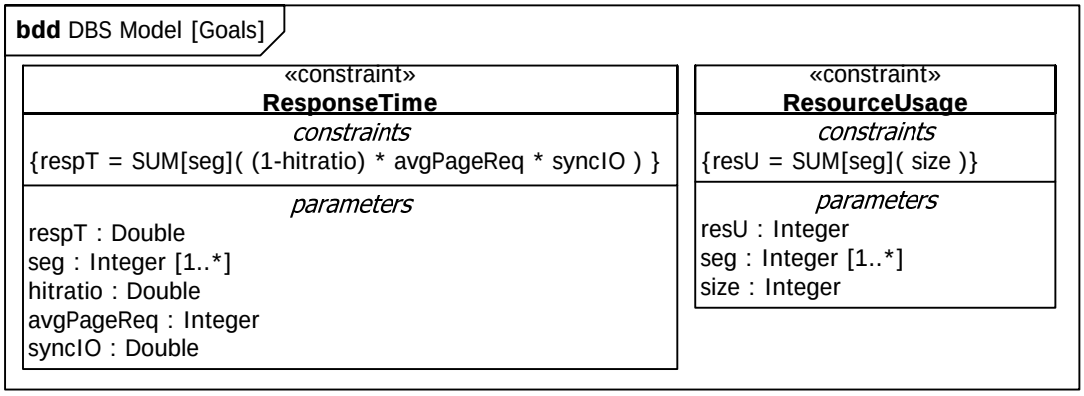


Figure 5.12: Running Example: Goal Functions

may be changed via configuration. Hence, the goal functions define an aggregation function *SUM* to summarize the values of all existing system buffer segments. For this reason the standard expressions are typically supported in the *ConstraintRules* of *ConstraintBlocks* with aggregation functions. The corresponding syntax is

$$AGG[aggVar](<expression>) \ ,$$

where *AGG* denotes the aggregation function (e.g. SUM, AVG), *aggVar* denotes a (multi-valued) variable that defines the identifiers of the instances, and *<expression>* defines the expression that has to be aggregated. For the example in Figure 5.12 the aggregation variables in the expressions (*seg*) refer to the IDs of the system buffer segments. The suffix *[1..*]* marks this parameter as multi-valued.

In order to be able to actually predict the goal values, these goal functions must be linked to corresponding sensor and effector values of the structural DBMS model. For this purpose again the SysML parametric diagram is used. To avoid having to refine the goal functions down to individual sensors and effectors, the goal functions can be added to existing parametric diagrams for the behavioural descriptions. Thus, the result values of existing constraints can be employed in order to define the goal function mappings. These goal function mappings are given in Figure 5.13. The *avgPageReq*, *syncIO*, *seg*, and *size* parameters of the *ResponseTime* and *ResourceUsage* constraints here are linked to the corresponding sensors of the DBMS model. In contrast, the *hitratio* parameter of the *ResponseTime* constraint is simply linked to the result value of the *HitratioPrediction* constraint instance. Thus, a self-management logic can determine that the response time can be optimized by modifying the buffer size effector of the buffer segments. As shown in the figure, the goal functions always must constitute the top of the constraint hierarchy, whereas the leafs must be sensors and effectors. It is important to note that the *SystemBuffer.ss* sensor that returns the set of system buffer segment IDs is linked to the constraint parameters like any other single-valued sensor. A self-management logic therefore has to correctly expand the aggregation functions and query the sensor values several times (once for every value in the multi-valued sensor's result).

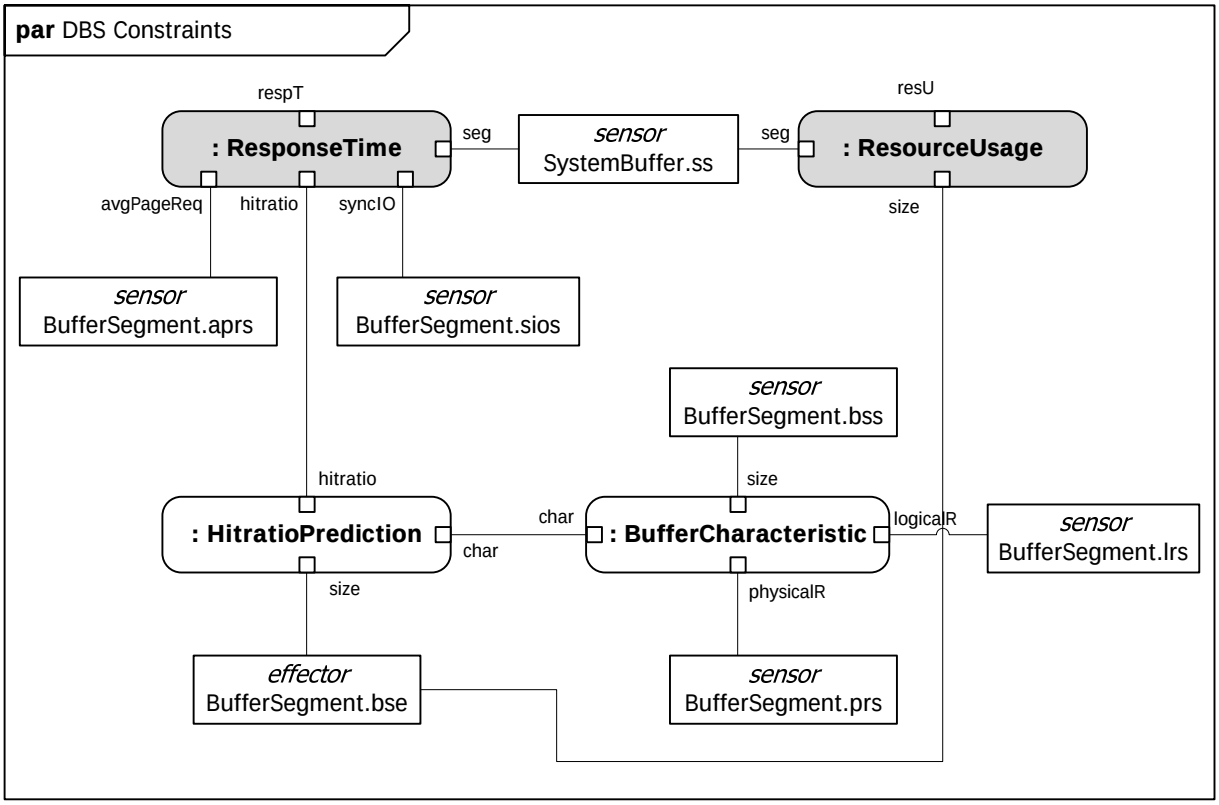


Figure 5.13: Running Example: Parameter Specifications for the Goal Functions

To allow a straight-forward declaration of goals with distinct values for specific service classes, the following rule syntax in the `ConstraintBlocks` has been chosen: The name of a goal function may be followed by a service class name in squared brackets. For example, the rule in Figure 5.12 can be rewritten as

$$respT[sc] = SUM[seg](1 - hitratio) * avgPageReq * syncIO \ .$$

The corresponding constraint parameter *sc* in this case of course also has to be connected to an element that returns the names of the service classes. The name of the service class property is then implicitly available as a parameter in all those sensors and effectors, which are (transitively) connected to the goal function constraint. For example, the sensor *BufferSegment.lrs* could consider the *sc* parameter to be able to return service-class-specific numbers of logical reads to the buffer segments.

5.4 DB2 System Model

After the general system modelling techniques have been introduced using the running example, this section describes the definition of a concrete system model for IBM DB2. The resulting system model comprises both a structural and behavioural description of DB2. The goal of the creation of the system model is two-fold: on the one hand it is intended as a proof of concept

to show that the system modelling techniques described in Section 5.3 can be applied for a concrete DBMS, and on the other hand the approach selected for creating the system model is intended to serve as a blueprint for other system models in the future.

As discussed in Section 3.3, the creation of a complete and exact system model is a challenging task that can only be attained by an iterative refinement of an initial coarse-grained model. This kind of coarse-grained initial model is constructed in the course of this section. The procedure for the creation of the model has been based on the selection of *one* key performance indicator (KPI) of the overall DBS, whose dependency on the DBS configuration is supposed to be described. The selected KPI is the *response time*, whereas all other possible KPIs like availability, throughput, or operation costs are not considered. After the selection of the KPI, a detailed analysis of the DB2 manuals ([Int06], [AFG+04], [Int09]) has been performed in order to identify the most important structural components of the DB2. For each of the components the configuration parameters with considerable impact on the response time KPI have been determined. Together with the required sensor information, these components build the coarse-grained structural system model, which is described in Section 5.4.1.

Following the identification of the major structural components, a quantitative mathematical model for the dependency of the KPI on the configuration of the components has been derived. The approach selected for this purpose is similar to the approach for the structural description, because the mathematical models are also built based on the theoretical descriptions given in the DB2 manuals (Section 5.4.2). These hypothetical models predict the share of the overall response time of every component in the structural model based on its sensor and effector values. The models have then been validated by performing experimental evaluations of the observable behaviour under different workloads and states. The evaluation framework that has been developed for this purpose and the corresponding evaluation results are described in Section 5.4.3.

The DB2 system model that is constructed in the following sections serves the purpose of a proof of concept. However, the results of the experimental evaluation in Section 5.4.3 show that – despite the simplifications and approximations that are made during the construction of the system model – quantitative predictions for the DBS performance are possible. Furthermore the resulting model serves as a basis for future refinements, and as a blueprint for system models for other DBMS.

5.4.1 Structural Description

The overall structure of IBM DB2 has been derived from the DB2 manuals. In particular, the Performance Guide for High Performance OLTP and BI [AFG+04] provides a detailed description of the DB2 components, their interaction, and their important tuning knobs. An overview of the resulting structural model is shown in Figure 5.14. The following subsections discuss the functionality and properties of the blocks in the model. For every component the most important effectors and sensors are given, too.

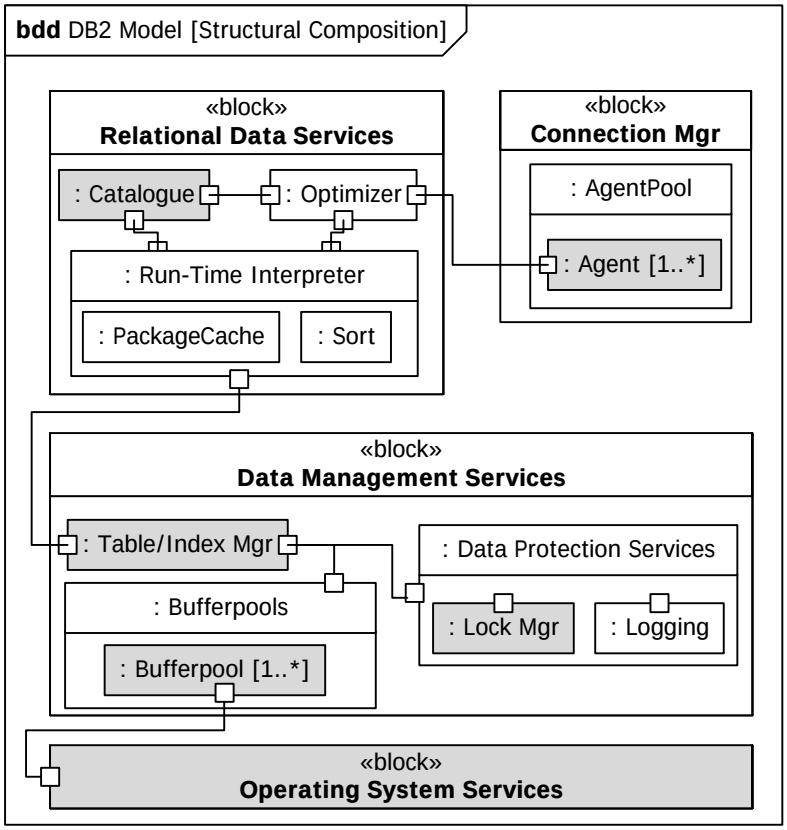


Figure 5.14: Overview of the Structural Description for IBM DB2

5.4.1.1 Connection Manager

The connection manager (*ConnectionMgr*) handles the connections and communication between the DBS and the client applications. For every application that establishes a connection to DB2 an *Agent* is created on the server-side. The agent is responsible for the execution of the client requests on the DBS and for returning the results back to the client. According to [Int06] the creation of the required agent is an expensive operation, which should be executed as rarely as possible. To avoid these costs for most connections, DB2 manages a pool of agents (*AgentPool*). Whenever an application requests a connection, one of the spare agents in the pool – if present – is assigned. An agent is returned to the pool as soon as the client connection is closed. Thus, new agents only have to be created when there are no spare agents available in the pool.

Effectors The most important configuration option for the connection management performance is the size of the agent pool (*NUM_POOLAGENTS*). The larger the pool is, the smaller is the probability that an agent has to be created and the smaller is the share of the response time caused by the connection. However, every agent requires a certain amount of memory, which is not available for other components then. Thus, increasing the pool size may increase the share of response time caused by other components.

Sensors The objective of agent pool size tuning is the minimization of the average connection establishment duration (*AVG_CON_DUR*). As this information cannot be directly retrieved via one sensor in DB2, it has to be calculated from two other monitoring elements: DB2 supports the monitoring of the connection request time (*appl_con_time*) and the timestamp of the connection establishment (*conn_complete_time*). The average connection establishment duration therefore has to be calculated as the average value of the difference of these two monitor elements. Another important sensor of the connection management is the total number of connections (*TOTAL_CONS*), which can be directly retrieved from the DB2.

Dependencies The performance of the connection management depends on the available memory for agents in the pool. Although memory is required by every component, the largest effects of resource competition can be expected with the system buffer and the sorting component.

In addition to the dependencies on other components there are also two configuration options specific to connection management, which have a significant impact on the behaviour, but which are not considered in this coarse-grained model: First, the behaviour of the connection management depends on whether or not the DB2 connection concentrator is activated. If the concentrator is activated multiple connections can share one agent. The concentrator is activated if the value of the effector *max_connections* is larger than the value *max_coord_agents*. In this model the connection concentrator is assumed to be deactivated. Second, the number of agents initially available in the pool can also be configured (*num_initagents*), which may be smaller than the actual pool size. Thus, up to *NUM_POOLAGENTS* agents are created only when they are required. To keep the model simple, the number of initial agents is assumed to be the same as the pool size.

5.4.1.2 Relational Data Services

The block *RelationalDataServices* covers the functionality of the two uppermost layers of the layered DBMS model described in [HR83]. On the one hand it comprises the *Optimizer* component, which creates efficient execution plans for the queries issued by the applications. On the other hand it also executes the generated plans in the *RunTimeInterpreter*, and comprises the meta-data *Catalogue*. The effectors and sensors of these components are discussed in the following.

Optimizer

The DBS optimizer is responsible for creating procedural execution plans for the descriptive SQL queries submitted by the client applications. The procedural execution plans consist of plan operators like table scans, index access, sorts, and aggregations. The implementations of these plan operators are provided by the Run-Time Interpreter of IBM DB2. In order to find execution plans that are as efficient as possible, the optimizer performs both a syntactical and

cost-based optimization. The overhead that the optimizer is allowed to cause for evaluating different query execution strategies can be controlled by the DBA.

Effectors The configuration parameter *DFT_QUERYOPT* defines the default optimizer level that is used for creating the execution plans. The higher the optimizer level, the more effort is spent on the optimization of the queries, because a larger set of heuristics and operator implementations are considered as alternatives for the execution plan generation. Thus, a high optimization class may cause more compilation overhead, but reduce the actual execution time (and therfore the overall response time). Consequently, the IBM DB2 documentation recommends the usage of low optimizer levels for OLTP environments and high optimizer levels for OLAP/DSS workloads, where the complex queries and large data volumes may cause very large statement execution times.

Sensors In order to assess the current configuration of the optimizer, two sensors are required: a sensor for the average compilation time (*AVG_COMPILE_T*) and a sensor for the average statement execution time (*AVG_EXEC_T*). These two sensor values can be calculated from the two DB2 monitor elements *prep_time_worst* and *total_exec_time*.

Dependencies The execution plans generated by the optimizer are stored in the package cache until they have to be replaced by another execution plan. Hence, a larger package cache may compensate the additional compilation times of larger optimizer levels, because the probability of execution plan re-uses are increased.

Sorts

Complex SQL queries often require the sorting of the underlying data. The most important factor of sorting is whether or not the sorting can be executed in memory. If the sorting cannot be performed in memory, an external sorting mechanism has to be used, i.e. intermediate results have to be written to disk.

Effectors Due to the significant overhead for writing intermediate sort results to disk, the choice of the sort heap size is a critical factor for the overall response time. The sort heap size is configured with the effector *SORTHEAP* and determines the amount of memory that may be used for sorting by every agent.

Sensors The important sensors that provide information about the performance of the sorting component are the total time spent for sorting (*TOTAL_SORT_TIME*) and the total execution time of all statements (*TOTAL_EXEC_TIME*). These two sensors can be directly retrieved from DB2 monitoring elements.

Dependencies As the performance of the sorting component mainly depends on the available memory, it obviously a depends on all other components that require memory. In particular, the system buffer performance will be degraded if the sort area is increased. But additional writes to disk for intermediate results may also affect the performance of read and write operations for the physical database or the logging data.

In addition to the dependencies on other components, the effects of changing the *SORTHEAP* effector also depends on whether or not shared sorts are activated. However, shared sorts are not considered in the coarse-grained DB2 model developed in this work and are therefore assumed to be switched off.

Package Cache

Being part of the Run-time Interpreter, the package cache is a memory area that caches execution plans for SQL statements. Whenever a SQL query is submitted to the DBS and the agent finds an appropriate execution plan for the query in the package cache, the query does not have to be re-compiled again.

Effectors The effect of the package cache on the overall response time is determined by its size, which is controlled by the DB2 parameter *PCKCACHESZ*. The larger the size of the package cache, the more execution plans can be stored and possibly re-used, and the more compilation time can be saved on an average.

Sensors The sensor that provides the important information for assessing the performance of the package cache is its hitratio *PCK_CACHE_HITRATIO*. As the DB2 does not provide a monitor element that directly reports this value, it has to be calculated as the ratio of the package cache inserts (pkg_cache_inserts) and the package cache lookups (pkg_cache_lookups).

Dependencies The benefit caused by a larger package cache depends on the actual compilation times for SQL queries: The greater the compilation times, the greater the benefit of an increased package cache size. Thus, there is a dependency between the package cache size and the optimizer level. In addition, there is of course a dependency to all other DB2 components whose performance depends on the available memory, e.g. the sorting and bufferpool components.

Catalogue

The system catalogue stores the meta data about the database. It is mainly required by the optimizer component for the compilation of queries and for checking authorization information. The persistent system catalogue information is cached by the DB2 in memory, and the size of the cache is subject to configuration. However, as the effects on the overall response time of

the catalogue cache size are considered small in comparison to the other system components, the catalogue cache size is not modelled in this coarse-grained DB2 model.

5.4.1.3 Data Management Services

The execution plans for SQL queries that are executed in the Run-Time Interpreter are based on the implementation of operators like index access and table scans. These operators provide access to the data stored in the DBS by retrieving individual records from the Data Management Services component. The *Table/Index Manager* component in the Data Management Services is responsible for offering the required record retrieval and storage functionality to these operators. It is therefore the task of the Table/Index Manager to extract the records from the physical pages stored in the system buffer. The system buffer may be split into multiple segments in DB2. These segments are referred to as *Bufferpools*. In order to provide the transactional properties of isolation and atomicity, the Data Management Services also comprise a *Locking* and a *Logging* component, which are grouped together as the *Data Protection Services*. The effectors, sensors, and dependencies of the components in the Data Management Services are discussed in the following paragraphs.

Table/Index Manager

The Table/Index Manager retrieves the data pages from the bufferpools and extracts the requested records from them. In addition, it applies any "sargable" predicates to them, i.e. all predicates which can be evaluated from checking the field values of the current record only (for details see [AFG+04]). The most important configuration option of the Table/Index Manager is the decision of which indexes would provide the most benefit for the current workload. As indexes are one of the most important measures for improving the performance of a DBS, the development of automatic indexing techniques has been in the focus of self-managing DBS research for a long time. Many approaches for this purpose have been published in the past (see Chapter 2). Due to this large amount of existing solutions to this problem, the modelling of an index benefit model has been excluded from the coarse-grained DB2 model in this work. However, the existing approaches could serve as a sound basis for the refinement of the DB2 system model in the future.

Bufferpools

The bufferpools component implements the DB2 system buffer. The functionality of this component has been described in detail in Section 5.1.

Effectors The size of the system buffer is one of the most important configuration parameters of a DBS. In DB2, there typically is a set of bufferpools, i.e. buffer segments, whose sizes can be controlled individually. Consequently, the sizes of the bufferpools are not set via a

single configuration parameter, but via a special DB2 command (`alter bufferpool`). In the following this effector is referred to as *BP_SIZE*.

Sensors The sensor which provides the information about the current performance of the bufferpool is its hitratio (*TOTAL_HIT_RATIO_PERCENT*). This value is directly available as a DB monitor element. In order to assess the impact of a bufferpool size change on the overall response time of the DBS it is furthermore important to know the number of logical references to the particular bufferpool. This information is provided by the *POOL_L_READS* sensor, which can be easily calculated from the *pool_data_l_reads* and *pool_index_l_reads* monitor elements. Furthermore, DB2 allows to assign particular data partitions to each of the bufferpools. Hence, the observed latencies for retrieving a data page from disk may vary between the bufferpools. The average synchronous I/O time for each bufferpool is returned by the sensor *AVERAGE_READ_TIME_MS*, which is available from the DB2 monitor element of the same name. As the number of bufferpools in a DB2 depends upon configuration, a sensor *BUFFERPOOL_IDS* that returns the IDs of the existing bufferpools is also required, of course.

Dependencies The bufferpool performance mainly depends on the available memory, i.e., its reconfiguration decisions interact with all other DBS components that require memory. However, there are also many other dependencies: if for example an index was created that avoids many scans of a large table, then the hitratio of the bufferpool might improve drastically. Likewise, the re-assignment of the data to a faster disk reduces the synchronous I/O time for that bufferpool and therefore reduces the overall response time. Also choosing a higher optimizer level might lead to more efficient execution plans, which again increase the hitratio because no data is read unnecessarily.

Logging

For performance reasons modifications of the data in a DBS are usually not forced into the physical database immediately at the commit of a transaction. Instead, the modifications are first cached in a logbuffer, and sequentially written to a logfile upon commit of a transaction. In case of a system crash it may therefore be required to restore the last transaction-consistent state of the DBS using a recovery mechanism.

Effectors Considering the goal function to be modelled (response time), the most relevant effector is the logbuffer size *LOGBUFSZ*. The greater the logbuffer, the less often its contents have to be flushed to disk before a transaction actually has issued a commit. For other goals of course other effectors are relevant, too, e.g. the frequency of logfile backups for availability or the logfile size for throughput.

Sensors In order to assess the performance of the logging component, the information on the number of times the logbuffer had to be flushed because it was full is most relevant. This

information is available via the *NUM_LOG_BUFFER_FULL* sensor, which can be directly retrieved from the corresponding DB2 monitor value. Furthermore, the total number of written log pages (*LOG_WRITES*), the total number of update operations (*NUM_UPD_STMTS*) and the time spent for writing the log pages (*LOG_WRITE_TIME*) are important sensor information for tuning the logbuffer size.

Dependencies As for the other components that require memory, the tuning decisions for the logging component interact with all other memory-dependent components. Furthermore, the group-commit-option is an important configuration option, which allows to delay the flushing of the logbuffer for a certain period of time in order to wait for other transactions to commit. For the sake of simplicity, this option is assumed to be switched off for the DB2 model constructed in this work.

Lock Manager

The locking component of the data protection services is responsible for the synchronisation of concurrent transactions. It manages the locks set on data objects and grants or denies the read/write operations of the transactions. As the effects on the *response time* can be assumed to be constant as long as the system is not congested, it is not considered in the DB2 model. In contrast, this component would be of highest importance for modelling the *throughput* of the system, because it has a significant influence on query concurrency.

5.4.1.4 Operating System Services

The operating system services are responsible for the persistent storage of the data in files and the access to the files using the operating system facilities. Hence, the many physical design options offered by DB2 (like data partitioning, assignment of data files to disks, clustered storage of the data, reorganisation) form the configuration decisions for this component. However, due to the complexity of these decisions a quantitative prediction model has not been investigated in this work. The operating system services component is therefore not part of the coarse-grained DB2 model.

By omitting the operating systems services configuration from the model, the underlying physical design is assumed to be constant. From the perspective of a self-management logic, this reduces the number of configuration options, i.e. the number of effectors. Hence, the partitioning, disk assignment, and clustering of the data cannot be changed by the self-management, because these DB2 configuration options are “hidden”. However, given that this physical design is not changed externally, the models for the other components form an overall model which is perfectly valid for a self-management logic.

5.4.2 Behavioural Description

As discussed above, the system model in this work focuses on the prediction of the response time as the only goal function. The following sections therefore describe quantitative models for predicting the response time depending on the workload, state and configuration. Only the components identified as relevant in the previous Section 5.4.1 are investigated for this purpose.

Section 5.4.2.1 first develops a model for predicting the overall DBS response time. The Sections 5.4.2.2-5.4.2.7 afterwards discuss quantitative response time models for every relevant DB2 component in detail.

5.4.2.1 Overall Response Time Model

The basic assumption for creating an overall response time model is that each of the DB2 components adds a certain delay to the query processing time. Hence, the overall response time function shown in Figure 5.15 models the response time as the sum of the response times caused by the DB2 components that have been identified as relevant for the structural model. That is, the delays caused by the connection management (RT_CON), bufferpools (RT_BUF), sorting (RT_SOR), logging (RT_LOG), and re-compilations due to package cache misses (RT_PKG) are summed up in order to estimate the overall response time of the DBS (RT_DBS). The optimizer level in this response time model has a special role: As the optimizer level determines the efficiency of the execution plans, it also determines how much effort is spent at the lower levels for actually processing a query. Thus, in the coarse-grained DB2 model, the optimizer level affects the overhead caused in the bufferpool, sorting and package cache components. In contrast, the delays caused by the connection management and logging components are not influenced by the optimizer level: While the connection management delay in the processing chain is caused before the optimizer level becomes relevant, the logging component causes overhead only for insert, update, and delete operations. In order to model the influence of the optimizer configuration on the overall response time, the quantitative model in Figure 5.15 therefore considers the optimizer level RDS as a factor that is applied to the delays of the affected components.

As described previously, the system model has to map the parameters of the goal functions to sensors and effectors in the structural system model. For the response time model, the parameters RT_CON, RT_BUF, RT_SOR, RT_LOG, and RT_PKG therefore have to be mapped to the sensors and effectors described in Section 5.4.1. But as performing this mapping in a single step is a complex task and the resulting model would be difficult to maintain, the mapping is instead performed in layers. Each of the following sections therefore analyses the behaviour of only one of the components in detail and describes a quantitative model that predicts the components's behaviour depending on the sensor and effector values. Every component-specific model provides one top-level constraint, whose "return value" matches the parameter required by the overall response time goal function. Figure 5.16 illustrates the parametrization of the response time goal function in a SysML parametric diagram.

