Self-Tuning Storage and Indexing for Native XML DBMSs

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Diplom-Informatiker Karsten Schmidt

Dekan des Fachbereichs Informatik:
Prof. Dr. Arnd Poetzsch-Heffter

Promotionskommission:

Vorsitzender:	Prof. Dr. Jens Schmitt
Berichterstatter:	Prof. Dr. Dr. Theo Härder
	Prof. Dr. Kai-Uwe Sattler

Datum der wissenschaftlichen Aussprache:
30. September 2011

D 386

For Claudia & Vincent
The center of my life.

Acknowledgements

A project like this thesis would not have been possible without the support of many people, and it is my pleasure to thank them.

First, and foremost, I would like to express my sincerest gratitude towards my advisor, Prof. Dr. Dr. Theo Härder. His broad knowledge and his continuous support have been of great value for me. His understanding, encouraging and personal guidance have provided an excellent basis for the present thesis. I would like to thank Prof. Dr. Kai-Uwe Sattler for his detailed review, constructive criticism and excellent advice during the preparation of this thesis and for accepting the role of the second examiner. I thank Prof. Dr. Jens Schmitt for taking the role of committee chair and providing a pleasant atmosphere during my defense.

Many students supported my research during the last years and I want to take the chance to thank them for their hard work and the fun we had. I want to mention Sebastian Potthoff, Felix Kling, Muhammad Mainul Hossain, Caetano Sauer, Martin Corrini and Ludger Overbeck.

Further, I would like to thank all the colleagues who gave me an excellent time in Kaiserslautern and made me feel right at home. Dr. Christian Mathis, for introducing me to the XTC project and for his continuous and invaluable support for my own research. Dr. Philipp Dopichaj, Volker Höfner, Daniel Schall and Dr. Andreas Weiner for their friendship and for proofreading parts of this thesis. In particular, I'm grateful to Dr. Jürgen Göres for his incredible level of proofreading and for the many fruitful discussions in the morning. I also want to thank all the other *Teeecke* members, Thomas Jörg, Joachim Klein, Nikolas Nehmer, Dr. Boris Stumm, Andreas Bühmann and Yi Ou that also contributed to making my stay in Kaiserslautern such a great time. Very special thanks go to my colleague and close friend, Sebastian Bächle, for our daily discussions on research and non-research topics, for sharing his programming wisdom and for enriching my ideas.

Most importantly, none of this would have been possible without the love and patience of my family. I am very grateful to my parents, Klaus and Annelie, for their support and encouragement. I wish to thank my entire extended family for providing a loving environment for me. My brother, grandparents and uncles were particularly supportive.

Last but not least, I owe my deepest gratitude to my wife Claudia, who means everything to me. She has been with me through all the happy and tough times, and this thesis wouldn't have been possible without her invaluable companionship. She always showed patience with me and full understanding, even when I was working late or weekends to meet the next paper deadline. Finally, I want to thank my son Vincent who kept me laughing when the laughs were running low and who reminds me that, all the evidence to the contrary, there is a life outside of computer science and academia.

Karsten Schmidt

Alzey, October 2011

Abstract

Today, database management systems (DBMSs) are ubiquitous and they often build the backbone of highly complex IT systems. Their flexibility, scalability, and configurability make them useful for a wide range of applications. From handheld devices to worldwide online systems, DBMSs operate reliably and are available 24/7. However, changing conditions such as (periodic) workload shifts, growing data volumes, increasing load, or the deployment of new applications may cause performance bottlenecks. To meet given performance goals such as transaction throughput or response time, configuration changes become inevitable.

A DBMS provides many tuning knobs that, even for human experts, it is challenging to quickly find a better, let alone the best, configuration. Here, self-tuning is capable of performing the necessary tasks just in time. A typical approach for self-tuning is to employ a cycle of system *monitoring*, performance *analysis*, and configuration *adjustments*. Moreover, simulation-based approaches are used to estimate the effects of configuration changes.

So far, research focused on self-tuning techniques for relational DBMSs and missed to face the emerging challenges imposed by the increasing demand of processing XML data natively with DBMSs. In this work, we investigate self-tuning techniques for native XML DBMSs (XDBMSs) in the context of our prototype "XTC – XML Transaction Coordinator".

In the course of this work, we present and benchmark our self-tuning concepts for XML storage, indexing, and processing. Therefore, we developed a full-fledged *monitoring* framework embedded into XTC. The typical, layered architecture of a DBMS guided our way of exploring tailored self-tuning techniques.

We developed algorithmic extensions for *database buffer* self-tuning that provide nearly accurate performance forecasts. Based on our *hotset* simulation, we improved the decision support for buffer pool resizing. Other techniques to optimize the IO load are related to XML storage. We present our novel, space-efficient, and high-performance *elementless* storage mapping for XML data. Our associated *path synopsis* structure is a concise structural summary that plays a central role in storage and query processing throughout this work. Structural and content *compression* techniques as well as *similarity* measures for XML are exploited to (self-) tune the storage configuration. Based on our elementless storage concept, we built a unique and flexible set of XML *indexing options*. Besides XML "standard" indexing, we provide tailoring for content and structural aspects as well as clustering, compression, and document coverage. On top of our storage and index layers, we developed a query-driven *Autonomous Indexing* (AI) framework. It is integrated into XTC and exploits so-called *virtual indexes* and the query optimizer. Indexes are automatically created, maintained, and dropped, while all self-tuning decisions are supported by proper statistics about queries and their optimization potential. Comprehensive experiments and analyses regarding performance, overhead, and interplay of self-tuning techniques are performed based on real-world and artificial datasets.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Database tuning has always been a fundamental aspect of database research and development: Many reasons may require changing a (database) system configuration. Either the workload (e.g., type of queries), the data (e.g., volume, schema, value distribution), or the system load (e.g., number of – possibly concurrent – users, hardware resource utilization) change. If the DBMS with its current hardware equipment and configuration is not able to silently cope with these changes, they will inevitably begin to (negatively) impact application performance, i.e., response times, and ultimately the throughput will degrade.

A common solution is the "KIWI" (Kill It With Iron) approach, i.e., to simply invest in more powerful hardware without actually addressing a specific component. However, while KIWI might appear straightforward and relatively cheap at first, given today's hardware prices, it has its limits. It totally ignores the (cheaper) option to reconfigure the system. Smarter alternatives try to identify the system bottleneck first, before undertaking tailored actions that are feasible using the current hardware, i.e., reconfigure a system.

Due to the diversity of database applications, DBMS vendors no longer change program code for individual customers (i.e., applications), but provide a wide array of configuration options to allow the customization of their product to the specific requirements of the database application during deployment. At the same time, the responsibility for application-dependent configurations was shifted from the developer to the customer (i.e., DB administrator). Current commercial DBMSs provide several hundreds of parameters, where finding a suitable (let alone the optimal) setup can occupy even experienced DB administrators for days [KLS+03].

Nowadays, IT systems need to be available 24/7 while providing fast reaction and response times. Although there are various kinds of IT systems ranging from in-house to worldwide connected online systems, they have something fundamental in common – the backbone of most IT systems are database management systems that are capable of reliably handling up to thousands of concurrent transactions at the same time. This also means that new applications are deployed into an existing IT landscape while the remaining infrastructure, database, and applications need to stay online. To prevent any downtimes or performance bottlenecks of already running systems, configuration adjustments need to be performed immediately when changes happen, i.e., the deployment of additional database applications. Obviously, a manual tweaking of the system by an administrator will take too long to prevent such situations. Consequently, only autonomous tuning mechanisms, built directly into the DBMS itself or close to it, can timely react to suddenly changing loads or requirements.

Automated tuning has been extensively studied by various communities, especially for computer and software systems; however, given the huge amount of parameters and their often

unpredictable interactions, the spectrum of automation is fairly wide. Often, it is quite difficult in practice to find a good balance between expected (positive) effects and limited loss in terms of uncontrolled or unwanted behavior. Also, the overhead of automation, both for system observation and reconfiguration, has to be taken into account. In complex environments and under varying (or unpredictable) workloads, pre-defined rules or more sophisticated knowledge bases and cost models are necessary to alleviate the search for a better or even optimal configuration.

Besides hardware and software changes, the data model and, thereby, the workload may change, too. Today's databases do not only deal with relational data anymore, as they did for more than 40 years. The most prominent data model evolution of recent years is probably XML – the *Extensible Markup Language*. Apart from the XML data and processing model, also query language(s) and, for instance, indexing are totally different from a conceptional point of view [FMM+07]. Still, this does not mean that DBMS developers need to discard everything they have learned – actually the opposite is true, it is wise to reuse existing and well-studied and understood techniques and concepts as far as possible. For instance, traditional DBMS development established the *layered architecture* – a hierarchical layout with isolated functionalities in each layer that are connected via well-defined interfaces – which has proven to be efficient for many reasons. For the purpose of self-tuning, the *separation of concerns*, supported by this architecture, forms "natural" groups of functionality that are the basis for identifying so-called *self-tuning* features. Many of the existing self-tuning techniques focus on a single component, i.e., layer or functionality in a DBMS. Such a local (i.e., component-wise) optimization is often possible by applying self-tuning, whereas a holistic (e.g., model-driven) optimization approach for a DBMS is still challenging.

Tuning a system can be done either online while the system is serving requests or offline during (scheduled) downtimes. In both scenarios, the evaluation of tuning measures is a crucial aspect. That means, tuning steps have to be evaluated to allow for comparing different tuning alternatives, learning the best reaction to different situations, predicting the outcome of tuning measures, and the validation of a tuning approach and its result. Adequate abstractions for resources and workloads as well as processing are required to attribute costs and benefits to certain configurations. To allow for evaluation, a cost model is essential. Certain issues like deferred impact or changing conditions while tuning make a sound evaluation difficult or even impossible. Moreover, they might reduce the significance of the results. In any case, self-tuning actions should ensure that system performance is not impaired, which is a tough demand.

The vision of self-tuning is simple: The user (or administrator) does not notice that tuning is taking place and ultimately might not even be aware of the internal tuning knobs anymore that are used by the system [WMHZ02]. Instead, the user can use more (abstract) high-level goals such as costs (i.e., dollars), energy consumption (i.e., Green IT), response times, throughput rates, or hardware and time budgets. Moreover, the system can recommend changes it is not able to perform by itself such as "add hardware" or "replace malfunction hardware" to fulfill those goals. Because the fundamental concept of self-tuning, as the prefix "self" indicates, is recursively applicable, the tuning components may control themselves in the future, i.e., self-tune the self-tuner. But this is still a vision. Eventually, those open research topics for self-tuning (database) systems may guide the way to gradually develop all the necessary techniques.

1.1 Tuning Principles

Left untouched, a DBMS's out-of-the-box configuration will rarely allow it to perform at its optimal performance. Consequently, in general, adjustments are necessary to adapt the system to specific workloads.

To guide these adjustments, initial system conditions have to be gathered and analyzed before tuning takes place to get a clear picture of current performance issues. This task is either done by requesting state information from the outside or by pushing *monitoring* information to the outside. On the one hand, actively requesting information may cause unnecessary interruptions and yield zero information when changes did not take place. On the other hand, pushing monitoring information to a client application may cause significant runtime overhead that can even skew the results. Therefore, *monitoring* has also to be tuned to strike a balance between obtaining monitoring data with a sufficient level of detail to draw the proper conclusions and avoiding a detrimental impact on the system when obtaining this data.

Knowing the state of a system allows analyzing it component-wise for individual parameters or with so-called *metrics* indicating certain performance characteristics, but not necessarily for an individual component. Different methods are common to *analyze* this information, as we will see in Section 3.2. Again, a human *expert* may perform the analysis or the system it*self*.

Adjusting the system configuration needs confidence that the changes' impact will result in a performance gain. Therefore, some preparation is necessary such as *simulating* the changes or *estimating* the impact. A similar approach to estimating is *predicting* the effects of a tuning measure. Knowing the *overhead* caused by the efforts for the state transition (i.e., reconfiguration) beforehand ensures that the expected gain is justifying the overhead. A new monitoring cycle may bring the evidence that the adjustments meet the expectations. However, the effects of a tuning measure may not be visible immediately, making their observation cumbersome.

Performing multiple tuning measures simultaneously complicates the identification and correct attribution of their effects, because they overlap or outweigh each other. A system *model* or *knowledge base* of tuning measures can be used to address those problems as we will see throughout this chapter.

Deciding about the optimal timing when the reconfiguration should take place is another tuning issue. On the one hand, the situation that motivated the tuning measure will require the change to be performed as soon as possible, as any delay will cause the current performance bottleneck to persist and decisions about the tuning measure to become outdated. On the other hand, a verification of the tuning measure, e.g., by simulating its effects beforehand to not jeopardize a production/mission critical system, might be advisable. If regular maintenance windows or times of low system load exist, enacting the tuning measure could also be deferred to affect the system's operation as little as possible. Similar to the question whether to do it immediately or during a maintenance period, is the problem of applying changes *online* or *offline*. Bringing a system down and doing all the necessary reconfigurations offline seems to be save, but causes comparatively long outages that are often not acceptable for production systems, where 24/7 availability is required. Doing it online, however, often requires more preparation, a cautious approach, and increases the chance of being faced with unforeseen problems. Online reconfiguration is further complicated by the fact that not all parameters

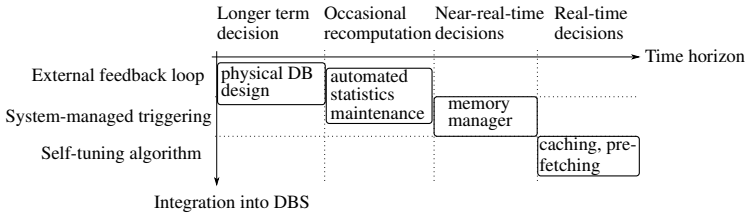


Figure 1.1: The spectrum of self-tuning [CW05]

or tuning knobs of a typical DBMS can be changed while the system is running, limiting the scope of possible changes for online tuning.

Environments, workloads, and requirements can change over time, which makes system configurations outdated and advocates tuning measures. Therefore, the cycle of *monitoring*, *analyzing*, and *acting* has to be done throughout the entire lifetime of a system.

1.2 Motivation

Nowadays, the deployment of database systems is ubiquitous. Embedded into or linked with complex systems, they are faced with varying application areas and workloads. A fundamental requirement to continuously provide high-performance data processing is adaptivity and scalability of a DBMS. But the manifold tuning options are manually unmanageable, which advocates automated support in form of self-tuning.

Because systems tend to be extremely complex, an holistic approach for self-tuning is still impracticable. For this reason, self-tuning targets at self-contained components, which however, may be connected to each other. Hence, the goal is to adjust each of them towards the optimal configuration, which is often not feasible in a reasonable time or at justifiable costs. To make self-tuning viable, it is enough to at least converge to the optimum.

For each component, a self-tuning technique has to be chosen, which may further dictate its level of DBMS integration and its decision's impact in terms of durability. For a selection of potential (self-)tuning areas, Figure 1.1 shows a possible classification for self-tuning. The authors of [CW05] argue that the more tightly coupled a (self-)tuning technique is employed, the more frequently the tuning decision can be refined. Considering the impact of certain tuning decisions, it would be desirable to always increase the integration, e.g., "move the physical DB design down", and to increase the recomputation frequency, i.e., move everything to the right.

Other important classifications comprise the online capabilities of a tuning measure, the potential of self-learning, or the field of application.

Self-tuning a DBMS component, as shown in the figure, implies tailored techniques and decision models. The majority of existing techniques is tailored to the domain of relational data management. The growing need to natively process XML data leads to new challenges and motivates our work to develop new techniques for XML-specific self-tuning. Of course, the lessons learned should be retained, such as data model independent tuning or well-known

tuning procedures. The research question is, whether or not a native XDBMS can be improved by a similar set of self-tuning features as relational systems.

Some of the most successful self-tuning techniques are based on the concept of combining *Monitoring*, *Decision*, and *Action*, which will be presented in more detail in Chapter 3. Basically, system performance is monitored, according to a tuning goal configuration changes are planned, and finally executed. This procedure is repeated while the system is running. Other successful techniques exploit simulations (often called "what-if" analysis) to identify alternative configurations providing better performance.

In this work, we apply those techniques for various components of a native XDBMS and integrate them allowing self-tuning decisions to be made online while avoiding administrative overhead and providing fast reactions to changing workloads and requirements.

1.2.1 Objectives

Tuning of database management systems (DBMSs) is a complex task that requires expertise in several computer science disciplines: One needs to have a thorough understanding of the hardware (e.g., IO or memory bandwidth) and system software (e.g., operating system) and their architectural implications. A deep understanding of the working principles of a DBMS is required, you (i.e., the administrators) have to know the (side-)effects when changing parameters. Furthermore, based on monitoring information and observations, one must be able to draw conclusions regarding the actual cause of the currently visible performance. A successful self-tuning mechanism for (X)DMBSs should mimic this approach.

The goal of this thesis is now to transfer these basic principles underlying (any kind of) manual DBMS tuning to autonomous self-tuning mechanisms tailored for native XML DBMSs. However, given the complexity and scope of this challenge even when targeting a specific data model, a single thesis cannot cover all aspects. Guided by a typically layered architecture of DBMSs, we therefore focus on the most essential aspects:

- *Analyze existing self-tuning techniques.* The range of possible self-tuning approaches is from theoretically formalized to brute-force trial and error. Depending on the DBMS itself, the tuning target, and access to DBMS internals, only certain tuning approaches are beneficial for performance-oriented self-tuning. Especially for the domain of *feedback-control-loop* techniques for online self-tuning, we require sophisticated *monitoring* and *analyzing* capabilities, as well as effective *tuning knobs* to adjust a DBMS.

- *Improve existing self-tuning approaches for buffer management.* Simulation and prediction are typically used to optimize buffer configurations, but often deliver bad accuracy in buffer performance forecasting. We develop lightweight algorithmic extensions to provide high-accurate performance forecasts for buffer configurations.

- *Develop XML-specific tuning options for XML storage and indexing.* XML query processing relies on fast and space-efficient XML document access. Therefore, we develop novel techniques to efficiently store/retrieve XML data and flexible index mechanisms

that are built on the same concepts. Due to possible storage space restrictions and maintenance overhead, the selection of document storage parameters is as crucial as the selection of indexes.

- *Integrate self-tuning techniques into a native XDBMS.* Fortunately, our system XTC allows to evaluate all ideas developed in this work. This requires a system-wide monitoring and event processing concepts, while the self-tuning logic is subject to the individual components for XML-specific *storage* and *indexing* as well as *buffer* management.

- *Show the effectiveness of tuning knobs, XML-aware storage and indexing techniques, and self-tuning effects.* For all techniques developed in this work, we assess their effectiveness by individual benchmarks using typical workloads. Especially, performance, space consumption, and the overhead of our tuning management are measured.

The domain of self-tuning comprises a variety of aspects such as system optimization, health, protection, and configuration. In this work, we consider the aspect of performance self-tuning, which incorporates the DBMS configuration and its optimization.

1.2.2 Overview

This dissertation is structured into three parts. The first part consists of two chapters: Chapter 1, which you are currently reading, contains the introduction and motivation for XML-specific DBMS self-tuning, and Chapter 2 presents fundamental concepts necessary for all following chapters. Especially, the aspects of *native* XDBMSs are emphasized in Section 2.4 and our prototype *XTC* is introduced in Section 2.5.

The second part of this work starts with Chapter 3, which first presents self-tuning approaches, followed by related work and our own XTC-specific self-tuning framework in Section 3.6. The layout of the main chapters in the second part is aligned to the layered architecture of a DBMS. In Chapter 4, non-XML-specific buffer self-tuning techniques are presented and evaluated. Based on that, Chapter 5 highlights and evaluates all the XML-specific storage self-tuning approaches, we implemented into XTC. Chapter 6 relies on those storage concepts and covers XML indexing options in Section 6.2 as well as query processing in Section 6.3. The second part concludes with Chapter 7, which presents and evaluates our autonomous indexing framework. Related work is always discussed within the chapters.

The last part consists of two chapters. In Chapter 8, a comprehensive self-tuning sample illustrates the interplay of major self-tuning techniques in case of changing workloads. Finally, the dissertation concludes with a summary and a discussion of future research aspects in Chapter 9.