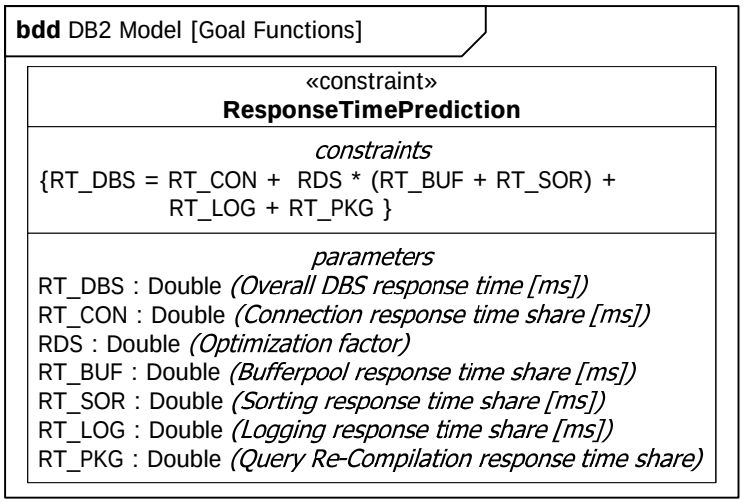


Figure 5.15: Overall DB2 Response Time Model Constraint

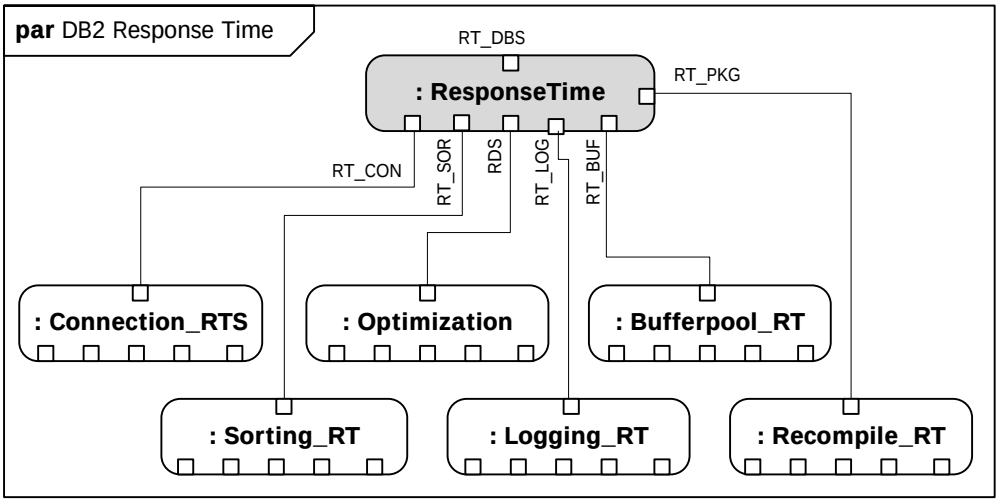


Figure 5.16: Overall DB2 Response Time Model Constraint Parametrization

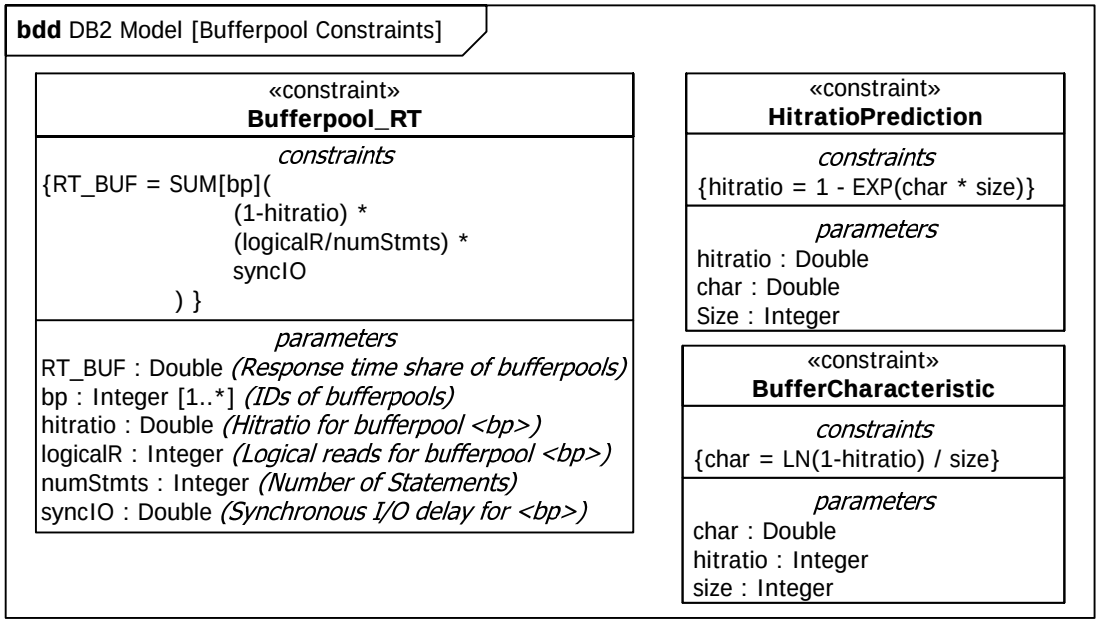


Figure 5.17: DB2 Bufferpool Response Time Model Constraint Definition

5.4.2.2 Buffer Management

The patent [BK09], which has been used as the source for the running example in Section 5.1, is owned by IBM. Hence, this existing quantitative model for estimating the response time share of the bufferpool is also used for the DB2 model. It approximates the response time as the sum of synchronous read times caused by cache misses in the bufferpool. As all other components of the DBMS are ignored, it perfectly suits the requirements of a bufferpool-specific overhead model. Figures 5.17 and 5.18 illustrate the adaptation of the corresponding constraint definitions for the DB2-specific sensors and effectors. One major difference is that DB2 offers a monitor element that directly allows the monitoring of the bufferpool hitratio. In addition, the average number of page requests in DB2 cannot be monitored directly, but has to be calculated from the number of statements and the total number of page requests.

5.4.2.3 Sorting

The quantitative model for the prediction of the sorting delay based on the DBS's workload and state is a simplified version of the model described in [DZ02]. It assumes that the sorting overhead decreases exponentially with the size of the available memory. As a simplification it therefore ignores the range of memory where sort operations can be executed with a one-pass technique. Furthermore, the sort time is assumed to be 0 when the sort can be executed entirely in memory. This approach is taken because DB2 reports only the overall sort time. Thus, the overhead from sorting in memory cannot be distinguished from the time spent for writing/reading intermediate results to/from disk and merging operations.

Due to the monitoring limitations in DB2, the entire sorting time that can be observed via the DB2 sensor is considered as overhead, which could be avoided if the sort was executed in

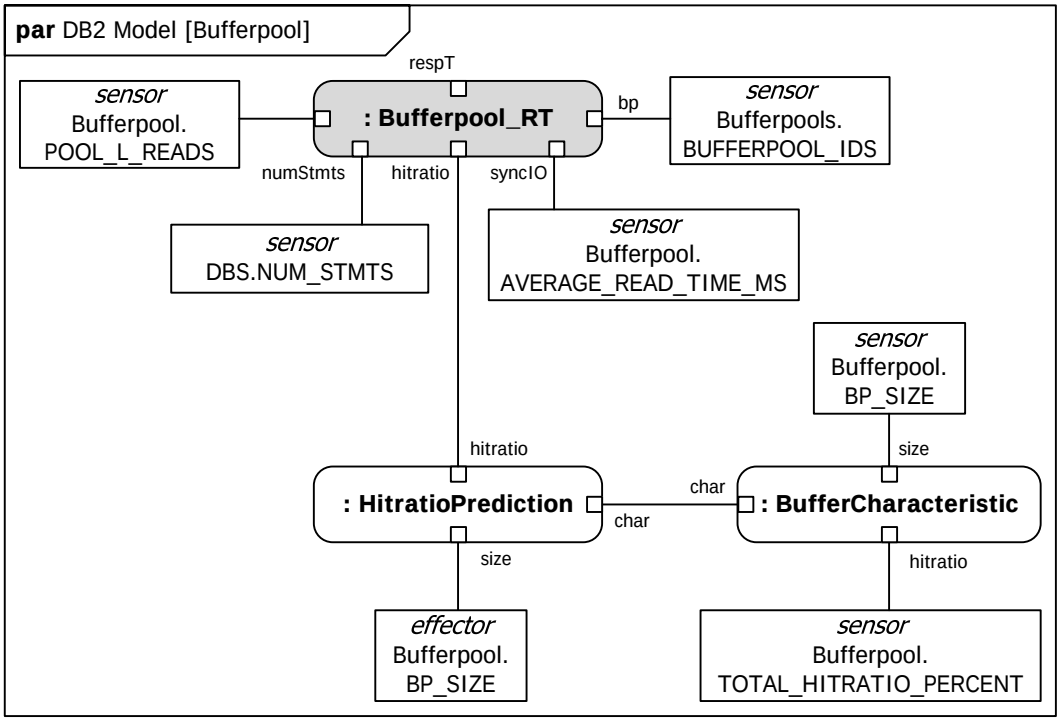


Figure 5.18: DB2 Bufferpool Response Time Model Constraint Parametrization

memory. Thus, the sorting overhead which could be avoided can be calculated as the ratio

$$overhead = \frac{sumSort}{sumExec} \,, \tag{5.5}$$

where $sumSort$ refers to the overall sort time and $sumExec$ to the overall execution time of the statements. Increasing the size of the sort area reduces the sort time by a certain amount z and therefore the execution time as well. So the new overhead can be defined as

$$overhead_{new} = \frac{sumSort - z}{sumExec - z} \,. \tag{5.6}$$

By rewriting this equation the time benefit z can be quantified as

$$z = \frac{overhead_{new} \cdot sumExec - sumSort}{overhead_{new} - 1} \,, \tag{5.7}$$

where $overhead_{new}$ depends on change of the sort area size. In order to estimate the changes of the response time the difference of z and the current sort time can be calculated, i.e.

$$sumSort_{new} = sumSort - z \tag{5.8}$$

$$= sumSort - \frac{overhead_{new} \cdot sumExec - sumSort}{overhead_{new} - 1} \tag{5.9}$$

$$= \frac{overhead_{new}(sumSort - sumExec)}{overhead_{new} - 1} \tag{5.10}$$

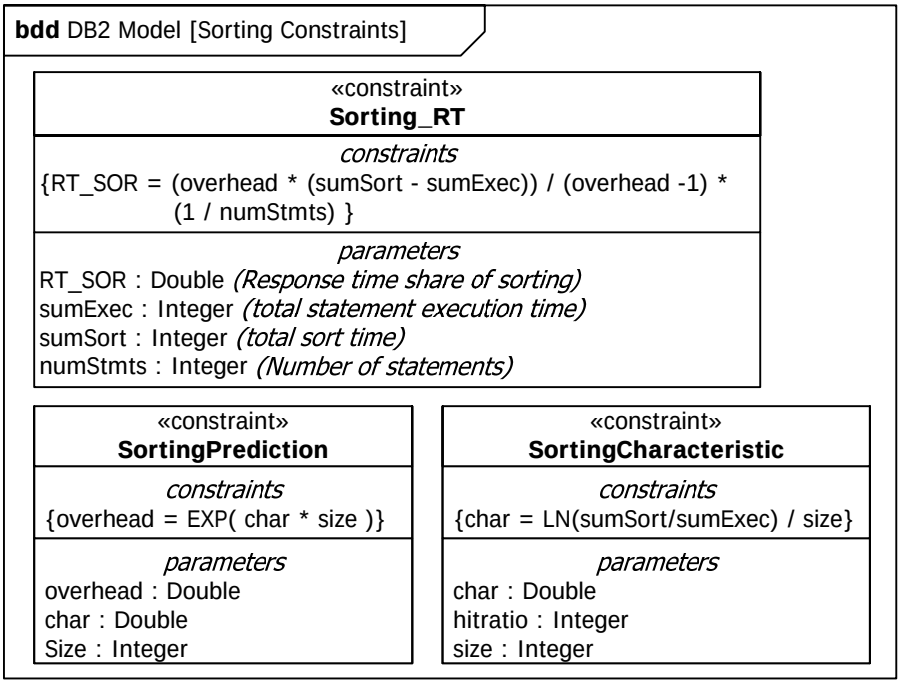


Figure 5.19: DB2 Sorting Response Time Model Constraint Definition

The resulting response time share estimation *Sorting_RT* per statement is shown in Figure 5.19. This figure also shows that for estimating the overhead reduction by increasing the sort area size the same exponential function as for the bufferpool hitratio is used (*SortingPrediction*). Of course, its concrete parametrization depends on the characteristics of the workload. The ConstraintBlock *SortingCharacteristic* therefore defines the calculation of the workload characteristics parameter. This rule has been formed by rewriting the rule from the *SortingPrediction* and replacing the *overhead* with its calculation rule from Equation 5.5. The parametrization of these constraints with the DB2-specific sensors and effectors is illustrated in Figure 5.20.

5.4.2.4 Connection Management

The basic rationale for the connection management model is that the most expensive operation is the creation of an agent. Hence, the agent creation time is assumed to account for the most significant share of response time in the connection management. The model of connection management therefore estimates the probability for the case that an agent has to be created because no agent is available in the pool.

Figure 5.21 defines the quantitative response time model of connection management in SysML ConstraintBlock elements. The probability *poolProb* that an agent is available in the pool is calculated as the ratio of the size of the agent pool (*poolSize*) and the average number of parallel connections (*parallelCons*) in the constraint *PoolAgentProbability*. The number of parallel connections is an important workload characterization of the DBS, which is expected to be low for OLAP environments and high for OLTP workloads, for example. From the *poolProb* information the constraint *ConnPrediction* approximates the average connection establishment

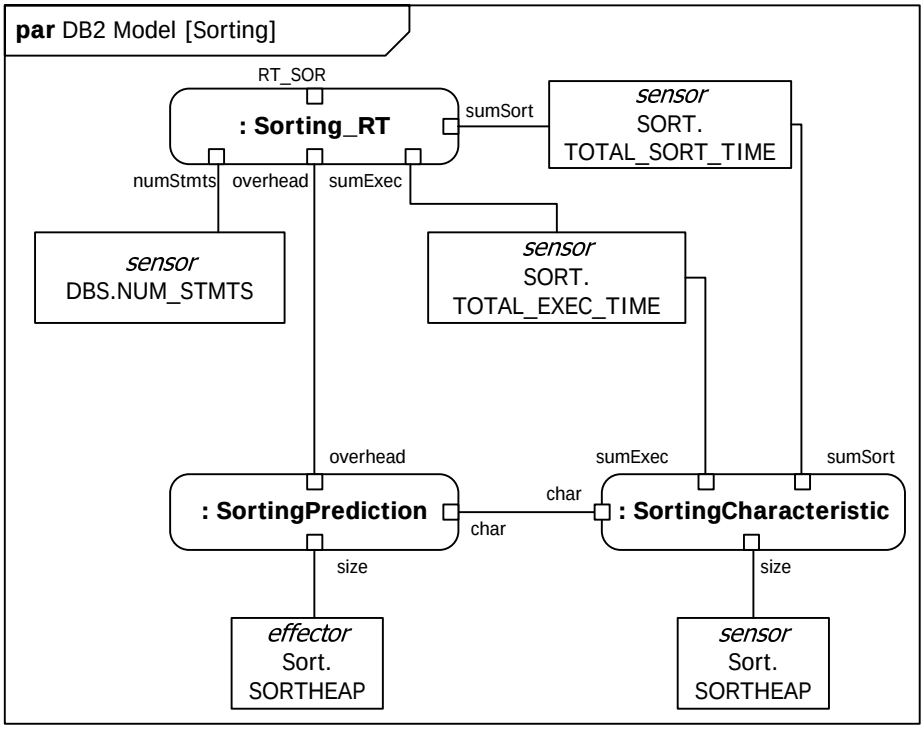


Figure 5.20: DB2 Sorting Response Time Model Constraint Parametrization

time by multiplying the time for creating an agent (*agentCrtTime*) with the probability that no pool agent is available. For the connection only has to be established once, but many statements may be issued via the connection, the connection establishment delay must be averaged over all statements of the connection (see top-level constraint *Connection_RT*). Figure 5.22 defines the mapping of the parameters to the DB2 sensors and effectors.

5.4.2.5 Logging

The quantitative model of the logging component has to predict the time that is required for logging the changes of update, insert and delete statements to disk. The basic assumption of the model is that for every modifying statement the costs appear at least once upon commit, i.e. group commits are not considered in the model. When statements modify a lot of records in the database and the log buffer is too small, then it may be necessary to flush the log buffer more than once, i.e. , additional overhead is caused. In this case the model approximates the logging costs to double (even if three or more log flushes are necessary). In addition, the model makes the simplifying assumption that insert, update, and delete operations are immediately committed. The evaluation results in Section 5.4.3.5 show that the predictions made from the model are reasonable despite these limitations. However, by refining the model in the future it should investigated whether or not more precise predictions can be retrieved for the logging overhead without these simplifications.

As shown in Figure 5.23, the response time share added by the logging component is modelled as average logging time *logTime*, which is doubled with a certain probability *logFullProb*. Of

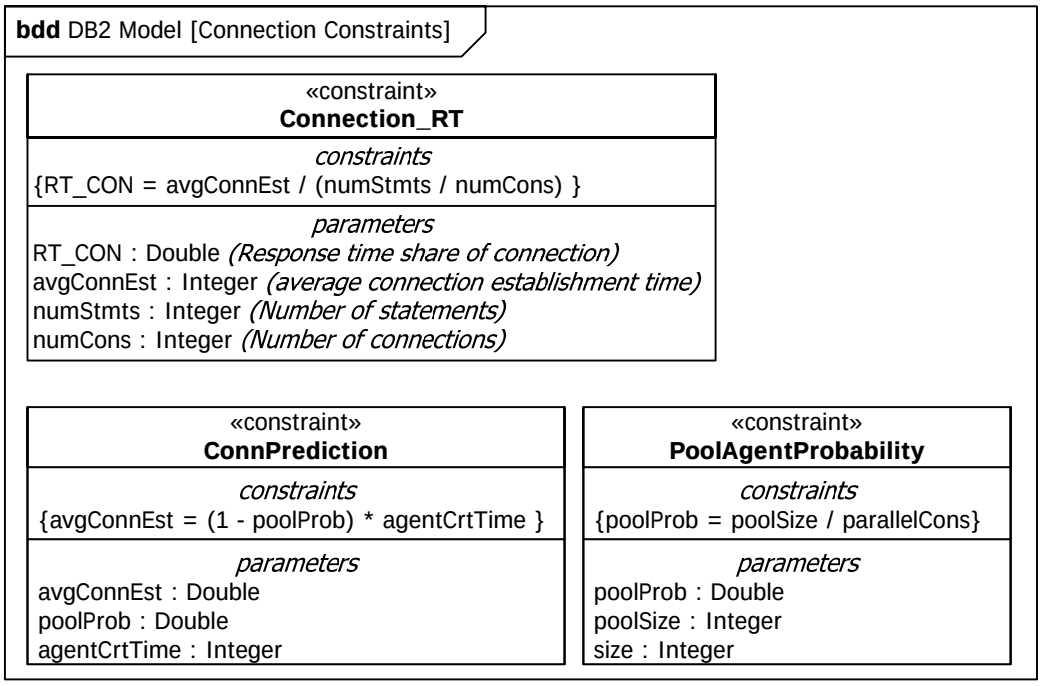


Figure 5.21: DB2 Connection Management Response Time Model Constraint Definition

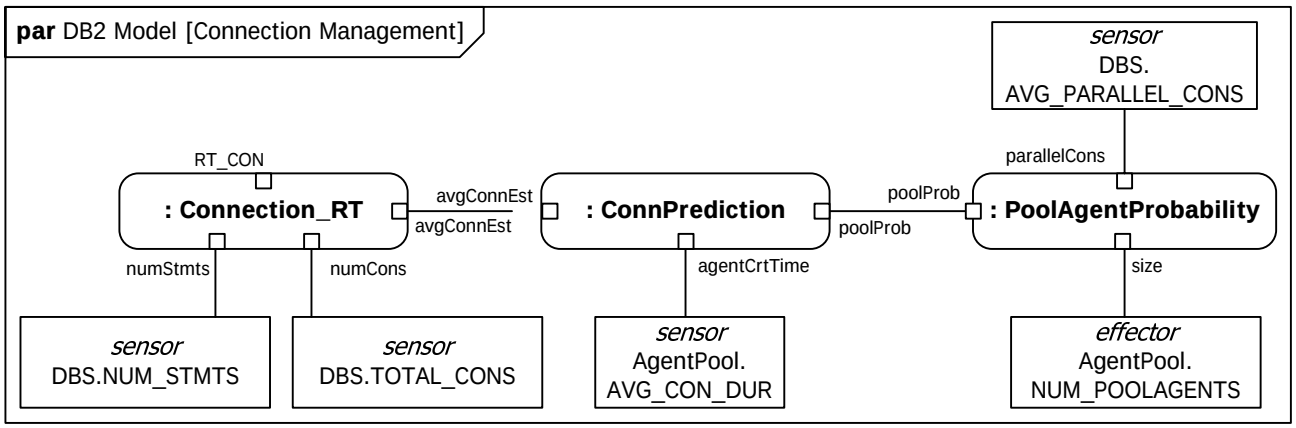


Figure 5.22: DB2 Connection Management Response Time Model Constraint Parametrization

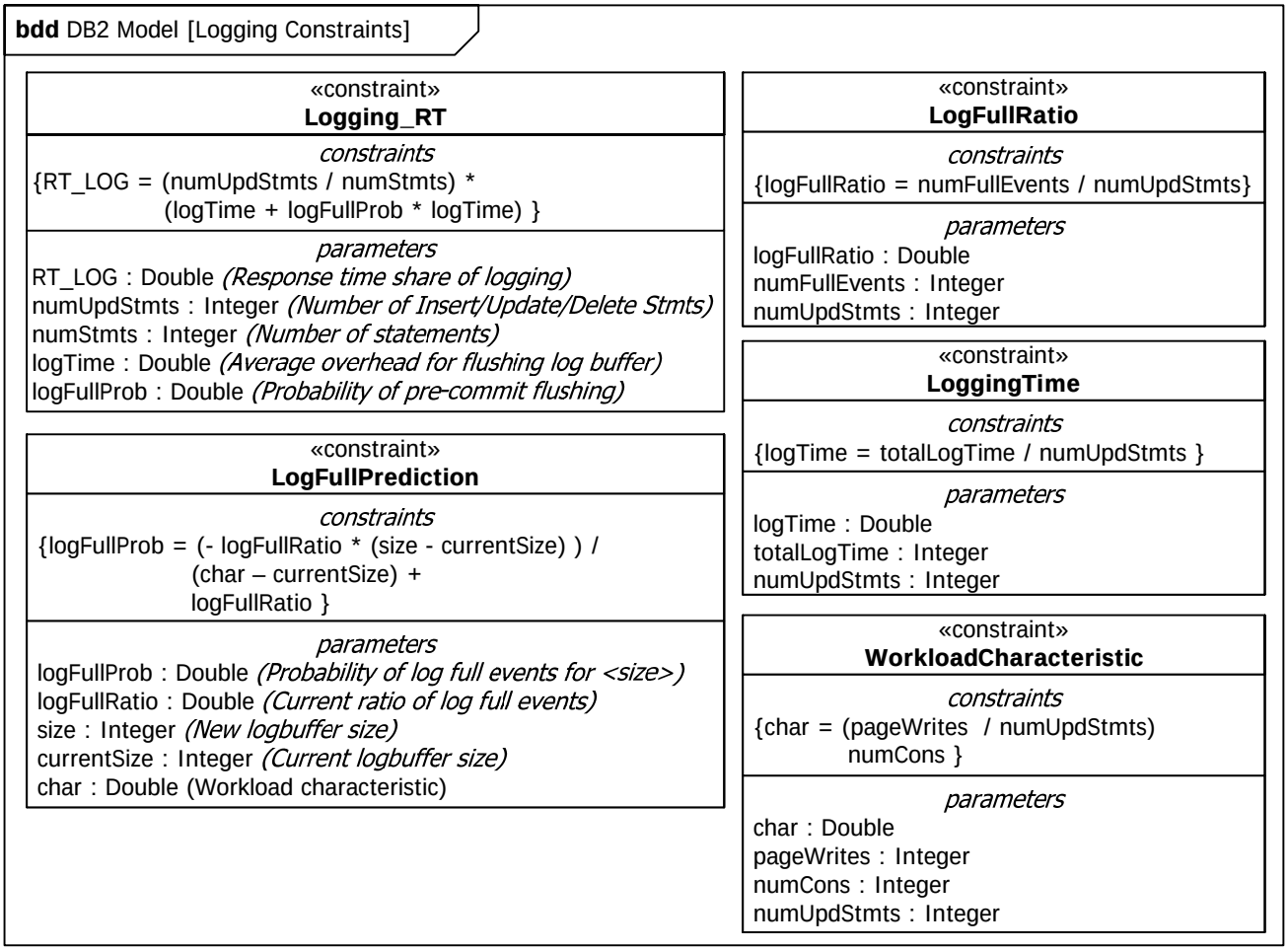


Figure 5.23: DB2 Logging Response Time Model Constraint Definition

course, the log overhead is only added for non-read operations. The average logging time per statement can be easily determined from the total log time and the number of non-read statements (as shown in constraint *LoggingTime*). The probability for log buffer overruns is more difficult to estimate. A linear approach has been chosen for this purpose, which determines the probability as a straight line defined by two points: The first point is the current state, which is defined by the current size of the log buffer *currentSize* and the ratio of log buffer full events (*logFullRatio*, see constraint *LogFullRatio*). The second point is the point where the ratio of log full events can be expected to be 0. This point depends on the workload of the system and is therefore referred to as the workload characteristic *char*. As shown in the *WorkloadCharacteristic* constraint, this characteristic is estimated by calculating the average number of log page writes (*pageWrites*) per non-read statement. To assure that the log buffer is sufficient for multiple applications operating in parallel, this value is multiplied with the average number of connections *numCons*. The parametrization of the constraints for the logging model is shown in Figure 5.24.

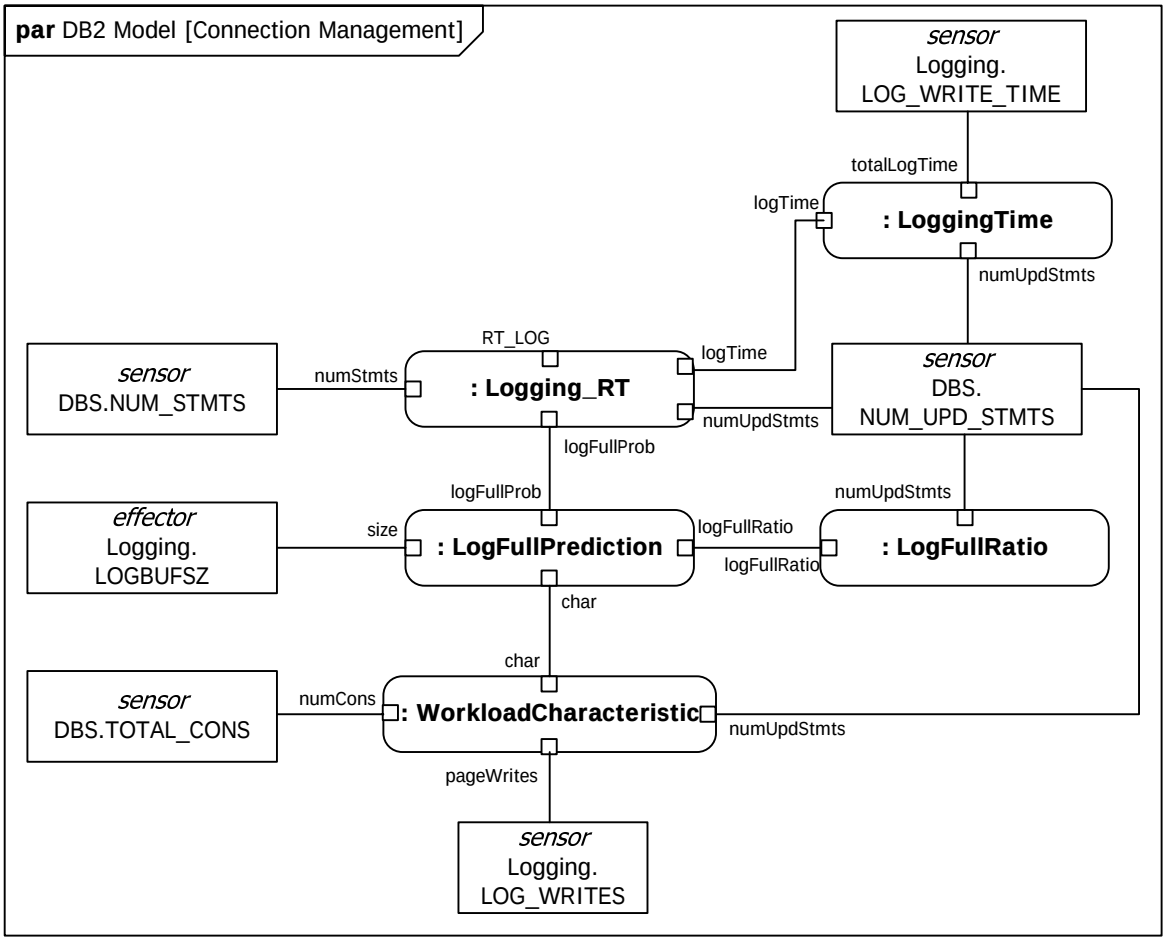


Figure 5.24: DB2 Logging Response Time Model Constraint Parametrization

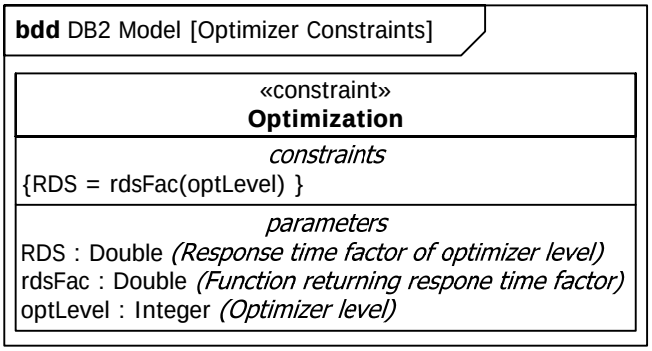


Figure 5.25: DB2 Optimizer Response Time Model Constraint Definition

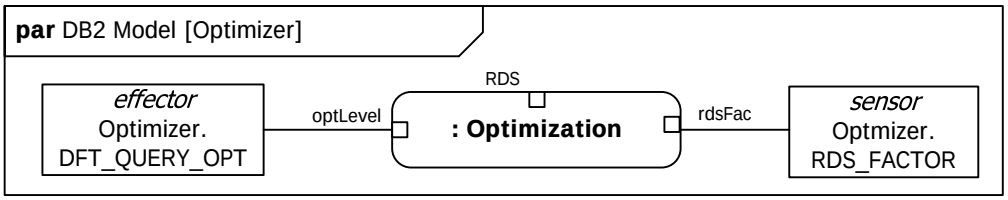


Figure 5.26: DB2 Optimizer Response Time Model Constraint Parametrization

5.4.2.6 Optimizer

In contrast to the other components, the goal of the optimizer model is not the quantification of the response time caused by this component. Instead, a factor that quantifies the effects of a particular optimization level on the performance of the affected components has to be determined. It can be assumed that the higher the optimizer level is chosen, the more efficient plans are generated and the less overhead is caused in the data processing components. The components that are affected by this factor in the DB2 model are the bufferpool and the sorting component. The effects of the optimizer level on the package cache response time implications are instead directly considered in the package cache model (see Section 5.4.2.6).