Chapter 2

Fundamentals

This chapter introduces the fundamental aspects of DBMSs in general and native XML DBMSs in particular, partially exemplified with the help of our own prototype DBMS – XTC. We will start with a brief historical digest of major developments in DBMS research that led to the current requirements for DBMS tuning. We will then give an architectural overview on DBMSs and discuss the properties that distinguish a *native* XML DBMS. These properties help to understand the different requirements when it comes to optimization and self-tuning of such a system.

2.1 From Custom Coding to DBMS Tuning

When the first DBMS systems were introduced in the mid-1960s (e.g., IDS [Bac09] and IMS[1]), the target application dictated the data model and data processing algorithms that had been immutably tailored to the available resources. This approach was adequate for the current quasi-static usage scenarios, where data structures and workloads would remain comparatively stable. These systems where never confronted with dynamic load situations, changing numbers of users, massively increasing database sizes, data model evolutions, and other unexpected usages or changes. Here, a performance analysis of such a static system would always deliver accurate and reliable predictions.

Obviously, in the long run, this custom-made approach is too costly, not only regarding the initial development costs, but also with respect to operational costs for system maintenance, hardware and software crash recovery, schema extensions, i.e., *schema evolution*, requiring new application code, hardware end-of-life requiring hardware or platform changes, and the potential of using data integration to support new applications is rigorously limited. To remedy these problems, to reduce time-to-market and consolidate development efforts that had so far been distributed across these individual solutions, the first generic DBMSs emerged that left the old "hand-built" systems in the dust.

Those versatile DBMSs, available today, contain, even off-the-shelf, a rich set of tuning knobs to customize their configurations. Because nowadays, applications mostly require tailored functionality provided by a standardized (and thus well-tested) product to ease its integration and to enable its exchangeability, a flexible and configurable system is preferred. Even if not all requirements are correctly anticipated during the development and rollout of a DBMS-enabled application, later changes in data volume, load shifts, or new application areas can easily be captured by reconfiguring the underlying DBMS.

[1] `http://www-01.ibm.com/software/data/ims/ims/`

Current requirements of production DBMSs call, in addition to the given ACID[2] properties, for availability, economical operation, and flexibility. *Flexibility* is needed when the DBMS is adjusted to new application areas such as Web, grid, or virtual environments and cloud-based computing, to new data models, and to new workloads. The second requirement – *economical operation* – depends on the input resource definition and accounts to the total cost of ownership. For instance, memory allocation, disk usage, CPU cycles, network load, or energy costs as well as staff costs may be appropriate to account for the operational costs. In contrast, the throughput or response time, peak load handling, and constant query performance are the measurable metrics for an application. However, the operational goal may be pursuing an efficient resource allocation or application performance, which are not stringently opposite goals.

High-*availability*, i.e., 24/7 operation with minimal downtime for maintenance, reconfiguration or optimization, is another common requirement for DBMSs. Specific techniques such as replication are used to ensure zero downtimes, which would however, induce algorithmic and management overhead.

We summarize any activities that aim at optimizing the DBMS, for one or several of these criteria, as DBMS tuning. Because, even in low-availability scenarios, taking a system offline for tuning measures is usually not an option, which is why most of the them have to be performed *online*, i.e., while the system is providing full service.

2.2 Customary DBMS Architecture

Almost all commercial and non-commercial DBMS architectures follow the five-layer reference design proposed by [HR83a], which is shown in Figure 2.1. To some degree, the realization of a DBMS is subject to non-functional requirements such as performance, data independence, and reliability, the layered architecture delivers an ideal concept to reduce its complexity by systematic abstraction. This concept comprises abilities to encapsulate logic and complexity within a certain layer, which can then be developed, improved, and verified in isolation. Each layer provides an interface to its upper layer while realizing certain functionality.

In the following, we will give a brief introduction into the goals of each layer (L1 to L5) and common design principles used to implement them. We will highlight various aspects that will later become options for effective (online) tuning.

The bottom layer L1 gives access to (external) physical storage devices such as hard disks or solid-state disks and encapsulates their physical details. File or partition handling, space allocation, and free space management are the essential tasks of this layer. Through its abstraction from physical devices (e.g., track and cylinder addressing), it provides equal-sized blocks as the unit of data to the upper layer L2. Here, *block size* is an important parameter for DBMSs.

In layer L2, a caching mechanism is employed to service requests from the higher layers using relatively fast main memory and to avoid as many (external) IO operations as possible, i.e., read/write operations to the external devices, which are usually several orders of magnitude slower. The typical units of data are segments or pages that most often have the same size as the underlying blocks of layer L1. A replacement algorithm is used to decide which

[2]The ACID paradigm was established by [HR83b] in 1983 and postulates *A*tomicity, *C*onsistency, *I*solation, and *D*urability for DBMSs.

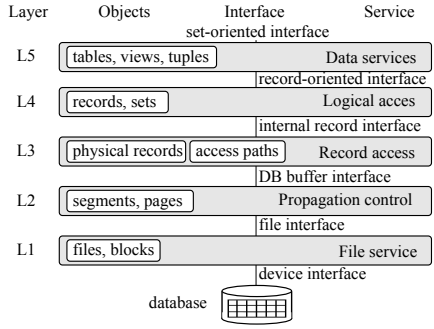


Figure 2.1: 5-layer DBMS architecture reference model [HR83a]

page remains in the cache or is removed from it, because typically, the number of cache slots is significantly smaller than the number of (external) data pages. Thus, the selection of an appropriate replacement algorithm and the number of pages available in the cache are the critical performance drivers in this layer.

Physical mapping of data to records and storage structures is realized in layer L3. Furthermore, it provides an internal interface to the internal records. Efficient mapping techniques may reduce space consumption and number of page accesses and, consequently, also the number of physical IO operations. Additional data structures such as B-trees or hash maps, can dramatically speed up selective data access.

Because the internal records of layer L3 still expose the details of how data is physically represented in pages, layer L4 now abstracts from this. It offers a logical view on data as external records. Furthermore, it operates on record sequences and employs additional access paths such as secondary indexes. Important aspects are the layout of various index types supporting certain access patterns.

Logical data structures are processed by layer L5, typically, as tables (relations), views, and tuples. The interface to this layer is almost equal to the API used for accessing and modifying data. That means, declarative queries are translated into imperative code, i.e., query plans containing operators. Those queries are optimized, inter alia based on statistical information about data and value distribution to efficiently process the query. This includes attempts to reduce the number of intermediate results, avoid sequential scanning of the records by making use of available indexes. In case of updates, those indexes need to be maintained as well, which can result in costly IO operations. Depending on the ratio of read and write accesses of the applications, index maintenance during data updates can offset the benefits obtained when reading data. Consequently, the index configuration is another performance-critical aspect suited for self-tuning.

Although most of the performance-critical parameters directly affect a certain layer, their trigger or impact is often visible throughout several layers due to dependencies between parameters and layers. Therefore, when optimization, i.e., parameter tuning, takes place, it is

wise to observe its impact on the entire system. Moreover, when tuning several parameters at a time, their correlations and dependencies are important to know.

However, for a specific workload, adjusting the tuning knobs is difficult because of the parameter dependencies and the sheer number of alternatives, let alone the identification of the right one to start with. However, most of the architectural issues and their tuning knobs share the same goal – reducing resource consumption such as IO, memory consumption, or CPU usage. In the following, we present the basic concepts that are necessary to identify bottlenecks and performance drivers based on the available resources and their interplay.

2.3 DBMS Cost Models

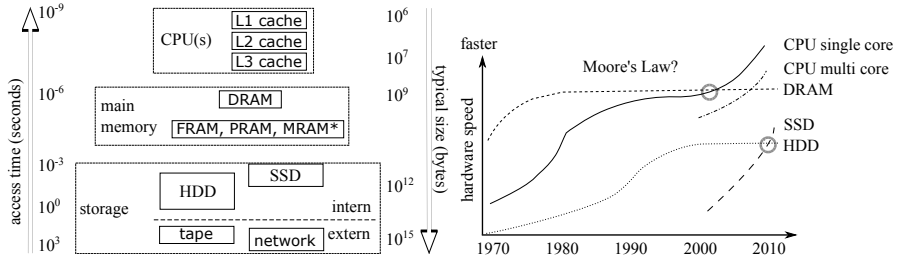
Evaluating tuning measures that are targeted at the reduction of processing time in a DBMS requires a mechanism that makes their impact visible and comparable. Nowadays, most DBMSs employ a so-called *cost model*.

Here, all operations performed during query processing, like disk IO, CPU cycles needed for query planning or processing are assigned with costs in a virtual cost unit, i.e., a "currency" for the usage of system resources. This virtual cost unit typically reflects the time ratios between certain hardware resource usages. For instance, how long does it take to fetch a single page from external storage, i.e., read access compared to the time it takes to calculate a join predicate, i.e., CPU usage. Moreover, DBMS operators such as scan, index access, join, sort, etc. cause certain resource consumptions. Obviously, the actual cost value assigned to each such operation is not fixed, but depends on the hardware available in the system, the system configuration[3] and the load situation, i.e., how many of the resources are free and can actually be assigned. In the following, we will give an overview of essential resources, a DBMS relies on, together with their interplay and their role in optimization.

2.3.1 Resources

Important resources for a DBMS comprise external storage, main memory, and CPU(s). Figure 2.2(a) shows the common hierarchy of memory and storage found in today's computer systems together with their performance and size characteristics. In general, the higher up in the hierarchy a certain kind of memory is found, the faster is its access time and also its throughput increases while, in contrast, its typical size decreases. That means, processing and optimization have to deal with the fact that the more memory is used or required, the slower access times and transfer rates will become. However, external storage such as hard disk drives (HDDs) or solid-state drives (SSDs) are today's method of choice for persistent storage. Although more and more storage systems are connected via networks, i.e., NAS, SAN, or online storage, basically they employ the same kind of storage media, namely disks. There are several technologies under development such as MRAM, PRAM, and FRAM that may become the next generation of non-volatile high-speed memory devices.

[3] The performance of a DBMS is, besides its own configuration, very sensitive to operating system parameters.



(a) Memory hierarchy (b) Hardware development

Figure 2.2: Hardware resource development and relationships.

The different levels of CPU caches, shown as L1, L2, and L3 in Figure 2.2(a) are tiny memory areas that provide extremely fast access. Optimized algorithms try to exploit the so-called cache line by aligning data portions according to cache sizes.

Important for the DBMS cost model are the relationships between memory components in terms of size and access speed. Although all of the resources can be extended to a certain degree, it may impose management overhead or is economically unviable. Thus, optimization has to figure out what is the best-possible resource distribution with the resources available to reduce bottlenecks. Later in this work, in Chapter 3, we will address the bottleneck issue in more detail.

Just like the memory, the number, speed, architecture, and features of CPUs can vary significantly. With the introduction of multi-core CPUs and technologies like hyper threading[4], capabilities for true parallel processing are becoming increasingly common. Exploiting these capabilities allows to improve DBMS performance. Thus, the more highly clocked CPU cores are available the more parallel tasks can be processed. However, data control flow dependencies, communication overhead, and locking of shared resources, put limits on the possible degree of parallelization.

2.3.2 Cost-based Optimization

Once all resources and their relationships in a DBMS have been identified, it is possible to specify a cost model for our system. Such a model allows us to base tuning measures not purely on heuristics and best-practices, but perform *cost-based optimization*, i.e., compare the cost of different alternatives to perform query processing and pick the ”cheapest“ one.

Flexible cost-based optimization is necessary, because hardware characteristics vary and change through evolution, see Figure 2.2(b). Unfortunately, Moore’s Law does not hold all the time, making a forecast for performance developments impossible. But even the hardware equipment of a system may change, too, thereby making a flexible cost model necessary.

[4]Commonly known as simultaneous multithreading (SMT), this technology enables each CPU core to run multiple threads in parallel.

Basic *metrics* for optimization count the number of (expensive) IO operations that can further be distinguished in writes, reads, and sequential or random access. Another important metric accounts the memory consumed, which can be measured in detail for each layer of the memory hierarchy. The third metric we are going to address in this work measures the number of CPU cycles spent for a certain task. Throughout the remainder of this work, we use the terms IO cost, memory cost, and CPU cost, respectively, to refer to these metrics.

Optimization does not only aim at improving OS or DBMS configurations, but also happens during runtime. Especially query execution heavily depends on runtime optimizations, which means the DBMS spends some time to find an optimal way of processing. Therefore, statistics about data are required as well as cost metrics for alternative ways of processing. The problem of query optimization will be presented in more detail in Section 2.4.2 and Section 6.3.

While query optimization happens whenever a query is executed, most configuration parameters can be changed online but, due to their overhead, not that frequently. Here, cost-based decisions have to confirm the (costly) adjustment to make its outcome improving the overall system performance.

The layered model, presented in Section 2.2 is perfectly suited to do cost-based optimization for each individual layer as we will see in the remainder of this work.

2.4 Native XML DBMS

With the increasing demand for XML-based data processing, relational DBMSs [STZ+99] and their object-relational successors were enhanced with features to better handle XML data. For instance, new data types and operators were added. The entire query processing kernel was extended[5] to deal with XML data that was often not stored in its parsed, tree-like form, but simply stored verbatim as text or as large object (CLOB or BLOB).

An early approach to remedy the problems of providing XML storage as add-on to existing systems was LORE [QWG+96, MAG+97], which started as a Lightweight Object REpository for semistructured data, i.e., today's XML. The LORE team realized that not only a tailored storage (i.e., object repository) is necessary but also a new kind of query language for semistructured data [AQM+97].

Just after LORE and other early prototypes had broken new ground in XML storage and processing, the W3C[6] began introducing standards for XML processing such as XPath [BBC+07] and XQuery [BCF+07] that combined and refined the ideas of these pioneers. New APIs like DOM [DOM05] and SAX [SAX04] were recommended as well to provide unified ways for applications to access and handle XML data.

In this chapter, we want to show why it is advisable to develop native XDBMSs instead of extending (object-)relational DBMSs. Furthermore, we want to sketch the differences of both worlds and the commons. Based on our own prototype XTC, we show how specific techniques for native XML processing can be integrated into a real XDBMS.

[5] In the beginning, these extensions were realized by UDFs instead of changing the DBMS kernel itself.
[6] `www.w3c.org`

Data Model Considerations

While XML is often perceived by its well-known textual representation using XML tags and attributes, the actual XML data model XDM [FMM+07] is independent of any specific kind of presentation. The introduction of XDM became necessary because Codd's relational model [Cod70] used so far was not suitable anymore. On the one hand, we have relations containing unordered tuples. On the other hand, we have a hierarchical (i.e., tree-oriented) structure containing (sequences of) nodes. The basic XDM node types are *element*, *attribute*, and *text*. Note, special node types such as *processing instructions* and *comments* are rarely used, and, for simplicity, we omit them in the following. Data typing is also different. Relational data types are typically string-based or numeric together with their typical operations[7]. In contrast, an XDBMS has data types for document (nodes), elements, attributes, character data and so on.

In the relational world, the only way how relationships between data can be modeled is value-based with so-called foreign keys. XML, on the other hand, has several alternative ways to model such relationships. Either the implicit nesting of elements and subtrees lead to an existence-dependent relationship or the explicit ID/IDREF datatypes, KEY/KEYREF identifiers, or XLink and XPointer mechanisms can be used to model (document-crossing) references and data dependencies.

Another important difference between XML and the relational world is the significance of the ordering: A relation is per definition an unordered set of tuples. Consequently, the sequence of tuples as they are returned by a query has no meaning, unless a specific ordering was imposed on the final query result by using an *order by* to specify which column should be used to define the result order. In XML, however, most kinds of nodes have a well-defined order among their siblings, which is completely independent from any actual data values of these nodes. That means, when processing XML data, an implicit ordering of nodes in document order is always present, if not stated otherwise by an explicit order requirement.

As these glaring differences between the XML data model and the relational data model show, the XML data model requires tailored techniques to be (efficiently) processed. This leads to the need for native XML storage layouts, operators, query languages, and APIs[8].

Native XML Data Processing

When processing XML data, various paradigms to store, query, translate, transform, evaluate, and create XML emerged. Depending on the actual usage scenarios, they have different advantages and disadvantages.

1. **Node-oriented processing**: The most fine-granular XML processing is based on direct navigation over the nodes of the XML document. The prominent example for such a navigational API is DOM [DOM05]. Using operations like getFirstChild(), called on a valid XML document node, delivers the logical first-child node if present, other navigational

[7] Conceptionally, a relation is a special data type, too, having customized operations such as creation, deletion, or selection [Gra02].

[8] Note, we do not explicitly look at XSD (XML Schema Definition), namespaces, and XSLT processing.

operations are getParent(), getNextSibling(), getValue(), setValue(), insertElement(), etc. The behavior of most operations depends on the node type they are applied to or are only available for some of the node types. Furthermore, DOM processing requires the entire document being available in memory and it is neither declarative nor stable in case of document structure modifications.

2. **Tree-oriented processing**: Because XML is hierarchically organized, a native way to process XML is to operate on entire documents or subtrees of a document. An API that follows such a paradigm may contain methods to store documents, retrieve documents, or add subtrees, remove subtrees, and update subtrees. Those abstractions are logically closely related to the relational world which is based on tuples. However, identifying and addressing subtrees instead of individual nodes may reduce labeling overhead (see Section 5.2), speed up XML processing, or simplify application use (e.g., serialization, deserialization, persistence APIs). Prominent APIs were developed such as *TAX - Tree Algebra for XML* [JLST02], *GTP - Generalized Tree Pattern* [CC03], *TLC - Tree Logical Classes* [PWLJ04] or an order-preserving *Pattern Tree Algebra* [PJ05].

3. **Stream-oriented processing**: Several techniques were developed to process XML documents as streams, i.e., nodes and subtrees are consecutively visible for the XML processor and only forward steps are allowed and possible. These approaches can further be separated in push- and pull-based variants. In a push-based API, like SAX (**S**imple **A**PI for **X**ML), which was originally developed for Java [SAX04], events are emitted to registered listeners. For instance, startElement(), endElement(), or startDocument() events are delegated to all listeners, while the parser keeps control. An often used combination for stream-based XML processing is the so-called *binary XML* representation of XML [BGJM04]. The main objective is to limit the data volume and speed up processing by using a more efficient binary encoding instead of the rather verbose XML text. A conceptual disadvantage of push-based streaming APIs is that they do not provide any means for modifying XML data.

   In a pull-based approach, the application is in control of the parsing process, such as StAX (**St**reaming **A**PI for **X**ML) [Fry03] following JSR 173. Furthermore, it allows for efficient XML access (especially using cursor API) and for writing entire XML streams, i.e., output XML data. However, no fine-grained manipulation functionality or update semantics are available.

4. **Declarative processing**: By the help of declarative languages like XQuery [BCF$^+$07], its subset XPath [BBC$^+$07], and its extension XQuery Update [AYBB$^+$08], XML data processing achieves the full power and flexibility that is necessary to do DBMS-enabled processing for querying and manipulating XML data. In Section 2.4.2, we will introduce in more detail the concepts that are required for this work.

Native XDBMSs are often built from scratch following the general principles of a layered architecture that we presented in Section 2.2. Before we present the most promising and most effective methods for tailoring the storage layer, buffer management, indexing layer, and query interface, we outline XML-specific languages and query processing concepts.

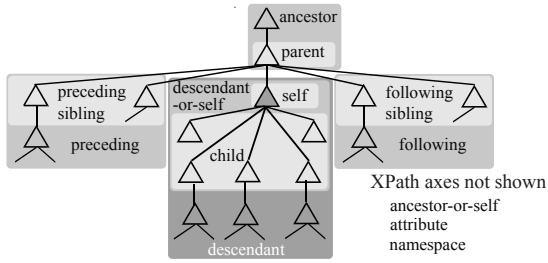


Figure 2.3: All 13 XPath axes according to [BBC+07].

2.4.1 Query Languages

Let us briefly introduce important query language aspects that we will need throughout this work. Because the XML data model is hierarchical, navigation along this tree structure plays an important role for querying XML. Therefore, XPath was developed to allow path-oriented[9] addressing and filtering of XML document structure and content, and became one of the cornerstones for the most successful languages. These languages include XSLT, XPointer, XQuery, and its update extension XQuery Update.

XPath

XPath is used to navigate through the hierarchical structure of XML documents and to evaluate node matches for given predicates. Therefore, XPath distinguishes between 7 different node types, which are root, element, text, attribute, namespace, processing instruction, and comment.

Different kind of XPath *expressions* exist, with the *location path* being the most important one. A location path consists of *location steps* assembled via "/". A location step itself consists of three parts:

- *axis* - Determines the direction of navigation. The 13 different axes specified by XPath [BBC+07] are shown in Figure 2.3.
- *node test* - Specifies node type and name of the nodes selected by the location step.
- optional *predicate* -XPath expressions can be used to further restrict the nodes selected by the surrounding location step. These expressions can be simple boolean expressions, filters, or nested location paths.

A typical location step starting at the context node has the following syntax:

$$axisName :: nodeTest[predicate]$$

[9] In the 70s, the PDP-11 file systems used paths to organize files in a hierarchy, which has been adopted since that by many other systems, too.

For instance, an XPath query selecting all "employee" nodes of a company.xml document that contain the text value "John Doe" can be written like this: $doc(\text{"}company.xml\text{"})/descendant :: employee[text() = \text{"}JohnDoe\text{"}]$. The *doc()* function delivers the entire document as a sequence of nodes, which are embedded into a logical document node. The following location step *descendant* filters all descendant nodes that match the nodeTest, which is in this case a so-called *nameTest* for "employee". Apart from that, a nodeTest may check for a certain node type or processing instruction. The bracket-enclosed predicate contains an additional filter expression. Here, the *text()* function returns the string value of the "employee" node, which is a serialization of the entire subtree's text content except attribute values. If the text value is equal to "John Doe", the employee node is added to the result sequence.

Compared to SQL's WHERE clauses, the predicates and nodeTest features of XPath are fairly similar[10].

XQuery

XQuery can be used to overcome the limitations of XPath. For instance, results can be altered, as well as variable bindings and iterations are possible. In this work, we do not exploit the full functionality of XQuery as it is a Turing-complete programming language [Kep04] and, therefore, we do not explain all aspects in detail, the interested reader may have a look at the specification [BCF+07]. However, the basic construct that is often used for simple XQuery expressions is *FLWOR* – `for`, `let`, `where`, `order by`, `return`.

- `for`: The `for`-clause allows to iterate over the elements of an input sequence to a variable. Multiple `for`-clauses are allowed that bind one or more variables to input sequences. These input sequences may be node sequences resulting from other XQuery expressions, which are typically XPath expressions or atomic values created by literals or constructor functions.

- `let`: The `let`-clause is similar to the `for`-clause, except that the variable binding is only valid for a specific iteration (within the "outer" `for`-loop). The `let`-binding is used to bind an entire sequence to a single variable.

- `where`: Within a `where`-clause, filter predicates are specified that are applied to the current values of bound variables within an iteration.

- `order by`: The `order by`-clause is used to order the result set (i.e., a sequence of items) of the current iteration.

- `return`: The `return`-expression defines the sequence of constructed elements that constitute the result of the entire expression.

Referring to the XPath example, a similar result using an XQuery FLWOR expression is achieved by the following query:

[10] Location steps are often evaluated using so-called structural joins, which are similar to the joins found in the relational world.

```
for $i in doc("company.xml")
  let $employee := $i/descendant::employee
  where $employee/text() = "John Doe"
  order by $employee/age
  return <emp>$employee</emp>
```

Note, an expression is the basic element of an XQuery statement. Common expressions are FLWOR expressions, path expressions (i.e., including XPath), element construction (e.g., in the sample query within the `return`-clause), conditional expressions, and function calls.

Later in this work, we will see how those XQuery statements are internally represented, translated, and analyzed.

XQuery Update

Although XQuery allows creating new node instances, it does not allow to modify existing ones. Therefore, the XQuery Update Facility was introduced. It extends XQuery by allowing *update expressions* to modify the state of existing nodes preserving their identity and to create modified copies of existing nodes with new identities.

The five *update expressions* `insert`, `delete`, `update replace`, `update rename`, and `transform` are recommended by the W3C [AYBB+08]. They can be used to insert or modify individual nodes or subtrees. Basically, a query part selects the target nodes that is followed by the modification step.

Because XQuery and its extension XQuery Update are based on XPath expressions, we will henceforth refer to them interchangeably.

2.4.2 Query Processing

As mentioned at the beginning of this chapter, native XDBMS processing incorporates its own APIs, operators, and processing model. Still, the approach to process declarative queries (given in, e.g., XPath or XQuery) is fairly similar to that of pure relational query processors, using the common three stages of *translating* the query, *optimizing*, and finally *executing* it. The entire process is sketched in Figure 2.4. The essential differences are usually limited to the internal representations of queries between the three steps. In the translation phase, the external **query string** is parsed and an AST (abstract syntax tree) is produced, which contains the syntax elements of the query. Throughout the sub-steps of *normalization*, *static typing*, and *simplification*, this syntax is checked and reduced to a canonical AST (i.e., a normalized core representation), which is finally translated into a **logical plan** containing logical operations according to a certain algebra. Such a plan may be described in XQGM, i.e., an XML extension of the classical query graph model (QGM) [HFLP89]. The logical plan serves as input to the *optimizer*, which reorganizes or replaces the algebraic operations with the goal of improving query performance and/or reduce resource utilization. Besides reducing the number of operations and the intermediate result size, it also identifies join options and generates alternatives for logically equivalent XQGMs. The plan identified as “the” optimal (i.e., cheapest) one is

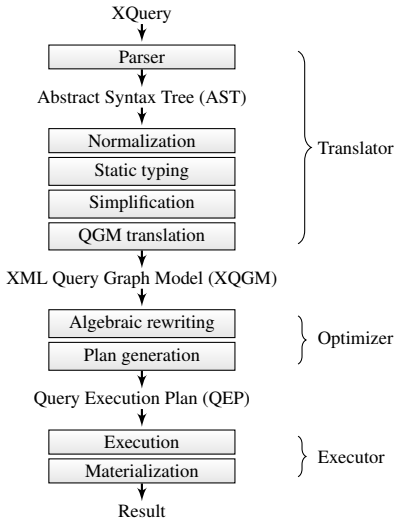


Figure 2.4: Query processing pipeline

then converted into a **physical plan**, i.e., algebraic operations are replaced by physical operations on data storage objects. This physical plan, referred to as QEP (query execution plan), is processed by the *query engine* and delivers a sequence of XML nodes. Eventually, these nodes are *materialized* to a given result format, such as string or XML node sequence.

2.4.3 Special Operators

Tree pattern matching is the most important aspect in XML query processing [MW99, Abi97]. Those query patterns define a certain structural relationship of named and unnamed nodes, which have to be found within the given XML document(s). Typically, these relationships are evaluated using joins on multiple node input streams, instead of costly per-node navigations. Therefore, optimizing XML queries requires efficient join operators – the so-called binary *structural join* and n-ary (holistic) *twig join*.

Structural Join

A structural join is used to find all occurrences of a structural relationship between two XDM sequences. The sample in Figure 2.5 shows the decomposition of a complex query into its five basic binary structural relationships. The individual binary relationships are evaluated and their results are joined. There are a plenty of tailored join algorithms for XML. Even relational systems employed special join algorithms, whose different characteristics are exploited by relational optimization techniques [STZ+99, FRK99, CS01]. Thus, native XDBMSs require cus-

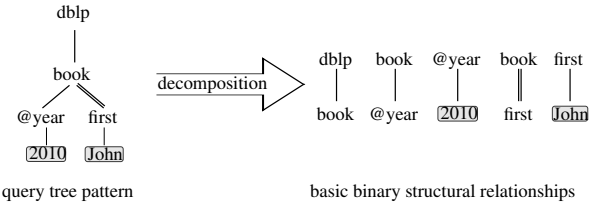


Figure 2.5: Structural join decomposition for query
$dblp/book[@year = \text{"2010"}]//first[text() = \text{"}John\text{"}]$

tomized implementations [MW99]. For instance, the *multi-predicate merge join* (MPMGJN) [ZND+01] is based on the inverted-list indexes from [MAG+97] and simply follows a traditional merge join algorithm. Alternatives try to exploit additional indexes, such as the *XISS* index approach in [LM01] providing the EE-Join and EA-Join. However, both approaches may perform unnecessary IO for matching structural relationships, because they scan the document or the same indexes multiple times.

Specialized join algorithms focus on a subset of structural relationships. For instance, a tree-merge join and a stack-tree join for ancestor and descendant evaluation only is proposed in [AKJK+02]. Other approaches try to exploit different kinds of indexes such as in [CVZ+02], where a B+-tree index and an R-tree index are used for structural join processing or the so-called XR-tree in [JLW03].

A practical way of processing structural joins is to exploit a prefix-based node labeling scheme and corresponding tailored indexes. A node labeling such as DeweyIDs can be used to evaluate the structural relationship for any two nodes. We will introduce DeweyIDs and XML node labeling in Section 5.2. The two input sequences to be joined may be filled by indexes. Customized XML indexes such as the path, element, or content indexes will be introduced in Section 6.2. Without appropriate indexes, all input sequences may also be generated by full document scans, as a fallback, typically requiring a lot more IO operations.

In our sample in Figure 2.5, the input sequences for "book" nodes and for "first" nodes include their node labels, which make it easy to evaluate their descendant qualification. As a general prerequisite for most forms of (structural) join processing, the input sequences need to be sorted in document order, which fortunately is the natural order when obtaining an input sequence via a document scan, and which is also ensured by appropriate indexes.

Twig Join

Besides simple, i.e., binary structural joins, so-called branching queries are common XML query expressions. The query on the left-hand side in Figure 2.5 is such a branching query. The idea of *twig joins* is to avoid the costly decomposition into multiple binary join cascades, and operating on multiple input sequences in parallel. A bunch of so-called twig join algorithms were developed that address the diversity of these query constructs in a holistic way. Nearly all of the algorithms rely on specialized index structures that avoid unnecessary IO. One of the

first algorithms, the TwigStack [BKS02] uses an XB-tree index. Further algorithms focusing on tiny differences emerged throughout the recent years, for instance FiST [KRML05], TJFast [LCL05], or Twig^{2}Stack [CLT$^+$06] based on GTP (Generalized Tree Pattern) [CC03].

However, the most important aspect of all twig algorithms is to support the evaluation of complex tree patterns using indexes [CLL05]. Furthermore, intermediate results are minimized or even avoided if not part of the final result. Some approaches optimize the evaluation of wildcard steps in XPath expressions, specific ordering, or certain axis evaluations.

2.5 XTC Prototype

This thesis and most of the concepts presented in it were mainly developed by extending the native XDBMS XTC - XML Transaction Coordinator [HH07], which was originally developed by our research group to evaluate XML locking protocols. Through the contributions of many researches, this initial prototype matured and evolved into a full-fledged XDBMS. In this section, we want to highlight the most important architectural aspects, in particular, those relevant for the addition of self-tuning mechanisms as they are subject of this thesis. First, the overall architecture of XTC is presented. It follows the original five layer design approach of [HR83a]. The essential elements of the architecture are sketched in Figure 2.6 including basic performance-critical parameters for the respective parts. Later in this work, we will give a detailed introduction to XTC's *storage mapping* concepts addressing XML verbosity (cf. Section 5.1), XTC's *node labeling* scheme using DeweyIDs (cf. Section 5.2), XTC indexing (cf Section 6.2), and its query processing pipeline (cf. Section 6.3).

L1 - File Services

The bottom layer of an (X)DBMS contains *IO managers* that typically operate on files or disk partitions (jointly referred to as "container" in the following) using a block-oriented interface for reading, writing, allocating and releasing equal-sized disk blocks. Obviously, the block size is the most important parameter as it specifies the maximum address space and is the basic unit of IO. It has to be specified when registering new data containers. Consistency-related parameters, for instance the usage of before image while writing (new) information can be controlled in this layer, too. An aspect with a significant impact on performance is the extent size of a container – as long as the container and the underlying file system allow dynamic file growth. Because extending a container file may require a lot of blocking IO operations to prepare physical device blocks and update metadata information such as the new container size and free space figures. However, sparse files may speed up this process by not allocating the physical blocks immediately but assigning them[11]. In this case, the runtime performance for writing these extents for the first time may degrade, because it includes the actual allocation of physical blocks. In XTC, we do not address container growth explicitly and instead rely on the underlying file system implementation; we simply assume that any IO for a certain container file is suspended, when the file size is increased.

[11] Sparse files allow to assign storage space, i.e., free space information is updated, without actually allocating the physical space immediately. On demand, physical space is claimed when (new) data needs to be stored in the file.

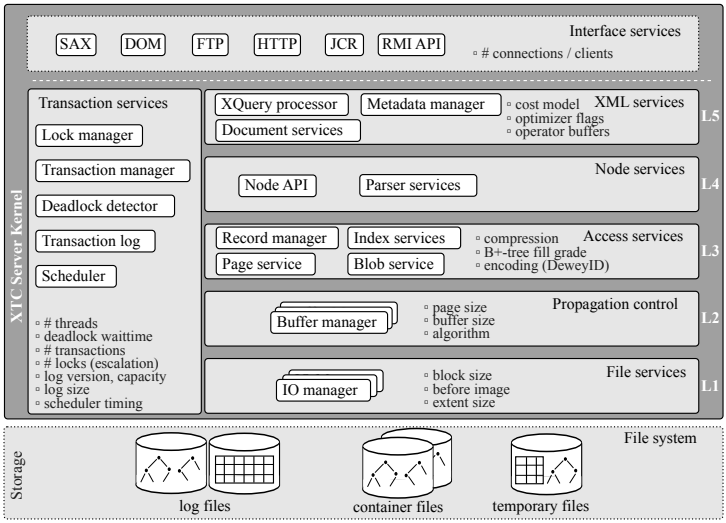


Figure 2.6: XTC architecture and basic performance parameters

L2 - Propagation Control

The propagation control layer L2 is responsible for buffer management. For each IO manager in L1, a corresponding buffer manager is instantiated and a specific memory pool is assigned, i.e., buffer pool. The buffer manager or, for short, buffer is operating on a page-oriented interface. As described in Section 2.2, the buffer fixes, unfixes, allocates, releases, reads, and writes (or *flushes*) pages. The most important parameters affecting system performance are the page size that has to be equal to the assigned IO manager's block size and the buffer size itself. The buffer size can be specified as an absolute amount of memory as long as it is a multiple of the page size or can be given as the number of pages. To control (when and) which pages are currently removed from the buffer (and written back to disk if they have been modified since they were loaded into the buffer) to make room for other pages, in the case of page faults[12], XTC's propagation control layer provides a selection of page replacement algorithms. In addition to well-known algorithms like LRU, LRU-k, GCLOCK, 2Q, which were designed for magnetic disks, the system also provides algorithms tailored to better address the properties of flash disks such as CFDC [OHJ10].

L3 - Access Services

The access services layer L3 is responsible for the management of database records. A record typically reflects a data item such as an XML node from the XDM info set. Nodes are identified

[12] If a component on a higher layer tries to access a certain page that is not currently in the buffer, we call this a page fault.

by DeweyIDs (cf. Section 5.2). As we will see, DeweyIDs of consecutive nodes have common prefixes, XTC makes use of this property by applying a prefix compression scheme on them. Special logic to transform between logical XDM node instances into physical byte arrays is located in that layer. Besides methods for individual record encoding and decoding, various page types are supported (i.e., key/value, slots) and separate BLOB (binary large objects) logic is available in XTC.

Additionally, the third layer implements access structures such as B^+-trees and lists working on records[13].

Performance can be influenced by adjusting the compression configuration for records and/or node labels, and by controlling parameters of the index structures, for instance, a B^+-tree's filling degree.

L4 - Node Services

For a reader familiar with DBMS architectures, our description of the layers L1 to L3 so far probably sounded not too different from the well-known standard architecture according to [HR83a]. Obviously, because these layers are not tied to a specific data model. With the node services layer L4, the distinguishing features of a native XDBMS over conventional relational systems will become much more apparent, because the node services solely operate on XML nodes and related APIs such as DOM and SAX. The DOM interface is node-oriented, i.e., all the navigation and manipulation is performed by calling corresponding methods on the DOM nodes. This interface consequently hides the internal details of node representation, index structures, and document references – "a node knows everything". However, nodes do not necessarily have physical counterparts, i.e., the system distinguishes between physical nodes (carrying valid node labels) and in-memory nodes (carrying no node labels).

The second essential task performed in this layer is the parser service that allows event-based and stream-oriented operations such as SAX and StAX processing. This interface is also used during index materialization, where simply an (index) listener is attached to a SAX scan on the document's root node. Furthermore, document storage, retrieval and subtree processing is using this interface because of its sequential scan semantics, it is clearly faster and easier to use compared to the navigation-oriented alternative provided by the node manager.

L5 - XML Services

The XML services layer L5 is responsible for the processing of the different XML query languages supported by our system, i.e., XPath and XQuery. Our XQuery processor follows the principles introduced in Section 2.4.2. In addition, XTC controls the application of optimization rules through configuration flags, and thereby limits the search space for alternative query plans. Moreover, the cost model is defined and implemented in that layer, which is responsible for estimating plan costs for IO, CPU usage, and main-memory consumption. Furthermore, the runtime behavior of many operators can be controlled, for example, by adjusting the amount of (temporary) memory available for intermediate results of sort and join operators. Besides

[13] B^+-trees are the fundamental storage structure in XTC, for documents, indexes, and metadata.

the given complexity of query processing, performance-critical aspects in layer L5 include the configuration of the various phases – optimization, planning, and execution.

In XTC, documents are organized in a virtual file system structure, offering users a familiar paradigm to handle their XML data. In addition to query optimization and processing, L5 is also responsible for providing the so-called *document services*, i.e., functionality to register (or store), import, rename, move, and remove documents, group them into document collections, and perform manipulations of the virtual directory structure itself.

All the metadata for a single XTC instance is managed in L5 by the *metadata manager*. This includes document and index information, storage and compression parameters, user management, and statistics. The entire metadata management is performed on a single XML document called "_master.xml". Users are not allowed to modify it directly. The logic to manipulate this document is mostly performed by internal XQuery calls and partially by hand-coded methods based on DOM due to performance aspects.

Transaction Services

Most of the ACID properties are ensured by the transaction services, which are operating in parallel to the aforementioned layers L1 - L5. The *lock manager* is responsible for translating lock requests, for instance, by the node manager, into different types of locks on the nodes forming a document. Due to XTC's history as a research testbed for concurrent transaction processing on XML documents, it does not offer a single, hardwired lock protocol, but instead offers a well-defined interface that allows different protocols to be implemented as pluggable components. Each lock protocol can define its own set of lock modes. Currently, XTC supports 12 lock protocols, such as the entire taDOM family [HHL06], but also a number of competing XML locking approaches. A performance-critical parameter in databases is the number of concurrently held locks, as each of them consumes main memory. Having more locks in the system will require an increasing amount of processing time for lock lookup, matching locks, protocol evaluation, and the creation of new lock objects. To keep the number of locks low, the well-known mechanism of lock escalation can be used: Once a certain threshold of fine-grained locks has been reached, they are replaced with fewer, more coarse-grained locks this way reducing the overhead for lock management, but also reducing the number of lock conflicts and thus the amount of concurrency in the system. The lock escalation thresholds are an essential tuning parameter of the transaction services component and can be used to balance the overhead of locking (memory, CPU) and its gain (concurrent XML document access and manipulation).

The second component is the *transaction manager*, which is demarcating the concurrent transactions within the system. The basic protocol allows applications to demarcate transactions with the common *begin*, *commit*, and *abort* operations, while the transaction manager itself takes care of doing appropriate redo and undo actions in case of abort or crash recovery. To fine-tune the transaction manager, the number of parallel running transactions and to a certain degree the number of parallel threads can be specified.

Parallelism in DBMSs is a crucial aspect controlled by lock protocols – so XTC does, too. In general, lock protocols may allow transactions to crosswise acquire locks which can make

them end up in a deadlock. Therefore, XTC employs a wait-for graph for transaction dependencies which is periodically analyzed. This deadlock detection may slow down transaction processing because it has to preserve some kind of "snapshot" of the wait-for graph while analyzing it, i.e., for a short time, transaction dependencies cannot be modified. While deadlocks are typically rare events, they become very costly when rolling back a loser transaction that already manipulated a lot of data. Therefore, the so-called waiting time between deadlock detection cycles can be adjusted in XTC.