Due to the complexity of the optimizer decision, finding a theoretical model for the effects of the optimizer level on the response times of the bufferpool and sorting components is a very complex task or even impossible. It is assumed that there is a sensor that "knows" the factors for all optimizer levels. This sensor could either learn the correct factor over time by trying different optimization levels and monitoring the effects. Alternatively, it could be equipped with a set of standard-factors that are derived from experimental evaluations of DB2 under different workloads. This approach is investigated in detail in Section 5.4.3.6.

The constraint for this simple optimizer model is illustrated in Figure 5.25. It shows that the factor that represents the effects of the optimizer level is returned by a custom function labelled *rdsFac*. This function takes the optimizer level *optLevel* as a parameter. The parametric diagram in Figure 5.26 shows that this function is expected to be implemented as a sensor.

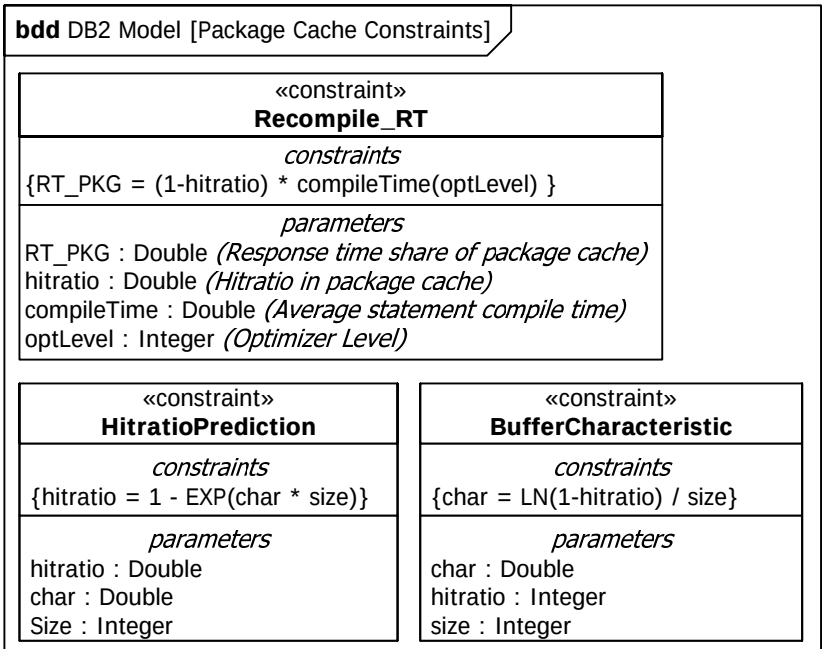


Figure 5.27: DB2 Package Cache Response Time Model Constraint Definition

5.4.2.7 Recompilation

As described in Section 5.4.1, the package cache buffers the execution plans generated by the optimizer for future use. If the package cache is large, there is a high probability that the execution plan for a submitted query can be found in the cache and does not have to be recompiled. Thus, the response time share added by the package cache is the average recompilation effort that is required due to package misses. To predict the number of package misses, again the exponential cache model already used for modelling the bufferpool and sorting behaviour can be employed.

The constraint definitions for the package cache model are given in Figure 5.27. It shows that to compute the response time share in constraint *Recompile_RT* the average compilation time for statements is required. However, this value depends on the current optimizer level, because a higher optimizer level implies a higher compilation time. As for the execution time, defining a theoretical model for the dependency of the compilation time is a very complex task. The package cache model therefore follows the same approach as the optimizer and assumes that there is a sensor (*compileTime*) which returns the required information. An experimental evaluation in order to determine the typical compilation time for different workloads is subject of Section 5.4.3.7. The remaining constraints *HitratioPrediction* and *BufferCharacteristic* are equal to those of the bufferpool and sorting models. Figure 5.28 illustrates the parametrization of the constraints with the sensors and effectors from the DB2 structural model. It is important to note that the *Recompile_RT* constraint depends on the *DFT_QUERY_OPT* effector setting, thus modelling the dependency to the optimizer configuration.

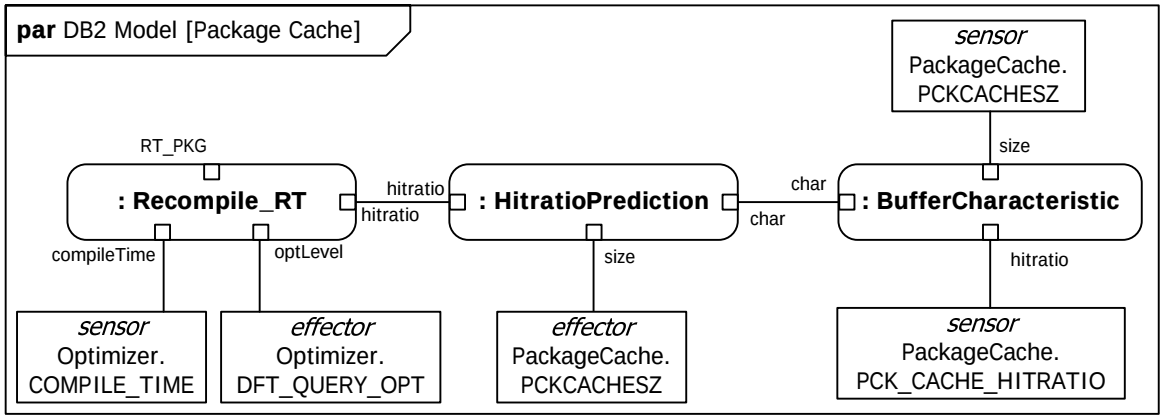


Figure 5.28: DB2 Package Cache Response Time Model Constraint Parametrization

5.4.3 Experimental Evaluation

The previous sections have developed a system model of IBM DB2. The model has been defined based on the information provided in DB2 manuals and research papers, and on general theoretical considerations about the mode of operation of the components. In this section the predictions of the DB2 system model are evaluated with respect to their accuracy. Section 5.4.3.1 introduces the evaluation framework developed for this purpose. Afterwards, Sections 5.4.3.2-5.4.3.7 present the evaluation results for each of the model components. Section 5.4.3.8 finally discusses the accuracy of the overall DBS response time prediction.

5.4.3.1 Evaluation Framework

The basic idea for evaluating the system model is to measure the effects of every single effector on the overall response time of DB2. In every test-run only one effector value is changed and the response time reaction on this change is examined. To ensure comparability the DBS is re-set to a standard configuration after each test-run. The effects of the test-run are compared to the predictions of the individual model components in order to judge their accuracy. To avoid biased results due to a workload dependency, the effects of each effector are evaluated for several workloads with different characteristics.

For the experimental evaluation of the system model the evaluation framework illustrated in Figure 5.29 has been created. The *evaluation control* implements the main evaluation functionality: It first sets the tested IBM DB2 to a standard configuration, which includes the installation of a standard image of the physical database (*DB backups*) and standard values for all parameter values. Afterwards, it changes one of the effectors from the system model to a new value and executes one of the standardized *workloads* on the system. While the workload is executed, the values of the relevant sensors and the response time are monitored and logged in regular intervals by the *results logger*.

To ensure that the evaluation results are valid for different workload scenarios, the evaluation is repeated for four different standardized workloads: TPC-C, TPC-H, TPC-W and DS2. As

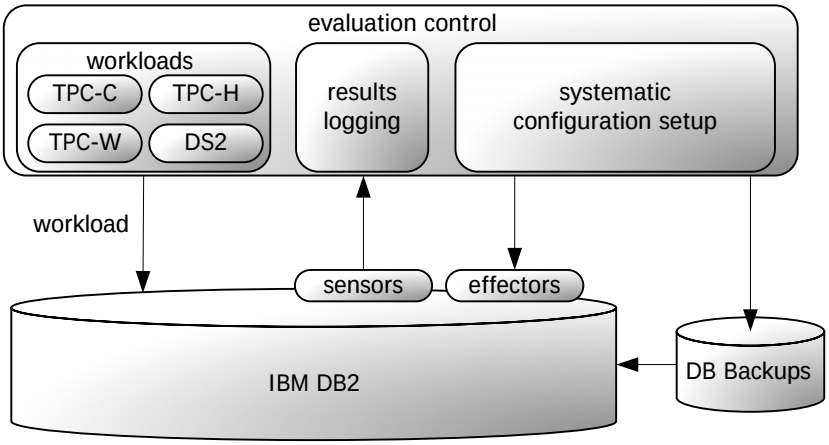


Figure 5.29: System Model Evaluation Framework Overview

described in Section 4.6.3.1, the TPC-C workload [Tra07] is a workload from a standardized DBS benchmark, which simulates an OLTP workload on the DBS. The TPC-H workload [Tra08] in contrast simulates an OLAP load on the DBS with long-running, complex queries. TPC-W [Tra02] is a standard for the evaluation of database-backed web servers, which defines browsing, shopping, and ordering loads on the web server. DS2 is a benchmark by Bell Laboratories [Lab08], which also evaluates the performance of web servers built upon a DBS. For both TPC-W and DS2 only the load caused on the DBS is relevant, whereas the workload on the web server is ignored.

All tests have been performed on a IBM DB2 Version 9.5 on an Windows XP operating system. The used hardware was a PC with a 4-Core CPU and 4 GB of physical memory. The load for the DS2 and TPC-H workloads has been generated using Apache JMeter [Pro10] with appropriate load specification. The TPC-C and TPC-W workloads have instead been generated using existing Java implementations of the benchmark specifications ([jTP09] and [Uni09]).

5.4.3.2 Bufferpool Evaluation Results

The bufferpool model predicts two important values: the hitratio for a given bufferpool size and the response time for the estimated hitratio. Figure 5.30 plots the observed hitratios for different bufferpool sizes in the test-runs. In addition, several predicted hitratios based on the workload characteristic calculated at a specific point in time are given in the plots. For example, the curve *Prediction 3000* represents the bufferpool hitratio estimation derived from the hitratio observed at a bufferpool size of 3000 pages. The plots show that in general the actual hitratio can be predicted quite well – while of course there are errors. The errors are large especially for predictions from small bufferpool sizes, e.g. the *Prediction 100* for the TPC-C workload. Using this estimation, the hitratio is assumed to be almost 100% for a bufferpool size of 1000 pages, although in fact it is around 50% only. These errors can be reduced by refining the model in a way that it considers more than one data point for calculating the workload characteristic.

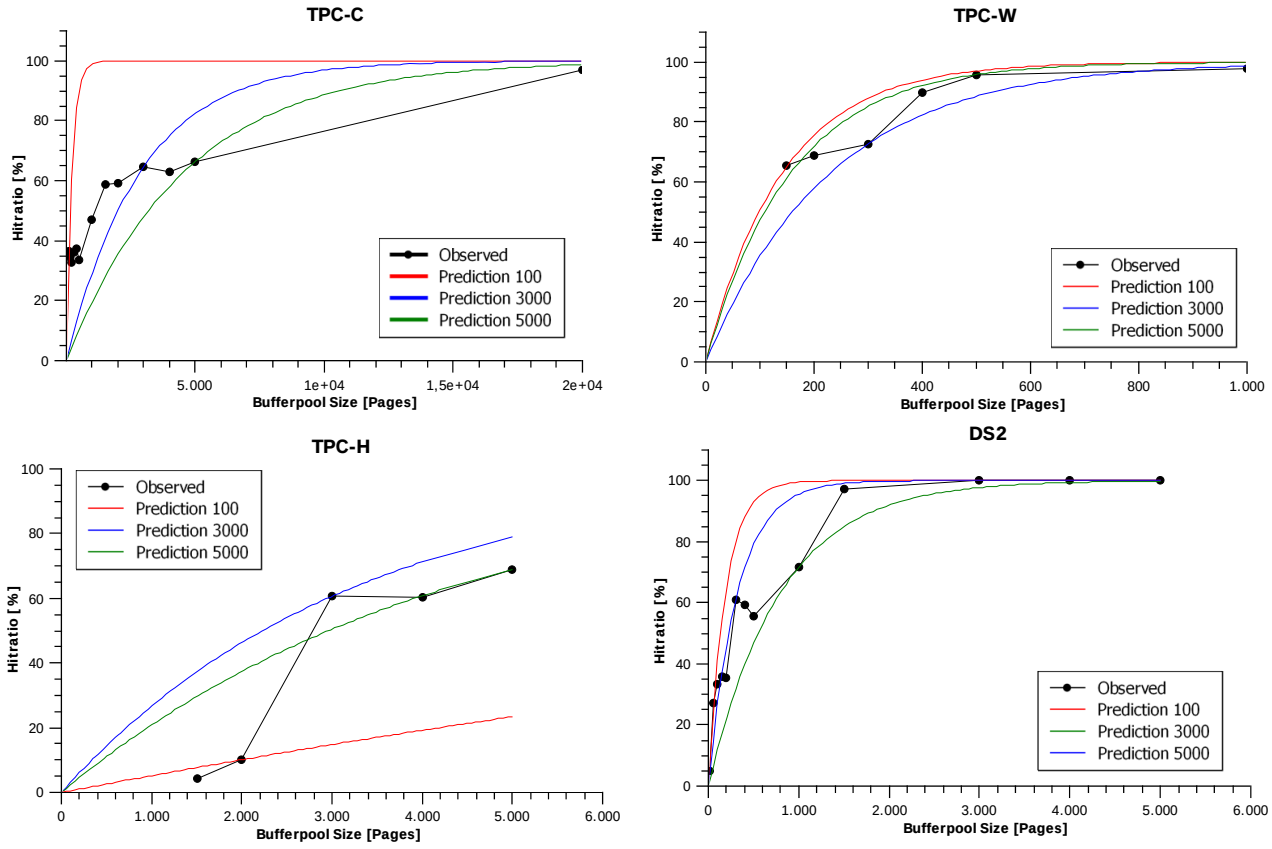


Figure 5.30: Bufferpool System Model Evaluation: Hitratio

The estimated and observed response times for different bufferpool sizes are given in Figure 5.31. It is important to note that in this figure only the response time has been estimated, whereas the actually observed hitratio has been used. From a theoretical point of view the model for the prediction is reasonable, because it estimates the response time from the product of the page misses and the average I/O time. However, as can be seen from the figure, the deviations of the actual response time from the estimated response time share can be large. The factor by which the actual response time is missed in the TPC-C, TPC-W, and DS2 scenarios is up to 2-5, and for the TPC-H scenario the factor is even around 13. However, it can also be seen that the general trend is well reflected by the predicted values.

5.4.3.3 Sorting Evaluation Results

The prediction of the response time share is performed with two models: In a first stage, the sorting overhead caused by insufficient sorting memory is estimated using an exponential function. From this overhead, the average sorting response time is approximated from the currently observed total execution and sorting times. For the evaluation of these models not the entire workloads defined in the evaluation framework have been used. Instead, only two queries requiring a significant amount of sorting have been selected (TPC-H query 13 and a reporting query on the DS2 database). Due to the large amount of data that has to be sorted

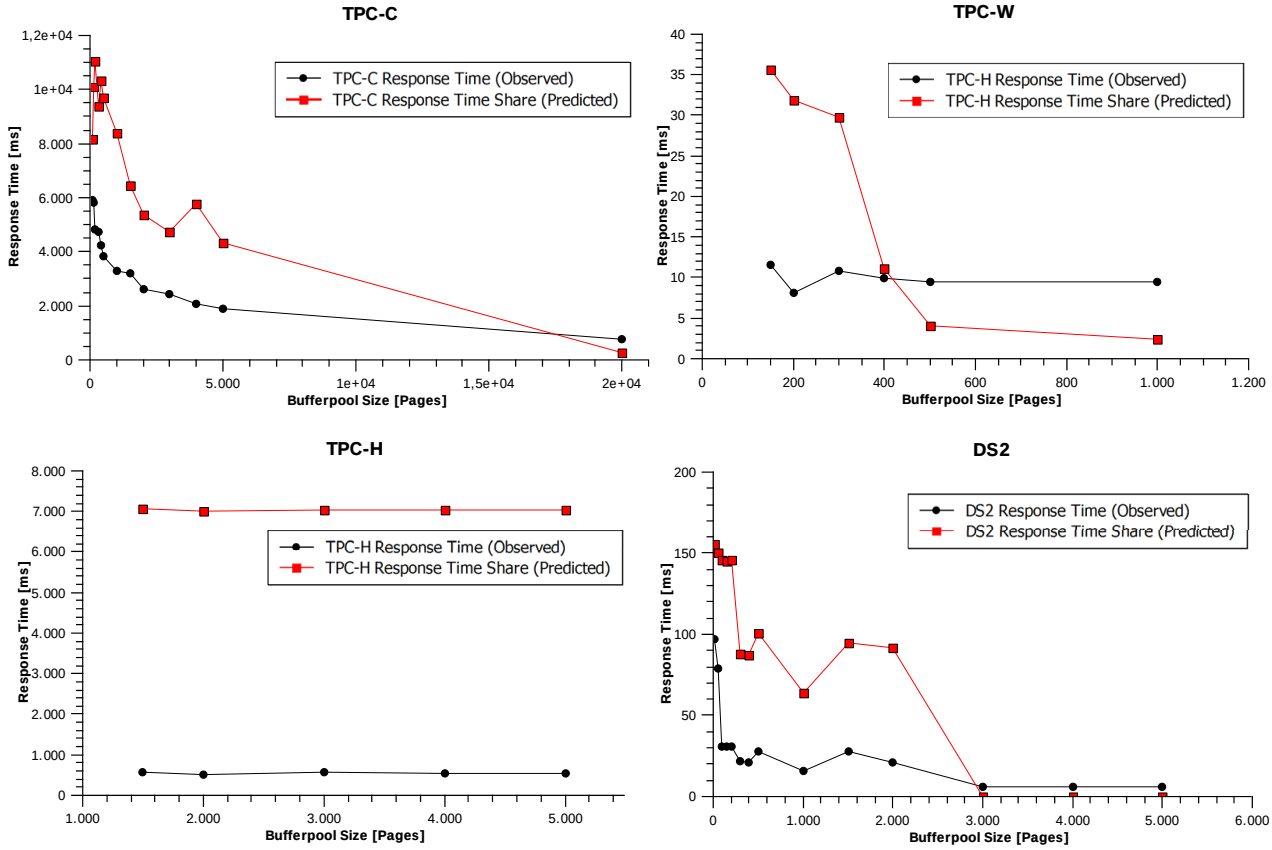


Figure 5.31: Bufferpool System Model Evaluation: Response Time

in these queries, the queries cause sort area overruns and thus flushing of intermediate results to disk.

The observed sorting overhead ratios for these two queries for different sort-heap sizes are illustrated in Figure 5.32. In addition, three exemplary predictions for the overhead are given. The *Prediction 1000*, for example, refers to the overheads predicted from the observations at a sort-heap size of 1000 pages. From the plots one issue of the overhead prediction model becomes obvious immediately: The model does not reflect the base sorting time that is required even when the sorts can be executed in memory completely. However, missing sensor information in the DB2 about the duration of internal sorting does not allow the identification of this base line. In addition, [DZ02] describes that the sorting area must have a certain minimum size before performance improvements can be observed. This knowledge is also not yet considered in the model.

The observed sorting times for the two queries are plotted in Figure 5.33. This figure also plots two sort time predictions for each query type. The predictions represent the sorting response times that would be estimated for the sorting characteristics (sort time, total execution time, overhead prediction) that have been observed at the indicated sort-heap sizes (1000/160000 and 100/500 pages). It can be seen that the predictions that are based on characteristics derived in the area where the sort-heap increases actually lead to an improvement of the sorting overhead are more precise (1000 pages for the TPC-H query and 100 pages for the DS2 query). This

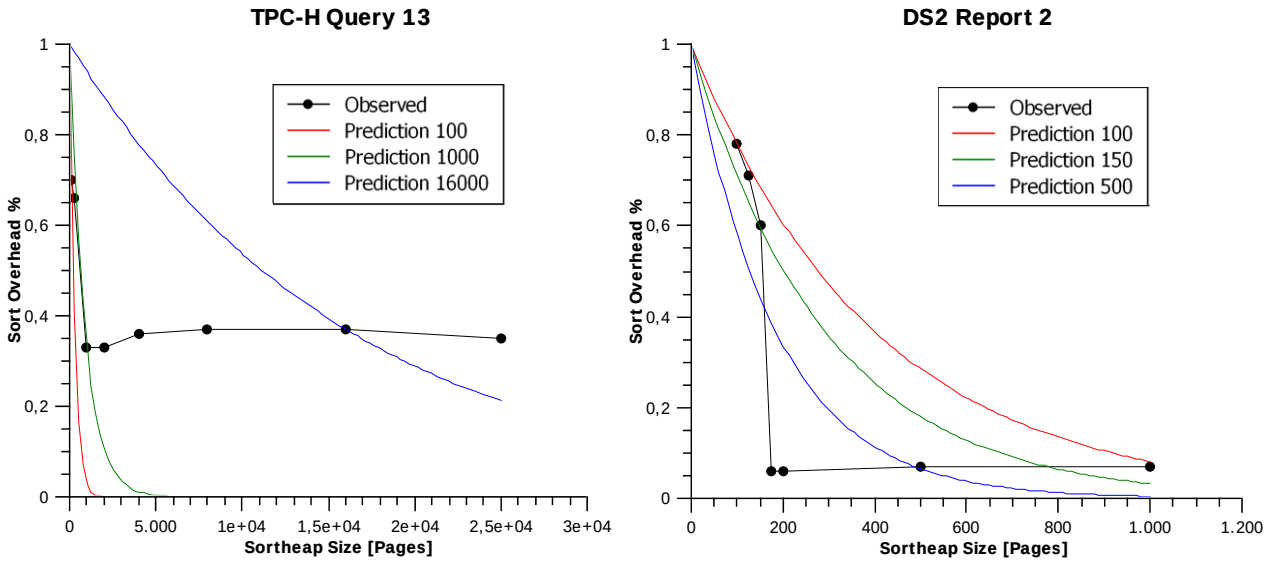


Figure 5.32: Sorting System Model Evaluation: Overhead

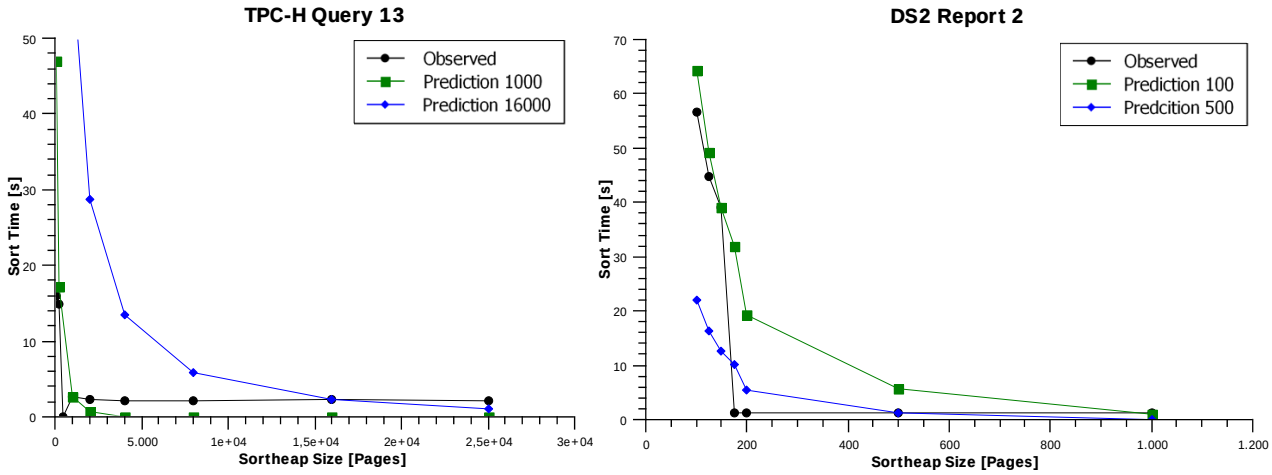


Figure 5.33: Sorting System Model Evaluation: Response Time

is caused by the fact that the base-line of sorting time is not considered in the model for the overhead prediction.

5.4.3.4 Connection Management Evaluation Results

The goal of the system model for the connection management component is the prediction of the effects of the agent pool size on the response time. According to the DB2 documentation [Int09], the creation of an agent causes a significant overhead, which should be avoided by maintaining an appropriately sized agent pool (especially for OLTP environments). For investigating the effect of the agent pool size on the observable response times, the evaluation framework described in Section 5.4.3.1 has been employed in a first step. However, for none of the workloads any effect of the agent pool size on the response times could be observed.

It has been assumed that the natural fluctuations in the processing times of the TPC-C, TPC-

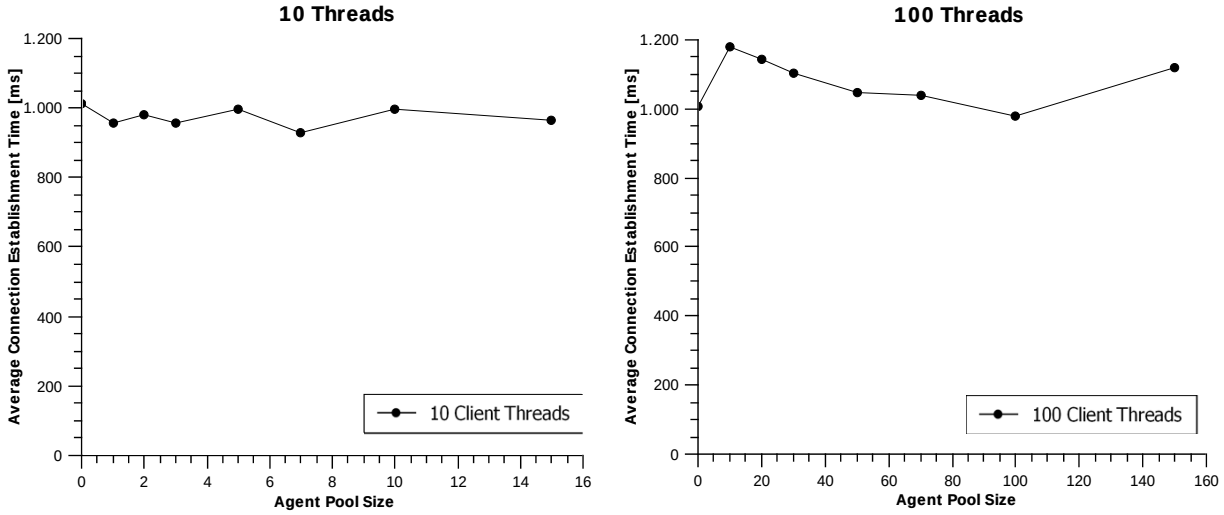


Figure 5.34: Connection Establishment Evaluation

H, TPC-W and DS2 workloads obscure the effects of the agent pool size on the response time. Hence, a second specialized test has been designed for evaluating the connection management: A Java application has been created which instantiates a configurable number of threads. Each thread then creates a separate connection to the DB2 and monitors the time for establishing the connection. The results of this test are given in Figure 5.34. It plots the average connection establishment times against the agent pool size for 10 and 100 parallel connections. As can be seen from the figures, a clear trend for lower connection establishment times with larger pool sizes can not be identified at all. Hence, the predictions of the connection management model developed in Section 5.4.2.4 do not have to be evaluated. The connection management component is instead excluded from the DB2 response time system model, because no effects on the response time could be observed in the experimental evaluation.