In order to provide full ACID conformance, XTC employs a *transaction log*. This log keeps track of (un-)committed changes and is written to disk before the actual data is manipulated (*write-ahead lock*, WAL). The log files are located in separate containers and, therefore, allow for separate tuning. XTC supports different log types, e.g., based on DB objects or data blocks. Only the total size of the log container needs to be specified, preferably in accordance with the underlying device's free space. In case of block-oriented logging, the block size has to be specified analogously to the block size parameter for file services. As log files are written sequentially and synchronously, it is in general advisable to place them on a dedicated physical device, because otherwise, the log writes can interfere with the randomly-distributed IO operations for the transaction themselves.

Although XTC does not provide a full workload management system, it supports certain scheduling features for parallel running tasks and jobs[14], as they are called in XTC. Coordinated by the *scheduler*, user and system transactions are assigned individual priorities[15] and *worker threads* process the different kind of jobs and are created, activated, suspended, and canceled by the scheduler. Besides the number of worker threads, the scheduler timings can also be adjusted. For instance, the polling frequency for the job queue can be aligned to the expected transaction income rate.

Interfaces

The XTC system provides several interfaces to work with XML or the DBMS itself. Without any additional driver, a user or application can interact with XTC via FTP or HTTP using an FTP client or browser, respectively. However, these interfaces are not that comprehensive and are in general only useful for a glimpse at XTC.

Using the XTC driver for Java, stream-based access is possible via the SAX interface and navigational access via the DOM interface. XML document manipulations are possible via the corresponding methods of the DOM interface, e.g., *setValue*(), *addX*(), *removeX*(), etc. These methods can be called at the client side and are automatically translated to corresponding manipulations of the server-side objects, i.e., DOM nodes.

The probably most powerful interface is the native RMI API, which provides XQuery interaction, internal method access, extendable procedure access, and a lot more utility functions.

[14] A job is created internally, either by a user transaction or a system transaction, which, however, may contain further sub-transactions. Tasks are more abstract and do not necessarily require a transactional context.

[15] During the development of a priority concept, we made the experience that Java's thread scheduling is heavily platform-dependent and JVM-dependent. Therefore, we extended the XTC job model with some control mechanisms to allow a coarse-grained priority-based scheduling. While the results were promising on Windows platforms, success on Linux platforms was limited.

A typical database application only requires the XTC driver containing these interfaces. Its use is fairly simple, because similar to JDBC connections, a connection object is returned after providing proper login and host information. Using this connection, new transactions and (X)queries can be sent to the server. Applications can decide whether they want to access the query results as serialized string representation, in DOM-style, or as node sequence. The result may further be extended with internal statistics such as query plans or query processing timings. Besides pure query-oriented features, the RMI API also supports analysis features to dig into data structures for debugging purposes. Moreover, many system parameters can be controlled via that interface, too.

XTC also includes an interface compliant to the Java Content Repository (JCR) specification, a standard for storing and interacting with content repositories [Nue06, Pre08]. It allows applications dealing with hierarchical content data to benefit from the strengths of a native XML DBMS.

2.6 Alternative XDBMS Systems

Systems for XML data management and (X)query evaluation can be classified in two major categories – (object-)relational and native. Due to their significantly different nature and the focus of our work, we highlight here the most important or successful native XDBMS systems that may be comparable to the XTC approach or yield further starting points for similar optimization approaches[16].

Because DBMS tuning heavily depends on the DBMS architecture, we provide an overview for the most successful native XDBMSs in Appendix A. Many of the systems presented there share some architectural design decisions. In nearly all cases, the storage subsystem is separated from the query processing part, which again advocates the development of customized tuning for each individual component. As we pointed out, not all systems fully support fundamental DBMS properties such as transaction management, indexing, buffer management, or full-fledged query processing, but their component-based or layered architectures offer similar starting points as XTC does for exploring self-tuning mechanisms in a native XDBMS.

Many other systems, such as BaseX [GHS07], eXist [Mei09], or Sedna [FGK06] have also a comparable architecture, as have the many relational systems with XML support, like DB2's pureXML [NvdL05], Oracle [BKKM00], SQL Server [PCS+05], MonetDB/XQuery [BGvK+06], or legoDB [BFH+02], which build their query processing and XML mapping on top of a relational storage and processing kernel. Thus, most of the (self-)tuning techniques developed in this work in the context of the XTC system are also applicable to a wide range of native XDBMSs.

[16] In [Mat09], comprehensive comparisons of XML processing capabilities and data model handling for a large set of DBMSs can be found.

Chapter 3

Self-Tuning – Challenges and Goals

This chapter gives an introduction to *Self-Tuning* and an overview of available online-tuning techniques for DBMSs. In this chapter, we will use the (X)DBMS architecture presented in Chapter 2 to identify certain tuning areas and explore suitable runtime tuning techniques. Although some of the concepts presented in this work are not restricted to the native XDBMS domain, in fact, neither implementation-specific aspects nor XML-specifics will impair the tuning measures, realization and evaluation is always based on our prototype XTC [HH07] to demonstrate the practical applicability of some ideas.

Implementing and evaluating self-tuning capabilities requires a platform that allows for extensions to monitor, control, and adjust the system at runtime. We therefore show the extensions we made to the XTC system – a framework that allows to integrate arbitrary self-tuning capabilities.

This chapter concludes with a comprehensive analysis of the architectural aspects of an (X)DBMS addressing optimization and self-tuning opportunities, which serve as detailed roadmap for the remaining thesis.

3.1 From Tuning to Self-Tuning

The major objective of tuning is to identify performance bottlenecks or unfavorable configuration settings to change the system configuration resulting in a performance boost. Today, performance analysis and the resulting tuning and reconfiguration measures are done by human administrators. However, this traditional tuning approach naturally has its limitations: With today's frequently changing usage scenarios or the fact that more and more systems are exposed to public users, a human administrator may simply not be able to react timely enough to sudden changes in system load or usage patterns. Further, such a scenario tends to put considerable stress on an administrator, increasing the likelihood of errors that further degrade performance or, in the worst case, topple the entire system. Finally, with the "exponentially" increasing number of information systems, the costs of such intensive 24/7 supervision by human experts will rapidly become economically infeasible.

Further, the whole process is highly depending on the individual knowledge and experience of few employees. In an industry known for its high employee churn rate, it can therefore be difficult to keep this expertise. So-called best-practice rules may offer a first point of help but, nowadays, complexity of system configurations and landscapes are hardly captured that way.

A system providing *self-tuning* capabilities[1] should support or, on the long run, substitute

[1] Especially software systems use auto-tuning to refer to the same concept as self-tuning.

the human expert. As a starting point, it is beneficial to "feed" the expert's knowledge into the system and create initial defaults out of it. Because a *self-tuning* system typically fulfills a certain optimization goal, it is straightforward to let it behave like a human pursuing this goal, i.e., monitoring, analyzing, and continuously optimizing its actions. By using what-if analyses or simulations, the system may be capable of doing performance forecasts for alternative configurations as well.

Eventually, a *self-tuning* system should provide the same or better tuning measures than a human, while operating faster, cheaper, and more reliable.

3.1.1 Offline vs. Online Tuning

Important for tuning and self-tuning is the question when is the right time to tune. You want to remedy a performance problem as soon as possible, but at the same time, you also want to avoid unnecessary interruptions on a production system or an outage at all. Tuning a running system may cause unforeseen and unwanted side effects that can incur additional performance penalties. Due to the inherent dangers of online tuning on a production system, it is common practice to test the new configuration settings on a dedicated test system or perhaps defer them according to a maintenance schedule.

Dependent on the DBMS workload and configuration, one has to weigh up the aforementioned risks of doing changes online with the potential benefits of reacting more quickly to a changing workload that currently negatively impacts system performance. For example, if the quality of service delivered by the system is reduced, but still within tolerable limits, it might be advisable to defer changing the configuration. If, however, users are already strongly affected by the current performance problems, waiting might not be an option.

Another issue that need to be taken into account is the unavoidable overhead caused by any form of online tuning facility, which is incurred not only when actually performing changes, but permanently during system runtime for monitoring and evaluating the current system status. Thus, this overhead has to pay off, i.e., the benefit of (online) tuning must exceed its costs.

In this work, our goal is to perform self-tuning measures online as often as possible, i.e., the DBMS – in our case, XTC – has to adjust the system while it is servicing user requests.

3.1.2 Problem Classes

Computer science theorists have intensely studied the problem space of DBMS tuning for several DBMS areas such as physical design tuning [PDA07, BC08], optimizer tuning [CN01, SLMK01], or memory tuning [SGAL$^+$06, BS11]. In almost all cases, the number of alternative configurations grows exponentially with each additional parameter[2], leading to extremely large search spaces. Due to this combinatorial explosion, finding the optimal configuration – or at least one that comes reasonably close to it – quickly becomes computationally intractable. For many of these problems, it was shown that they are NP-complete or at least NP-hard. In [Com78, PS83], the authors show that the optimal index selection (short ISP - *Index Selection Problem*), i.e., the decision which indexes to create for a given schema and for a DBMS is

[2] Additional parameters are constantly added, typically through product updates, which increases the search space, too.

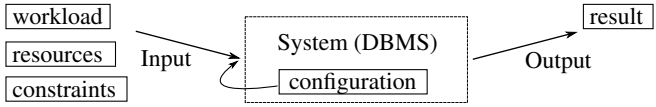


Figure 3.1: System view of a DBMS

NP-complete. Fortunately, in practice many constraints typically reduce the search space: For example, the resource budget, i.e., the amount of memory or CPU power available, is limited and therefore any configuration alternative violating this budget can be dropped without further inspection. At the same time, in addition to such external constraints, certain DBMS constraints, like restrictions in the value ranges or parameter dependencies, constitute another problem class, so-called *constraint satisfaction problems* (CSP) often identified as combinatorial optimization problems [BR03]. The aforementioned space restrictions are typical optimization criteria for the class of *bin packing and knapsack* problems [GGU72, CGJ97]. All these problems are well-known examples of NP-complete problems.

When optimization time is critical, it may further be important to limit the maximum search time. Important for time pruning is that the search continuously improves the current solution, i.e., the more time spent for search the better the output.

Brute-force solutions such as probing a lot of possible configurations [DTB09] are not desirable due to their non-polynomial runtime for explorative search.

Although most of the tuning problems cannot be solved in polynomial time, fast reductions are always possible at least when not attempting to find the optimal solution, but restricting oneself to finding one that is reasonably close enough to the optimum, or just sticking to solutions that are "good enough" for the current problem situation.

3.2 Self-Tuning

Self-tuning, as already indicated in this chapter, is about tuning measures conducted by a system itself. Research in this area; particularly in the context of DBMSs, has a long history, which we will briefly sketch in the following, before we present various approaches and concepts to accomplish self-tuning.

But let us clarify some terminology first. The goal is to optimize a system that may be regarded as a black box getting some *input* and producing an *output*. This simplified view of a system, in our case the DBMS, is depicted in Figure 3.1. The *input* consists of the actual *workload*, *constraints*, available *resources*, and the *current configuration*. On the other hand, the *output* interesting for self-tuning is the new system configuration. First, it is important to know the workload or at least its essential characteristics that define its impact on the system performance. User interaction (i.e., query processing), the data itself, and DB tool execution typically form the workload of a DBMS. As already motivated, this workload can change rapidly and unexpectedly within very short time frames. Second, constraints have to be met such as maximum connections, operating system capabilities, or prioritizations for certain requests. An

often fixed parameter is the availability of hardware resources, which are statically assigned[3] and therefore make the life for self-tuning easier. The third essential input parameter is the currently active DBMS configuration. The direct link between input and output is constituted by the DBMS configuration and forms a kind of cycle.

All techniques presented in the next section(s) are dealing with the problems to identify reasonable changes in the configuration and how to apply them.

3.2.1 A Brief History of Autonomous Computing

Research on autonomous computing systems dates back almost a quarter of a century and was never focused specifically on the context of DBMSs. An example of an early form of self-managing appeared in the context of communication networks: As part of the ARPANET approach, the predecessor of today's ubiquitous internet technology, [RP86] developed a kind of self-managing network where routing decisions could be performed ad-hoc based on the current state of the different alternative routes.

Today's common notion for this area of research was coined during a speech of Paul Horn from IBM. Inspired by the principles of the human autonomic nervous system, he compared its hierarchical nature to that of computing systems [Hor01].

Projects such as DASDA2 from DARPA focused on optimizing the architecture of large distributed software systems to meet dependability and adaptability requirements based on monitoring [COWL02]. Furthermore, preventing attacks, vulnerabilities, and provide rock-solid high-available software that automatically adjusts itself and scales by self-reconfiguration is the common goal of many research projects up to now [Gar02, Geo05].

When IBM introduced the concept of autonomic computing, they identified four major properties of a self-managing system: *self-configuration, self-optimization, self-healing*, and *self-protecting* [KC03]. Over the course of the next sections, we will shed a light on the meaning of these terms.

The database community quickly adapted the ideas of self-management and self-tuning [WMHZ02, CN07]. Despite good initial progress in this area, a truly holistic tuning perspective, which takes all aspects that control the performance of a DBMS into consideration, has not been achieved so far. Most of the techniques do an isolated optimization of certain resources or components of a DBMS.

In the remainder of this section, we will discuss the basics for self-* properties, namely *monitoring, analyzing*, and *adjusting*.

Self-* Properties

According to [KC03], self-* properties of self-management are distinguished as follows:

- **Self-Configuration**: Self-configuration aims at setting up hardware and software to work properly in their designated environment without human intervention. Installation

[3] New processing concepts operating in virtual environments such as cloud computing or pay-as-you-go can dynamically add or remove certain resources. For instance, CPU(s) can be activated on demand (e.g., Capacity on Demand) or re-assignments of main memory for a virtual machine are immediately effective.

of network and computing equipment is followed by software configurations. Already during the initial setup of a (software) system, autonomic computing provides means to specify high-level objectives for its operation purpose, but not how they are actually accomplished, i.e., the tasks necessary to fulfill a certain objective. For instance, fault-tolerance and scalability can be achieved by automatically exploit and identify (added or failed) computing resources.

- **Self-Optimization or Self-Tuning**: As already introduced in Section 3.1, self-tuning targets at the adjustment of the large number of tuning knobs of a running system. Commonly focusing on performance goals, the system tries to automatically fulfill this goal through parameter changes. A system may learn from its own actions as well as act proactively to avoid predictable bottlenecks.

- **Self-Healing**: Software bugs and hardware failures are unavoidable, and a self-managing system has to cope with them. Self-healing techniques provide mechanisms to detect and trace those problems to keep the system in a consistent state or at least to support the problem solving by diagnostic information or probabilistic evaluations.

- **Self-Protection**: Either attacks from the outside or failed attempts of self-healing may require the system to protect itself from their effects. This includes actively blocking attacks after they have been discovered or avoid further damage by protecting affected system parts.

In this thesis, we will primarily address the tuning and configuration aspects of autonomic computing. However, most of the concepts and facilities that we establish over the course of this work, in particular the monitoring and dependency analysis, can also serve as a foundation for other self-properties such as protection and healing. For the remainder of this work, we will use the terms management, tuning, and optimization interchangeably.

Next, because various approaches exist to address self-tuning of systems, we will highlight the most prominent ones that have emerged in literature.

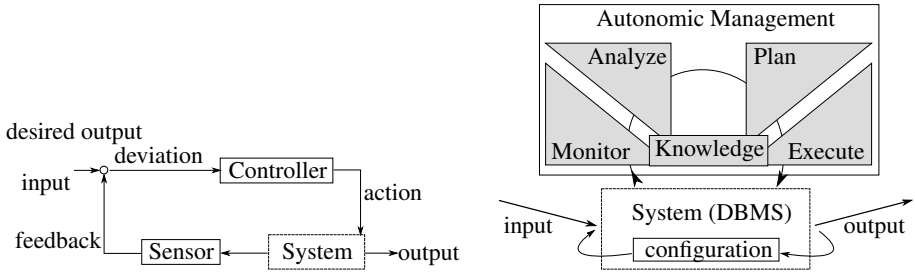
3.2.2 Feedback Control Loop – MAPE-K

The basic idea of a feedback control loop as shown in Figure 3.2(a) is to control the target system by continuously taking its output into account. Classic system theory uses the term sensor for monitoring and requires a controlled process variable. In this case, the regulation is focusing on minimizing the deviation for the given target variable(s).

Adopting the feedback control loop from system theory, IBM's approach named MAPE [KC03] distinguishes between four phases, namely *M*onitoring, *Analyze*, *P*lan, and *E*xecute (MAPE) as shown in Figure 3.2(b). Operating in parallel to the normal DBMS processing, the MAPE approach acts as a feedback controller to improve system performance.

Now let's briefly have a look into the four (different) phases of the MAPE approach:

- **Monitoring**: Hardware, software, and network components are monitored, where the frequency of measurements and the number of monitored system parameters must be



(a) Feedback control loop

(b) MAPE-K cycle for *Autonomic Computing* [KC03]

Figure 3.2: Feedback-based self-tuning schemes (extension of Figure 3.1)

chosen carefully to achieve the necessary level of detail and precision while keeping overhead minimal.

- **Analyze**: In the first step of the *control* part, monitoring data can be filtered, transformed, and aggregated to improve its analysis. Based on desirable target values or thresholds for the monitored parameters, the analysis may find violations or correlations, which are then subject to a more detailed inspection.

- **Plan**: The second step of the *control* part may exploit knowledge about the system behavior combined with the already analyzed monitoring data to decide what actions are necessary to improve the performance, i.e., a goal fulfillment.

- **Execute**: Eventually, the planned actions have to be executed by changing the system parameters. Depending on the type of changes, the effects may be immediately visible in the next monitoring round or they show up only with – sometimes considerable – delay.

Note, as indicated in the figure, the different phases may be tightly coupled with the actual system, i.e., at least monitoring and execution require access to the system's internals.

To support the analyze and plan phases, the MAPE cycle can optionally be extended by a knowledge component (MAPE-K) [IBM04], which reflects expert knowledge, experience (e.g., gained through learning), conditions, or further system-relevant limitations.

A similar feedback control loop approach is implemented in *OceanStore's* "Cycle of Introspection" [KBC+00] that refers to the elements of its control cycle as *observation, optimization*, and *computation*. Taking historical records of system behavior by collecting events and current activity monitoring into account, periodical pattern extraction guides the optimization module.

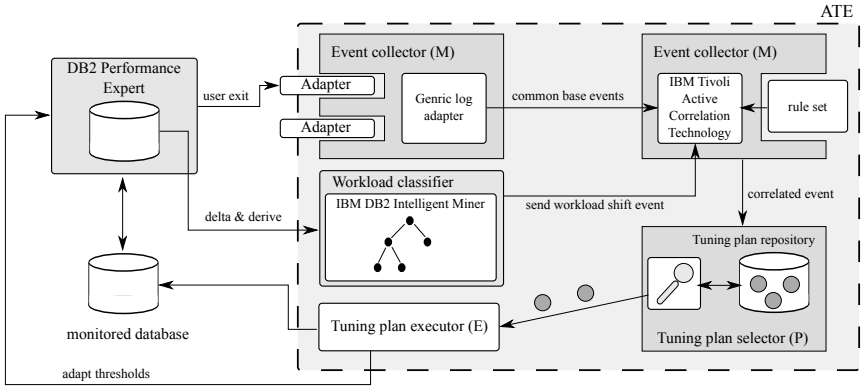


Figure 3.3: Autonomic Tuning Expert (ATE) infrastructure [WRRA08]

3.2.3 Rule- or Policy-based Management

Tuning a system by rule application is a straightforward approach, aside from developing a suitable mechanism or language to define such rules or policies for a certain scenario. So-called high-level goals (cf. Chapter 1) may guide the rule definition and search [KM07]. Rules can also be defined based on SLAs (service level agreements) or business demands, as well as experience.

One of the first approaches that can retroactively be understood as a restricted form of self-tuning is query optimization, although aspects commonly understood as essential, in particular feedback loops or reoptimization during (query) runtime adaptation were usually missing. The current state of the art in query optimization heavily relies on rules explicitly specifying how to restructure certain query patterns [HFLP89, PHH92] [4].

Similar approaches emerged for physical design tuning, workload management, and memory management [RS91].

A major drawback of fixed rules is their non-existing adaptation capabilities to different environments or changing environments. Although parameters, e.g., cost models, may be adjusted, the underlying rule will always perform the same adaption. While this lack of flexibility can be problematic, the decisive advantage is that rule-based approaches have proven to deliver reliable configurations even in unstable system situations [QSF+07].

Capturing best-practice and expert knowledge as a set of rules applicable to DBMS tuning knobs may be used to turn any system into a self-tuning system as long as a suitable infrastructure for rule application exists, i.e., a set of utilities or DBMS commands.

Fully integrated autonomic tuning, using guidance gathered through collecting expert knowledge and formalizing it, is presented in [WRRA08]. This approach builds on DB2 Performance

[4] Only the search for the best plan can be regarded as 'self'-tuning, but, nevertheless, it still relies on up-to-date statistics and cost models which partially became self-tunable throughout the years.

Expert[5] and proposes the architecture of an *Autonomic Tuning Expert* (ATE). Figure 3.3 shows the entire infrastructure of ATE. Similar to the feedback control loop presented in Section 3.2.2, ATE monitors events resulting from log analysis or online event capturing. The control component tries to find correlations and exploits the expert knowledge that is modeled separately.

In addition to the generic, system-independent expert knowledge, individual workload analysis and query analysis are used to extract information about the system's specific performance characteristics. This information is stored in the knowledge base and considered for future adaptations, that way providing a means to continuously extend and update the knowledge base.

3.2.4 Multi-Agents

Traditional computer systems are centralized, whereas today's large-scale computing systems are distributed across a, possibly very large, number of nodes. These nodes are often hetereogeneous and often have some degree of autonomy. To be able to handle the considerably increased complexity of optimizing and tuning such systems, it is common to handle measures relevant for the individual nodes locally, while only aspects relevant for the system as a whole are handled globally.

Local optimization covers aspects like data storage, buffering, and memory management, whereas global optimization attends to query routing, data distribution, security and safety issues, and workload balancing. The tuning components on the individual nodes are often referred to as agents; they operate autonomously and are proactive and goal-oriented to solve a common problem.

In [TCW+04], a prototype of a tuning approach for distributed systems called *Unity* is proposed that realizes autonomic system behavior by having goal-driven *self-assembly*, *self-healing*, and real-time *self-optimization* capabilities. Each *autonomic element* has its own high-level goals, e.g., response time or throughput rates, which need to be "mapped" to resource utilization. Therefore, utility functions are used to express a specific level of goal fulfillment (e.g., response time) reached by a given resource assignment. A so-called *arbiter* tries to cooperatively assign tasks, i.e., resources.

Another multi-agent system is *OceanStore* by [KBC+00]. Its goal is to provide reliable, scalable, secure data storage and access using a non-secure, non-reliable "ocean" of computers[6]. Although its primary design goals are not autonomic computing, it employs many mechanisms from the autonomic computing world to achieve its goals. Each site (i.e., server) is an agent in this ocean, the autonomy of agents allows them to fail anytime, which makes global optimizations fairly difficult. To improve performance and reliability, OceanStore continuously monitors usage patterns, regional outages, and denial of service attacks and reacts to them by proactively moving data.

Most multi-agent systems assume that all agents are peers with identical capabilities. Taking this concept to a single system by considering components as specialized agent, enables reusing multi-agent optimization techniques within a non-distributed system. For instance,

[5] http://www.redbooks.ibm.com

[6] This notion of a flexible computing system is known as cloud computing today.

each individual self-tuning feature is an individual agent. The obviously challenging part is to make them cooperate by defining common APIs, metrics, and goals.

3.2.5 Economical Models

In contrast to agent-based systems discussed in the previous section, where cooperation is necessary and assumed, in economics, the participants appear as competitors. A *market* is used to "sell" and "buy" desired goods, i.e., computing resources or computing tasks. The pricing is either guided by supply and demand via a brokering or auction processing.

So-called service providers announce their offers on a central market or broker. Service requesters can supply computing tasks and additional constraints such as processing time or costs. The broker (or market) is responsible for finding a suitable match, i.e., a member fulfills the task under the prescribed conditions. In computing environments, quality of service requirements play an important role [LNPM98]. Another goal is achieving an optimal resource allocation, i.e., economic operations, as [KS89] showed for file allocations.

In [WHH+92], a Xerox research system called *Spawn* uses costly simulation tasks to harness idle computing resources in a distributed network of heterogeneous computer workstations. Task priority requirements are used for payment and fairness of resource distribution was examined. The economical aspect deals with priorities and resource utilization to fulfill the tasks.

Communication costs and computation-site awareness play a central role in Berkeley's Mariposa project [SAL+96]. In those days, wide-area network (WAN) configurations had to observe data transfer volumes in the KB range. Other important aspects are site-specific load situations and processing capabilities. The autonomic members of the systems cannot be controlled by a system-wide cost-based query optimizer, instead only local query optimizers are embedded. Typical for a market-driven model, tasks can be put up for auction, so each site can bid for them and a central broker is assigning tasks to the winners. The task initiators (i.e., users) specify a cost budget in form of a curve, i.e., the more time is spend for processing, the less the initiator is willing to pay for it. Besides query processing and network impact analysis, Mariposa employs a storage management similar to [KS89] that trades so-called fragments of data. By observing historical access, future access patterns are anticipated and the fragment trading is a proactive optimization for improving locality.

Although the underlying market model of these approaches is fairly similar, its application is different due to divergent high-level goals.

3.2.6 Genetic Algorithms and Multi-criteria Optimization

From a more theoretical perspective, the problem of DBMS optimization can be modeled as a so-called multi-criteria optimization problem. Multiple criteria such as low memory consumption, high IO rates, and response time limits have to be fulfilled at the same time. However, some of them are contrary to each other, i.e., improving the fulfillment of one criterion reduces the degree of fulfillment for others.

Multi-criteria optimization problems are often solved or at least approached with genetic algorithms (GA). In a GA, tuning knobs and configuration parameters can be described by a

vector $\vec{x} = [x_1, x_2, \ldots, x_n]$. For each criterion c_i, its fulfillment is represented by an objective function f_j that receives the parameter vector as input $f_j(\vec{x})$ and can be used to minimize, maximize, or evaluate a certain $\vec{x}$. Usually, there exist several near-optimal solutions of $\vec{x}$ fulfilling the criteria to a similar degree, i.e., they do not dominate each other, which is also called *pareto optimum* [Coe00, BC08].

Applying weights to different criteria can facilitate the configuration search by reducing the number of pareto solutions.[7] The (weighted) goal value for each criterion c_j is specified by G_j. Now, each deviation of the objective function f from the given goal G_j can be calculated and leads to the following minimization problem:

$$min \sum_{i=0}^{n} |f_i(\vec{x}) - G_j|$$

The really challenging parts here are the specification of a feasible G for each goal and the specification of weights for each goal fitting the actual system and user demands.

Probably one of the most difficult problems in multi-objective optimization is determining how to measure the quality of a solution and to compare several solutions. For instance, ranking may help to order them according to their goal fulfillment. The overhead necessary to establish the solution is often omitted.

Game Theory

The foundation of *game theory* dates back to [Neu28]. Game theory typically analyzes two-player games, either in a cooperative or a non-cooperative manner. A game has several variables assigned to the individual players, whereas each player aims to fulfill its objective function, i.e., maximum or minimum goal. A player is only capable of altering its variable while taking the other player's decisions, i.e., variables as fixed values into account.

In game theory, most often *non-cooperative games* are analyzed where multiple players try to optimize their objective functions in a turn-based schedule [Neu28]. Many algorithms try to find a stable equilibrium condition, i.e., according to their objectives, players can not improve their fulfillment as long as the others stay unchanged – this principle is called *Nash equilibrium*. Although cooperative game theory exists [Rao87], its application is similar to multi-agent approaches and when generating multiple equilibria, it easily turns into a multi-criteria optimization problem.

Eventually, game theory is partially similar to traditional ways of modeling self-management but has various strengths in employing player roles, analyzing reachability, and dealing with non-cooperative games. In our DBMS-oriented scope of self-tuning and self-management, these paradigms do not fit that naturally, for which reason we do not consider game theory in more detail.

[7] Skyline queries are a mechanism to find those pareto situation when faced with multiple objectives.

3.2.7 Languages

As we have mentioned in the previous sections, specifying constraints and goals is not always an easy task, however, some language support may improve this situation. Such a language can be used to describe and define a system, express constraints, service agreements, and even goals [KC03].

There exist a lot of architecture description languages (ADLs) and notations that support dynamic architectures. Some of them even support formal architecture-based analysis and reasoning. However, most of these languages emerged in the software architecture domain such as [MRT99]. Ponder [DDLS01], which has its origin in the area of networking, is a language that defines pre-specified actions (so-called plans) triggered by (complex) events (i.e., state changes of the underlying system). These simple *event-action* handlers can also be found in OceanStore's [KBC+00] event handler mechanism written in a domain-specific language.

A separate language domain is formed by so-called requirements description languages (RDLs). [GKW04] presents a combination of the ADL and RDL approaches into what they call an architectural prescription language (APL) and suggest its use as a *reconfiguration language* for systems. More sophisticated approaches extend ADL with feedback and change mechanisms to build self-adaptive (software) systems [BMM+04]. System components are modeled by behaviors (e.g., collaboratively and hierarchically) and employ a communication mechanism (send, receive).

Besides software architecture languages presented so far, tailored DBMS-optimization languages exist, too. For instance, in [BC08], a constraint language is introduced to express physical design tuning constraints with *Assertions*. Its usage is similar to SQL, because it includes similar constructs like aggregations, filters, generators (for binding), and nesting. Using a formal model, an SQL-based language to describe view and index configurations is presented in [BC07]. It includes special transformation operators such as merge and reduce, and it allows to derive minimum as well as closure properties of a configuration.

3.2.8 Summary of Existing Approaches

While few proposals dedicated to the field of self-tuning in DBMSs exist, our survey of the literature yielded many concepts that could be applied to this area, too, i.e., which component or what kind of tool to use.

Table 3.1 lists the most promising techniques and summarizes their essential properties for use in the self-tuning of DBMSs. For each approach, we specify whether or not it is suitable for *online* tuning, can be made *multiuser-aware*[8], the DBMS components it can be applied to, and the operational goals it can optimize. Although many of the techniques are promising, some can be ruled out due to their inapplicability for multiuser scenarios – the typical use case for a DBMS.

Another challenge is that most techniques address only individual components of a DBMS. To successfully combine them, their dependencies have to be known to exploit their individual strengths and avoid their weaknesses and prevent them from interfering with each other. In

[8] Unfortunately, multiuser capabilities are often neglected or not tested, when self-tuning techniques are proposed.

Table 3.1: Self-tuning areas and goal contribution capabilities

Technique (related work)	**Online**	**Multiuser-aware**	**Components**	**Operation goals****
Design advisor [SGS03, ZRL+04, ZZL+04, PDA07, BC08]	–	–	workload, storage	response time
Workload models [MEW06, HR07, SH08, EM09]	/	–	workload, scheduler	throughput
Self-tuning memory [SGAL+06]	+	/	memory pools	throughput, resource usage
Buffer tuning [THTT08, BS11]	+	+	buffer pools	throughput, memory usage
Statistic management [CN01, SLMK01, AHK+04]	+	/	query optimizer	response time
Index selection [CN97, VZZ+00, LSSS07, EAZ+08b, SH10]	–*	–	index configuration	response time
What if analysis [CN98, DTB09]	–	–	physical design	throughput, response time

– not available + available / unknown but possible

*) online recommendation available, but no online management (decision, creation, deletion)

**) none of the techniques directly addresses costs or energy consumption, but most of them can be extended for that

the next section, we will present some insight into DBMS components and their self-tuning dependencies.

3.3 Dependencies

Typically, a (database) system is a composition of several components interacting with each other, i.e., calling functions, sending and receiving data. A self-tuning feature in a DBMS can now be considered as yet another component. However, while the other components are isolated from each other as much as possible, e.g., by following a design like the common 5-layer architecture (cf. Section 2.2), a system-wide self-tuning component must inevitable know about the implementation details of the other components. However, this poses new problems such as error propagation or interlaced dependencies, which may also lead to cyclic dependencies. Therefore, we believe that self-tuning enabled components should be linked based on the existing cost model concept, i.e., resource consumption and timings may be fed as additional inputs to any component, which allows to integrate this information into self-tuning decisions.

Of course, dependencies exist between components, either direct or indirect. Those dependencies may constraint resource usages or block processing (e.g., due to synchronous waits), influence parallelism, etc. Further, while the common components interact with each other more or less top-down along the layers triggered by a user request, self-tuning actions operate independent of the system's normal operation or other self-tuning actions. Therefore, we

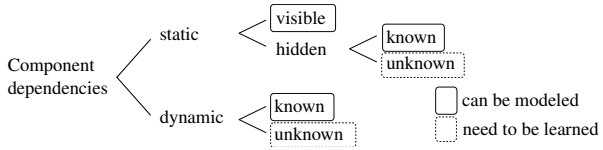


Figure 3.4: Classification of component dependencies

distinguish between (software system) component dependencies and the self-tuning dependencies.

3.3.1 Component Dependencies

Component dependencies usually arise by moving data or by invoking functionality across component borders. Our classification in Figure 3.4 shows the different kinds of dependencies. At the top level, we distinguish between *static* and *dynamic* dependencies.

Static Dependencies

Initial modeling must comprise all the statically visible component relationships, such as memory allocation for data structures where the size is known or CPU overhead for deterministic algorithms, whenever functionality of one component calls another one (i.e., a new client session initiates buffer allocations, or the query processor's operator requires CPU cycles for sorting and disk access is necessary).

But there are also hidden correlations that can be revealed by doing experiments or by reasoning (transitive dependencies). Those dependencies are hard to model because they are often conditional, i.e., their existence may vary. For instance, additional indexes are used to speed up query processing, however, contention and locking may be increased when indirectly accessing data values first via an index. Here, lock escalation interferes with query planning assumptions.

Dynamic Dependencies

Dependencies that are not always present must be modeled differently. For instance, when a (self-tuning) feature can be turned on or off, or an alternative implementation for a component/algorithm is selected (e.g., different lock granularities, swap-enabled algorithms vs. non-swapping algorithms, non-blocking eager strategies vs. blocking strategies), the dependency can vary from non-existent to strong. Still, if such a dependency is known, it can be modeled as optional. In cases where such a dependency is not known in advance, inference at runtime is the only way to capture them in the model. Dynamic dependencies may also occur for a short time through misinterpretation of external factors. They can cause the entire model to fail, e.g., if high system loads caused by external programs or administrative intervention are misleadingly dedicated to a prior parameter adjustment.

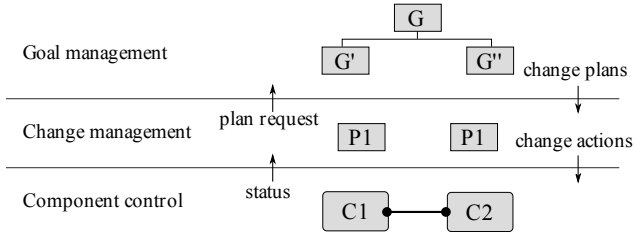


Figure 3.5: Self-management system architecture [KM07]

Architecture for Dependency Modeling

A coarse-grained view on a complete system and its components for self-tuning, based on abstraction and generalization, is given in Figure 3.5. Although the approach [KM07] targets at the feedback-control flow in a goal-oriented self-tuning architecture, it already indicates that components are correlated with each other. In order to analyze and thereby consider dependencies, a model of interacting components is required, which is out of scope for this work.

3.4 Online Self-Tuning Challenges

Enabling self-tuning mechanisms within a DBMS is not necessarily beneficial in all cases. The additional overhead, wrong assumptions or estimations made may even downgrade system performance. In the following, we present some challenges and how we limit their negative impact.

3.4.1 Search Space

Self-tuning is trying to explore the search space of feasible system configurations to find one that better suits the current system use. As motivated in Section 3.1.2, the search space is fairly large in nowadays DBMSs.

One way to reduce the search space is to quickly prune impossible or bad “search areas”. As a first measure, impossible configurations (i.e., configurations that exceed parameter ranges, have contradicting or incompatible setting etc.) can be removed from the search space. Another measure is to skip the analysis of entire configurations and variations if the already fixed part for those would already result in a performance degradation. Therefore, the expected performance figures can be mapped to a cost metric making configurations directly comparable. In [BC05], a relaxation-based approach is proposed to incrementally improve a configuration by taking certain constraints into account. Using this technique, redundant analyses are limited and, thereby, the search effort is reduced but not necessarily the search space.

Most offline tools for physical DB design tuning limit the maximum search time for a configuration. Here, two things have to be considered: first, search should permanently find equal

or better configurations. Second, one has to determine when to stop the search, for instance, by declaring a quality measure to control this decision[9].

Bringing the search time-based pruning to the online self-tuning world, puts additional emphasis on the search effort itself. The overhead of self-tuning (i.e., monitoring, analysis, actions) has to be compensated by its benefits. Especially, analysis, i.e., the question how often to do a re-evaluation and adaptation of the system configuration and how long this is allowed to take, has a significant impact. Later, we will show that the frequency of analysis steps can also be adjusted to the probability of successful improvements.

3.4.2 Prediction Quality

Probably all self-tuning mechanisms intend to increase the performance of future processing. Based on historical information, which is often weighted in a step-wise manner, i.e., the more recent history is weighted higher, *assumptions* about the future are made. But not all input parameters – see Section 3.2.2 – remain stable and unforeseen inputs may reduce the expected performance increase. Thus, it is important to not only consider the configuration with the maximum performance possible but also evaluate its likelihood. Moreover, a lower bound estimation for worst-case situations may limit the risk of negative configurations at all.

Two areas are relevant for predictions: workload model and cost model. Workload models are used to deal with repeating workloads or certain patterns in the workload [MEW06]. These models distinguish between certain types of workload, e.g., OLTP and OLAP, and relate them to a timeline. Some approaches also model dependencies between initial workloads, (internal system) consequences, and future workloads, e.g., using Markov chains as in [HR07]. A very common example that is used to motivate workload shifts, which can be predicted, is to distinguish between working hours (having a lot of OLTP load) and night hours when only batch jobs process large volumes of data (OLAP-like). A very simple approach is presented in [EM09], where the authors assume repeating workload patterns as a mandatory requirement for any reasonable form of predicting tuning measures.

Cost model predictions deal with internal system costs for certain tasks. Here, the workload is taken for granted and processing costs and behavior are predicted. A lot of different approaches exist that use either histograms, neural networks, or more unusual techniques for cost modeling [HLS05]. But in principle, all of them rely on simple cost accounting, data volume statistics, and weights for the cost producers that can be adjusted dynamically.

Similar to Section 3.4.1, frequency, duration, and history span inspected are crucial parameters to the prediction itself. Most important is, as indicated earlier, that the recent history may be more significant and thereby more suitable for predictions. The challenge is now to identify the optimal time spans used for prediction and the proper weighting of more current against more recent data.

Evaluating the prediction quality is necessary to improve it by adapting the tuning model, i.e., workload model and cost model.

[9] For instance, within 5 minutes find a better configuration, but each 30 seconds the prior configuration has to be surpassed by 10%.

Eventually, predictions are a crucial part of self-tuning when faced with dynamic environments and changing workloads. However, prediction quality, overhead, and chance of even degraded system performance have to be considered all the time.

3.4.3 Delay Effects

Tracking the effects of self-tuning actions is important to obtain "feedback" regarding prediction quality and actual degree of performance change. As mentioned above, a particular difficulty is to correctly attribute the observed changes in system performance and behavior to the different configuration changes that were made. However, the effects of a specific tuning measure are not visible immediately. Consequently, tuning guides for databases recommend to only change one parameter at a time, wait for the change to show its effects, assess them, and either correct the setting or proceed with the next parameter, until optimal performance is obtained. However, for many situations, e.g., with variable system load, such an approach is not always feasible. System performance demands for immediate action that often involves changing many different configuration parameters. The challenges now are to identify tuning tasks that can safely be performed in parallel and to correctly observe their individual impact.