5.4.3.5 Logging Evaluation Results

The system model for predicting the logging overhead is based on predicting the probability of situations where the log buffer is too small to hold all pages modified within a transaction depending on the logbuffer size (see Section 5.4.2.5). From this probability it estimates the average logging time per statement, i.e. the response time share added by the logging component. For this purpose it multiplies the logbuffer overrun probability with the average log write time. In order to validate the precision of this model, only the TPC-C and DS2 workloads have been used. The TPC-H contains only queries and therefore does not cause any logging overhead, whereas the modifications from the TPC-W workload affect a very small number of records only, so that no logbuffer overruns are caused even for the smallest possible logbuffer size configuration.

For each of the TPC-C and DS2 workloads, two sample predictions of the logbuffer overrun probability are shown in Figure 5.35. For the TPC-C workload, the predictions made from

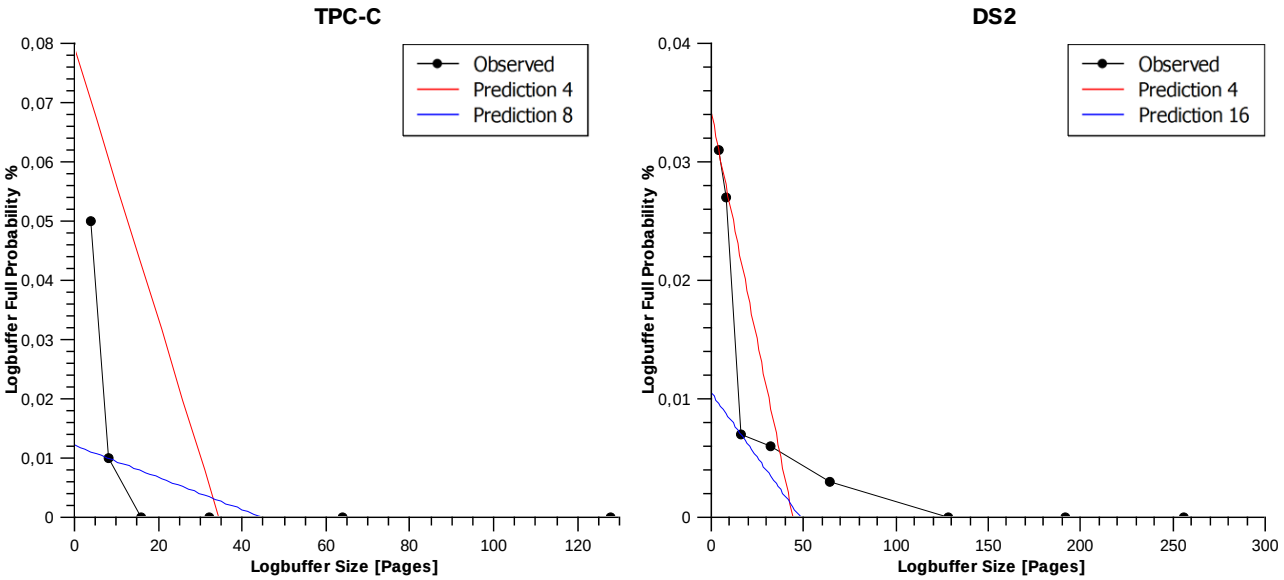


Figure 5.35: Logging System Model Evaluation: Logbuffer Full Probability

the DBS state that could be observed as a logbuffer size of 4 and 8 pages are shown. The intersections of the linear prediction with the x-axis represent the points where no overruns are expected for the current workload of the system. These logbuffer sizes are calculated as the workload characteristic according to the rules given in Section 5.4.2.5. However, as can be seen from the figures, the actually observed overrun probabilities do not follow the linear dependencies suggested by the model. Still, the approximations of the model are close to the actual probabilities and do not fail by orders of magnitude. The amount of memory that would be allocated unnecessarily in the TPC-C scenario in order to completely avoid logbuffer overruns would be less than 30 pages, i.e. typically less than 120KB.

The observed logging times for the TPC-C and DS2 workloads are plotted in Figure 5.35. This figure also plots two logging time (i.e. response time share) predictions for each of these workloads. The predictions represent the logging response times that would be estimated for the logging characteristics observed at the indicated logbuffer sizes (4/8 and 4/16 pages). The TPC-C figure shows that the estimated logging time from *Prediction 8* is much higher than the estimations derived from the characteristics derived from the observations that have been made at a logbuffer size of 4 pages. These higher values are caused by a higher average log time (42.7ms instead of 28.8ms) that have been observed for this logbuffer size. In addition, it can be seen from the figure that the logging time decrease estimated from the model is too small. This underestimation shows that the simplifying assumption made in Section 5.4.2.5 – that the logging costs are at most doubled when there is a logbuffer overrun – is too restrictive. Except for the outlier at a logbuffer size of 400 the same observations hold true in the DS2 scenario. However, the absolute values of the average logging overhead is much smaller in this case, because the ratio of modifying statements is much smaller ($\approx 3\%$ modifications for DS2 compared to $\approx 50\%$ for TPC-C).

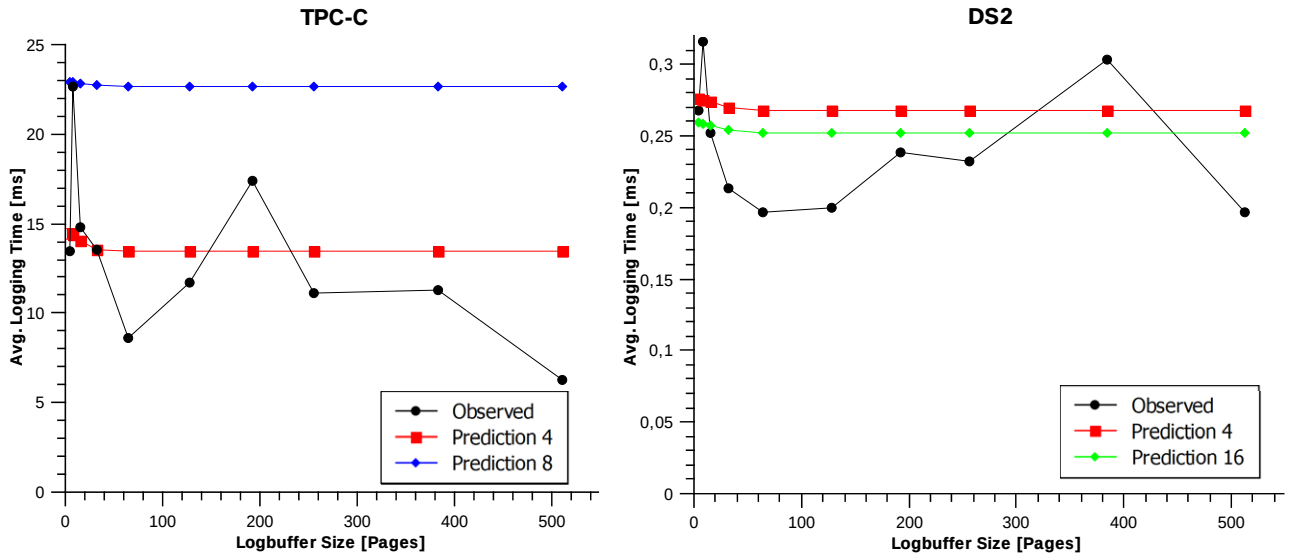


Figure 5.36: Logging System Model Evaluation: Response Time

5.4.3.6 Optimizer Evaluation Results

The approach that has been taken for modelling the effects of the optimizer differs significantly from the other components: Instead of creating a model for the added response time share, a factor that affects the response times estimated at the lower query processing layers has been modelled. As finding a theoretical model for this factor is prevented by the complexity of the optimizer decisions, an experimental approach has been chosen in Section 5.4.2.6. Hence, the effects of the possible optimization classes (0,1,2,3,5,7,9) on the response time had to be determined for each of the workloads in the evaluation framework. From these observations a response time factor has been calculated. For 5 is the default optimization level in DB2, this optimization class is defined as factor 1. In theory, smaller optimization classes should yield larger response time factors and vice versa.

The observed response times for the evaluation workloads are plotted in Figure 5.37. The figure shows that the observed response times do not support the theoretic assumptions in general. For TPC-H there is at least a significant decrease in the response times starting from optimizer level 3. All other workloads do not follow this pattern. The TPC-C workload even exposes a significant increase in the response time at optimizer level 9. As the effects of the optimization class largely depend on the workload, it is obvious that finding a fixed set of factors that is valid for any workload is not possible. Instead, the factor has to be determined for every particular DBS workload separately. The resulting optimization factors for each of the evaluation workloads are given in Table 5.2 (the TPC-H workload at optimizer level 0 caused too large response times and therefore did not complete within a reasonable time period).

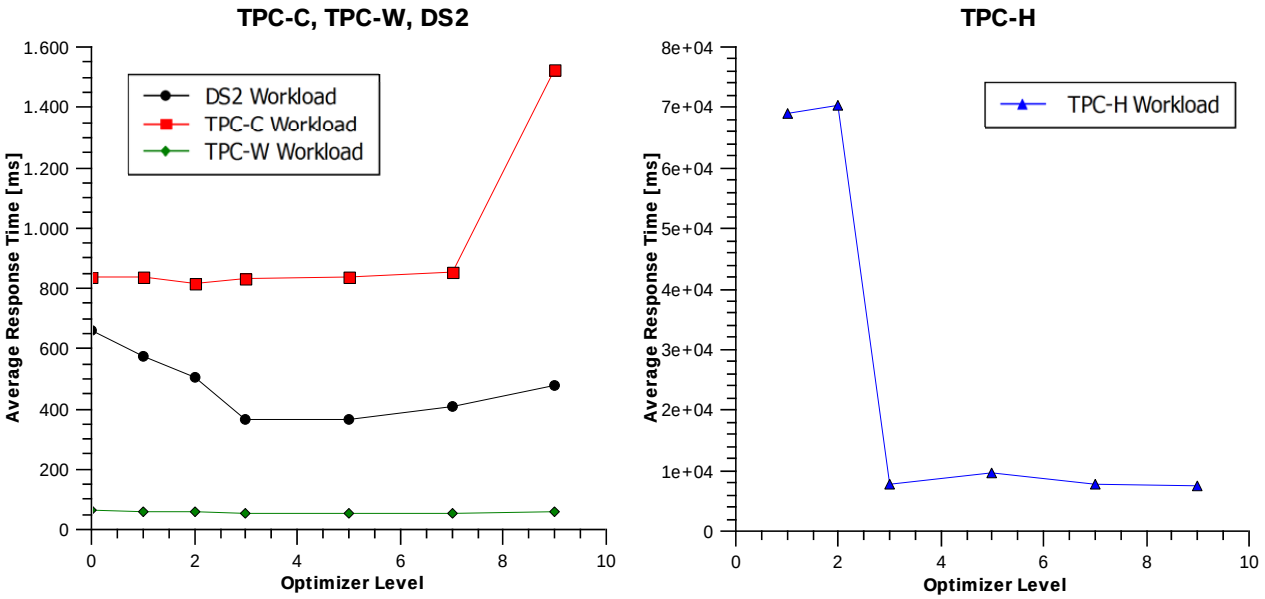


Figure 5.37: Optimizer System Model Evaluation: Response Time

Table 5.2: Response Time Factors for Optimizer Levels

Optimizer Level	TPC-C	TPC-W	DS2	TPC-H
0	1.00	1.17	1.82	-
1	1.00	1.13	1.57	7.17
2	0.97	1.08	1.38	7.31
3	0.99	0.98	1.01	0.80
5	1.00	1.00	1.00	1.00
7	1.02	1.06	1.12	0.82
9	1.82	1.08	1.32	0.78

5.4.3.7 Recompilation Evaluation Results

The model for the recompilation overhead discussed in Section 5.4.2.7 comprises two important aspects: On the one hand it predicts the number of package cache misses depending on the package cache size, and on the other hand it estimates the recompilation effort depending on the optimizer level. As for the optimizer level's influence on the data processing at the lower layers, its influence on the response time is also modelled as a factor which has to be determined in an experimental way. Hence, the effects of the optimizer level on the average compilation times have been determined for each of the workloads defined in the evaluation framework. Theoretically, the compilation times should increase with the optimizer level. However, as shown in Figure 5.38, this theoretical assumption could not be observed for the sample workloads. Consequently, the resulting compilation time factors listed in Table 5.3 (the default optimizer level 5 is defined as factor 1 again) also do not match the expected behaviour. As for the optimizer's response time factor, these results show that the compilation time factors have to be determined experimentally for every environment and workload.

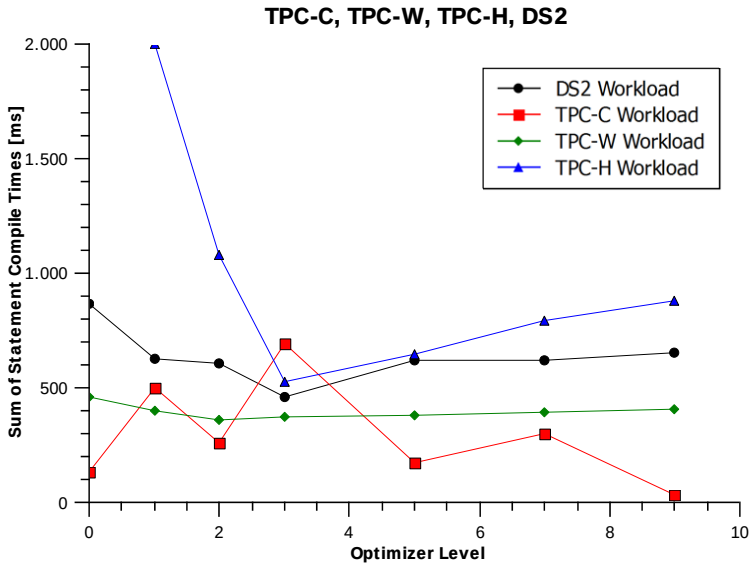


Figure 5.38: Recompilation System Model Evaluation: Compile Time

Table 5.3: Compilation Time Factors for Optimizer Levels

Optimizer Level	TPC-C	TPC-W	DS2	TPC-H
0	0.78	1.20	1.36	-
1	2.94	1.04	1.01	3.09
2	1.51	0.95	0.98	1.67
3	4.05	0.97	0.74	0.81
5	1.00	1.00	1.00	1.00
7	1.76	1.03	1.00	1.23
9	2.05	1.06	1.05	1.36

In order to evaluate the model for the prediction of the hitratio in the package cache, the DS2, TPC-W and TPC-C implementations have been used. The TPC-H workload has been omitted because in its implementation in the evaluation framework every query is executed only once in every testrun. Hence, the probability of finding the execution plans for the queries in the package cache is necessarily 0. The observed package cache hitratios for the workloads are given in Figure 5.39. For every workload also three predictions from the observations at particular package cache sizes are plotted. The *Prediction 32*, for instance, illustrates the predicted hitratios derived when executing the workloads with a package cache size of 32 pages.

From the plots in Figure 5.39 it can be seen that the predictions for the DS2 and TPC-W workloads are close to the actual observations. However, also two weaknesses of the model can be identified: First, the predictions from larger package cache sizes (*Prediction 2048*) cause larger errors than the predictions from smaller package cache sizes. Second, although the observed hitratios do not exceed a value of 90% even for large package cache sizes, the predictions always assume that is possible to achieve a hitratio of 100%. This second problem becomes especially apparent for the TPC-C workload. For this workload the package cache hitratio that can be observed does not exceed 29%, even if the package cache size is set to the

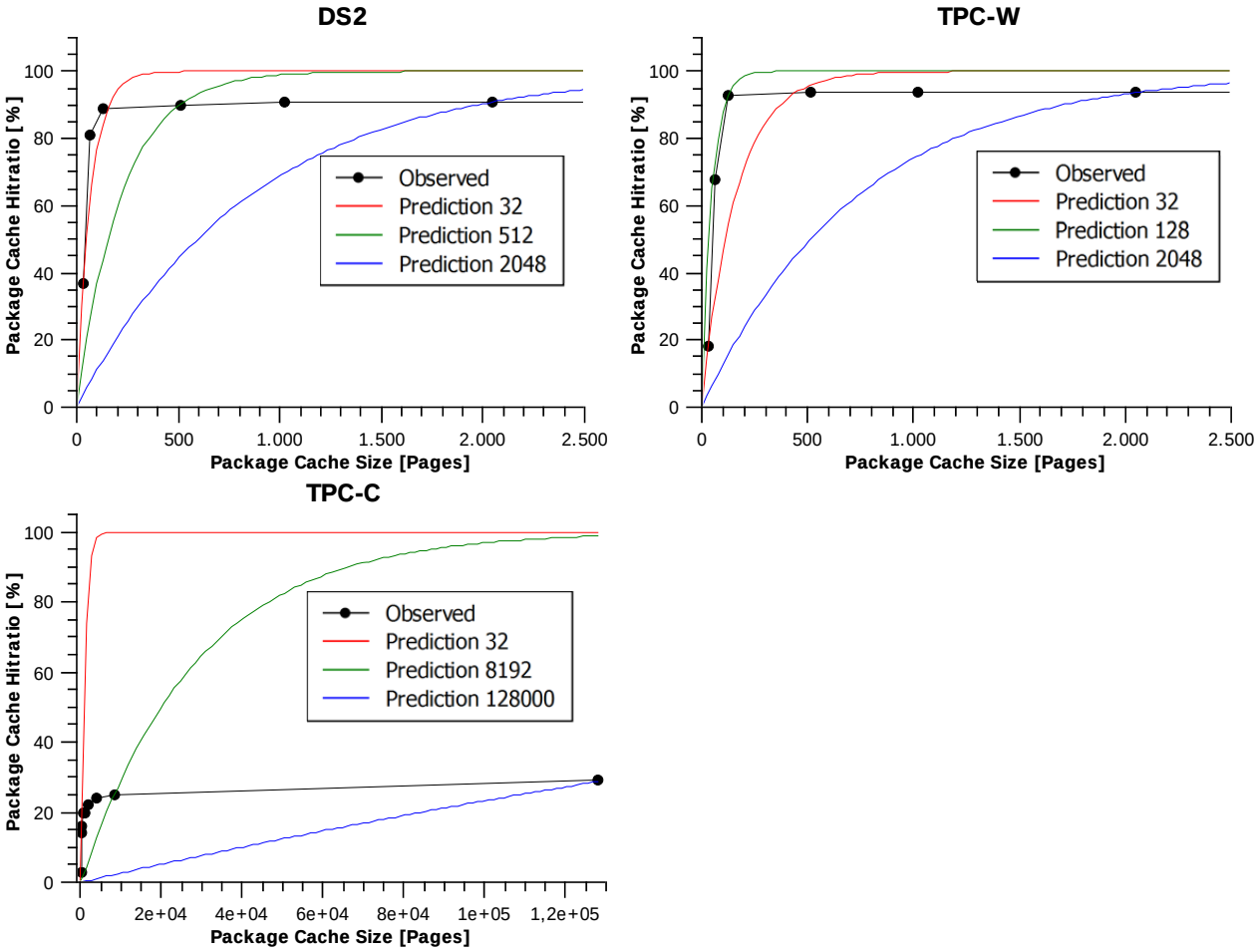


Figure 5.39: Recompilation System Model Evaluation: Package Cache Hitratios

maximum size allowed by DB2 (128000 pages). Hence, the predictions for this workload are imprecise.

In order to estimate the response time share added by the recompilation overhead due to package cache misses, the model developed in Section 5.4.2.7 multiplies the expected ratio of package misses with the average compile time for the selected optimizer level. The ranges of average compile times that have been observed from the DB2 sensors across all optimizer levels have been [1.0ms;1.2ms] for TPC-C, [6.0ms;6.2ms] for DS2, and [10.5ms;16.0ms] for TPC-W. Figure 5.40 in contrast shows the changes in the overall response time that has been observed for different package cache sizes. These response time plots show that the absolute values of the compile times are much too small. For the DS2 workload, for instance, the response time may decrease by more than 300ms, whereas the reported compilation times are around 6ms. Hence, the model from Section 5.4.2.7 for the effects of the package cache size does not fully explain the observed behaviour. The experimental results instead hint that there must be other effects of the package cache size on the response time, which have not been considered satisfactorily.

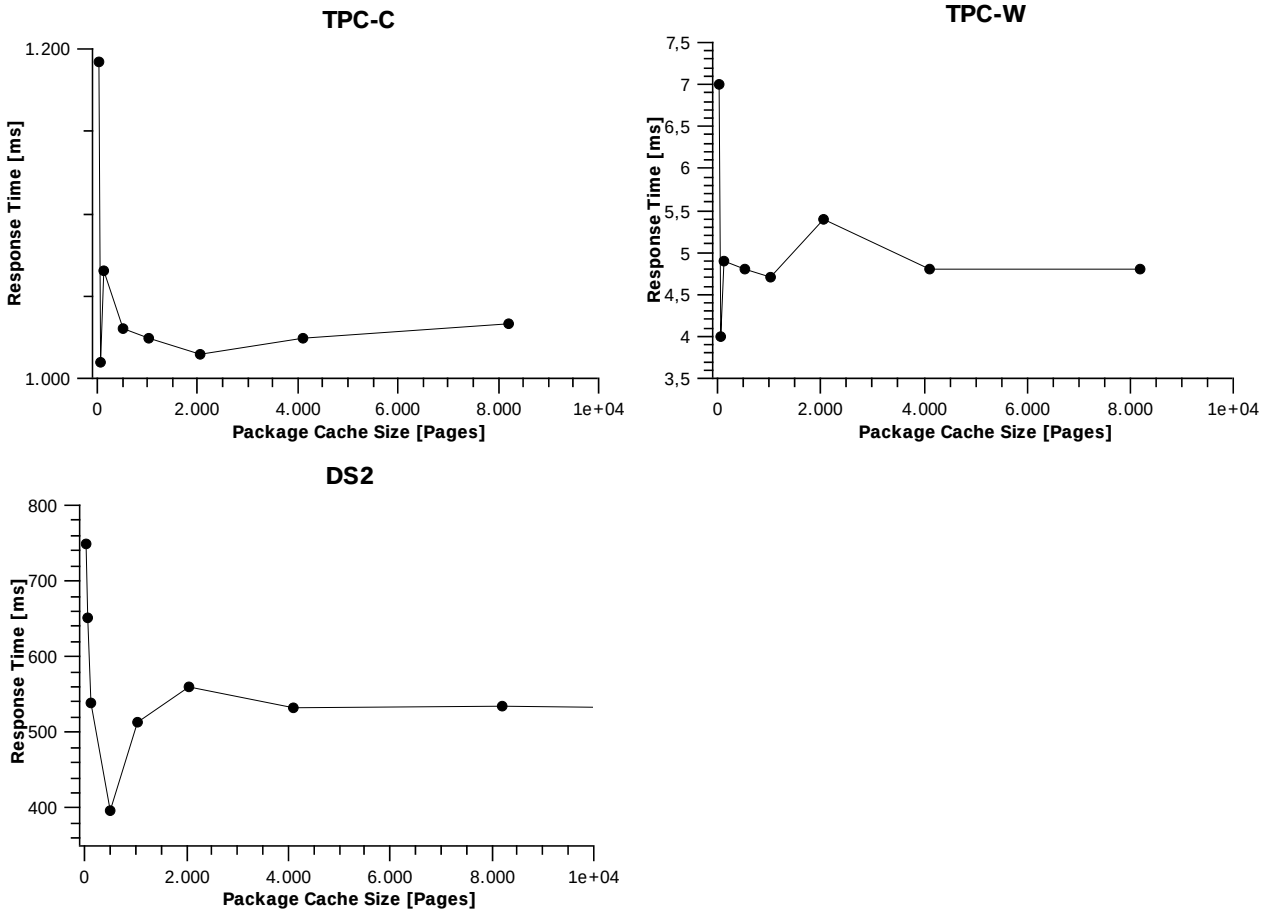


Figure 5.40: Recompilation System Model Evaluation: Response Time

5.4.3.8 Overall Response Time Evaluation Results

The previous sections have evaluated the precision of the behavioural descriptions and response time models for six selected components of the IBM DB2 DBMS. The results have shown that some of the models are already suitable to predict the response time implications of the DBS configurations. In particular, the *bufferpool*, *sorting*, and *logging* models have shown to predict response times that resemble the actual behaviour of the system quite well. However, it has also become obvious that it is not sufficient to evaluate these models based on a single observation of its behaviour. The evaluation should instead consider multiple performance observations from the past in its analysis. In contrast to the bufferpool, sorting, and logging models the *recompilation* model has exhibited some deficiencies and therefore requires refinement in the future. Although for the *connection management* a sound theoretical model has been found, no actual effects of the agent pool size on the response times of the DBS could be observed in the experimental evaluation. Hence, the corresponding DBMS component might be completely omitted from the system model. The decisions of the *optimizer* component are so complex that the definition of a mathematical model has been considered as impossible. Instead, an experimental approach has been selected for this component, whose evaluation has shown that the effects of the optimizer configuration largely depend on the workload.

Table 5.4: Overall Response Time Prediction Results

	TPC-C	TPC-W	DS2	TPC-H
Observed Response Time	3585ms	9.9ms	412ms	814ms
Predicted Response Time	8377ms	3.56ms	279ms	7070ms

Despite the flaws of the presented system model, also an evaluation of the overall response time model has been performed. For this purpose, the overall response time predicted by the system model has been compared to the actual response times for the TPC-C, TPC-W, TPC-H and DS2 workloads. However, as only some of the component models currently provide predictions that are sufficiently precise, it has not been the goal to cover a broad range of DBS configurations in the evaluations. Instead, the evaluation has been intended to test whether or not the magnitude of the predicted response times matches the actual observations. Hence, only a single configuration with the default DB2 parameter settings has been used (bufferpool size: 1000 pages; optimizer level: 5; sortheap size: 250 pages; logbuffer size: 8 pages; package cache size: 1024 pages; agent pool size: 100 agents). The observed and predicted response times for this configuration are given in Table 5.4. The results on the one hand show that at least the order of magnitude of the predictions for the TPC-C, TPC-W and DS2 workloads matches the actually observed response times. On the other hand, the precision of the DB2 model is not yet sufficient to provide guarantees that performance goals will be met. Further evaluations and model refinements will be necessary for this purpose in the future.

5.5 Related Work

Research in the area of self-managing database systems focuses on the automation of individual administration tasks like memory management [SGAL+06] or index selection [BC07]. These functions do not consider relationships to other autonomic managers, side effects or high-level goals. In contrast, the system model described in this section allows the modelling of all (important) DBS components. The interdependencies between them can be modelled intuitively by applying multiple constraints to effectors. Recently, also works on meeting response-time goals for multiple service classes in DBS have been published ([KSA+08], [NMP+06]). These approaches strictly focus on admission control for queries. The described system model solution in contrast aims at implementing the right DBS configuration to meet the goals rather than delaying the execution of queries with a lower importance level.

Models with quantitative descriptions of the managed resource behaviour by now have mainly been used for bufferpool management in DBS. These models are used to predict the bufferpool hit ratio ([THTT08], [CFW+95]) or even the DBS response time ([BK09], [BCL96]) depending on the bufferpool size. These models are helpful preliminary works for creating the behavioural descriptions and response time models of individual DBMS components. They can be used as a first step towards a full DBS model, which serves as a knowledge base for DBS self-management.

With the described approach, it is possible to describe and refine these models in a graphical way. Thus, they can be extended to also cover other configuration and physical design options.