Identifying *parallel tuning actions* may be alleviated by knowing dependencies between tuning knobs. What a human expert knows by experience can be provided to a self-tuning mechanism by a system model. Thus, self-tuning may explore variations of tuning measures or complete configurations [DTB09] and reason about them.

The second challenge, *delay effects* of tuning actions consists of two aspects. A change with an overall positive impact may momentarily degrade system performance, or its effects may not appear immediately but only after a considerable amount of time. Therefore, changes should not be discarded as inappropriate too quickly, solely based on the first observations. Adding further to the problem, unanticipated workload shifts or parallel tuning actions by a human expert may additionally distort the identification and attributation of performance changes, making self-tuning evaluation a considerable challenge.

3.5 Self-Tuning in DBMSs

All major DBMS vendors have integrated self-tuning features into their recent product releases. Most of these products evolved over several decades, which led to the inclusion of literally thousands of configuration parameters. This enormous degree of customizability allows them to be fine-tuned to specific environments. More and more of these parameters are made accessible to online (self-)tuning to improve system performance and to make the life of DB administrators easier. In this section, we will survey the self-tuning capabilities of existing commercial systems and research projects in academia. Besides their online tuning capabilities, we examine which tuning areas are addressed.

3.5.1 IBM DB2

DB2 supports physical design decisions by the *DB2 Design Advisor* [ZRL+04, ZZL+04], a component that exploits the query optimizer to evaluate multiple alternative configurations. This tool operates offline and requires weighted sample queries and constraints fed into it. In the end, it produces recommendations for indexes, MQTs, clustering, and partitioning.

Essential for query optimization are statistics, which can be gathered autonomously and validated using LEO (Learning Optimizer) of DB2 for LUW (Linux Unix Windows) [SLMK01, AHK+04]. Cost-model-based assumptions of query processing timings are compared to actual runtimes and, in case of deviations, the basic cost model or cardinality estimation feature of the query optimizer are adjusted.

Many other often isolated tools are developed by the SMART (Self Monitoring And Resource Tuning) project [LL02], which is the database part of IBM's *autonomic initiative* [Hor01]. Although the project goal is to develop system-wide self-* capabilities, so far, the different tools emerging from SMART are not integrated with each other and are thus operating independently.

The probably most important online self-tuning feature of DB2 is the STMM (Self-Tuning Memory Manager) [SGAL+06]. Dynamic memory pools in DB2 can be put under the control of STMM, for instance, buffer pools and query caches. Based on control theory, in particular, multi-input multi-output (MIMO) controller and oscillation dampening (OD) controller, STMM tries to estimate time savings if additional memory is made available to different buffer pools and caches. Iteratively approaching to an optimal memory distribution, STMM transfers memory portions based on a cost-benefit decision supported by a control model. Thereby, STMM achieves fast convergence times, a rapid adaptation to changing environments, and stable response in case of noise. STMM can be used to operate beyond database instance boundaries.

3.5.2 Oracle Database

Similar to IBM's Design Advisor, Oracle Database delivers the SQL Access Advisor for physical design tuning. The only difference to IBM's tool is that Oracle allows to integrate the SQL Tuning Advisor. This advisor can analyze query plans and statistics to recommend DB index structures, statistic changes, or SQL changes and creates so-called SQL profiles for reuse.

Oracle also includes *Automatic Shared Memory Management* (ASMM). However, ASMM capabilities are less profound than STMM's, because ASMM operates with fixed maximum boundaries and requires instance restarts for changes taking effect. Further, sort pools are not covered at all.

Performance monitoring is a strong point of Oracle's DB because the Automatic Workload Repository (AWR) gathers performance statistics and the Automatic Database Diagnostic Monitor (ADDM) is a DB-wide performance diagnosis tool [DRS+05]. It builds a classification tree of various DB timings (accumulated in the tree) using a wait model and a time model.

3.5.3 Microsoft SQL Server

Microsoft research is putting an emphasis on physical design tuning [CN97, BC05, BC08] and ships its Database Tuning Advisor. Compared to DB2 and Oracle, it allows for more space and time constraints while analyzing the configuration offline. It employs so-called "what-if" indexes. Tuning Advisor recommendations can be automatically implemented.

The entire *AutoAdmin* project is based on the "what-if" approach [CN98] and paved the way for further self-tuning research by Microsoft. For instance, autonomic statistic gathering and management was developed [CN01].

In its current incarnation, the memory management is fairly basic. For several memory pools such as buffers, connection contexts, logs, procedures, etc., the minimum and maximum memory sizes can be changed, which are respected whenever memory is (de)allocated on demand.

Similar to DB2's and Oracle's monitoring capabilities, SQL Server also contains a Physical Design Alerter indicating performance bottlenecks.

For all commercial DBMSs presented, those performance alerts can automatically be processed and even some actions can be triggered; at least notifications are send out to a DBA.

3.5.4 Academia

The academic research in self-tuning is often confined to a subset of a DBMS and most often is limited to explore their ideas on their own prototypes or existing open source systems, with PostgreSQL[10] being the most wide-spread. A lot of research is done in the area of data placement [MD97, LKO+00, YAA07] and index self-tuning [SGS03, RPBP04, HKL05, SAMP06, LSSS07], i.e., physical design [PDA07]. Another well-studied area is buffer management self-tuning [NTA05, THTT08, BS11], which however is only a part of the whole memory management that is addressed by commercial self-tuning mechanisms.

Many different approaches exist for modeling and predicting workloads and their changes [ACN06, MEW06, HR07, EM09]. However, most of them are limited to predict user input happening next (i.e., load, query "size", and type) or classifying workload types but fail to give viable tuning hints.

Eventually, a kind of brute-force approach for configuring DBMS parameters, as in [DTB09], seems to be the only project that is able to address the tuning of all aspects of a DBMS at a time.

3.6 Self-Tuning Framework in XTC

In this section, we show the steps and extensions that are necessary to enable our XDBMS XTC for MAPE-like self-tuning features. To lay the foundations for developing, implementing, and evaluating them, we have to extend XTC's basic functionality with suitable *monitoring* and *analysis* functionality.

In this process, the already existing "traditional" database monitoring and configuration management has to be expanded to cover the complete DB software system. Additionally,

[10] `www.postgresql.org`

a cost model is required that addresses each component of the system and make their resource use and runtime behavior comparable to establish a foundation for reconfiguration decisions, i.e., *planning*. In the following sections, we introduce our approach to provide such a system-wide monitoring facility for XTC and further aspects for a Java-based DBMS like XTC. We show how we exploit existing technologies and how we deal with their inherent flaws.

Based on the monitoring facility, we then present our analysis framework for cost-based tuning decisions and how to realize it in a native XDBMS. The remaining aspects of our MAPE approach for XTC are partially addressed in this section and intensely covered in the following chapters.

3.6.1 Monitoring in XTC

Usually, there are two reasons for monitoring a database system. First of all, performance aspects and runtime behavior observations reveal information about the system's inner state and allow to identify potential bottlenecks. As a second aspect, constant monitoring of the system is necessary to ensure security, safety, and integrity demands. However, this work only considers performance-critical issues.

Monitoring has to be lightweight, i.e., avoid unnecessary overhead, which is usually directly related to the amount and type of data being collected as well as the frequency of monitoring.

Starting with simple change detection or threshold and counter parameters, each component of the system is enabled to send *events* to a specific monitoring component where they are *dispatched* to be collected or processed immediately. The event receivers not necessarily reside on the same machine as the DBMS backend. As a result, the increased communication overhead and network latency have to be taken into account. One way to reduce this overhead is to employ a UDP-like send process, i.e., events are sent asynchronously, without their reception being acknowledged by the receiver. While this allows events to be lost, such an approach is favorable for non-blocking event emissions as long as enough information is received. In contrast, synchronous event processing is costly but guarantees to capture all events.

All the monitoring spots can be enabled or disabled during runtime. Events can be classified according to two orthogonal dimensions, the component the event pertains to, and the type of information conveyed by it. Thus, we have the following event classes:

- **Parameter Change**: Based on thresholds or value change, an event encapsulating the old and new value of a parameter is emitted.

- **Statistic**: Statistic events are sent periodically or when requested by an event receiver. For instance, the current fill level of a specific memory pool is frequently communicated.

- **Message**: Internal processing situation or actions, which can be important to notice. Different from the event types before, messages need not carry any concrete payload other than their actual type. For instance, a client connection is established or a rollback was initiated are typical messages.

and the following component identifiers:

- **INFO**: This is not really a component identifier, it is more a kind of debug level identifier having informational character.

- **MEMORY**: Because a lot of different memory areas are employed in XTC, their current states (i.e., load) and sizes, often in relation to the total memory available, are denoted by this identifier.

- **BUFFER**: Although buffer pools are memory areas, too, their characteristics are special in a way that it is beneficial to have a distinct identifier for them. For instance, hit and miss ratios or replacement algorithm counters are different for simple memory pools such as for sort memory, connection contexts, or lock area.

- **AI**: This abbreviation stands for *Auto Indexing*, which identifies all events related to the autonomous indexing framework of XTC, presented in Chapter 7.

- **SERVER**: All events that cannot be assigned to a specific component are covered by this system-wide identifier.

This classification may be refined in the future and presents only our current state in XTC's event modeling. Each event also carries a unique *eventID* that is necessary for automated event handling, i.e., dispatching, grouping, filtering, etc.

As soon as an event is created, a timestamp is automatically attached to this event, depending on the dispatcher or receiver.

Synchronous event processing is only allowed for internal receivers, i.e., DB system components or tuning components due to its blocking character. In contrast, all events send via network are processed by a simple FIFO buffer queue, which may drop events if the registered receivers are inactive.

Clients and components can register for certain events by specifying a filter (e.g., class, component, eventID) and listen for incoming events.

There is no specific distinction between internal and external events, however, most often configuration-based events (e.g., workload, storage layout) are processed internally, whereas alert-style events (e.g., index creation, buffer threshold) are often monitored from the outside.

Component Model

Every component in our database system carries specific properties. These properties may *describe the state* of a value (e.g., high/low watermark or current IO and memory usage statistics), *indicate a change* (e.g., the start or end of a function), or *count a value* (e.g., time period such as execution time or event frequencies such as query incoming ratio).

Monitoring values are preprocessed by a component, before an *event* is send to the central monitoring or dispatching service. Besides this pushing mechanism, every component can return its memory statistics and CPU timings on request.

3.6.2 Analysis in XTC

With the monitoring data collected from the individual system components, it is now possible to perform a detailed analysis of the overall system state. However, the typically huge amount of collected data requires aggregation or filtering to further limit the number of events that can be kept for analysis. Simple predefined rules can automatically trigger actions to reconfigure the system.

Cost Model

The internal cost model works on a component base and aims at predicting the cost to reconfigure the system, i.e., component(s), compared to the estimated benefits. System resources covered by the cost model are memory consumption, IO cost, and CPU cost.

Although the computational overhead for the cost model is usually low or negligible, due to the pruning step of the state space reduction, it has to be considered when building the cost model. Moreover, the effort and cost for the monitoring itself have to be taken into account, too. Obviously, the total cost for monitoring, analysis, and cost model calculation must be kept below the benefits expected from the self-tuning facility.

As Java does not directly provide comparable CPU usage statistics in the manner of a UNIX-style process accounting, we have to transform the given CPU times. As input we use the detailed information about runtime and blocking time the JVM provides for each thread. In addition, threads may be blocked due to scheduling displacement or while waiting for IO operations. However, measurements for these aspects are not provided directly by the JVM, but are an important input for our cost model and must therefore be acquired differently:

Based on the recorded CPU runtimes, we can compute the proportional CPU time of every single thread by taking snapshots periodically for all active and passive threads. Combined with the number of processors and elapsed system time, a per-thread CPU-usage can be calculated.

Besides CPU usage, heap loads of various threads are also collected when taking snapshots. Together with our event-based IO start or stop notification, a complete cost model for all database components is achievable.

Before a new system configuration is put into effect, the cost model must also provide an estimate about the cost for the reconfiguration. For instance, the redistribution of memory between components produces costs that need to be accounted for the reconfiguration decision. Furthermore, to avoid a thrashing of the reconfiguration process, i.e., a permanent reconfiguration of the system, it is advisable to set a certain threshold which the expected benefits of the reconfiguration have to surpass before a reconfiguration is actually applied. Naturally, this threshold should be at least as high as or higher than the cost for the configuration change.

3.6.3 Plan and Execute in XTC

XTC's self-tuning features (i.e., controllers) will be presented throughout the following chapters, which implement the *P*lan and *E*xecute parts of MAPE. The controllers *decide* which action(s) should take place to *reconfigure* the system. The entire loop remains active for the complete time a self-tuning feature is activated. Note, there is no explicit central *knowledge*

repository defined as the MAPE-K model recommends, because each tuning component keeps its own (history-based) knowledge base, if necessary.

3.6.4 Implementation Aspects for MAPE

First, we sketch some basic considerations for Java-based monitoring options.

A common method to obtain CPU usage and IO waiting times is to use a profiler or debugger tool. However, these tools are strictly meant for use during development, as they incur a significant performance overhead, are not flexible enough during runtime, may prevent internal Java tuning (e.g., JIT), and may also interfere with parallel thread processing.

Obtaining detailed monitoring information is further complicated since the JVM on top of modern operating systems such as Windows and Unix-like systems does not directly expose the kernel scheduler nor does it provide ways to obtain accurate per-thread CPU times.

To implement monitoring in XTC, we naturally aimed to use the existing Java APIs as far as possible. However, for some challenges, existing APIs did not provide the required functionality or had other significant drawbacks.

JVMTI and JMX

Application developers rely on those tools to analyze algorithms for errors and potential bottlenecks. Complex systems, such as DBMSs, require sophisticated tools to analyze their components and effects on each other. Thus, the JVM Tooling Interface (JVMTI) is useful for lightweight profiling of several components. One of the new features in the context of JVMTI is the byte code instrumentation (BCI). With some limitations, it allows to change compiled code during runtime or to pre-execute commands before class (un)loading. The most severe restriction, however, is that to gain useful results, a deep understanding and knowledge of the internals is needed. One needs to know where to instrument your binary code and this is obviously not always wanted or possible.

To overcome the drawbacks of JVMTI and to provide high-level monitoring interfaces, the second API – JMX (Java Management Extension) – was developed. Basically, Web systems and distributed applications should benefit from JMX, due to its external instrumentation. To allow an application to be monitored and managed via JMX, the application developer has to provide so-called MBeans[11] or MXBeans for complex data structures. These application-provided MBeans can now be registered by an MBean server. This server handles all incoming requests for monitoring or instrumentation from external monitoring or management clients and calls the pre-specified MBean methods. Standardized tools can explore a complete M(X)Bean interface and dynamically invoke their methods or adjust their values.

The JMX architecture is based on a poll mechanism, which means that the monitoring client has to periodically query the state and changes. The polling paradigm is augmented by a simple notification extension enabling the M(X)Bean component to send messages to clients. However, the client still has to poll the message queue, which can reside on another machine. The benefit of this approach is that it avoids blocking the production system.

[11] MBeans are Java Beans used for monitoring purposes but also for control.

JMX aims to be used as *the* management interface of any Java-based software system. The main features target the code instrumentation from outside. That means, with code substitutions at the level of JVM byte code or with pre- and post-method calls, the behavior of the running software could be influenced. But the MBean is not comparable to a manager of a database system. Usually, an MBean is a setter/getter Object with public methods. When calling these methods from the outside (i.e., management client), the caller is not aware of mandatory conditions (transactions). Therefore, we will restrain the usage of MBeans to the read-only part of an external monitoring component[12].

Thread Monitoring

As modern computer hardware increasingly obtains its performance gain from increasing the number of CPU cores available, instead of increasing the speed of the individual core, multi-threaded software is becoming a necessity to benefit from these improvements. Consequently, a monitoring solution must be able to cope with a massively multi-threaded environment.

In Java, thread monitoring comprises of the recorded CPU times (active runtime) and the thread states provided by the JVM. Here, we can exploit JMX's platform MBean to query thread-based CPU times and states. With a simple method call, the accurate CPU time (given in nanoseconds) each thread has used can be obtained.

In contrast, memory consumption is stated for the whole VM and not per thread. Still different JVM memory pools can be evaluated individually. To obtain memory usage statistics per thread, some additional efforts are necessary. We therefore monitor instances and sizes of data structures when they were created in form of new objects. Arrays, as they are commonly used either directly or indirectly, encapsulated within another data structure, are easily accounted. For more complex structures, some kind of low-overhead heuristics is needed to determine their space consumption. With this approach, we are able to obtain fairly accurate memory usage data per individual thread, in addition to the CPU usage provided directly by the JVM. Note, thread-based monitoring does not directly provide component-based resource statistics necessary for self-tuning of individual components.

3.6.5 Logging and Reporting in XTC

The overhead implied by JMX and the polling-based usage of the MBean API prevents its application within the XTC system. There we need thread-based [13] cost information. Moreover, JMX/JVMTI does not provide any information about IO blocking time that is mandatory for a complete cost model.

[12] This monitoring component is a visual aid for database administrators or users to visualize internal load states. Due to the loose coupling of this client, we prevent interaction with the backend.

[13] In the XTC system, so-called jobs (single tasks) were enclosed into a thread. Thus, multiuser access is mapped to a multi-threaded application.

Reporting API

In XTC, we have a system-wide event reporting that allows pushing events to registered event listeners, typically clients or tuning components. A major design goal was to specify simple and clean interfaces that allow us to easily integrate new monitoring options, i.e., a new type of system events.

The basic interface encapsulating events is depicted in Listing 3.1. Event type and originator, (i.e., component) together with a unique name or eventID must be specified when generating new events.

Listing 3.1: Reportable event interface

```
interface Reportable<T> extends Serializable {
  enum ReportComponent {AI, BUFFER, MEMORY, INFO, SERVER};
  enum ReportType {Parameter, Statistic, Event};
  String getName();                //unique name of metric to report
  ReportComponent getComponent();
  ReportType getType();
  T getEvent();
}
```

Components or internal XTC tools may implement the ReportProvider interface, shown in Listing 3.2. This interface is required when you want to query possible events, similar to the MBean approach of JMX, which can be used for event polling.

Listing 3.2: Report provider interface (Poll Mode)

```
interface ReportProvider {
  List<Reportable> getReportables();
  Reportable getReportable(String name);
}
```

The XTC kernel has to instantiate at least one event broker, which is called STReporter – *S*elf *T*uning *Reporter*. Its interface is shown in Listing 3.3. Any internal or external client may register at this broker and can optionally provide an event filter. Global event control (i.e., for all connected clients) is possible via the *enable()* and *disable()* methods, which switch on and off the event emission for entire components or certain event groups. The STReporter employs an event buffer, does the dispatching, timestamping, and filtering for all event sources and destinations using this central hub.

Listing 3.3: Self-tuning reporter interface

```
interface STReporter {
  void addEvent(Reportable<?> event);
  String start(String clientURL, EventFilter filter);
  void stop();
  String getStatus();
  boolean enable(String featureName);
  boolean disable(String featureName);
}
```

For the client side, a simple interface, consisting of two methods, needs to be implemented to receive XTC events and arbitrary string messages. The two methods are shown in Listing 3.4.

Listing 3.4: Reporting client interface (Push Mode)

```
1 interface ReportClient extends java.rmi.Remote {
2   void sendString(String event) throws RemoteException;
3   void sendEvent(Reportable<?> event, long timestamp) throws RemoteException;
4 }
```

From everywhere in XTC, a static reference to *STReporterImpl* that implements the *STReporter* interface can be used to generate a new event. For instance, the following line of code is sufficient to send multiple query timing information at once:

```
STReporterImpl.getInstance().addEvent(new ReportEvent<long[]>(ReportComponent.
    INFO,"MetaDataMgr.queryTiming", queryTimings));
```

Observe how the use of Java Generics allows us to provide events with virtually arbitrary data types as payload. The only requirement is that the data type must be serializable (i.e., implement the java.io.Serializable interface).