A quantitative model for an entire DBS is proposed in [NTA05]. The model provides predictions for both the throughput and the response time of a DBS depending on the amount of memory and the CPU power that are assigned to the DBS. Both predictions are based on simple models of the CPU and I/O usage. The I/O usage is estimated by a model of the bufferpool, which estimates the page misses under a LFU page replacement strategy. This bufferpool model is combined with a storage model approximating the disk seek and read times for cache misses. The CPU model simply assumes a linear dependency between performance and clock cycle. In contrast to the DB2 model described in this work, the DBS model in [NTA05] is based on custom sensor information, which has been integrated into the Microsoft SQL Server. This additional sensor information logs transaction-specific information, allowing the identification of transaction traces (an example for a transaction trace is given in [NTA06]). The evaluation results of the model show that precise predictions for the response time and throughput under different resource-assignments are possible for a TPC-C workload. However, the model proposed in [NTA05] exclusively focuses on the assignment of resources to the DBS. The adaptation of configuration parameters for meeting the SLAs is not considered.

A different approach towards DBS system models describes [Sch09]. The author plans to identify functional groups within the DBMS, which are intended to perform a local self-optimization. The local results are then planned to be propagated to find a global optimum. However, concrete methods for describing the system models and a technique for the evaluation of the functional groups and their interdependencies have not been presented yet. The solution followed in this work in contrast describes all dependencies within a single system model. Thus, the need to consolidate component-specific results is avoided, because the self-management logic can consider all dependencies directly in its decisions.

The Common Information Model [Dis08] provides standardized management of IT systems, independent from the manufacturer and technology. Among others, CIM defines an abstract model for database systems. However, the CIM database system model only describes general information about the DBS, like the instance name, version, the responsible DBA, and the current values of configuration parameters. The internal structure of the DBMS and a quantitative description of the system behaviour are not part of the model. Although of course the CIM database model could be extended in this way by using the UML mechanisms, CIM currently could only be considered as an alternative modelling solution to using SysML.

The IBM Autonomic Computing Toolkit (ACT) [JLHY04] stores information about the resource managed by an autonomic manager in a resource model. This resource model defines the properties of resources, and stores additional information like check cycles, thresholds, and dependencies. However, reconfigurations cannot be automatically derived from the model. Instead, a decision tree script must be provided, which implements this knowledge. The reason for this limitation is that the resource model does not provide a mathematical model of the behaviour of the resource under different configurations.

Experimental approaches for determining the behaviour of a DBS have recently been published in [DTB09] and [BBD+09]. These approaches employ systematic methods to select DBS configurations that might improve the performance of the DBS configurations. For each of the selected configurations a performance test is executed, and the results of the test are exploited in order to determine the next candidate configuration. The algorithms for selecting the tested configurations are designed to cover a broad range of the DBS configuration space. Due to the overhead caused by the required performance tests, the described approaches assume that there is a spare standby-system, which may be used for the performance tests exclusively. Similarly, [TBA10] and [GKD+09] use an experimental approach to determine the expected system loads for specific queries and query combinations from the DBS workload. However, all of these approaches have in common that they do not employ mathematical models for describing the performance implications of DBS configurations. Hence, the experimental evaluations have to be re-executed in every particular DBS environment, although the general behaviour of the DBMS components does not change (only its parametrization). Instead, the system model approach described in this work is based on the paradigm of using general mathematical models wherever possible. Only when the definition of a mathematical model is not possible (e.g. for the optimizer level effects on the response time), an experimental solution is chosen.

5.6 Conclusions

This section has presented an approach for creating a system model as a knowledge base for DBS self-management. Following the approach by Chaudhuri and Weikum [CW06], the system model predicts the performance of the DBS under all possible configurations and workloads. In particular, it provides solutions to the problem

$$workload \times config \rightarrow performance$$

by defining mathematical models for the DBS behaviour. However, due to the complexity of today's DBMS the definition of a system model that completely and precisely predicts the system performance under all possible configurations will not be attainable. The system modelling solution presented in this section therefore supports the incremental system model development by following a graphical modelling approach. Thus, approximate, coarse-grained models can be created in a first step and then refined afterwards.

The requirements towards a system modelling solution have been identified in Section 5.2. Table 5.5 lists these requirements and shows that all of these requirements are met by the described approach. The *refineability* requirement is met by choosing the graphical SysML modelling language. As SysML is an extension of the well-known UML language, existing models can be easily understood by DBMS experts and refined with little overhead. Also the representation of the *hierarchical structure* of the DBMS is naturally supported by the package concepts and aggregation relationships of the SysML language. Furthermore, SysML

Table 5.5: System Modelling Requirements

Requirement	Status
Refineability	✓
Hierarchical Structure	✓
Touchpoint Specification	✓
Constraints	✓
Goal Functions	✓
Hardware Model	(✓)

modelling tools like TOPCASED (http://www.topcased.org) allow to control the presented modelling details by zooming into or out of model components. As shown in Figure 5.8 on page 153, the *touchpoint specifications* can be created using the SysML model elements, too. Thus, the system model comprises all the information that is required by a self-management logic for accessing the sensors and effectors. In addition, also *constraints* on the effector values can be defined. Using the SysML `ConstraintBlock` element, restrictions on the effectors (like domain restrictions or dependencies on other model elements) can be defined in terms of mathematical expressions. With the parametric diagram SysML provides an intuitive way to connect the parameters of these expressions to the model elements. As shown in Figures 5.12 and 5.13 (page 156), the same mechanism can be used to represent *goal functions* in the model. Although it has not explicitly been investigated in this section, the modelling technique could also be used to create a *hardware model*. Every hardware component would have to be represented as a separate SysML `Block`, where the attribute values describe the characteristics of the component (e.g. average I/O time of disks).

Based on the developed system modelling technique, this section has also presented an approach for creating concrete DBS system models. In particular, a system model for the response time of the IBM DB2 has been developed. The classical scientific method has been employed for this purpose: In a first step, models have been created for the effects of selected DB2 components on the response time by taking into account the descriptions in the manuals, theoretical assumptions, observations, and previous publications. Afterwards, the models have been validated by a comparison of the predictions to the actual behaviour of the DB2 under different configurations and workloads.

The results have shown that the performance of some of the components (bufferpool, sorting, logging) is already approximated quite well, while for others (recompilation, optimizer) additional experimental evaluations and model refinements would be necessary. From these results it could be concluded that the modelling and evaluation for DBMS components on the lower layers of the DBMS (see Section 3.1) is easier than for components on the higher layers. However, with the small number of experimental evaluations performed so far this can only be stated as an assumption. Much more detailed and extensive evaluations would be necessary to prove this assumption in the future.

The evaluation of the models has also shown that it will be important to extend the number

of observations that represent the DBS workload and state information during the analysis. In other words, it will not be sufficient to parametrize the model with the current sensor value only. Instead, a history of sensor values should be considered during the model evaluation in order to increase the quality of the predictions.

Despite the remaining problems, the definition of the DB2 response time model has shown three important aspects: First, the selected modelling approach provides all the required concepts for defining a concrete system model. Second, the approximation of the DBS behaviour using mathematical models is possible. Third, in order to completely predict the overall performance of a DBS, extensive experimental evaluations under different workloads and system environments are required.

In the future the DB2 model has to be refined and extended by additional components that have been excluded for the coarse-grained model developed in this section. Every refinement in the model has to be thoroughly tested for its effect on the accuracy of the response time prediction. Only then it will be possible to judge the required level of detail that has to be represented in the DB2 response time model. The interesting question of whether or not the required level of detail in a system model can be defined in general can only be answered by furthermore extending the DB2 model to other key performance indicators, and by creating models for other DBMS, too.

6 Goal-Driven Reconfiguration Analysis

With the modelling technique described in Chapter 5, it is possible to easily develop and refine DBS system models. Whenever a workload shift is detected or there is a risk of missing user-defined high-level goals, the information in the system model should be evaluated in order to determine a new optimal DBS configuration. This section presents an approach for automatically deriving optimal configurations from the knowledge stored in a DBS system model [HR11]. It discusses how the required information is extracted from the knowledge base, how it is parametrized with the current sensor information, and how appropriate effector settings are automatically derived. The solution is designed so that changes to the system model do not require an adaptation of the self-management logic. Thus, evolutions or refinements of the system model can be made without having to adapt the self-management logic.

Section 6.1 discusses the most important requirements towards an automatic reconfiguration analysis. From these requirements, Section 6.2 identifies Multi-Objective Optimization (MOO) techniques as an adequate approach for deriving the DBS configurations. The construction of the necessary objective functions from the information in the system model is the subject of Section 6.3. Choosing an actual DBS configuration from the solution sets returned by the MOO algorithms is discussed in Section 6.4. Evaluation results for the running example (see Section 5.1) are presented in Section 6.5. Related work is discussed in Section 6.6 before Section 6.7 concludes with a summary.

6.1 Reconfiguration Analysis Requirements

The following paragraphs discuss the most important requirements towards a self-management logic based on a system model. An overview of these requirements is given in Figure 6.1.

Extraction The self-management logic has to extract all the required self-tuning information from the system model. Thus, it has to determine the goal functions, constraints, and effectors and sensors without prior knowledge of the managed DBS. In order to allow the easy integration of new sensors and effectors into the system model, the access information for them has to be derived from the system model, too.

Hierarchical Structure One of the most important characteristics of the system modelling approach described in Section 5 is its support for hierarchically structured models. While this

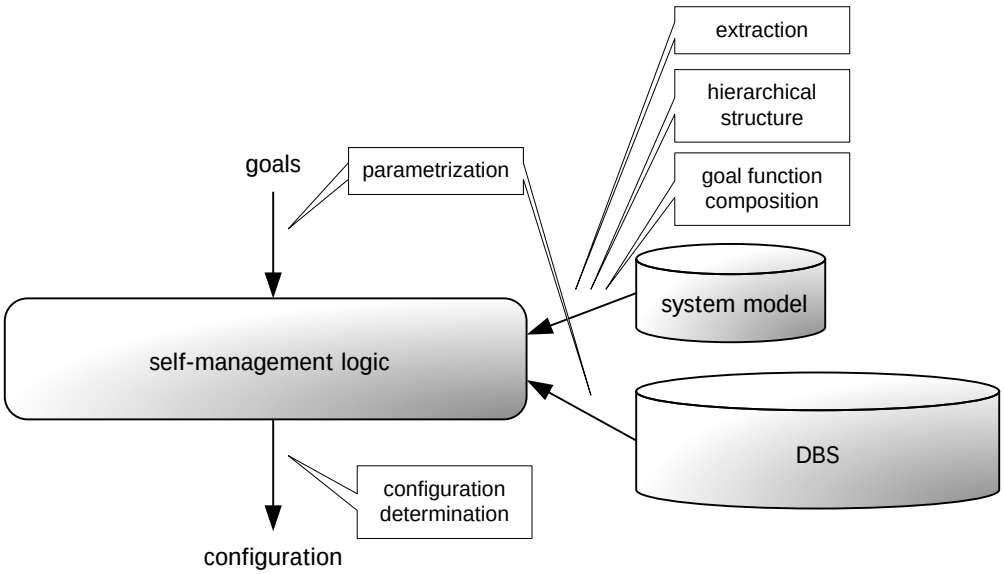


Figure 6.1: Reconfiguration Analysis Requirements Overview

hierarchical structure on the one hand makes it easy for the DBMS expert to create and refine the system model, it on the other hand requires the self-management logic to descend into all levels and compose an overall model from the information.

Goal Functions As described in Section 5.3, goal functions in a system model predict the values of the high-level goals depending on the sensor and effector values. However, in order to allow an easy definition of the goal functions, they may be composed from existing behavioural descriptions in the system model. Hence, the parameters of the goal functions have to be replaced with the corresponding behavioural descriptions.

Parameterization The goal functions refer to sensors and effectors in order to express the workload-dependency and configuration-dependency of the model. In order to derive a concrete configuration (i.e. effector settings) for a DBS at runtime, the sensor parameters have to be replaced with the current values of the sensors. Furthermore, as discussed in Section 3.5, a history of sensor values should be considered during the model evaluation in order to increase the quality of the predictions.

Configuration Determination The main functional requirement towards the reconfiguration analysis in the self-management logic is the determination of a DBS configuration that meets the high-level goals in the best possible way. Considering the system modelling approach followed in this work, this task can be re-phrased as the task of finding settings for all effectors in the system model, given the current values of the sensors.

Service Classes In many real-world DBS usage scenarios, there is not a single goal value for goals like the response time or the throughput, but there are separate values for different

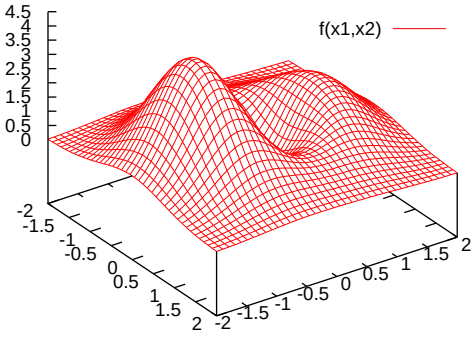


Figure 6.2: Single-Objective Optimization

user groups or applications. The definition of service-class-specific goal functions has been previously described in Section 5.3. The self-management logic has to take into account these service-specific goals during the reconfiguration analysis.

6.2 Design

As described in Section 6.1, the main functional requirement towards a self-management logic is to determine a configuration for the DBS that meets the high-level goals in the best possible way. For this task, the values of the goal functions have to be either minimized (response time, costs, ...) or maximized (throughput, availability, ...). The sensors referenced in the goal functions have to be replaced with the current sensor values, i.e. they become constants in the goal functions that have to be optimized. The referenced effectors in contrast constitute the parameters in the goal functions. Thus, every goal function is a function $f(\mathbf{x})$ where $\mathbf{x}$ is an n-dimensional *decision vector* $x = (x_1, x_2, ..., x_n)$ and every x_i represents an effector that is referenced in the body of the goal function.

Figure 6.2 illustrates the surface of an exemplary goal function ($f(x_1, x_2) = (x_1^2 + 3x_2^2 - x_2) \cdot e^{1-(x_1^2+x_2^2)}$). The task of optimizing a goal function $f(\mathbf{x})$ is the task of finding a global maximum (or minimum, respectively) for the function. As can be seen from Figure 6.2, the function $f(x_1, x_2)$ exhibits one global maximum and a local maximum. For finding a solution to such an optimization problem, several algorithms have been developed (see [CVL07]). A simple heuristic is the *hill climbing* algorithm, for example. This algorithms starts at a random point of the goal function surface and determines the highest function value in the neighbourhood, which is then selected as the next starting point. This is continued until no higher function value is found in the neighbourhood anymore. Thus, the hill climbing algorithm often only finds local optima. Other algorithms like *tabu search* [Glo89] therefore continue searching for other, possibly superior optima even after they have found an optimum. To avoid excessive computation times, the number of steps in tabu search is typically limited.

Unlike the single-objective optimization problem discussed above, the reconfiguration analysis in the self-management logic has to consider multiple independent goal functions. A set of

effector settings that optimizes one of the goal functions will therefore usually not also be the optimal setting for the other goal functions. In particular, the goal functions may even be opposing, i.e., improving the value of one goal function may cause another goal function value to downgrade. For example, adding more memory to a DBS server will on the one hand decrease the response time, while on the other hand it increases the operation costs. Hence, the reconfiguration logic has to consider multiple, possibly opposing goal functions at the same time and find an optimal tradeoff that meets all high-level goals.

As outlined in Section 3.4, the set of goal functions that the reconfiguration logic has to take into account constitutes an *objective vector* $F(\boldsymbol{x}) = (f_1(\boldsymbol{x}), \dots f_k(\boldsymbol{x}))$ of goal functions $f_1(\boldsymbol{x})$ to $f_k(\boldsymbol{x})$. In addition, the reconfiguration analysis has to consider the set of constraints $E(\boldsymbol{x}) = (e_1(\boldsymbol{x}), \dots e_k(\boldsymbol{x}))$ defined in the system model. The set of optimal tradeoff solutions for this problem is referred to as the Pareto optimum P^*. A solution $\boldsymbol{x}$ is a member of the Pareto optimum if $F(\boldsymbol{x})$ is not *dominated* by any other $F(\boldsymbol{x'})$ for a solution $\boldsymbol{x'}$. For a minimization problem, the dominance of an objective vector $F(\boldsymbol{x'})$ over another objective vector $F(\boldsymbol{x})$ is defined as

$$F(\boldsymbol{x'}) \preceq F(\boldsymbol{x}) := \forall i : f_i(\boldsymbol{x}) \leq f_i(\boldsymbol{x'}) \wedge \exists i : f_i(\boldsymbol{x}) < f_i(\boldsymbol{x'}) \ . \tag{6.1}$$

For a maximization problem, the Pareto dominance is defined accordingly as

$$F(\boldsymbol{x'}) \succeq F(\boldsymbol{x}) := \forall i : f_i(\boldsymbol{x}) \geq f_i(\boldsymbol{x'}) \wedge \exists i : f_i(\boldsymbol{x}) > f_i(\boldsymbol{x'}) \ . \tag{6.2}$$

Thus, a solution dominates another solution if all of the goal function values are at least as good as the ones of the competitor, and at least one of its goal function values is strictly better. Formally, the Pareto optimum P^* for a minimization problem therefore can be defined as

$$P^* := \{\boldsymbol{x} | \neg \boldsymbol{x'} : F(\boldsymbol{x'}) \preceq F(\boldsymbol{x})\} \ . \tag{6.3}$$

An example for a Pareto optimum for two goal functions has been given in Figure 3.7 on page 46.

Due to the large search space, a complete search for all possible configurations for the Pareto optimum is usually impossible for multi-objective optimization problems. Instead, meta-heuristics like multi-objective evolutionary algorithms, particle swarms or multi-objective simulated annealing are used. As the largest number of algorithms has been developed for multi-objective evolutionary algorithms, this technique has been selected for deriving the Pareto optimal DBS configurations during the reconfiguration analysis. Introductions to evolutionary algorithms are given in [SD08] and [CVL07], for example. Evolutionary algorithms operate on a set of individuals, where every individual represents a solution to the optimization problem. The chromosome of each individual is encoded from the values of the decision variables represented by the individual. For every individual a fitness function can be used to quantify how well the solution represented by the individual fits the problem requirements. In order to determine the individuals of the Pareto optimum, evolutionary algorithms resemble the process of natural

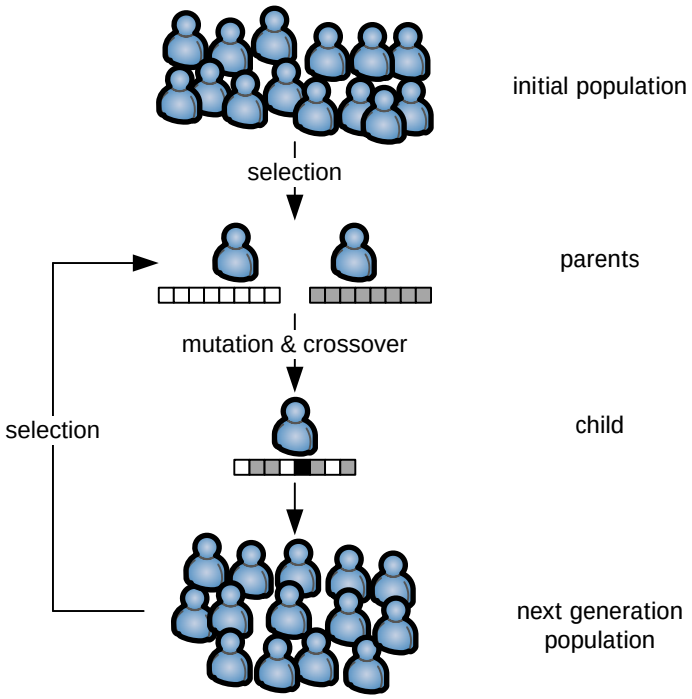


Figure 6.3: Evolutionary Algorithms Overview

selection in evolution theory: From a set of randomly selected initial individuals, they choose two parents and create a new individual by combining their chromosomes. In order to generate solutions that have a high probability of providing a good fitness value, the parents for every new generation of individuals are not selected randomly. Instead, only parents which are not dominated by other individuals are considered as parents. From these, the individuals with the highest fitness values are determined with techniques like binary tournament or probabilistic binary tournament (see [SD08]). After the parents of a new individual have been selected, their chromosomes have to be combined to form a new individual in the next generation. For this purpose usually first a random mutation of selected genes of the parents' chromosomes is performed. Afterwards, a combination of the parents' chromosomes is performed using techniques like single-point crossover or uniform crossover (see [SD08]).

An overview of the mode of operation of multi-objective evolutionary algorithms is given in Figure 6.3. The figure shows that evolutionary algorithms create the individuals in generations, i.e. a predefined number of individuals is created from an existing population in each step. Every member of the next generation is created using the discussed parent selection and crossover operations (the chromosomes of the parents and the child are illustrated as a set of squares in the figure). The creation of new generations is continued until either a threshold for the maximum number of generations has been reached, or until a convergence of the solutions can be observed. The individuals in the final generation then represent the (approximated) Pareto optimum. For DBS self-management they therefore represent a set of possible DBS configurations, which all meet the high-level goals and which represent an optimal tradeoff between the goal functions. From these solutions the self-management logic then has to choose one.

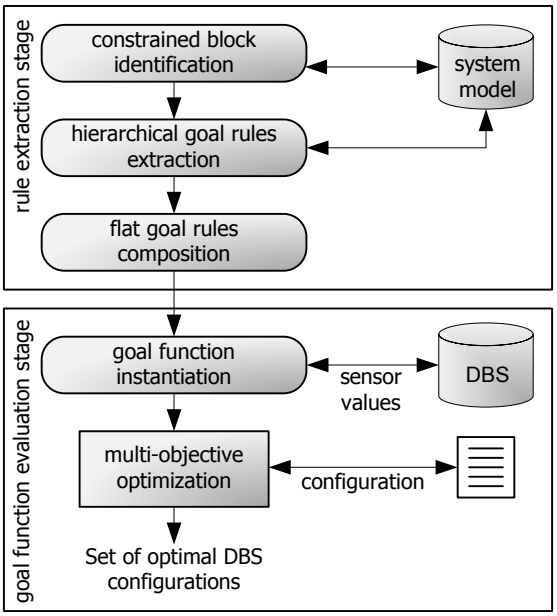


Figure 6.4: Two-staged goal function creation process

There is a rich set of existing algorithms (e.g. NSGAII, PAES, SPEA2; see [CVL07]) and frameworks (e.g. ParadisEo, EvA2, jMetal) that can be employed to derive solutions to MOO problems. However, these frameworks and algorithms all expect a set of "flat" mathematical expressions as input objective functions and constraint definitions. In contrast, the objective functions in the DBS system model are hierarchically structured in order to keep the modelling complexity manageable. The tasks of deriving the required expressions from the system model, enriching them with the current sensor values, and instantiating them for the applicable service classes are the subject of the following Section 6.3. Section 6.4 discusses the selection of a concrete DBS configuration from the solution set.

6.3 System Model Analysis

In order to apply MOO techniques to the hierarchically structured SysML model, a two-staged transformation process has been designed, which is illustrated in Figure 6.4. The first stage of this goal-function creation process comprises the extraction of the relevant goal calculation rules from the SysML model. It only has to be performed once for a particular SysML model. In the second stage, the actual goal functions are instantiated from these rules and enriched with the current sensor and configuration values. In the following, the individual processing steps of these two stages are described in detail. For the sake of clarity, all model validation checks (e.g. for cycles in the rule definitions) are excluded from these descriptions. Furthermore, only the more complex task of extracting and instantiating the goal functions is described, whereas effector constraints are omitted.

The first step *constrained block identification* of the rule extraction stage is the identification of all goal values that have been applied to a SysML `Block` in the model. For this purpose, all

Algorithm 6.1: Determine relevant Goal Functions

Input: SysML model M
Output: Set of goal functions gf

```
1  goalParam ← getConfig(M, ConfType.Goal);
2  forall the goals g ∈ goalParam do
3      constrainedBlocks ← getConstrainedBlocks(g);
4      forall the blocks b ∈ constrainedBlocks do
5          goalAttribute ← b.getGoalAttr(g);
6          constraintProperty ← goalAttribute.getOtherConnectorEnd();
7          constraintBlock ← constraintProperty.getContainer();
8          nestedRule ← createRule(b, constraintBlock, null, null);
9          flatRule ← nestedRule.flatten();
10         gf.add (flatRule);
11     end
12 end
13 return gf
```

goal configuration elements (defined in a designated SysML package) are located. Afterwards, all those `ConstraintBlock` elements in the model are identified, which have a `Property` with a type equal to one of these configured goals (*goal attributes*). For each of these goal attributes, the calculation rule must have been defined using the goal function declaration technique described in Section 5.3. Hence, these `Properties` must have a `Connector` to a `ContraintProperty` in a SysML `ConstraintBlock`. This `ConstraintBlock`, which defines the top-level rule of the objective function, is passed on to the second step.

In the *hierarchical goal rules extraction* step the calculation rule of a particular goal attribute is extracted from the SysML model. The extraction algorithm that has been developed for these hierarchical rule definitions is shown in Algorithm 6.2. After extracting the rule string from the `ConstraintBlock` (line **1**), this algorithm investigates its properties. For every `ConstraintProperty` in the block, it determines the model element that it is connected to (**2-3**). If the element is a terminal value, i.e. a sensor, effector or configuration value, then this property and its access information is extracted from the model and stored as the value of the property (**10-11**). If the connected element is a `ConstraintProperty` in a `ConstraintBlock`, then the value of the property is again a rule, which is created by recursively calling Algorithm 6.2 (**6-8**). No return value is assigned to properties which resemble return values of nested rules. These return values are identified by a connection to the parent element, which is passed as a parameter to this algorithm (**4-5**). In all cases, the property information is added to the rule object(**12**).

After all the required information has been extracted from the SysML model, flat mathematical expressions are created in the *flat goal rules composition* step. Furthermore, this step connects the variable names in the rule string to the rule properties. The processing in this step is described in Algorithm 6.3. For each token in the rule string this algorithm determines

Algorithm 6.2: Extract Rule Information from SysML model

Input: Block *constrainedBlock*, ConstraintBlock *constraintBlock*, Property *parent*
Output: Nested rule *rule*

```
1  rule.setRuleString(constraintBlock.getOwnedRule());
2  forall the properties p ∈ constraintBlock do
3  |  otherEnd ← constrainedBlock.getConnectedElement (p);
4  |  if otherEnd = parent then
5  |  |  p.setType(NESTED_RESULT);
6  |  else if otherEnd.getContainer().isConstraintBlock() then
7  |  |  p.setType(RULE);
8  |  |  p.setValue(ExtractRule(constrainedBlock, otherEnd.getContainer(), p));
9  |  else
10 |  |  p.setType(TERMINAL);
11 |  |  p.setValue(otherEnd.getTerminalInfo());
12 |  rule.add(p);
13 end
14 return rule
```

whether it is a number, an operator, an aggregation function or a property. In the former two cases, the value is simply added to a sorted list of rule components (*ruleComponents*; **2-3**). For aggregation functions, the aggregation rules cannot be applied to the rule string at this stage of processing yet, because the values of the aggregation variable (e.g. the number of bufferpools) are only available at runtime. Hence, only the aggregation information (aggregation variable, aggregation function, start token, end token) is added to the resulting *flatRule* (**4-5**). For properties, the processing rules depend on the type of property: References to rules must be replaced by the contents of the other rule, so Algorithm 6.3 is called recursively. This approach ensures that the contents of the nested rule are added before the processing of the current rule is continued (**10-11**). Result values in nested rules are not required for rule composition and may be skipped (**8-9**). If the property is a terminal (i.e. sensor, effector or configuration value), then it is first checked whether or not it resembles a goal attribute. If so, the optimization operator ($<$ or $>$) and a possible service class property, which must be stored for future reference, are determined (**14-17**). Otherwise, the property is simply added to the overall list of rule components (its value is determined at runtime). However, as the same property names may have been used in sub-rules, the original property name must be replaced with a unique name (**19-20**).