Logging API

XTC uses the established log4j[14] API for logging services. Property files are required to configure designated logging spots within XTC. For instance, the following example will create a log entry for query cost statistics, if an adequate log appender matching the package pattern and log level is enabled:

```
if (log.isInfoEnabled()) log.info(" query costs: "+(int)(qCost)+" time:"+qTime);
```

While log4j cannot be used directly for our monitoring extensions based on *STReporter* and *events*, it is straightforward to implement an event listener to do the actual logging. An event listener, i.e., client, can easily filter events using regular expressions or named event classes/components to dump them to a given log file. A sample client is provided together with XTC, called *STReportClientConsole*, which furthermore allows to remotely connect to XTC (i.e., register event listeners) and generates user-defined log file formats.

3.7 Challenges and Opportunities

This work is targeting the challenges imposed by self-tuning a native XML DBMS. Therefore, we demarcate the important and interesting areas, which are promising in terms of self-tunability for XML data processing. According to the layered DBMS model (shown in Figure 2.6 on page 21) and our prototype XTC, we will describe our approaches between layer L2 and L5 step by step. We restrict ourselves to these layers as the operating system and file management (layer L1) as well as user interfaces (above L5) build the "natural" barrier for our tuning considerations. However, the characteristics of these layers, e.g., the available IO performance, or the kind of user queries and the actual workload the system is subjected to, will be discussed where they have implications for our own work.

[14] The Jakarta log4j project. `http://jakarta.apache.org/log4j/`

Buffer

The first and bottom layer that lends itself to self-tuning is the DB buffer. Because here XML data structures are not visible, the techniques presented for buffer tuning are also applicable to relational and other systems employing multiple buffer pools. In Chapter 4, we will show the fundamental concepts for page-oriented buffer management, before introducing several (standard) replacement algorithms. Based on these algorithms, we will extend them to improve buffer resize options and decisions. The impact of optimal buffer configurations will be demonstrated with performance measurements for various IO patterns, buffer sizes, and replacement algorithms. Autonomous memory balancing for buffer pools requires cost metrics and a decision model, which will be presented as well.

Storage

Storage options for XML data will be the focus of our work pertaining to layer L3. Beginning with XML document mappings issues, Chapter 5 presents our approaches for efficient XML node labeling and encoding. Furthermore, compression techniques for XML structure and content play an essential role in reducing space consumption and thereby IO. Because some of these features can be (self-)tuned, we explore possibilities for the XML domain, which we also have realized in XTC.

Secondary access paths are important for (X)DBMSs. We have implemented standard index options and developed our own additional index features to improve the secondary access path selection. Customization of different index types helps to reduce space consumption and maintenance costs while improving query support by orders of magnitude. In Chapter 6, XTC's index options are presented and benchmarked as well as fundamentals for XTC's cost-based query processing.

Query-driven Index-Tuning

The most advanced self-tuning feature in XTC is our autonomous index management, presented in Chapter 7. It is operating on top of the storage layer and all index options we have in XTC. This feature heavily exploits XTC's query optimizer for cost-based index decisions. Thus, index maintenance costs and query benefits are evaluated while keeping an eye on space constraints. The impact and gain of query-driven index-tuning will be proved with experiments running industry-standard XML benchmarks.

Interplay of Self-Tuning Features

Bringing individual techniques and evaluations presented throughout this thesis into a single scenario is the aim of Chapter 8. For a series of workload shifts, the effects and the interplay of our indexes self-tuning and buffer self-tuning are analyzed. With the help of this analysis, we will discuss the next steps of XDBMS self-tuning and towards an integrated system model supporting cooperative (X)DBMS-wide self-tuning in our outlook, in Chapter 9.

Chapter 4

Buffer Tuning

One of the most important and critical aspects of the storage subsystem in a layered DBMS is the buffer management layer controlling the buffer pool configuration. Being the IO interface to external devices, this layer needs not only to be tailored to the available resources but also to the (expected) workloads, data, and data volumes. In Section 2.2 and 2.5, we have shown that typically multiple buffer pools are assigned to multiple storage containers. Thus, the distribution of the available main memory is extremely important for the entire IO performance. Changing workloads or requirements to the IO subsystem, different device characteristics, and new data objects or increasing data volumes may frequently degrade the buffer performance where only online reconfigurations may relief those penalties.

Self-tuning the buffer configuration can be approached from two perspectives. On a fine-grained basis, the *replacement algorithm* itself may adjust its configuration, which aims at optimizing a single buffer pool. Whereas *buffer pool sizing* requires a sophisticated approach to control the memory distribution for all pools.

In this chapter, we present basics of database buffer management, show important aspects for self-tuning capabilities, and present our approach for *performance forecast* and for *dynamic management* [BS11, SB11] in Section 4.3 and Section 4.4, respectively. The chapter concludes with an evaluation, demonstrating that simulation-based self-tuning of buffer configurations is beneficial for various workload scenarios.

4.1 Buffer Management

Efficient XML data processing using an XDBMS requires the same smart buffer management as relational systems do. Although query processing and the data model are totally different, the fundamental interfaces – buffer frames and disk pages – are identical. Only streaming systems such as [PC03, YLL+07] may benefit from XML-tailored buffer mechanisms[1]. Therefore, our buffer techniques are not restricted to the XML data model. We describe essential characteristics and parameters necessary for (self-)tuning options, which are presented in the remainder of this chapter and in Chapter 5.

[1] XML (sub)trees of varying sizes may require to be completely buffered to allow analyses, which is typically not the case for relational data operating on a (fixed-size) tuple base.

4.1.1 Working Principle

Basically, a buffer reduces the number of IO operations by caching external pages in its assigned portion of main memory. The major goal is to minimize external page *fetches* by exploiting the principle of locality [EH84]. Whenever a (logical) database page is requested, the buffer searches through its cached frames (i.e., pages) for it. In case of a hit (i.e., page found), a transaction may "lock" the page with a *fix* until the page is not needed anymore and release it by invoking an *unfix*. If a new page needs to be allocated or a request fails (i.e., miss), an empty frame is required to load the external page. If there is no empty frame available, a page needs to be flushed in case of modifications (i.e., dirty pages); otherwise, unmodified and non-fixed pages (i.e., clean pages) can simply be dropped to free up buffer space again. Note, the minimization of page fetches comes along with the goal of minimizing page flushes, both reduce the number of physical IOs compared to logical IOs.

4.1.2 Replacement Algorithms

The replacement algorithm is responsible to choose a so-called *victim* page that needs to be dropped or flushed to free up a buffer frame. This is typically triggered when a logical page request causes a miss and no empty frame is available to directly load the page into the buffer.

The major objective of replacement algorithms is to minimize the buffer misses for a given number of buffer frames (i.e., buffer size) and sequences of requests to the buffer. In case of a miss, the page with the lowest probability to be referenced again is replaced, i.e., flushed or dropped and updated with the requested page.

A replacement algorithm is not only based on the principle of locality but also on the assumption that the recent past of buffer requests is an indicator for the near future. Therefore, the age of a buffered page and the number of references (i.e., fixes) to it are accounted to support victim selection. Another important requirement is that an increase of buffer frames will definitely not degrade the miss rate of a buffer [GST70].

Now, we will briefly introduce frequently used (classes of) replacement algorithms found in most DBMSs.

LRU-based

The *Least Recently Used* (LRU) strategy takes the last time a page was referenced into account when selecting victims. It can simply be implemented with a stack of page pointers, having the pointer of the most recently referenced page on top. Page pointers at the end of the stack typically refer to the victims, while pointers that are already contained in the stack may be moved to the top in case of a new reference. However, having sequential access patterns (i.e., scans), cyclic references, or multiple references, the drawbacks of LRU become visible by not taking this additional information into account.

Therefore, slight variants exploit a small history by accounting the k *least recently references* or k *least recently unfixes* [DT90, OOW93]. For instance, the LRU-K algorithm [OOW93] remembers the last *k* references and timestamps of a page to be more "scan resistant". The history of already evicted pages is remembered for a certain period called *retained information*

period (*RIP*) to allow its reuse in case of a re-reference. Victim search in LRU-K is confined to pages that are buffered for at least a predefined *correlated reference period* (*CIP*), because this avoids that pages are immediately evicted after their first reference. A victim page is chosen based on its maximum backward k-distance, i.e., the earliest reference remembered within a page's history. The configuration $k = 1$ causes LRU-K to behave as the simple LRU algorithm.

LRD-based

The *Least Reference Density* (LRD) strategy accounts the number of page references within a given interval [EH84]. This reference counter is used to calculate the reference density by dividing it by the total number of references within the same interval. Reference densities are simply ranked to select the lowest one as victim. The problem of blocking pages, i.e., pages that have frequently been used a long time ago, is solved by regularly decrementing the reference counters.

CLOCK-based

Based on a FIFO (First In First Out) queue, clock algorithms iterate through the list of page references and use a bit or a counter (e.g., Generalized CLOCK [NDD92]) indicating the frequency of references. The bit is switched or the counter is decremented as long as the bit is set or the counter larger than zero, respectively.[2] GCLOCK algorithms are easy to implement and provide results comparable to LRU-based algorithms.

2Q

The 2Q algorithm [JS94] is a combination of a FIFO queue and an LRU chain. 2Q imitates LRU-2 and delivers similar performance, but avoids the algorithmic complexity of LRU-2. Upon the first reference, pages are added to the FIFO queue (denoted $a1$). In case of a re-reference, a page is moved to the head of the LRU chain (denoted *am*). The rationale behind 2Q is that only pages that are referenced several times either stay in the LRU chain or are promoted to it. In contrast, pages referenced only once within the recent history, are dropped earlier by the FIFO queue. A history extension for 2Q splits the FIFO queue to keep track of already evicted page references [JS94]. However, this raises similar queue sizing problems as for LRU-K.

ARC and CAR

The ARC algorithm (Adaptive Replacement Algorithm) [MM03] is based on a two-stage model similar to 2Q. An improved version based on CLOCK-like techniques is CAR (Clock with Adaptive Replacement) [BM04] that provides the same hit/miss performance as ARC, but has the advantage that it also "removes the cache hit serialization problem of LRU in ARC"

[2] You may also consider a clock-like cycle where page references are residing at the clock's numbers. Whenever the clock hand points to the next reference either its counter is decremented or the page is flushed/dropped.

[BM04]. We will focus on ARC, because the advantage of CAR is especially important for main-memory caches, which is not in the center of interest for this work.

ARC employs two LRU chains L_1 to filter out scans and L_2 to retain hot pages for re-reference. A parameter p, which is adapted at runtime, controls the chain sizes. Both chains are divided into top T_1, T_2 and bottom B_1, B_2, respectively. Only the top lists contain pages, which are actually cached while the bottom lists only remember page identifiers of evicted pages. The lists are allowed to permanently grow and shrink as long as their total size does not exceed the buffer capacity. The parameter p is used to control the target size of T_1, which has the same target size as B_2 and vice versa. If a page selected as victim originates from one of the top lists, its reference is promoted to the respective bottom list (i.e., remembered as history). As long as $|T_1| \geq p$, victims are selected from the tail of T_1, otherwise from the tail of T_2. Page hits in list B_1 advocate to increase p, because a larger T_1 is recommended to give chance for a second reference to the same page. In contrast, a hit in B_2 leads to a decrease of p.

Conclusion

Selecting the best replacement algorithm strongly depends on the current or expected workload and its own overhead induced by its complexity. However, for standard database applications and use cases, the presented classes of algorithms successfully dominate most of the default installations. A lot of very special algorithms or variations of existing ones emerged, too, for instance LFU (Least Frequently Used), MRU, or flash memory-oriented versions of CFLRU [PJK+06]. However, the replacement strategy is only half of the truth for buffer performance, its configuration is important, too [LWF77], and will be therefore discussed next.

4.1.3 Buffer Pool Configuration

Nowadays, DBMSs allow for many fine-grained buffer-related configuration options. Not only that external data containers[3] (i.e., files or disk partitions) may have multiple buffers assigned but also operators such as sort, (hash) join, query plans, or lock tables have their individual buffers assigned leading to the so-called *Buffer Pool Configuration Problem* [XMP02]. The size of a page-oriented buffer is simply the product of page size and the number of buffer frames. Therefore, the only constraint is that buffer frames must provide enough space to cache at least one external page of the assigned data container.

Object mapping is an important aspect w.r.t. buffer impact on query performance, especially when you think of different (external) storage devices providing different IO characteristics, sizes, or prices. Often used indexes and large volumes of user data may be buffered in different buffers, because they also reside in different containers. Moreover, a kind of stripping known from RAID configurations can be realized by spreading data across multiple buffer pools, too.

Distributing the available memory along all buffers may become challenging when the workload for the buffers varies. That means, some buffers may provide more benefit than others and their combined performance gain would improve if they have different sizes. This problem –

[3] DBMSs typically distinguish between user data, indexes, log entries, or long fields and therefore advocate the usage of multiple data containers at the same time.

optimal memory partitioning – is NP-complete and strongly depends on the current workload situation, which however, may rapidly change [MLZ+00].

The replacement algorithms of different buffers are not necessarily the same. When certain buffers are assigned for special purposes, they may require or benefit from tailored strategies. Even the underlying devices may also have impact on the selection of the replacement algorithm, for instance, for solid-state disks [PJK+06].

Eventually, many of the various tuning knobs for buffer pools can be changed online, i.e., dynamic buffer management without downtime is possible. But the ideal configuration is hard to achieve due to changing workloads and, if at all, difficult to predict [DYC93, BS11], especially in ad-hoc scenarios.

4.2 Self-Tuning Buffer Management Approaches

The importance of optimal buffer management in databases has clearly been emphasized for more than 20 years. In [EH84], many aspects were analyzed such as the underlying disk model, search strategies within a buffer, different propagation algorithms, page layout impact, and concurrency issues. Hence, the complexity of buffer management makes it impossible to find an optimal configuration that works for all kinds of workloads and system environments.

To self-tune a buffer configuration, [JCL90] used so-called priority hints. Depending on the access type (e.g., scan or index), buffered pages were differently handled for replacement. Thus, the dominating field of LRU-based propagation algorithms could be outplayed. However, those techniques do not reflect self-tuning behavior and they only address a single issue of dynamic buffer management – victim selection.

Another class of self-tuning techniques observes load situations in the DBMS and the buffer allocation strategy itself to predict future performances. In [NFS95], the authors give a theoretical base for combining the buffer allocation problem and the impact of access patterns.

Continuing the idea of combining multiple configuration areas, in [SGAL+06] a system-wide memory management was presented. Major DBMS components that can allocate variable-sized portions of the main memory are controlled by a feedback-control mechanism. Thus, memory assignments can be conveyed between different components, for instance, between the sort heap and a buffer pool. However, such a controller-based approach slowly adapts itself to rapidly changing situations due to stability demands avoiding configuration thrashing.

Current self-tuning research concentrates on fine-grained analytics. For instance, an analytically derived equation that relates miss probability to buffer allocation is presented in [THTT08]. Although those equations support the reconfiguration decision, they do not address the entire buffer or memory management. Moreover, their scope is often too narrow and self-tuning may get stuck in a local optimum. Real-world workloads contain peak situations or changing characteristics (e.g., access patterns, data volume). Here again, the requirement for a stable system is contrary to fast reaction times necessary when (short) peak situations occur.

In [DTB09], a kind of brute-force approach determines the optimal configuration for an entire DBMS by testing the configuration space step by step. This can hardly be done online due to the overhead and this approach is incapable of reacting to (short) rapid changes.

Most of the existing buffer tuning techniques can be classified either into *goal-oriented* techniques or *simulation-based* techniques. In the following, we will briefly introduce the main concepts of both approaches, before we identify actual issues for buffer performance forecasts.

4.2.1 Goal-oriented Buffer Tuning

For *goal-oriented* buffer tuning, so-called performance goals have to be defined. Typical goals that can be defined are the *average response time*, a *maximum workload specific latency*, or the *maximum miss rate*. Such a goal usually holds for a single buffer pool, but may also apply for all buffer pools. At runtime, the goal fulfillment is observed and, if necessary, memory is reallocated to increase the goal fulfillment.

The *Fragment Fencing* approach [BCL93] observes the reference frequency of certain *workload classes* and a special *working storage region* used for sort and join operations. This idea, however, was improved by the *Class Fencing* approach that is "more responsive, more robust, and simpler to implement" [BCL96]. It introduced the *Hit Rate Concavity* concept and extended the notion of workload classes for buffer pools by also allowing the assignment of multiple workload classes to the same data.

Based on the random access response times (RT), the so-called *Goal Satisfaction* algorithm [yCFW+95, Ten95] uses simple performance indexes (PI) for each buffer pool. By specifying a performance goal (GOAL), the index is obtained as follows: $PI = \frac{RT}{GOAL}$. The algorithm requires hit statistics (HIT) for various buffer sizes (SIZE) and the average time needed in case of a buffer miss (delay) to define the following formula:

$$RT(SIZE) \approx (1 - HIT(SIZE)) \times \text{delay} + HIT(SIZE) \times 0$$

Because those hit statistics typically are not available, they are approximated by the following equation based on [Bel66][4]: $HIT(SIZE) = 1 - a \times SIZE^b$. After each tuning interval, the values for *delay*, *a*, and *b* are adjusted based on observations. To tune the buffer configuration, the buffer pools (BP) are ordered by their expected *PI* when being resized by a fixed $\triangle$ size. As long as the worst and best ranked buffer pool are not the same, they are pairwise resized as follows (for the full documentation we refer to [yCFW+95]):

1. $\triangle$ = a constant number of buffer frames
2. $PI_i(\text{size}_i)$ denotes the buffer pool BP_i's performance index with pool size size_i
3. $min_{BP} = BP_j | PI_j(\text{size}_j - \triangle)$ is minimum
4. $max_{BP} = BP_i | PI_i(\text{size}_i + \triangle)$ is maximum
5. If $(max_{BP} \neq min_{BP})$ and
   $(PI_j(\text{size}_j - \triangle) < PI_i(\text{size}_i + \triangle))$ or
   $max(PI_j(\text{size}_j), PI_i(\text{size}_i)) > max(PI_j(\text{size}_j - \triangle), PI_i(\text{size}_i + \triangle)))$ then

[4] Characteristics are extrapolated based on (at least) two reference points using two parameters *a* and *b*.

$$\text{size}_i = \text{size}_i + \triangle \text{ and}$$
$$\text{size}_j = \text{size}_j - \triangle.$$

The algorithm ensures to only change memory allocations if the expected result yields improved performance indexes. Experiments by the authors have shown that their dynamic assignment is better than the static default configuration. Unfortunately, they used the same goal of 0.15 (i.e., 85% hit probability) in all cases.

The *Dynamic Reconfiguration Algorithm* (DRF) [MLZ+00] targets at performance goals for *transaction classes*, i.e., transactions (T) operating on the same database objects and having the same performance goal are grouped into a class. Similar to the previous algorithm, the DRF uses an *achievement index* (AI) to express the goal fulfillment for a buffer *i*:

$$AI_i = \frac{\text{goal average response time for } T_i}{\text{actual average response time for } T_i}$$

If *AI* is between 0 and 1 the transaction class does not fulfill its performance goal. After a tuning period, the algorithm identifies pairwise buffers with the largest expected performance gain when being increased and with the lowest negative impact in case of downsize. To calculate the expected cost increase and decrease for a certain transaction class, the following *cost estimation equation* for its buffers (B) is defined:

$$C_i = \sum_{j=1}^{b} L_i(B_j) \times costLR_j(m)$$

The formula takes the number of *logical reads* (L) and their costs (LR) for a given buffer size (m) into account. A detailed description how to derive *costLR* based on read/write numbers and probabilities, page reference counters, and CPU use can be found in [MLZ+00]. Although this approach takes dirty pages and asynchronous IO into account and delivers promising results, the authors also mention that the success strongly depends on the specified performance goal.

The major problem of all goal-oriented approaches is that the user is required to set appropriate performance goals for certain sets of queries (or transactions), which is comparably difficult as manual database buffer tuning [SGAL+06].

4.2.2 Simulation-based Buffer Tuning

Simulation-based techniques try to avoid wrong assumptions or estimations of buffer hit rates. Because incorrect estimations easily lead to suboptimal tuning, buffer simulations are employed to run through alternative buffer configuration(s). Capturing buffer traces or having adequate simulation data available is the most challenging aspect [THTT08], especially when doing online buffer tuning. Either a familiar workload is running that makes data collection obsolete, incrementally changing the buffer allocation leads to an increasing accuracy for access figures [SGAL+06], or the buffer manager simulates (another) replacement policy [DRS+05, NTA05, SGAL+06], which allows a replay of the trace against alternative buffer configurations. Even the approach in [THTT08], which is capable of predicting hit and miss

rates for various replacement policies by simply changing a couple of parameters, requires adequate buffer request traces.