In order to derive appropriate DBS configurations from the flat goal rules, the self-management logic has to fill in the actual sensor and configuration values at runtime in the *goal function instantiation* step. This stage has to take into account service-class-specific goal functions (see Section 5.3), because there may be multiple goal values for different user groups or applications. In order to consider these different goal values during the optimization process, the self-management function has to create a separate goal function instance for every service

Algorithm 6.3: Conversion of Nested Rule to Flat Rule Representation

Input: Nested rule *rule*, Flat rule *flatRule*

```
1  forall the tokens token ∈ rule.getRuleString() do
2      if isNumber(token) or isOperator(token) then
3          flatRule.ruleComponents.add(token);
4      else if isAggregationFunction (token) then
5          flatRule.ruleComponents.getAggType(getAggStart(token), getAggEnd(token),
           getAggFunction(token), getAggProperty(token));
6      else
7          property ← rule.getProperty (token);
8          if property.getType() = NESTED_RESULT then
9              skipOperator();
10         else if property.getType() = RULE then
11             FlattenRule(property.getValue(), flatRule);
12         else if property.getType() = TERMINAL then
13             if property.isGoal() then
14                 if property.isScConstrained() then
15                     scPropName ← property.getScPropertyName();
16                     flatRule.setScProperty(rule.getProperty(scPropName));
17                 flatRule.setoptOperator (nextToken());
18             else
19                 property.setUniqueName();
20                 flatRule.ruleComponents.add(property);
21
22
23
24 end
```

class. In each instance the sensor values then reflect the values for this particular service class. The MOO thus derives a configuration that meets all goal functions, i.e. all goal values for all service classes.

In addition to service classes, the *goal function instantiation* step also has to consider the aggregation functions. As described above, only the aggregation information (aggregation variable, aggregation function, start token, end token) is determined in the static *rule extraction stage*. The value of the aggregation variable may depend on a runtime sensor (e.g. the IDs of the bufferpools existing in the DBS). Hence, this information has to be determined at runtime in the *goal function instantiation* step, and the aggregation function then has to be expanded accordingly. For example, the sum of all pages in all existing bufferpools has to be calculated by adding the sensor value for the size of every existing bufferpool.

Algorithm 6.4 summarizes the instantiation of a flat rule for a particular service class. For every rule component *rc* the algorithm checks its type: If it is an aggregation, then all rule components in the aggregation range have to be appended once for every value of the aggregation variable (**5**), and concatenated according to the aggregation function (**6**, e.g. summarized for a

Algorithm 6.4: Instantiation of Goal Functions.

Input: Service class name *serviceClass*, Rule components *ruleComponents*, Map of aggregation variables *aggVars*, Result string *result*

```
1  pos ← 0;
2  while pos < ruleComponents.size() do
3      rc ← ruleComponents[pos];
4      if rc is Aggregation then
5          forall the values v of rc.getAggProperty() do
6              concatenateAgg();
7              aggVars.put (rc.getAggPropName(), v);
8              InstantiateRule(serviceClass,
               ruleComponents.getRange(rc.getAggStart(), rc.getAggEnd()), aggVars,
               result);
9          end
10         aggVars.remove (rc.getAggPropName());
11         pos ← rc.getAggEnd();
12     else if rc is Property then
13         if rc.isSensor() or rc.isConfiguration() then
14             result.append (getValue (rc, serviceClass, aggVars));
15         else if rc.isEffector() then
16             rc.setEffectorIncarnationName(aggVars);
17             storeEffectorIncarnation(rc, aggVars);
18             result.append(rc.getEffectorIncarnationName());
19         pos ++;
20     else
21         result.append(rc);
22         pos ++;
23
24 end
```

SUM function). Hence, Algorithm 6.4 is called recursively on the aggregation range in this case (**8**). The current value of the aggregation variable also has to be passed as a parameter (**7**). If *rc* is not an aggregation but a sensor or configuration then its value can simply be read and appended to the resulting mathematical expression *result* (**13-14**). It is important to note that these values may depend on both the current service class and the set of aggregation variables. If *rc* refers to an effector, then this effector name has to be added to the *result* and marked as a decision variable (**17-18**). However, as an effector may depend on the current values of the aggregation variables, unique names must be constructed beforehand (**16**; e.g. sizes of buffer-pools: "Size_BP1", "Size_BP2", ...). All numbers and operators in the rule components may simply be added to the resulting mathematical expression (**20-22**).

With the described two-staged approach the constraints from the SysML model only have to be extracted and converted to flat rules once, whereas at runtime these flat rules can be quickly instantiated. After their instantiation, the goal functions are passed to the MOO

algorithm, which automatically determines a set of optimal solutions to these functions (the Pareto optimum). However, the goal functions extracted from the system model only constitute the objective vector $F(\boldsymbol{x}) = (f_1(\boldsymbol{x}, \ldots f_k(\boldsymbol{x}))$ of the multi-objective optimization problem. The set of constraints $E(\boldsymbol{x}) = (e_1(\boldsymbol{x}), \ldots e_k(\boldsymbol{x}))$, which also has to be taken into account by the self-management logic, has to be extracted from the system model accordingly. In order to indicate that a particular solution does not satisfy the constraints, a value of 0 can be passed to the MOO algorithm as the fitness value, for instance. Thus, solutions that violate domain restrictions on effectors or dependency rules between the effector values are excluded from the solution set.

6.4 Solution Selection

Section 6.3 has described an approach for deriving a set of goal functions $F(\boldsymbol{x})$ and a set of constraints $E(\boldsymbol{x})$ from a hierarchical SysML system model. The effectors of the DBS constitute the parameters $\boldsymbol{x}$ of these goal functions and constraints, whereas all references to sensors are replaced with constant values. By passing these goal functions to an MOO algorithm, e.g. a multi-objective evolutionary algorithm, a set of optimal trade-off solutions can automatically be derived. All of these solutions (i.e. DBS configurations) meet the high-level goals defined by a DBA and they are members of the Pareto optimum.

Although all of these DBS configurations are optimal, only one of them can be selected for the implementation in the DBS. Hence, a strategy is required for selecting a solution from the solution set. As all of the solutions are optimal, it would of course be possible to randomly choose one of the configurations. However, changing the configuration of a DBS often causes change costs. For instance, changes to the physical design may require long-running index creation tasks. Changes to configuration parameters may even require a shutdown and restart of the DBS to become effective.

Due to the overhead that may be caused by changes to effectors, the selection of a configuration from the solution set determined by the MOO algorithm should not be made randomly. Instead, the expected change costs of the various solutions should be compared. A simple approach is the comparison of the candidate configurations to the current configuration of the DBS. The configuration that requires the smallest number of effector changes could then be chosen as the result of the reconfiguration analysis. Although this approach will significantly reduce the number of required effector changes, it does not consider the fact that the change costs between the effectors differ largely. Some changes, e.g. to online-adaptable configuration parameters, may not cause any change costs at all. Hence, the system model should contain information about the expected change costs for every effector. Thus, the reconfiguration analysis could easily compare the expected change costs for all solutions and choose the DBS configuration that causes the least expected overhead.

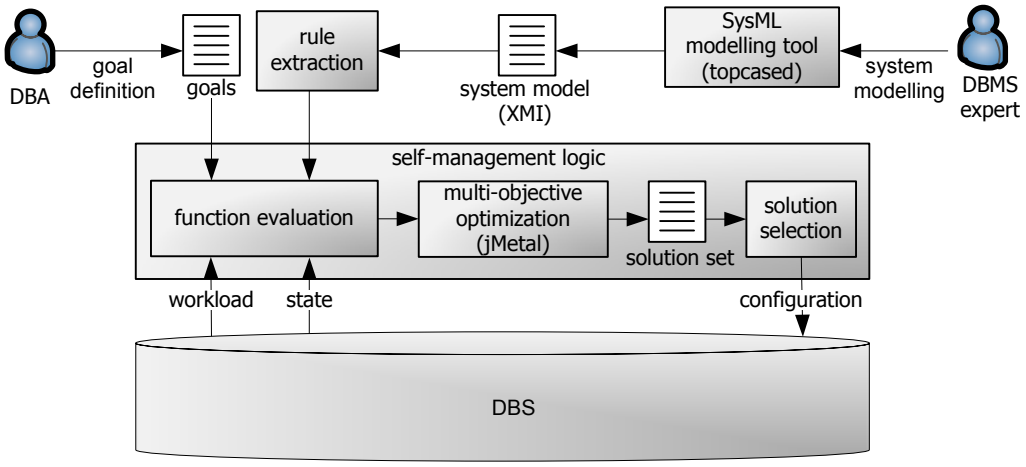


Figure 6.5: System model definition and evaluation prototype

6.5 Evaluation

For the evaluation of the reconfiguration analysis concepts the infrastructure illustrated in Figure 6.5 has been created. The system model is supposed to be created by a *DBMS expert* with in-depth knowledge of a particular DBMS. For system modelling the open source tool *TOPCASED* (http://www.topcased.org) is used, which natively supports the SysML modelling language. This tool can store an SysML model as a *XMI* file, which serves as an excellent basis for the extraction of the flat goal rules according to the first stage described in Section 6.3. The – unparametrized – flat goal rules are then passed to the *function evaluation* stage of the *self-management logic*. Here the flat goal rules are used to re-instantiate the goal functions in every reconfiguration analysis run, including their parametrization with the *workload* and *state* of the *DBS* (IBM DB2) and *DBA*-defined *goal* information. Afterwards, the goal functions are passed to the MOO, which derives a set of Pareto optimal solutions (i.e. effector values) for them. For performing the MOO, the *jMetal* framework ([DNL+06]) has been chosen, because it provides a rich set of MOO algorithm implementations. It is then the tasks of the self-management logic to select one of the DBS configurations from the solution set.

The running example introduced in Section 5.1 has been used to evaluate the functional and non-functional properties of the described reconfiguration analysis approach. Thus, a system model has been created, which predicts the response time of a DBS based on the size of the system buffer. To be able to compare the predictions to the real results of an existing DBS, IBM DB2 has been used in the evaluation environment. Consequently, the created system model reflects the structure, sensors and effectors of IBM DB2, and it comprises all the required system model contents identified in Section 5.2. In particular, it has been defined using a hierarchical structural approach (similar to Figure 5.6 on page 152), and it defines all required sensor and effector information. SysML constraints have been used to store the knowledge about the estimated hitratios according to Equation 5.1 (similar to Figures 5.9 and 5.10). Furthermore, two goal functions have been defined: *ResponseTime* estimates the response time according to

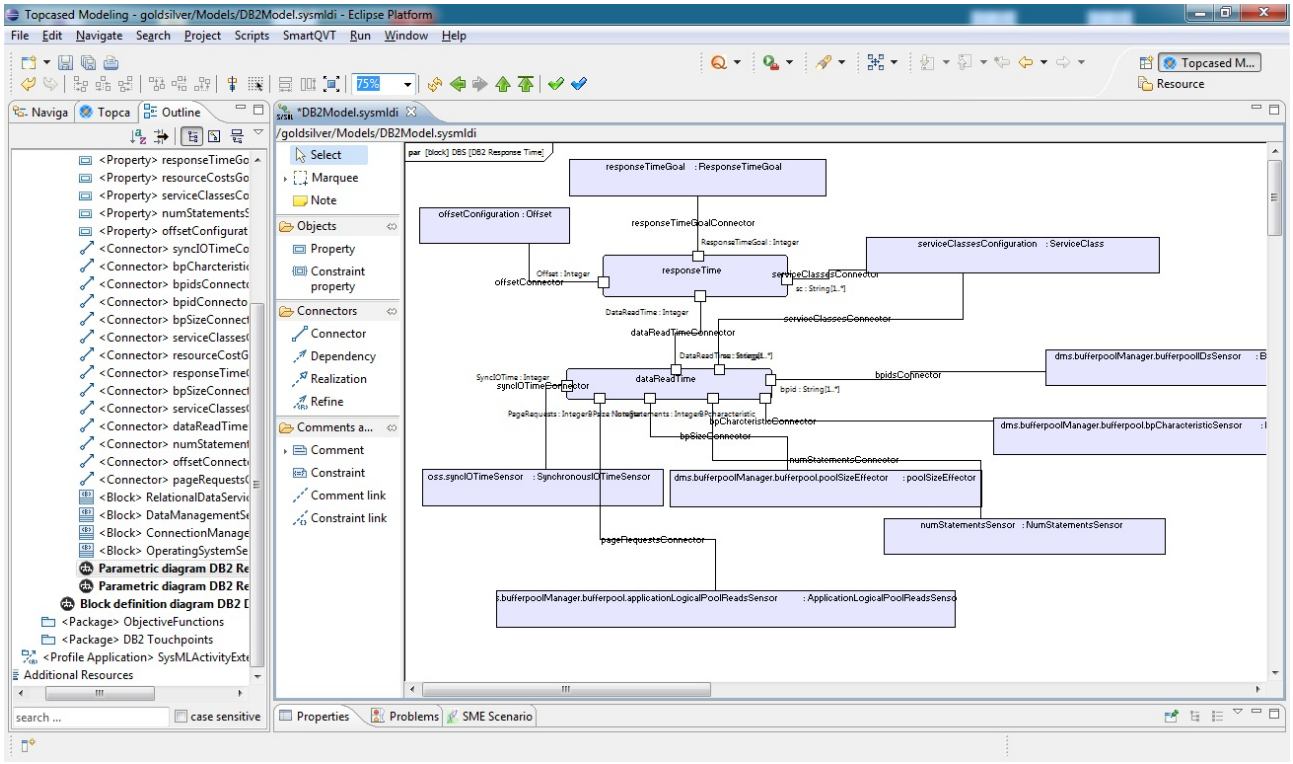


Figure 6.6: Parametric diagram for a response time goal function (running example) in TOPCASED

Equation 5.2, and *ResourceCosts* summarizes the bufferpool sizes according to Equation 5.4. An overview of the response time parametric diagram for the running example in TOPCASED is shown in Figure 6.6.

In order to evaluate the functionality and performance of the described reconfiguration analysis approach, the testbed shown in Figure 6.7 has been set up. Two distinct applications have been simulated with load generators. The workload manager [CCI+08] of IBM DB2 Version 9.7 has been configured to automatically assign all workloads to two service classes (*gold* and *silver*). As shown in Figure 6.7, the load of these two service classes has resulted in unevenly distributed access to the two bufferpools *BP1* and *BP2*. Each of the bufferpools had been configured to serve one tablespace, where *tablespace1* had only 60% of the amount of data in *tablespace2*.

Applying the system model to the evaluation scenario results in three goal function instances: One response time goal function for each service class (*gold* and *silver*), and one resource costs goal function for the overall memory usage. Exemplary solutions sets that have been determined by the prototype for these goal functions are depicted in Figure 6.8 and Figure 6.9. All results were produced using the NSGAII algorithm [DPAM02] with a population size of 100 and a maximum of 25000 generations, because the NSGAII algorithm has shown to provide good results for the running example scenario. A detailed analysis of the results of various MOO algorithms has been performed in [Pog09]. The results in Figure 6.8 were computed for the goal values ResponseTime[Gold]<8ms, ResponseTime[Silver]<15ms

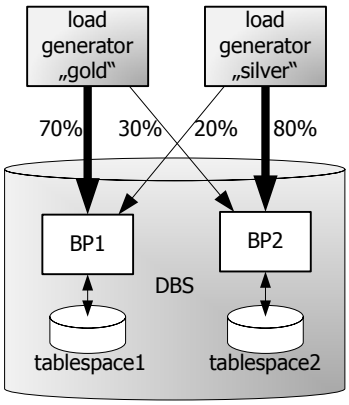
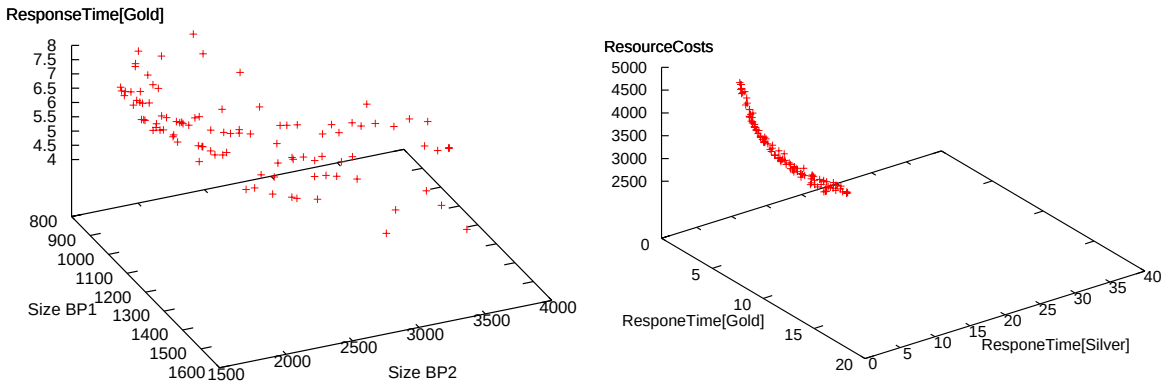


Figure 6.7: Evaluation scenario overview

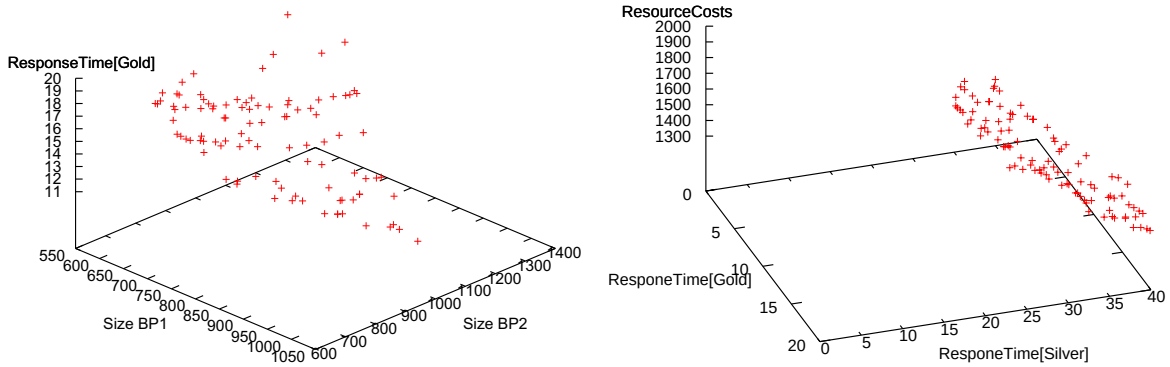


(a) ResponseTime(Gold) estimations (b) Pareto-front

Figure 6.8: Illustration of solution set for goals:
ResponseTime[Gold]<8ms; ResponseTime[Silver]<15ms;
ResourceCosts<5000pages

and ResourceCosts<5000pages. Figure 6.8a shows the expected response time for service class *gold* depending on the sizes of BP1 and BP2. Figure 6.8b depicts the corresponding Pareto optimum, where each data point resembles a combination of the effectors Size_BP1 and Size_BP2. Figure 6.9 shows the same information as Figure 6.8, but for the goal values ResponseTime[Gold]<20ms, ResponseTime[Silver]<40ms, and ResourceCosts<2000.

The reconfiguration analysis for a DBS is a complex and expensive task, which must not be executed continuously, but only when goals are missed or when there is a workload shift. With the described reconfiguration analysis approach, the reconfiguration analysis time is the sum of the time required for instantiating the goal functions and the execution time of the multi-objective optimization. For the system model of the running example, the goal instantiation (including the 27 DBS queries for filling in the sensor values) took 0.6s on an average. The execution time of the MOO-algorithm depends on the selected algorithm and its configuration. Figure 6.10 shows the execution times for some MOO-algorithms that have been observed for different population sizes and numbers of generations. The extraction of the system model from the XMI file (only required after system model changes) took 0.7s on an average. All experiments were made on a single-core PC (CPU cycle: 3GHz, Memory: 3GB).



(a) ResponseTime(Gold) estimations (b) Pareto-front

Figure 6.9: Illustration of solution set for goals:
ResponseTime[Gold]<20ms; ResponseTime[Silver]<40ms;
ResourceCosts<2000pages

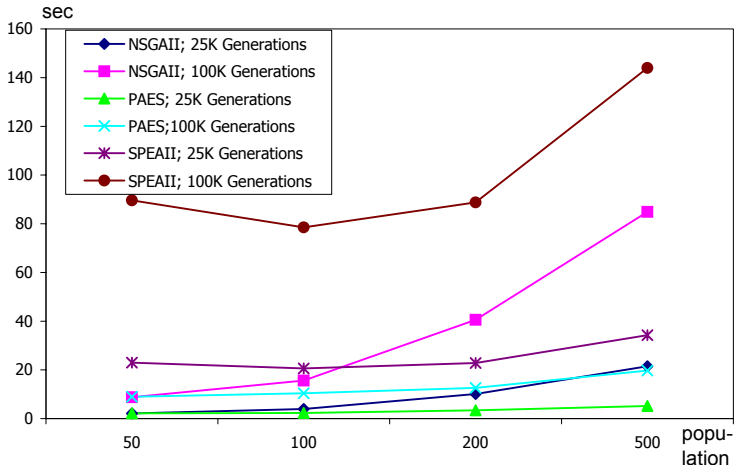


Figure 6.10: Execution times for MOO-algorithms under different configurations

In order to validate the quality of the configurations in the solution sets, the computed DBS configurations have been implemented in the IBM DB2. Afterwards, the predicted response times have been compared to the actual response times when re-executing the evaluation workloads. A set of sample results is shown in Table 6.1. The first four columns show the response times for gold (RT_g) and and silver (RT_s) requests observed for a given initial configuration (bufferpool sizes $BP1$ and $BP2$). From the characteristics observed with this configuration, reconfiguration analysis have been performed for different response time goals (G_g for gold; G_s for silver) and allowed resource costs (G_{res}). For each goal, Table 6.1 reports three exemplary solutions from the solution set in columns 8 to 11 (RT_g and RT_s here denote the *estimated* response times). The final two columns report the actual response times that have been observed after implementing the corresponding configuration in the DBS (all solutions have been derived using the NSGAII algorithm with a population of 100 and 25000 generations). The results show that despite the simplistic system model the response time can be predicted sufficiently well. For the response time of the gold workload, for example, the average absolute difference between

Table 6.1: Response Time Prediction Accuracy Validation

Initial Configuration				Goal Definitions			Solutions				Observed RTs	
BP1	BP2	RT_g	RT_s	G_g	G_s	G_{res}	BP1	BP2	RT_g	RT_s	RT_g	RT_s
350	350	40	73	10	20	5000	765	1144	10	18	12	14
							1008	1249	8	15	9	13
							1514	3485	4	4	3	4
				20	40	2000	582	616	19	40	13	18
							542	1052	17	23	12	15
							872	1127	9	18	11	13
500	500	30	46	10	20	5000	948	1246	10	20	10	12
							1026	2033	7	10	6	7
							1503	3496	4	5	4	4
				20	40	2000	648	590	20	40	15	20
							831	915	12	26	11	15
							910	1003	11	23	11	14
750	750	12	15	10	20	5000	1012	665	10	17	11	15
							873	875	10	11	11	15
							1151	1567	5	5	7	9
				20	40	2000	736	419	20	37	29	51
							756	622	15	21	17	19
							1034	898	8	10	10	14

the estimated and the actual response time is 2ms (the average relative difference is 18%). Of course, the system model needs to be extended in the future to also predict the effects of other important configuration changes, e.g. to the physical design, sorting, logging, and optimizer.

6.6 Related Work

Using multi-objective optimization for the self-optimization of a DBS is a novel approach, which (to the best of knowledge) has not been published in other works before. Previously, this subject in database systems has only been seen from the query processing point of view, i.e. as the retrieval of Pareto-optimal result sets from a DBS (*skyline queries*; e.g. [KRR02], [BG04]). The focus in these works has been on creating efficient query processing algorithms in the DBS, not the application of multi-objective optimization techniques for self-management. Likewise, there are no known sources on extracting and composing objective functions from a DBS system model. The lack of related work in this area is caused by the fact that – except the position paper [Sch09] by Schmidt – no other approaches for creating a *quantitative* DBS system model are known.

In the area of complex IT infrastructure management, rule-based frameworks like Accord [LP06] have been developed. Like the Autonomic Computing Toolkit described in Section 5.5, these frameworks require the administrator to define a set of actions and the conditions under which they are fired (*ECA-rules*). The same approach is taken in policy management frame-

works like [BGJ+04]. So the decision about the necessary reconfigurations to meet business goals is not derived from a quantitative description of the behaviour, but has to be predefined for specific scenarios by an administrator. To overcome this limitation the Accord framework has been extended with a Limited-Look-Ahead-Controller [BPL+06]. However, the controller is limited to the optimization of a single objective function.

In the area of web service composition there has been research on the usage of multi-objective optimization in order to meet SLAs, e.g. in [WCSO08] and [CWC05]. Depending on the QoS requirements, multiple concrete web services are composed to realize an abstract business process. Compared to databases, the objective functions for web service composition are rather simple and the configuration alternatives are limited. Thus, they do not allow the graphical modelling of a hierarchically structured system model, which is essential to keep the modelling complexity manageable for database systems. Instead, the objective functions are expected to be predefined and hard-coded into the self-management logic.

6.7 Conclusions

This section has presented an approach for automatically deriving DBS configurations that meet high-level goals from a DBS system model. As multiple, possibly opposing goal functions may be defined for a DBS, MOO techniques have been identified as an appropriate solution for this purpose. These heuristics determine a set of Pareto optimal solutions for a given set of objective functions. By considering the high-level goals as constraints, configurations that would not meet the goals are automatically excluded from the solution set. Employing the MOO techniques, the major challenges for the evaluation of DBS system models in the reconfiguration analysis have been reduced to deriving the objective functions from a DBS system model, instantiating them at runtime, and selecting one of the solutions from the solution set. Solutions to these challenges have been presented in Sections 6.3 and 6.4.

Section 6.1 has discussed the requirements towards a reconfiguration analysis solution in detail. Table 6.2 lists these requirements and shows that all of these requirements are met by the described approach. The *extraction* requirement demands that all information required for self-management is extracted from the system model, without prior knowledge about the DBS. The algorithms described in Section 6.3 perform exactly this task: they first determine the goals that have been defined for the DBS, and then extract all the necessary rules and touchpoint specifications. In particular, they are able to descend into the *hierarchical structure* of the system model and derive all rules, sensor and effector information from the lower levels, too. Parameters in the rules that do not refer to terminal values (i.e. sensors or effectors) but to behavioural descriptions are replaced accordingly. Thus, flat *goal functions* are composed, which refer to sensor values and effectors only. While the effectors represent the decision variables of the optimization problem, the references to sensors in the goal functions are replaced with their current values. After this *parametrization* they appear as constant values

Table 6.2: System Modelling Requirements

Requirement	Status
Extraction	✓
Hierarchical Structure	✓
Goal Functions	✓
Parametrization	(✓)
Configuration Determination	(✓)
Service Classes	✓

in the goal functions. It is important to note that the current self-management solution only supports single values for the parametrization, whereas the evaluation of the system model in Section 5.4.3 has shown that precise prediction results can only be achieved by considering an entire history of sensor values. The extension of the parametrization concepts for this purpose therefore should be examined in the future. The parametrized goal functions are then passed to the actual *configuration determination* algorithm. Existing implementations of multi-objective optimization algorithms are used for this purpose. However, these algorithms do not determine a single solution, but a set of Pareto optimal solutions, which all represent optimal tradeoffs between the goal functions. Possible strategies for choosing one of these solutions have been discussed in Section 6.4. Choosing the configuration that causes the least change costs has been identified as the most appropriate strategy. However, an analysis of the change costs for all effectors, their representation in the system model and an algorithm for their comparison is still missing. Different goals for *service classes* in contrast are fully supported by the described algorithms. This requirement is met by instantiating the service-class-specific goal functions once for every service class.

Using multi-objective optimization techniques for deriving configurations in a self-managing DBS is a novel approach that (to the best of knowledge) has not been followed in other works before. In contrast to existing solutions it has several advantages: First, it does not require the specification of fixed plans, which are executed under certain conditions (like ECA-rules in policy-controlled frameworks, for example). Second, the implementation of the reconfiguration analysis does not have to be adapted when the system model changes. All algorithms designed in this solution are generic, i.e., they can be applied to any SysML system model that follows the system modelling technique described in Chapter 5. Third, the reconfiguration analysis is not limited to a particular DBMS component. Instead, all components are considered as long as their effect on the goal values is quantified in the system model. Fourth, the result of the reconfiguration analysis is not only a set of optimal solutions, but also the set of the expected values for each of the goal functions. The developed reconfiguration analysis solution therefore is an excellent basis for the development of system models in the future, because DBS configurations can be immediately derived from these models. By implementing the resulting configurations and comparing the goal values to the predictions, the accuracy of models can be judged very quickly.

7 Conclusions

This work has presented concepts for the self-management of relational database systems. The following Section 7.1 provides a summary of the contributions. Section 7.2 gives an outlook on future studies.

7.1 Summary of Contributions

In a nutshell, the contributions of this work to the area of DBS self-management are as follows:

- *System-wide Self-Management:* A novel framework for the self-management of DBS has been developed. In contrast to existing approaches this framework allows a system-wide view on all configuration decisions in the DBS. Using a system model as an external knowledge base, the self-management framework is highly flexible and can be easily adapted to a concrete DBMS.

- *System Model Definition:* The work has described a highly intuitive method for the definition of system models. It has identified the necessary system model contents and illustrated the usage of the graphical modelling language SysML for system model definitions.

- *DB2 System Model:* To illustrate the applicability of the system modelling concepts, a coarse-grained system model for IBM DB2 has been developed. The DB2 model predicts the response time of DB2 depending on its configuration and workload. Experimental evaluations have shown the applicability of the approach.

- *Goal-Driven Reconfiguration Analysis:* A set of generic algorithms for the evaluation of system models has been developed. Using these algorithms, any DBS system model can be used in order to derive DBS configuration that meet high-level goals. Multi-objective optimization techniques are employed for this purpose.

- *Lightweight Workload Analysis:* In order to detect the points in time when a reconfiguration analysis is required for a DBS, the work has developed an analysis framework for the workload of relational DBS. This framework allows the lightweight monitoring of the workload for significant changes. It therefore allows the quick adaptation of the DBS configuration to workload changes, while it restricts the analysis overhead for stable workloads to a minimum.

The following paragraphs discuss how these contributions solve for the open challenges in DBS self-management, which have been identified in Section 2.4:

Self-Optimization When comparing the traditional DBMS technology with the characteristics of an autonomic system, then self-optimization is the most important challenge for research and development. In particular, self-optimization requires a continuous adaptation of the DBS configuration to changes in the workload. An analysis of the existing self-management functions has shown that off-line maintenance tools are not an adequate means to reach this goal, because the DBA has to know when their execution advisable. Instead, on-line self-management functions are required, which continuously monitor some sensor information from the DBS in order to immediately identify benefits of possible reconfigurations. However, currently only a small fraction of the DBS configuration is under control of on-line self-management functions.

This work has presented a self-management approach that significantly differs from the existing solutions. Instead of developing further component-specific or administration-task-specific on-line self-management functions, an integrated, system-wide approach towards DBS self-management is followed. This integrated approach has been designed with only a single self-management loop, which monitors and analyses the workload and state of the overall DBS. Whenever it detects the need for a reconfiguration, the self-management logic automatically triggers the execution of a multi-objective optimization algorithm, which determines a new optimal configuration of the DBS. This optimization is based on a system model, which quantitatively predicts the behaviour of the DBS under different configurations and environmental conditions. By giving a first coarse-grained model for the IBM DB2, Section 5 has shown that the prediction of the DBS performance using a system model is possible.

In order to extend the self-optimization capabilities to other DBS components, the proposed solution does not require the development of additional on-line self-management functions. It is only necessary to integrate a quantitative description of their behaviour into the system model. All other aspects of self-management are automatically provided by the framework: the identification of the points in time when a reconfiguration analysis is required (workload shift detection), the parametrization of the model with the current sensor values, the computation of optimal configuration values for the DBS components, the consideration of side effects on other components, and the implementation of the new values in the DBS configuration. In order to make the required extensions of the system model as intuitive as possible, a graphical modelling language is supported by the framework. Thus, it is expected that the developed framework will greatly simplify the development of new self-management functionality for DBS.

Goal-Independency The vision of autonomic computing demands *all* systems in an enterprise to automatically adapt to high-level goals and policies that are defined by an administrator. However, none of the existing DBS self-management solutions is capable of considering goal values for response time, throughput, or costs in its analysis. Instead, all of them perform a best-effort performance optimization.

The integrated self-management framework described in this work is explicitly designed to be goal-driven: On the one hand, a reconfiguration analysis is supposed to be triggered when goal values are missed. On the other hand, every new configuration meets the goal values in the best possible way. The support for goal values integrates naturally with the multi-objective optimization approach chosen for deriving the DBS configurations from the system model, because goal functions can be easily integrated into the system model as constraints. These goal functions then form the subjects of the optimization, where the parameters of the goal functions represent configuration parameter settings. Thus, only solutions that meet the goal values will be identified by the self-management logic as possible DBS configurations.

Interdependency Today's DBMS are complex systems with lots of interdependencies between their components. Hence, changes to the configuration of one component may have side effects on other components, especially if the parameters affect the usage of a shared resource. However, considering all side effects of a reconfiguration would cause a high overhead. Today's on-line self-management functions therefore analyse the performance and workload related to one configuration parameter or one system component only, without regarding side-effects.

With the system-wide self-management framework described in this work, the challenge of considering side-effects during the reconfiguration analysis can be solved easily. All parameters that influence the behaviour of a component can be represented in the system model in a intuitive way. Of course, a quantitative model of the effects of these parameters on the components behaviour has to be defined as a prerequisite. Thus, the influence of a single parameter on the performance of multiple components can be directly derived from the system model by following its connections to performance constraints. The reconfiguration analysis therefore can easily determine the system-wide effects of changing the parameter's value.

Overhead With existing solutions every single on-line self-management function separately monitors some particular sensor information, analyses this information and initiates appropriate reconfiguration actions when required. Thus, every on-line self-management function causes overhead, which adds to the usual statement execution time. Most of the time this analysis overhead will be wasted, because the usage and state of the DBS are almost constant, and therefore no changes to the configuration are required.

Using the workload shift detection approach, the analysis overhead for possible reconfigurations can be restricted to situations where there actually is a strong hint that the usage of the DBS has changed. While the workload is stable (and the goals are met), only the workload shift detection algorithm has to be executed. The experimental evaluations have shown that the workload shift detection concepts developed in this work can be executed continuously with very little overhead. Of course, the reconfiguration analysis that is performed when a workload shift has been detected employs heavy-weight optimization algorithms. But as workload shifts are expected only rarely, the usual statement processing will not be affected by this overhead. Furthermore, the fact that a workload shift has been detected indicates that the result of the

reconfiguration analysis is likely to find a new configuration resulting in an increased DBS performance.

Workload-pattern Unawareness Current on-line self-management functions in DBS make their reconfiguration decisions based on the past observations, i.e. they assume that the workload remains identical in the future. But in the presence of periodic workload changes, e.g. between day-time and night-time, the self-management functions could anticipate upcoming changes and refrain from reconfigurations shortly before a workload change. However, there are currently no concepts for considering periodic workload changes in the existing self-management solutions.

By storing all workload models, the workload shift prediction solution developed in this work maintains a knowledge base of historic workloads. Based on this knowledge periodicity detection techniques have been designed and evaluated, which allow the prediction of upcoming periodic workload changes. This information provides a valuable input to the self-management logic. Furthermore, the knowledge about re-occurring workloads in the future may be used to assign concrete DBS configurations to the workload models, thus avoiding the need for an expensive reconfiguration analysis.

Overreaction As discussed in Section 2.4.6, on-line self-management functions in DBS often lack a precise model of their managed component. They implement rules-of-thumb algorithms instead. If the rules-of-thumb assumptions are incorrect in certain environments, then the self-management functions may overreact. As a result, the performance of the DBS may even downgrade after the tuning decision.

With the approach presented in this work, quantitative models of the managed components are mandatory. For every goal function it must be precisely defined how the expected value depends on the sensor and effector values. Thus, overreactions due to incorrect rules-of-thumb assumptions are avoided. The self-management framework furthermore supports the development of the required models by allowing an immediate testing of the model evaluation results.

7.2 Outlook

The goal-driven, system-wide self-management framework developed in this work provides the necessary concepts to overcome the issues faced with traditional DBS self-management solutions. Currently the framework of course exists only in the status of a research prototype. There are many areas where additional studies and experimental evaluations are required before the self-management framework may be considered sufficiently mature for usage in production systems.

Reliable Service-Level Management Section 3.1 has identified five challenges that have to be considered when creating the integrated, system-wide self-management framework developed

in this work. While the challenges *Lightweight Workload Monitoring*, *System Model Definition*, and *System Model Evaluation* have been investigated in detail in this work, the challenge of *Reliable Service-Level Management* has been excluded from the studies. The subject of this challenge is to reliably ensure that the high-level goals defined by the DBA will be met under all conditions. Thus, it is important to detect possible goal violations as early as possible, because only then the reconfigurations can be triggered in time to avoid goal violations. The goal value monitoring component therefore could employ trend detection or other time series analysis techniques. A reliable service-level management should be thoroughly designed in a future work.

Anticipatory State Monitoring In addition to the *Reliable Service-Level Management*, also the challenge *Anticipatory State Monitoring* has been excluded from the investigations in this work. This challenge requires a continuous lightweight monitoring of internal DBS state information of components which may demand administrative actions even though the workload is stable. The fragmentation of the data over time, for example, may require a reorganization of the data without any changes in the workload. State information will typically not change dramatically within a short period of time, but it will show a trend over time, which should be observed by the self-management logic. Upcoming maintenance operations should then be scheduled in time before problems actually appear, and for time periods when little workload is expected on the DBS. The identification of the internal state information that has to be monitored by the self-management logic and the design of an appropriate monitoring technique should be investigated in the future.

Model extension and refinement The system model for IBM DB2 presented in Section 5.4 is a coarse-grained model, which considers the most important components and configuration parameters only. The evaluation results for this exemplary model have shown that some aspects of the behaviour are described sufficiently well, whereas other parts obviously still need refinement. It has shown that in order to precisely predict key performance indicators like the response time or throughput extensive experimental evaluations of the DBMS with varying workloads will be required in the future. As a result of these evaluations the model has to be refined and extended where appropriate. The groundwork for these experimental evaluations has been developed in this work: First, a testbed has been created, which provides a basic set of benchmark and custom workloads on different database schemas. Second, the self-management logic can be used in order to derive configurations from candidate models, together with the estimated goal values. Thus, candidate models can be easily validated by implementing the configurations and comparing the estimated performance values against the actual observations. Nevertheless, the experimental evaluation will require a large amount of time for the iterative refinement of the model.

Goal/Policy-Framework Integration As illustrated in Section 2.1, the vision of autonomic computing is larger than the self-management of an individual computing resource like a DBS. It rather demands that the entire IT-infrastructure of an enterprise should be self-managing according to a set of high-level business goals. To keep the complexity of breaking down the enterprise-level goals to configuration parameter values manageable, a hierarchical structure is proposed in [IBM05], which refines the goals at every level. Finally, the managed resources at the leaf-node level will have to accept resource-specific goal definitions and autonomically find configurations that meet these goal values. So the DBS should be able to exchange goal definitions and performance reports with other self-management components in the IT-infrastructure. As the current solution only accepts a set of goal values from a flat configuration file, a future work should examine existing policy- or goal-definition frameworks for their applicability to the integrated self-management solution developed in this work.

Capacity Planning for DBS An important challenge for the management of data centres is the capacity planning. Given a set of SLA definitions by the customer, the operators of a data centre have to choose the correct physical hardware for an application or infrastructure service. For DBS, it is expected that the concepts developed in this work will greatly support this task: On the one hand the workload model histories provide the operators with a detailed description of the load of the DBS, and with the time intervals when this workload usually occurs. On the other hand, the system model allows the prediction of the performance of the DBS under different configurations. Thus, the operator can predict how well the given SLAs will be met on a particular hardware. The application of the developed techniques for capacity planning should be investigated in the future.

Bibliography

[ABCN06] Agrawal, Sanjay ; Bruno, Nicolas ; Chaudhuri, Surajit ; Narasayya, Vivek R.: AutoAdmin: Self-Tuning Database SystemsTechnology. In: *IEEE Data Engineering Bulletin* 29 (2006), Nr. 3, S. 7–15

[ACK+04] Agrawal, Sanjay ; Chaudhuri, Surajit ; Kollár, Lubor ; Marathe, Arunprasad P. ; Narasayya, Vivek R. ; Syamala, Manoj: Database Tuning Advisor for Microsoft SQL Server 2005. In: [NÖK+04], S. 1110–1121

[ACN00] Agrawal, Sanjay ; Chaudhuri, Surajit ; Narasayya, Vivek R.: Automated Selection of Materialized Views and Indexes in SQL Databases. In: Abbadi, Amr E. (Hrsg.) ; Brodie, Michael L. (Hrsg.) ; Chakravarthy, Sharma (Hrsg.) ; Dayal, Umeshwar (Hrsg.) ; Kamel, Nabil (Hrsg.) ; Schlageter, Gunter (Hrsg.) ; Whang, Kyu-Young (Hrsg.): *Proceedings of* 26^{th} *International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2000. – ISBN 1558607153, S. 496–505

[ACN06] Agrawal, Sanjay ; Chu, Eric ; Narasayya, Vivek: Automatic physical design tuning: workload as a sequence. In: Chaudhuri, Surajit (Hrsg.) ; Hristidis, Vagelis (Hrsg.) ; Polyzotis, Neoklis (Hrsg.): *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM Press, 2006. – ISBN 1595932569, S. 683–694

[AFG+04] Alur, Nagraj ; Farrell, Peter ; Gunning, Philip ; Mohseni, Saeid ; Rajagopalan, Swaminaathan ; International Business Machines Corporation (Hrsg.): *DB2 UDB ESE V8 non-DPF Performance Guide for High Performance OLTP and BI*. 1. Armonk, NY, USA: International Business Machines Corporation, 2004. – Redbook

[AHL+04] Aboulnaga, Ashraf ; Haas, Peter J. ; Lightstone, Sam ; Lohman, Guy M. ; Markl, Volker ; Popivanov, Ivan ; Raman, Vijayshankar: Automated Statistics Collection in DB2 UDB. In: [NÖK+04], S. 1146–1157

[AHWY03] Aggarwal, Charu C. ; Han, Jiawei ; Wang, Jianyong ; Yu, Philip S.: A Framework for Clustering Evolving Data Streams. In: [FLA+03], S. 81–92

[AHWY04] Aggarwal, Charu C. ; Han, Jiawei ; Wang, Jianyong ; Yu, Philip S.: On Demand Classification of Data Streams. In: Kim, Won (Hrsg.) ; Kohavi, Ron (Hrsg.) ; Gehrke, Johannes (Hrsg.) ; DuMouchel, William (Hrsg.): *Proceedings of the* 10^{th} *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA : ACM Press, 2004. – ISBN 1581138881, S. 503–508

[ANY04] Agrawal, Sanjay ; Narasayya, Vivek R. ; Yang, Beverly: Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In: Weikum, Gerhard (Hrsg.) ; König, Arnd C. (Hrsg.) ; Dessloch, Stefan (Hrsg.): *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM Press, 2004. – ISBN 1581138598, S. 359–370

[BBD+09] Babu, Shivnath ; Borisov, Nedyalko ; Duan, Songyun ; Herodotou, Herodotos ; Thummala, Vamsidhar: Automated Experiment-Driven Management of (Database) Systems. In: Fox, Armando (Hrsg.): *Proceedings of the* 12^{th} *Workshop on Hot Topics in Operating Systems*, Online Proceedings, 2009

[BBR+07] Bensch, Michael ; Brugger, Dominik ; Rosenstiel, Wolfgang ; Bogdan, Martin ; Spruth, Wilhelm G. ; Baeuerle, Peter: Self-Learning Prediction System for Optimisation of Workload Management in a Mainframe Operating System. In: Cardoso, Jorge (Hrsg.) ; Cordeiro, José (Hrsg.) ; Filipe, Joaquim (Hrsg.): *Proceedings of the* 9^{th} *International Conference on Enterprise Information Systems* Bd. AIDSS, 2007. – ISBN 9728865894, S. 212–218

[BC06] Bruno, Nicolas ; Chaudhuri, Surajit: To tune or not to tune?: a lightweight physical design alerter. In: [DWL+06], S. 499–510

[BC07] Bruno, Nicolas ; Chaudhuri, Surajit: An Online Approach to Physical Design Tuning. In: *Proceedings of the* 23^{rd} *International Conference on Data Engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2007. – ISBN 1424408032, S. 826–835

[BCL96] Brown, Kurt P. ; Carey, Michael J. ; Livny, Miron: Goal-Oriented Buffer Management Revisited. In: Jagadish, H. V. (Hrsg.) ; Mumick, Inderpal S. (Hrsg.): *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM Press, 1996. – ISBN 0897917944, S. 353–364

[BDDY09] Belknap, Peter ; Dageville, Benoît ; Dias, Karl ; Yagoub, Khaled: Self-Tuning for SQL Performance in Oracle Database 11g. In: *P*roceedings of the 25^{th} International Conference on Data Engineering[con09], S. 1694–1700

[BG04] Balke, Wolf-Tilo ; Güntzer, Ulrich: Multi-objective Query Processing for Database Systems. In: [NÖK+04], S. 936–947

[BGJ+04] Bhide, Manish ; Gupta, Ajay ; Joshi, Mukul ; Mohania, Mukesh ; Raman, Shree: Policy Framework for Autonomic Data Management. In: *P*roceedings of the 1^{st} International Conference on Autonomic Computing[con04], S. 336–337

[BJR08] Box, George E. ; Jenkins, Gwylim M. ; Reinsel, Gregory C.: *Time Series Analysis - Forecasting and Control*. 4. Heidelberg/Berlin, Germany : Springer-Verlag, 2008. – ISBN 9780470272848

[BK09] Bildhäuser, Hans-Jürgen ; Karn, Holger: *System and Method to Improve Processing Time of Databases by Cache Optimization*. 03 2009

[BLR02] Bernstein, Philip A. (Hrsg.) ; Loannidis, Yannis E. (Hrsg.) ; Ramakrishnan, Raghu (Hrsg.): *Proceedings of* 28^{th} *International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2002 . – ISBN 1558608699

[BN93] Basseville, Michèle ; Nikiforov, Igor: *Detection of abrupt changes: theory and application*. 1. Engelwood Cliffs, NJ, USA : Prentice Hall, 1993. – ISBN 0131267809

[BPL+06] Bhat, Viraj ; Parashar, Manish ; Liu, Hua ; Khandekar, Mohit ; Kandasamy, Nagarajan ; Abdelwahed, Sherif: Enabling Self-Managing Applications using Model-based Online Control Strategies. In: *Proceedings of the* 3^{rd} *IEEE International Conference on Autonomic Computing*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2006. – ISBN 1424401755, S. 15–24

[Buc09] Buckler, Andrew D.: *Workload Periodicity Analyzer for Autonomic Database Components*. 03 2009 Patent US 7,509,336 B2

[But06] Butz, Tilman: *Fourier Transformation for Pedestrians*. 1. Heidelberg/Berlin, Germany : Springer-Verlag, 2006. – ISBN 354023165X

[CCI+08] Chen, Whei-Jen ; Comeau, Bill ; Ichikawa, Tomoko ; Kumar, Sadish ; Miskimen, Marcia ; Morgan, H T. ; Pay, Larry ; Väättänenn, Tapio: *DB2 Workload Manager for Linux, UNIX, and Windows*. 1. International Business Machines Corporation, 2008. – ISBN 0738485381

[CFW+95] Chung, Jen-Yao ; Ferguson, Donald ; Wang, George ; Nikolaou, Christos ; Teng, Jim: Goal-oriented dynamic buffer pool management for data base systems. In: *Proceedings of the* 1^{st} *International Conference on Engineering of Complex Systems*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1995. – ISBN 0818671238, S. 191–198

[CGN02] Chaudhuri, Surajit ; Gupta, Ashish K. ; Narasayya, Vivek: Compressing SQL workloads. In: [FMA02], S. 488–499

[CN97] Chaudhuri, Surajit ; Narasayya, Vivek R.: An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In: Jarke, Matthias (Hrsg.) ; Carey, Michael J. (Hrsg.) ; Dittrich, Klaus R. (Hrsg.) ; Lochovsky, Frederick H. (Hrsg.) ; Loucopoulos, Pericles (Hrsg.) ; Jeusfeld, Manfred A. (Hrsg.): *Proceedings of* 23^{rd} *International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1997. – ISBN 1558604707, S. 146–155

[CN98] Chaudhuri, Surajit ; Narasayya, Vivek R.: AutoAdmin 'What-if' Index Analysis Utility. In: Haas, Laura M. (Hrsg.) ; Tiwary, Ashutosh (Hrsg.): *Proceedings ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM Press, 1998. – ISBN 0897919955, S. 367–378

[con04] *Proceedings of the* 1^{st} *International Conference on Autonomic Computing*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2004 . – ISBN 0769521142

[con07] *Proceedings of the* 23^{rd} *International Conference on Data Engineering Workshops*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2007 . – ISBN 1424408320

[con09] *Proceedings of the* 25^{th} *International Conference on Data Engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2009 . – ISBN 9780769535456

[Cor10] Corporation, Microsoft: *Memory Management Architecture*. 2010. – http://msdn.microsoft.com/en-us/library/cc280359.aspx; Online; 28.11.2010

[CT65] Cooley, James ; Tukey, John: An Algorithm for the Machine Calculation of Complex Fourier Series. In: *Mathematics of Computation* 19 (1965), Nr. 90, S. 297–301

[CT07] Chen, Yixin ; Tu, Li: Density-Based Clustering for Real-Time Stream Data. In: Berkhin, Pavel (Hrsg.) ; Caruana, Rich (Hrsg.) ; Wu, Xindong (Hrsg.): *Proceedings of the* 13^{th} *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA : ACM Press, 2007. – ISBN 1595936097, S. 133–142

[CVL07] Coello, Carlos ; Veldhuizen, David V. ; Lamont, Gary: *Evolutionary Algorithms for Solving Multi-Objective Problems*. 2. Heidelberg/Berlin, Germany : Springer-Verlag, 2007. – ISBN 0306467623

[CW06] Chaudhuri, Surajit ; Weikum, Gerhard: Foundations of Automated Database Tuning. In: Liu, Ling (Hrsg.) ; Reuter, Andreas (Hrsg.) ; Whang, Kyu-Young (Hrsg.) ; Zhang, Jianjun (Hrsg.): *Proceedings of the* 22^{nd} *International Conference on Data Engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2006. – ISBN 0769525709, S. 104

[CWC05] Chang, Wei-Chun ; Wu, Ching-Seh ; Chang, Chun: Optimizing Dynamic Web Service Component Composition by Using Evolutionary Algorithms. In: Skowron, Andrzej (Hrsg.) ; Agrawal, Rakesh (Hrsg.) ; Luck, Michael (Hrsg.) ; Yamaguchi, Takahira (Hrsg.) ; Morizet-Mahoudeaux, Pierre (Hrsg.) ; Liu, Jiming (Hrsg.) ; Zhong, Ning (Hrsg.): *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2005. – ISBN 076952415X, S. 708–711

[DD06] Dageville, Benoit ; Dias, Karl: Oracle's Self-Tuning Architecture and Solutions. In: *IEEE Data Engineering Bulletin* 29 (2006), Nr. 3, S. 24–31

[DDD+04] Dageville, Benoît ; Das, Dinesh ; Dias, Karl ; Yagoub, Khaled ; Zaït, Mohamed ; Ziauddin, Mohamed: Automatic SQL Tuning in Oracle 10g. In: [NÖK+04], S. 1098–1109

[DHP+05] Diao, Yixin ; Hellerstein, Joseph L. ; Parekh, Sujay ; Griffith, Rean ; Kaiser, Gail ; Phung, Dan: Self-Managing Systems: A Control Theory Foundation. In: *Proceedings of the* 12^{th} *International Conference on the Engineering of Computer-Based Systems*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2005. – ISBN 0769523080, S. 441–448

[Dis08] Distributed Management Task Force (Hrsg.): *Common Information Model (CIM) Infrastructure*. 2.5.0a. Portland, OR, USA: Distributed Management Task Force, 2008. – Specification

[DNL+06] Durillo, Juan J. ; Nebro, Antonio J. ; Luna, Francisco ; Dorronsoro, Bernabé ; Alba, Enrique: jMetal: A Java Framework for Developing Multi-Objective Optimization Metaheuristics / Departamento de Lenguajes y Ciencias de la Computación, University of Málaga. 2006 (ITI-2006-10). – Forschungsbericht. – Technical Report

[DPAM02] Deb, Kalyanmoy ; Pratap, Amrit ; Agarwal, Sameer ; Meyarivan, T.: A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. In: *IEEE Transactions on Evolutionary Computation* 6 (2002), Nr. 2, S. 182–197

[DRS+05] Dias, Karl ; Ramacher, Mark ; Shaft, Uri ; Venkataramani, Venkateshwaran ; Wood, Graham: Automatic Performance Diagnosis and Tuning in Oracle. In: *Proceedings of the* 2^{nd} *Biennial Conference on Innovative Data Systems Research*, Online Proceedings, 2005, S. 84–94

[DTB09] Duan, Songyun ; Thummala, Vamsidhar ; Babu, Shivnath: Tuning Database Configuration Parameters with iTuned. In: *Proceedings of the VLDB Edowment* 2 (2009), Nr. 1, S. 1246–1257

[DWL+06] Dayal, Umeshwar (Hrsg.) ; Whang, Kyu-Young (Hrsg.) ; Lomet, David B. (Hrsg.) ; Alonso, Gustavo (Hrsg.) ; Lohman, Guy M. (Hrsg.) ; Kersten, Martin L. (Hrsg.) ; Cha, Sang K. (Hrsg.) ; Kim, Young-Kuk (Hrsg.): *Proceedings of the 32nd International Conference on Very Large Data Bases*. New York, NY, USA : ACM Press, 2006 . – ISBN 1595933859

[DZ02] Dageville, Benoît ; Zaït, Mohamed: SQL Memory Management in Oracle9i. In: [BLR02], S. 962–973

[EM04] Elnaffar, Said S. ; Martin, Patrick: An Intelligent Framework for Predicting Shifts in the Workloads of Autonomic Database Management Systems. In: *Proceedings of International Conference on Advances in Intelligent Systems*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2004. – ISBN 2959977688, S. 1–8

[EMSL08] Elnaffar, Said ; Martin, Patrick ; Schiefer, Berni ; Lightstone, Sam: Is it DSS or OLTP: automatically identifying DBMS workloads. In: *Journal of Intelligent Information Systems* 30 (2008), Nr. 3, S. 249–271. – ISSN 0925–9902

[EPBM03] Elnaffar, Said ; Powley, Wendy ; Benoit, Darcy ; Martin, Patrick: Today's DBMSs: How autonomic are they? In: *Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, IEEE Computer Society Press, 2003. – ISBN 0769519938, S. 651–655

[FHWY04] Fan, Wei ; Huang, Yi an ; Wang, Haixun ; Yu, Philip S.: Active Mining of Data Streams. In: Berry, Michael W. (Hrsg.) ; Dayal, Umeshwar (Hrsg.) ; Kamath, Chandrika (Hrsg.) ; Skillicorn, David B. (Hrsg.): *Proceedings of the 4th SIAM International Conference on Data Mining*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 2004. – ISBN 0898715687, S. 457–461

[Fin99] Fink, Gernot A.: Developing HMM-Based Recognizers with ESMERALDA. In: Matousek, Václav (Hrsg.) ; Mautner, Pavel (Hrsg.) ; Ocelíková, Jana (Hrsg.) ; Sojka, Petr (Hrsg.): *Proceedings of the 2nd International Workshop on Text, Speech and Dialogue* Bd. 1692. Heidelberg/Berlin, Germany : Springer-Verlag, 1999 (Lecture Notes in Computer Science). – ISBN 3540664947, S. 229–234

[Fin08] Fink, Gernot: *Markov Models for Pattern Recognition*. 1. Heidelberg/Berlin, Germany : Springer-Verlag, 2008. – ISBN 3540717668

[FLA+03] Freytag, Johann C. (Hrsg.) ; Lockemann, Peter C. (Hrsg.) ; Abiteboul, Serge (Hrsg.) ; Carey, Michael J. (Hrsg.) ; Selinger, Patricia G. (Hrsg.) ; Heuer, Andreas (Hrsg.): *Proceedings of 29th International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2003 . – ISBN 0127224424

[FMA02] Franklin, Michael J. (Hrsg.) ; Moon, Bongki (Hrsg.) ; Ailamaki, Anastassia (Hrsg.): *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM Press, 2002 . – ISBN 1581134975

[FST88] Finkelstein, Sheldon J. ; Schkolnick, Mario ; Tiberio, Paolo: Physical Database Design for Relational Databases. In: *ACM Transactions on Database Systems* 13 (1988), Nr. 1, S. 91–128

[FTARS06] Ferrer-Troyano, Francisco J. ; Aguilar-Ruiz, Jesús S. ; Santos, José Cristóbal Riquelme: Data Streams Classification by Incremental Rule Learning with Parameterized Generalization. In: Haddad, Hisham (Hrsg.): *Proceedings of the 2006 ACM Symposium on Applied Computing*. New York, NY, USA : ACM Press, 2006. – ISBN 1595931082, S. 657–661

[GC03] Ganek, Alan G. ; Corbi, Thomas A.: The dawning of the autonomic computing era. In: *IBM Systems Journal* 42 (2003), Nr. 1, S. 5–18. – ISSN 0018–8670

[GKD+09] Ganapathi, Archana ; Kuno, Harumi A. ; Dayal, Umeshwar ; Wiener, Janet L. ; Fox, Armando ; Jordan, Michael I. ; Patterson, David A.: Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In: *P*roceedings of the 25^{th} International Conference on Data Engineering[con09], S. 592–603

[Glo89] Glover, Fred: Tabu Search–Part I. In: *Journal on Computing* 1 (1989), Nr. 3, S. 190–206

[GRCK07] Gmach, Daniel ; Rolia, Jerry ; Cherkasova, Ludmila ; Kemper, Alfons: Workload Analysis and Demand Prediction of Enterprise Data Center Applications. In: *Proceedings of the* 10^{th} *IEEE Workload Characterization Symposium*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2007. – ISBN 1424415618, S. 171–180

[HAH01] Huang, Xuedong ; Acero, Alex ; Hon, Hsiao-Wuen: *Spoken Language Processing*. 1. Upper Saddle River, NJ, USA : Prentice Hall, 2001. – ISBN 0130226165

[HEK09] Hartung, Joachim ; Elpelt, Bärbel ; Klösener, Karl-Heinz: *Statistik: Lehr- und Handbuch der angewandten Statistik*. 15. München, Germany : R. Oldenbourg Verlag, 2009. – ISBN 9783486590289

[HGR08] Holze, Marc ; Gaidies, Claas ; Ritter, Norbert: Erkennung signifikanter Laständerungen für autonome Datenbanksysteme. In: *Datenbank Spektrum* 8 (2008), Nr. 27, S. 27–36. – ISSN 1618–2162

[HGR09] Holze, Marc ; Gaidies, Claas ; Ritter, Norbert: Consistent On-Line Classification of DBS Workload Events. In: Cheung, David Wai-Lok (Hrsg.) ; Song, Il-Yeol

(Hrsg.) ; Chu, Wesley W. (Hrsg.) ; Hu, Xiaohua (Hrsg.) ; Lin, Jimmy J. (Hrsg.): *Proceedings of the* 18^{th} *International Conference on Information and Knowledge Management*. New York, NY, USA : ACM Press, 2009. – ISBN 9781605585123, S. 1641–1644

[HHR10] Holze, Marc ; Haschimi, Ali ; Ritter, Norbert: Towards workload-aware self-management: Predicting significant workload shifts. In: *Workshop Proceedings of the* 26^{th} *International Conference on Data Engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2010. – ISBN 9781424454440, S. 111–116

[HK06] Han, J. ; Kamber, M.: *Data Mining: Concepts and Techniques*. 2. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2006. – ISBN 1558609016

[Hob03] Hobbs, Lilian: *Performance Tuning using the SQLAccess Advisor*. 1. Redwood Shores, CA, USA, 2003. – White Paper

[Hor01] Horn, Paul: Autonomic Computing: IBM's Perspective on the State of Information Technology / International Business Machines Corporation. 2001. – Forschungsbericht

[HR83] Härder, T. ; Reuter, A.: Concepts for Implementing a Centralized Database Management System. In: Schneider, H.J. (Hrsg.): *Proceedings International Computing Symposium on Application Systems Development* Bd. 13. Stuttgart, Germany : B.G. Teubner, 1983 (Berichte des German Chapter of the ACM). – ISBN 3519024322, S. 28–59

[HR07] Holze, Marc ; Ritter, Norbert: Towards Workload Shift Detection and Prediction for Autonomic Databases. In: Varde, Aparna S. (Hrsg.) ; Pei, Jian (Hrsg.): *Proceedings of the First Ph.D. Workshop in the* 16^{th} *ACM Conference on Information and Knowledge Management*. New York, NY, USA : ACM Press, 2007. – ISBN 1595938329, S. 109–116

[HR08] Holze, Marc ; Ritter, Norbert: Autonomic Databases: Detection of Workload Shifts with n-Gram-Models. In: Atzeni, Paolo (Hrsg.) ; Caplinskas, Albertas (Hrsg.) ; Jaakkola, Hannu (Hrsg.): *Proceedings of the* 12^{th} *East European Conference on Advances in Databases and Information Systems* Bd. 5207. Heidelberg/Berlin, Germany : Springer-Verlag, 2008 (Lecture Notes in Computer Science). – ISBN 3540857129, S. 127–142

[HR09] Holze, Marc ; Ritter, Norbert: System Models for Goal-Driven Self-management in Autonomic Databases. In: Velásquez, Juan D. (Hrsg.) ; Ríos, Sebastián A. (Hrsg.) ; Howlett, Robert J. (Hrsg.) ; Jain, Lakhmi C. (Hrsg.):

Proceedings of the 13^{th} *International Conference on Knowledge-Based and Intelligent Information and Engineering* Bd. 5712, Springer-Verlag, 2009 (Lecture Notes in Artificial Intelligence). – ISBN 9783642045912, S. 82–90

[HR11] Holze, Marc ; Ritter, Norbert: System Models for Goal-Driven Self-management in Autonomic Databases. In: *Data & Knowledge Engineering* (2011). `http://dx.doi.org/10.1016/j.datak.2011.03.001`. – DOI 10.1016/j.datak.2011.03.001. – ISSN 0169–023X. – in press

[IBM05] IBM: *An architectural blueprint for autonomic computing.* 2005

[Int06] International Business Machines Corporation (Hrsg.): *DB2 Version 9 for Linux, UNIX, Windows - Performance Guide.* 1. Armonk, NY, USA: International Business Machines Corporation, 2006. – Manual

[Int09] International Business Machines Corporation (Hrsg.): *DB2 Version 9.5 for Linux, UNIX, and Windows - WindowsSystem Monitor Guide and Reference.* 1. Armonk, NY, USA: International Business Machines Corporation, 2009. – Manual

[IY94] Ichino, Manabu ; Yaguchi, Hiroyuki: Generalized Minkowski Metrics for Mixed Feature-Type Data Analysis. In: *IEEE Transactions on Systems, Man, and Cybernetics* 24 (1994), Nr. 4, S. 698–708. – ISSN 0018–9472

[JLHY04] Jacob, Bart ; Lanyon-Hogg, Richard ; Yassin, Devaprasad Nadgirand A. ; International Business Machines Corporation (Hrsg.): *A Practical Guide to the IBM Autonomic Computing Toolkit.* 1. Armonk, NY, USA: International Business Machines Corporation, 2004. – Redbook

[JM80] Jelinek, Frederick ; Mercer, Robert L.: Interpolated Estimation of Markov Source Parameters from Sparse Data. In: Gelsema, Edzard S. (Hrsg.) ; Kanal, Laveen N. (Hrsg.): *Proceedings of the International Workshop on Pattern Recognition in Practic.* Amsterdam, The Netherlands : North-Holland Pub. Co., 1980. – ISBN 0444861157, S. 381–397

[JMF99] Jain, A. ; Murty, M. ; Flynn, P.: Data Clustering: A Review. In: *ACM Computing Surveys* 31 (1999), Nr. 3, S. 264–323. – ISSN 0360–0300

[jTP09] jTPCC: *jTPCC - Open Source Java implementation of the TPC-C benchmark.* `http://jtpcc.sourceforge.net/`. Version: 2009. – accessed 2009-08-01

[Kar09] Karakaya, Okan: *Evaluierung von Modellierungssprachen zur Realisierung eines Systemmodells für Autonome Datenbanksysteme*, Universität Hamburg, Diplomarbeit, 2009

[Kat87] Katz, Slava: Estimation of Probablilities from Sparse Data for the Language Model Component of a Speech Recognizer. In: *IEEE Transations on Acoustics, Speech, and Signal Processing* 35 (1987), Nr. 3, S. 400–401. – ISSN 1053–587X

[KC03] Kephart, Jeffrey O. ; Chess, David M.: The Vision of Autonomic Computing. In: *IEEE Computer* 36 (2003), Nr. 1, S. 41–50

[KJ00] Klinkenberg, Ralf ; Joachims, Thorsten: Detecting Concept Drift with Support Vector Machines. In: Langley, Pat (Hrsg.): *Proceedings of the* 17^{th} *International Conference on Machine Learning*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2000. – ISBN 1558607072, S. 487–494

[KL51] Kullback, Solomon ; Leibler, Richard: On Information and Sufficiency. In: *The Annals of Mathematical Statistics* 22 (1951), Nr. 1, S. 79–86

[KLS+03] Kwan, Eva ; Lightstone, Sam ; Schiefer, K. B. ; Storm, Adam J. ; Wu, Leanne: Automatic Configuration for IBM DB2 Universal Database. In: Weikum, Gerhard (Hrsg.) ; Schöning, Harald (Hrsg.) ; Rahm, Erhard (Hrsg.): *Tagungsband der 10. Konferenz für Datenbanksysteme für Business, Technologie und Web* Bd. 26, Gesellschaft für Informatik, 2003 (Lecture Notes in Informatics). – ISBN 3885793555, S. 620–629

[KLSW02] Kwan, Eva ; Lightstone, Sam ; Storm, Adam ; Wu, Leanne: *Automatic Configuration for IBM DB2 Universal Database*. 1. Armonk, NY, USA, 01 2002. – Performance Technical Report

[Kol08] Kolaczkowski, Piotr: Compressing Very Large Database Workloads for Continuous Online Index Selection. In: Bhowmick, Sourav S. (Hrsg.) ; Küng, Josef (Hrsg.) ; Wagner, Roland (Hrsg.): *Proceedings of the* 19^{th} *International Conference on Database and Expert Systems Applications* Bd. 5181. Heidelberg/Berlin, Germany : Springer-Verlag, 2008 (Lecture Notes in Computer Science). – ISBN 3540856535, S. 791–799

[KRR02] Kossmann, Donald ; Ramsak, Frank ; Rost, Steffen: Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In: [BLR02], S. 275–286

[KSA+08] Krompass, Stefan ; Scholz, Andreas ; Albutiu, Martina-Cezara ; Kuno, Harumi A. ; Wiener, Janet L. ; Dayal, Umeshwar ; Kemper, Alfons: Quality of Service-enabled Management of Database Workloads. In: *IEEE Data Engineering Bulletin* 31 (2008), Nr. 1, S. 20–27

[KT81] Krichevsky, Raphail E. ; Trofimov, Victor K.: The Perfromance of Universal Encoding. In: *IEEE Transactions on Information Theory* 27 (1981), Nr. 2, S. 199–207

[Lab08] Laboratoies, Dell: *Dell DVD Store Database Test Suite*. `http://linux.dell.com/dvdstore/`. Version: 2008. – accessed 2008-03-03

[LB04] Lightstone, Sam ; Bhattacharjee, Bishwaranjan: Automated design of multidimensional clustering tables in relational databases. In: [NÖK+04], S. 1170–1181

[Lin91] Lin, Jianhua: Divergence Measures Based on the Shannon Entropy. In: *IEEE Transactions on Information Theory* 37 (1991), Nr. 1, S. 145–151

[LLH+06] Lightstone, Sam ; Lohman, Guy M. ; Haas, Peter J. ; Markl, Volker ; Rao, Jun ; Storm, Adam ; Surendra, Maheswaran ; Zilio, Daniel C.: Making DB2 Products Self-Managing: Strategies and Experiences. In: *IEEE Data Engineering Bulletin* 29 (2006), Nr. 3, S. 16–23

[Llo82] Lloyd, Stuart: Least Squares Quantization in PCM. In: *IEEE Transactions on Information Theory* 28 (1982), Nr. 2, S. 129–137. – ISSN 0018–9448

[LNK+03] Lahiri, Tirthankar ; Nithrkashyap, Arvind ; Kumar, Sushil ; Hirano, Brian ; Kant Pate and, Poojan K.: *The Self-Managing Database: Automatic SGA Memory Management*. 1. Redwood Shores, CA, USA, 2003. – White Paper

[LP06] Liu, Hua ; Parashar, Manish: Accord: a programming framework for autonomic applications. In: *IEEE Transactions on Systems, Man, and Cybernetics* 36 (2006), Nr. 3, S. 341–352. – ISSN 0018–9472

[LS08] Leeuwen, Matthijs van ; Siebes, Arno: StreamKrimp: Detecting Change in Data Streams. In: Daelemans, Walter (Hrsg.) ; Goethals, Bart (Hrsg.) ; Morik, Katharina (Hrsg.): *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases, Part I* Bd. 5211. Heidelberg/Berlin, Germany : Springer-Verlag, 2008 (Lecture Notes in Computer Science). – ISBN 9783540874782, S. 672–687

[LSSS07] Lühring, Martin ; Sattler, Kai-Uwe ; Schmidt, Karsten ; Schallehn, Eike: Autonomous Management of Soft Indexes. In: *P*roceedings of the 23^{rd} International Conference on Data Engineering Workshops[con07], S. 450–458

[Mac67] MacQueen, James: Some Methods for Classification and Analysis of Multivariate Observations. In: Cam, Lucien M. L. (Hrsg.) ; Neyman, Jerzey (Hrsg.): *Proceedings of the* 5^{th} *Berkeley Symposium Mathematical Statistics and Probability* Bd. 1. Berkley/Los Angeles, CA, USA : University of Claifornia Press, 1967, S. 281–297

[MEW06] Martin, Patrick ; Elnaffar, Said ; Wasserman, Ted: Workload Models for Autonomic Database Management Systems. In: Dini, Petre (Hrsg.) ; Ayed, Dhouha (Hrsg.) ; Dini, Cosmin (Hrsg.) ; Berbers, Yolande (Hrsg.): *Proceedings of the*

International Conference on Autonomic and Autonomous Systems. Los Alamitos, CA, USA : IEEE Computer Society Press, 2006. – ISBN 0769526535, S. 10

[MLR03] Markl, Volker ; Lohman, Guy M. ; Raman, Vijayshankar: LEO: An autonomic query optimizer for DB2. In: *IBM Systems Journal* 42 (2003), Nr. 1, S. 98–106

[MM00] Maloof, Marcus A. ; Michalski, Ryszard S.: Selecting Examples for Partial Memory Learning. In: *Maschine Learning* 41 (2000), Nr. 1, S. 27–52

[MRHA08] Mateen, Abdul ; Raza, Basit ; Hussain, Tauqeer ; Awais, Mian M.: Autonomic Computing in SQL Server. In: Lee, Roger Y. (Hrsg.): *Proceedings of the* 7^{th} *IEEE/ACIS International Conference on Computer and Information Science*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2008. – ISBN 9780769531311, S. 113–118

[MRHA09] Mateen, Abdul ; Raza, Basit ; Hussain, Tauqeer ; Awais, Mian M.: Autonomicity in Universal Database DB2. In: Miao, Huaikou (Hrsg.) ; Hu, Gongzhu (Hrsg.): *Proceedings of the* 8^{th} *IEEE/ACIS International Conference on Computer and Information Science*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2009. – ISBN 9780769536415, S. 445–450

[MW47] Mann, Henry B. ; Whitney, Donald R.: Individual Comparisons by Ranking Methods. In: *The Annals of Mathematical Statistics* 18 (1947), Nr. 1, S. 50–60

[NMP+06] Niu, Baoning ; Martin, Patrick ; Powley, Wendy ; Horman, Randy ; Bird, Paul: Workload adaptation in autonomic DBMSs. In: Erdogmus, Hakan (Hrsg.) ; Stroulia, Eleni (Hrsg.) ; Stewart, Darlene A. (Hrsg.): *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative Research*. Indianapolis, IN, USA : IBM Press, 2006, S. 13

[NÖK+04] Nascimento, Mario A. (Hrsg.) ; Özsu, M. T. (Hrsg.) ; Kossmann, Donald (Hrsg.) ; Miller, Renée J. (Hrsg.) ; Blakeley, José A. (Hrsg.) ; Schiefer, K. B. (Hrsg.): *Proceedings of the* 30^{th} *International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2004 . – ISBN 0120884690

[NTA05] Narayanan, Dushyanth ; Thereska, Eno ; Ailamaki, Anastassia: Continuous Resource Monitoring for Self-Predicting DBMS. In: *Proceedings of the* 13^{th} *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2005. – ISBN 0769524583, S. 239–248

[NTA06] Narayanan, Dushyanth ; Thereska, Eno ; Ailamaki, Anastassia: Challenges in building a DBMS Resource Advisor. In: *IEEE Data Engineering Bulletin* 29 (2006), Nr. 3, S. 40–46

[NY07] Nishida, Kyosuke ; Yamauchi, Koichiro: Detecting Concept Drift Using Statistical Testing. In: Corruble, Vincent (Hrsg.) ; Takeda, Masayuki (Hrsg.) ; Suzuki, Einoshin (Hrsg.): *Proceedings of the* 10^{th} *International Conference on Discovery Science* Bd. 4755. Heidelberg/Berlin, Germany : Springer-Verlag, 2007 (Lecture Notes in Artificial Intelligence). – ISBN 9783540754879, S. 264–269

[Obj08] Object Management Group (Hrsg.): *Systems Modeling Language*. 1.1. Needham, MA, USA: Object Management Group, 2008. – Specification

[OMM+02] O'Callaghan, Liadan ; Meyerson, Adam ; Motwani, Rajeev ; Mishra, Nina ; Guha, Sudipto: Streaming-Data Algorithms for High-Quality Clustering. In: Agrawal, Rakesh (Hrsg.) ; Dittrich, Klaus (Hrsg.) ; Ngu, Anne H. (Hrsg.): *Proceedings of the* 18^{th} *International Conference on Data Engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2002. – ISBN 0769515312, S. 685–694

[OO08] Ordonez, Carlos ; Omiecinski, Edward: FREM: Fast and Robust EM Clustering for Large Data Sets. In: Nicholas, Charles (Hrsg.) ; Grossman, David (Hrsg.) ; Kalpakis, Konstantinos (Hrsg.) ; Qureshi, Sajda (Hrsg.) ; Dissel, Han van (Hrsg.) ; Seligman, Len (Hrsg.): *Proceedings of the* 11^{th} *International Conference on Information and Knowledge Management*. New York, NY, USA : ACM Press, 2008. – ISBN 1581134924, S. 590–599

[PN93] Polana, Ramprasad ; Nelson, Randal: Detecting Activities. In: *Proceedings of the International Conference on Computer Vision and Pattern Recognition*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1993. – ISBN 081863880X, S. 2–7

[Pog09] Poggensee, Sven: *Systemmodelle zur Bestimmung optimaler Konfigurationen für Autonome DBS mittels evolutionärer Algorithmen*, Universität Hamburg, Diplomarbeit, 2009

[PRD+04] Parekh, Sujay ; Rose, Kevin ; Diao, Yixin ; Chang, Victor ; Hellerstein, Joseph ; Lightstone, Sam ; Huras, Matthew: Throttling Utilities in the IBM DB2 Universal Database Server. In: *Proceedings of the 2004 American Control Conference* Bd. 3. Los Alamitos, CA, USA : IEEE Computer Society Press, 2004. – ISBN 0780383354, S. 1986–1991

[Pro10] Project, The Apache J.: *Apache JMeter*. `http://jakarta.apache.org/jmeter/`. Version: 2010. – accessed 2010-08-10

[Rab09] Rabinovitch, Gennadi: Policy-Based Coordination of Best-Practice Oriented Autonomic Database Tuning. In: Dini, Petre (Hrsg.) ; Gentzsch, Wolfgang (Hrsg.) ; Geraci, Paul (Hrsg.) ; Lorenz, Pascal (Hrsg.) ; Singh, Krishna (Hrsg.): *Proceedings of the Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2009. – ISBN 9781424451661, S. 55 –60

[Rei09] Reinhardt, Stephan: *Entwicklung und Untersuchung verschiedener Methoden zur Erkennung von Laständerungen in Datenbanksystemen*, Universität Hamburg, Diplomarbeit, 2009

[RZML02] Rao, Jun ; Zhang, Chun ; Megiddo, Nimrod ; Lohman, Guy: Automating physical database design in a parallel database. In: [FMA02], S. 558–569

[SAMP07] Schnaitter, Karl ; Abiteboul, Serge ; Milo, Tova ; Polyzotis, Neoklis: On-Line Index Selection for Shifting Workloads. In: *P*roceedings of the 23^{rd} International Conference on Data Engineering Workshops[con07], S. 459–468

[Sch09] Schmidt, Karsten: Goal-Driven Autonomous Database Tuning Supported by a System Model. In: *Proceedings of the SIGMOD Workshop on Innovative Database Research*, 2009, S. 708–711

[SD08] Sivanandam, S. ; Deepa, S.: *Introduction to Genetic Algorithms*. 1. New York, NY, USA : Springer US, 2008. – ISBN 9783540731894

[SG07] Sebastião, Raquel ; Gama, João: Change Detection in Learning Histograms from Data Streams. In: Neves, José (Hrsg.) ; Santos, Manuel F. (Hrsg.) ; Machado, José (Hrsg.): *Workshop Proceedings of the* 13^{th} *Portuguese Conference on Aritficial Intelligence* Bd. 4874. Heidelberg/Berlin, Germany : Springer-Verlag, 2007 (Lecture Notes in Computer Science). – ISBN 9783540770008, S. 112–123

[SGAL+06] Storm, Adam J. ; Garcia-Arellano, Christian ; Lightstone, Sam S. ; Diao, Yixin ; Surendra, M.: Adaptive Self-Tuning Memory in DB2. In: [DWL+06], S. 1081–1092

[SGS03] Sattler, Kai-Uwe ; Geist, Ingolf ; Schallehn, Eike: QUIET: Continuous Query-driven Index Tuning. In: [FLA+03], S. 1129–1132

[SLMK01] Stillger, Michael ; Lohman, Guy M. ; Markl, Volker ; Kandil, Mokhtar: LEO - DB2's Learning Optimizer. In: Apers, Peter M. G. (Hrsg.) ; Atzeni, Paolo (Hrsg.) ; Ceri, Stefano (Hrsg.) ; Paraboschi, Stefano (Hrsg.) ; Ramamohanarao, Kotagiri (Hrsg.) ; Snodgrass, Richard T. (Hrsg.): *Proceedings of* 27^{th} *International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001. – ISBN 1558608044, S. 19–28

[SS00] Shumway, Robert H. ; Stoffer, David S.: *Time Series Analysis and Its Applications*. 1. New York, NY, USA : Springer-Verlag, 2000. – ISBN 0378989501

[SSG04] Sattler, Kai-Uwe ; Schallehn, Eike ; Geist, Ingolf: Autonomous Query-driven Index Tuning. In: *Proceedings of the* 8^{th} *International Database Engineering and Applications Symposium*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2004. – ISBN 0769521681, S. 439–448

[TBA10] Tozer, Sean ; Brecht, Tim ; Aboulnaga, Ashraf: Q-Cop: Avoiding Bad Query Mixes to Minimize Client Timeouts under Heavy Loads. In: Li, Feifei (Hrsg.) ; Moro, Mirella M. (Hrsg.) ; Ghandeharizadeh, Shahram (Hrsg.) ; Haritsa, Jayant R. (Hrsg.) ; Weikum, Gerhard (Hrsg.) ; Carey, Michael J. (Hrsg.) ; Casati, Fabio (Hrsg.) ; Chang, Edward Y. (Hrsg.) ; Manolescu, Ioana (Hrsg.) ; Mehrotra, Sharad (Hrsg.) ; Dayal, Umeshwar (Hrsg.) ; Tsotras, Vassilis J. (Hrsg.): *Proceedings of the* 26^{th} *International Conference on Data Engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2010. – ISBN 9781424454440, S. 397–408

[THTT08] Tran, Dinh N. ; Huynh, Phung C. ; Tay, Yong C. ; Tung, Anthony K. H.: A new approach to dynamic self-tuning of database buffers. In: *ACM Transactions on Storage* 4 (2008), Nr. 1, S. 1–25. – ISSN 1553–3077

[Tra02] Transaction Processing Performance Council (Hrsg.): *TPC BENCHMARK W (Web Commerce) Specification*. 1.8. San Jose, CA, USA: Transaction Processing Performance Council, 2002. – Specification

[Tra07] Transaction Processing Performance Council (Hrsg.): *TPC BENCHMARK C Standard Specification*. 5.9. San Jose, CA, USA: Transaction Processing Performance Council, 2007. – Specification

[Tra08] Transaction Processing Performance Council (Hrsg.): *TPC BENCHMARK H (Decision Support) Standard Specification*. 2.8.0. San Jose, CA, USA: Transaction Processing Performance Council, 2008. – Specification

[Tri04] Triola, Mario F.: *Elementary Statistics*. 1. USA : Pearson Education, 2004. – ISBN 0321181964

[Uni09] University of Wisconsin: *TPC-W Java Implementation*. `http://tpcw.deadpixel.de/`. Version: 2009. – accessed 2009-08-01

[VZZ+00] Valentin, Gary ; Zuliani, Michael ; Zilio, Daniel C. ; Lohman, Guy M. ; Skelley, Alan: DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In: *Proceedings of the* 16^{th} *International Conference on Data*

Engineering. Los Alamitos, CA, USA : IEEE Computer Society Press, 2000, S. 101–110

[WCSO08] Wada, Hiroshi ; Champrasert, Paskorn ; Suzuki, Junichi ; Oba, Katsuya: Multiobjective Optimization of SLA-aware Service Composition. In: *Proceedings of the 2008 IEEE Congress on Services - Part I*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2008. – ISBN 0769532868, S. 368–375

[Wei08] Weilkiens, Tim: *Systems Engineering with SysML/UML*. 1. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008. – ISBN 0123742749

[Wei09] Weiss, Christian H.: *Categorical Time Series Analysis and Applications in Statistical Quality Control*. 1. Berlin, Germany : dissertation.de – Verlag im Internet GmbH, 2009. – ISBN 9783866244429

[WHYZnt] Wu, Yongwei ; Hwang, Kai ; Yuan, Yulai ; Zheng, Weiming: Adaptive Workload Prediction of Grid Performance in Confidence Windows. In: *IEEE Transactions on Parallel and Distributed Systems* (preprint). – ISSN 1045–9219

[Wil45] Wilcoxon, Frank: Individual Comparisons by Ranking Methods. In: *Biometrics Bulletin* 1 (1945), Nr. 6, S. 80–83

[WMHZ02] Weikum, Gerhard ; Mönkeberg, Axel ; Hasse, Christof ; Zabback, Peter: Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In: [BLR02], S. 20–31

[WW40] Wald, Abraham ; Wolfowitz, Jacob: On a Test Whether Two Samples are from the Same Population. In: *The Annals of Mathematical Statistics* 11 (1940), Nr. 2, S. 147–162

[YAH05] Yao, Qingsong ; An, Aijun ; Huang, Xiangji: Finding and Analyzing Database User Sessions. In: Zhou, Lizhu (Hrsg.) ; Ooi, Beng C. (Hrsg.) ; Meng, Xiaofeng (Hrsg.): *Proceedings of the* 10^{th} *International Conference on Database Systems for Advanced Applications* Bd. 3453. Heidelberg/Berlin, Germany : Springer-Verlag, 2005 (Lecture Notes in Computer Science). – ISBN 3540253343, S. 851–862

[YCHL92] Yu, P.S. ; Chen, M.-S. ; Heiss, H.-U. ; Lee, S.: On Workload Characterization of Relational Database Environments. In: *IEEE Transactions on Software Engineering* 18 (1992), Nr. 4, S. 347–355. – ISSN 0098–5589

[YWZ06] Yang, Ying ; Wu, Xindong ; Zhu, Xingquan: Mining in Anticipation for Concept Change: Proactive-Reactive Prediction in Data Streams. In: *Data Mining and Knowledge Discovery* 13 (2006), Nr. 3, S. 261–289

[ZRL+04] Zilio, Daniel C. ; Rao, Jun ; Lightstone, Sam ; Lohman, Guy M. ; Storm, Adam ; Garcia-Arellano, Christian ; Fadden, Scott: DB2 Design Advisor: Integrated Automatic Physical Database Design. In: [NÖK+04], S. 1087–1097

[ZZL+04] Zilio, Daniel C. ; Zuzarte, Calisto ; Lightstone, Sam ; Ma, Wenbin ; Lohman, Guy M. ; Cochrane, Roberta J. ; Pirahesh, Hamid ; Colby, Latha ; Gryz, Jarek ; Alton, Eric ; Liang, Dongming ; Valentin, Gary: Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In: *P*roceedings of the 1^{st} International Conference on Autonomic Computing[con04], S. 180–188