One of the most cited simulation approaches can be found in [SGAL+06]. The authors propose a *simulated buffer pool extension* (SBPX) to estimate the *cumulative save time* for buffer upsizing. This extension is simply an overflow buffer for the page identifiers of the most recently evicted pages. The overflow buffer must, of course, have its own strategy for victim selection. The authors of SBPX recommend here a strategy "similar to that of the actual buffer pool" [SGAL+06]. Inspired by this idea, we implemented an improved version into XTC [BS11, SB11], which is briefly described in the following.

When a page miss in the actual buffer occurs, the extension checks if the page identifier is found in the overflow buffer, i.e., whether the page would have been present in a larger buffer. In this case, we can account a "saving" potential for upsizing. Further, we must now maintain the overflow buffer. The page identifier of the actual evicted page is promoted to the overflow buffer that in general requires to evict another page identifier from the overflow buffer. This replacement is not exactly the same as a real miss in the simulated larger buffer. The identifier of the requested page causing the miss could have been present in the larger buffer. In the course of continuous requests, however, also a larger buffer must evict pages. Thus, a replacement in the overflow buffer can be regarded as a "delayed" replacement effect. In the case of a page hit in the actual buffer, no further bookkeeping is required, because the locality principle suggests that the replacement strategy in a larger buffer holds a superset of the pages present in a smaller one. Listing 4.1 shows a sketch of the modified page fix routine.

Listing 4.1: Modified page fix algorithm for upsize simulation

```
1 Frame fix(long pageNo) {
2   Frame f = mapping.lookup(pageNo);
3   if (f != null) {
4     strategy.refer(f);            // update replacement strategy and statistics
5   } else {
6     Frame of = overflowMapping.lookup(pageNo);
7     if (of != null) {
8       overflowMapping.remove(of.pageNo);      // update overflow hit statistics
9     } else {
10      of = overflowStrategy.victim();
11      overflowBuffer.remove(of.pageNo);      // update overflow miss statistics
12    }
13
14    Frame v = strategy.chooseVictim();
15    strategy.copyStateTo(overflowStrategy);
16    v.copyStateTo(of);                  // transfer page identifier to overflow
17    overflowMapping.put(of.pageNo, of);
18    mapping.remove(v.pageNo);     ...     // replace page in frame v
19    strategy.referAsNew(v);       // update replacement strategy and statistics
20    mapping.put(pageNo, v);
21  }
22  return f;
23 }
```

The problem of this approach is that replacement decisions for two separate buffers in combination compared to a single large buffer are not necessarily the same. Thus, the forecast quality of upsizing simulations depends on one aspect: When a page is evicted from the actual buffer and promoted to the overflow area, we must be able to transfer "state" information

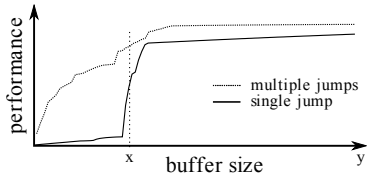


Figure 4.1: Buffer speed-up trend for different access patterns

(e.g., hit counters, chain positions, etc.) from the actual replacement strategy into the overflow strategy (lines 14 and 16). Otherwise, the overflow strategy behaves different.

The most difficult aspect of simulation-based buffer tuning is the fact that actual IO timings and request sequences strongly depend on processing times and page fetch/flush times. For instance, devices where the pages need to be fetched from or written to can be idle or under load, which is not covered at all by only simulating hit and miss rates. In the next section, we show further issues related to simulation-based buffer tuning.

4.2.3 Forecast Issues

Forecasting the buffer behavior for alternative buffer sizes is still a challenging task and we identified several weak points in the existing approaches [BS11, SB11].

- The first weakness is that buffers do not scale linearly with their pool size. Existing approaches simulate or estimate the performance gain for buffer upsizing, but not for buffer downsizing. The extrapolation of downsize behavior may be totally wrong leading to bad tuning decisions. The most obvious example is a repeating scan of x pages occurring at a buffer of size y. As long as $x > y$, the buffer delivers a weak performance. In contrast, as soon as $x \leq y$, i.e., all the requested pages fit into the buffer, the second scan is fully buffered resulting in perfect buffer performance. Such a "jump" or non-uniform behavior is sketched in Figure 4.1.

- Hit/miss ratios are the standard quality metrics for buffers, because they are cheap to assess and express the actual goal of buffer use: IO reduction. Unfortunately, they are useless for performance forecasts, i.e., they even do not allow to make simple extrapolations for growing or shrinking buffer sizes. To illustrate this fact, let us assume the following scenario for a given buffer size of 5 and LRU-based replacement. At the end of a monitoring period, we observed 5 hits and 10 misses. At least two different access patterns may have led to these statistics:

  Scenario 1: $1,2,3,4,5,1,1,1,1,1,6,7,8,9,10,...$

  Scenario 2: $1,2,3,4,5,1,2,3,4,5,6,1,2,3,4,...$

  In the first scenario, 5 hits are attributed to repeated accesses of page 1, whereas, in the second scenario, the hits are attributed to 5 different pages (1, 2, 3, 4, 5). For the

same scenarios and a buffer of size 2, we get completely different hit (h) and miss (m) statistics:

Scenario 1: $m,m,m,m,m,m,h,h,h,h,m,m,m,m,m,...$
Scenario 2: $m,m,m,m,m,m,m,m,m,m,m,m,m,m,m,...$

Obviously, scenario 1 obtains a better hit rate with 4 hits to 11 misses than scenario 2 without any hit. If we increase the buffer to 6 pages, the picture turns again:

Scenario 1: $m,m,m,m,m,h,h,h,h,h,m,m,m,m,m,...$
Scenario 2: $m,m,m,m,m,h,h,h,h,h,m,h,h,h,h,...$

Now, we observe 5 hits to 10 misses for scenario 1 and 9 hits to 6 misses for scenario 2. This example shows that hit/miss numbers or page/benefit metrics do not allow for correct extrapolations, because the order of page requests and the hit frequency distribution are important. Thus, self-tuning relies on monitoring and sampling of data where current buffer use is taken as an indicator for the future. Information relevant for resizing forecasts such as reuse frequencies, working set sizes, or noise generated by scans cannot be expressed in single numbers.

- As we already indicated, simulating separate buffer pools bear certain problems. The first problem results from buffer operations (i.e., fetch, flush, fix, unfix, etc.) typically having different runtimes. In a simulation scenario, alternative hit/miss numbers gained through simulation do not necessarily last for the same duration as the actual tuning period. That means, either more or less buffer operations could be processed. Moreover, the impact of a different buffer IO is not accounted in terms of the interplay of transaction dependencies and device loads.

  Modeling only the additional buffer size for upsize forecasts in a separate buffer easily leads to wrong results. Sophisticated replacement algorithms are sensitive to the actual buffer size. Simply adding the performance figures of two distinct buffers (e.g., [SGAL+06]) is therefore unwise.

- An important issue that is often excluded from self-tuning mechanisms are short-term memory consumers such as sorting or joins. Although a lot of research was done to improve sort and join algorithms for DBMSs, their system impact in terms of memory thrashing is often neglected. The only parameter that can be tuned online is the main memory area reserved for sorting or join buffers (e.g., hash tables). Most systems allocate fixed-size portions of main memory for those operations after an optimized query was translated into a final execution plan. However, estimation errors or long running queries may overly bind memory capacities. Even in systems where sort operations are "sourced out" to a distinct buffer pool, saturation effects may turn the pool into a permanently assigned memory area of constant size. If those pools were integrated into a self-tuning mechanism, weak reaction times due to system *monitoring* demands may cause the system to miss or exceed ideal configurations (see Figure 4.2). But sort operations are typically one-shot operations, which means, as soon as the sorting is performed,

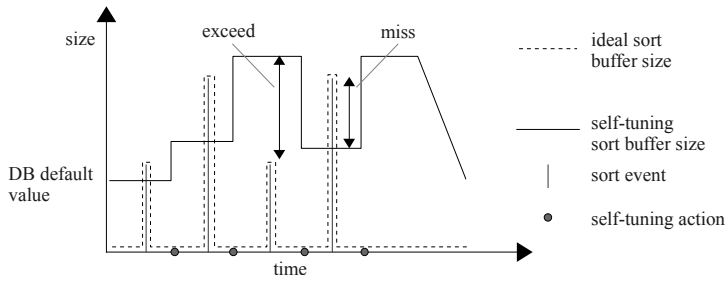


Figure 4.2: Short-term memory consumers cause suboptimal self-tuning decisions

the data is read only once by the caller (i.e., a transaction). Although intermediate runs of external sorts may cause IO for the same data multiple times, the sort result is accessed only once by a single client, often in a sequential manner, and, as a consequence, the resulting access pattern is deterministically ordered. In contrast, buffer replacement algorithms typically try to avoid read misses and buffers allow shared access as well as multiple modifications of the same buffer frame – which is not required for short-term memory consumers [OS11].

The sample scenario fort sort buffer adjustments in Figure 4.2 illustrates the problem, caused by self-tuning when *sort events* (vertical bars) of different sizes occur. Three different effects may happen: (1) The sort buffer size exceeds the required space due to over-(re)acting, or (2) sort buffer size is below the optimum due to eager adjustments, and (3) (not shown) a conservative strategy may retain constant sort buffer sizes over long periods which is similar to static buffer management.

Despite the forecast issues we identified, the basic assumption is that buffer increase should never result in a performance decrease. This justifies most of the existing approaches for buffer upsize forecasts. Even if their predicted performance gain holds off, it should not decrease at all.

Downsize forecasts are more critical, because they may lead to unwanted performance penalties. The ideal starting point for buffer forecasts is the replacement algorithm used for a buffer. Its statistics incorporate a lot more information about these relevant aspects than any other performance marker. Today, substantial research has already been done to develop adaptive replacement algorithms, hence, it is safe to assume that such algorithms are operating “optimally” for the available memory. The question is how to leverage this implicit knowledge for performance forecasts. As we will demonstrate in the next section, it is difficult, but not impossible, to get reliable estimates for buffer downsizing. In combination with already known simulation methods for the estimation of buffer upsizing, we can then build a lightweight framework for dynamic buffer management.

4.3 Lightweight Performance Forecasts

The goal of buffer replacement algorithms is the optimized utilization of data access locality, i.e., to keep the set of the currently hottest pages that fits into memory. Accordingly, a smaller buffer is assumed to keep an "even hotter subset" of the pages that would be present in the actual buffer. Based on this assumption, we denote a subset of pages in a buffer of size n as $hotset_k$, if it would be kept in a smaller buffer of size k. The key idea of our approach is to keep track of this hotset during normal processing. When a page is found in the buffer and belongs to the hotset, it would have been a hit in the smaller buffer, too. However, if a requested page is in the current buffer but not in the hotset, the smaller buffer would need to evict another page, which must be, of course, part of the current hotset and load the requested page from disk. Here, we only have to maintain the hotset. The page that would have been evicted from the smaller buffer is removed from the hotset and the requested page is added to it. Each swap is accounted as a page miss for the simulated smaller buffer.

Listing 4.2: Modified page fix algorithm for downsize simulation

```
1 Frame fix(long pageNo) {
2   Frame f = mapping.lookup(pageNo);
3   if (f != null) {
4     if (!f.hotSet) {
5       Frame v = strategy.chooseHotSetVictim();
6       f.hotset = true;            // swap frame to hotset
7       v.hotset = false;
8       strategy.swapHotset(f, v);  // update simulated statistics
9     }
10    strategy.refer(f);            // update replacement strategy and statistics
11  } else {
12    Frame v = strategy.chooseVictim();
13    mapping.remove(v.pageNo);     // replace page in frame v
14    if (!v.hotset) {
15      Frame hv = strategy.chooseHotSetVictim();
16      hv.hotSet = false;          // swap frame to hotset
17      v.hotSet = true;
18      strategy.swapHotset(f, v);
19    }
20    strategy.referAsNew(v);       // update replacement strategy and statistics
21    mapping.put(pageNo, v);
22  }
23  return f;
24 }
```

Of course, a page miss in the current buffer would also be a page miss in a smaller buffer. Accordingly, we have to select a replacement victim for both the current buffer and the (simulated) smaller buffer. The real victim page is now replaced with the new page and swapped with the virtual victim of the smaller buffer into the hotset. The modified page fix algorithm is shown in Listing 4.2.

Note that a real replacement victim is generally not expected to be part of the current hotset, because this would imply that the replacement strategy evicts a page more recently accessed. In some algorithms, however, such counter-intuitive decisions might be desired, e.g., to explicitly rule out buffer sweeps through large scans. Then, we must not maintain the hotset at all.

Obviously, the overhead of this approach is very small. We only need a single bit per buffer frame to flag the hotset membership and must determine a swap partner, when a new page

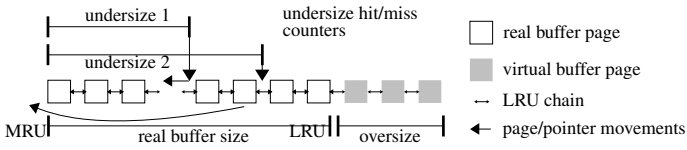


Figure 4.3: LRU-based hotset simulation with overflow extension

enters the hotset[5]. Furthermore, the simulation does not influence the quality of the current buffer, i.e., the strength of the replacement strategy is fully preserved. As said, the choice of the hotset victim is dependent on the used replacement strategy to reflect the behavior of the strategy in a smaller buffer correctly. In the following, we will look at hotset victim determination for our families of replacement algorithms. In particular, we want to know if it is possible to predict replacement decisions for a smaller buffer based on the implicit knowledge present.

4.3.1 Algorithmic Extensions

To demonstrate the effectiveness of our approach, we integrated the hotset simulation into several replacement algorithms. In the following, we sketch the most important aspects.

- **LRU** Typically, LRU is implemented as a doubly-linked list as shown in Figure 4.3. On request, a page is simply put to the head, i.e., MRU position of the chain. Thus, LRU finds its replacement candidate always at the tail, i.e., LRU position. Accordingly, the first k pages of the LRU chain in a larger buffer of size n are identical with the k pages in the simulated smaller buffer of size k and the hotset victim page is found at the k-th position of the head. The overhead of pointer dereferencing to position k can be avoided with a marker pointer (called *hotset LRU*) that can be maintained at low cost. Hence, the hotset victim is guaranteed to be identical to the victim as in the smaller buffer and the simulation is precise. Evidently, the simplicity of LRU even allows to easily simulate at the same time the effects when the current buffer would be reduced to different smaller sizes, which is especially useful for precise step-wise tuning decisions. Here, it is sufficient to place a marker at each desired position.

- **LRU-K** The victim page in LRU-K is determined by the maximum backward K-distance, i.e., the page with the earliest reference in the history vector. Thus, although implemented differently, LRU-K behaves for $K = 1$ as LRU. The hotset victim is chosen accordingly as shown in Listing 4.3. Note that implementations of LRU-K usually maintain a search tree for that. For simplicity, we present here the modification of the unoptimized variant as in the original paper.

  Due to the history update algorithm described in [OOW93], more than one victim candidate can exist. This could become a problem for our simulation, because a real buffer might choose a different victim than simulated. Therefore, we simply evict the candidate

[5] We applied the *bit flag* idea also to the SBPX buffer, which reduced the number of hash map lookups.

with the least recent reference (line 12). As the timestamp of the last access is unique, our simulation will be accurate here. Instead, the choice of *RIP* turns out to become a problem. If the garbage collection for history entries is not aligned, pages that re-enter the smaller buffer will be initialized differently than in simulation, which may affect future replacement decisions.

Listing 4.3: LRU-K hotset victim selection

```
1 Frame chooseHotSetVictim() {
2   long min = t;
3   long minLast = Long.MAX_VALUE;
4   Frame v = null;
5   for (int i = 0; i < pages.length; i++) {
6     Frame p = pages[i];
7     History h = p.history;
8     if ((p.hotSet) && (t - last > CIP)) {
9       long last = h.last;
10      long dist = h.vector[k - 1];
11      if ((dist < min)
12          || ((dist == min) && (last < minLast))) {
13        victim = p;
14        min = hist.vector[k -1];
15      }
16    }
17  }
18  return v;
19 }
```

- **GCLOCK** The determination of a hotset victim for GCLOCK is straightforward: We simply have to iterate over the frames and look for the first hotset page whose reference counter would drop below zero. Obviously, this occurs in the page with the minimum reference counter. The algorithm is sketched in Listing 4.4.

Listing 4.4: GCLOCK hotset victim selection

```
1 Frame chooseHotSetVictim() {
2   Frame v = null;
3   int h = clockHand;
4   for (int i = 0; i < size; i++) {
5     Page p = circle[(++h % size)];
6     if (p.hotSet) {
7       if (p.count == 0) {
8         return v;
9       } else if ((v == null) || (p.count < v.count)) {
10        v = p;
11      }
12    }
13  }
14  return v;
15 }
```

Again, this only approximates the behavior of a smaller buffer with GCLOCK for two reasons: First, the angular velocity of the clock hand in a smaller buffer is higher because there are fewer frames. Second, the circular arrangement of buffer frames makes the algorithm inherently dependent on the initial order. Thus, victim selection is not only a matter of the page utilization, but also a matter of clock-hand position and neighborship of frames. Using a second clock hand (i.e., pointer) walking solely over the hotset frames

is necessary to address differing round trips. However, swapping of frame positions when the hotset is maintained would influence the behavior of GCLOCK in the actual buffer – a circumstance, we want to avoid. To improve forecast quality, we implemented the smaller circle, i.e., the hotset, with forward pointers for hotset pages that point to the logical next one. In case of swapping (see lines 8 and 21 in Listing 4.2), only the forward pointer and a hotset counter for that page need to be maintained. Later, we will show that these minor efforts can lead to almost perfect estimations.

- **2Q** Due to the sizing problems for the FIFO queue and the LRU chain in the standard algorithm, we used a simplified variation of 2Q where all buffer frames are assigned to the LRU chain and the FIFO queue only stores references to the pages in the LRU chain. Consequently, it serves like an index for the LRU chain to identify pages referenced only once so far. Victims are primarily selected from the FIFO queue to replace those pages earlier. A subtlety of 2Q is here that the FIFO queue must not be drained to give new pages a chance for re-reference and promotion to the LRU chain. The minimum fill degree of the FIFO queue is a configurable threshold. For simulation, we must therefore count the number of hotset entries in the queue, to be able to decide when a smaller buffer would pick a victim from the FIFO queue and not from the LRU chain. Also, the threshold must be the same for both sizes. Although this results in uniform retention times in the FIFO queue for differing LRU chain sizes, it is acceptable to some degree, because the threshold models the granted window for references of new pages. The hotset victim selection is sketched in Listing 4.5. Depending on the number of hotset entries in the $a1$ list (line 4), the victim is chosen either from the $a1$ list (lines 4–15) or from the *am* list (lines 17–22). Note that both LRU hotset pointers may refer to the same buffer page, which requires maintenance (i.e., moving the pointer) also for the unaffected queue (lines 5,6 and lines 18,19, respectively).

Listing 4.5: 2Q hotset victim selection

```
1 Frame chooseHotSetVictim() {
2   Frame v;
3   if ((a1.numberOfHotsetEntries() > threshold)) {
4     v = a1.hotsetLRU();          // victim from FIFO
5     if (v == am.hotsetLRU()) {
6       am.hotsetLRU() = am.hotsetLRU().amNext;
7       // if both list's hotset LRU pointers are equal move the am's, too
8     }
9     a1.hotsetLRU() = a1.hotsetLRU().a1Next;
10    while (!a1.hotsetLRU().hotSet) {
11      a1.hotsetLRU() = a1.hotsetLRU().a1Next;
12        // skip non-hotset page to reposition a1's hotset LRU pointer
13    }
14  } else {  // inverse logic for LRU list:
15    v = am.hotsetLRU();          // victim from LRU
16    if (v == a1.hotsetLRU())
17      a1.hotsetLRU() = a1.hotsetLRU().a1Next;
18    am.hotsetLRU() = am.hotsetLRU().amNext;
19    while (!am.hotsetLRU().hotSet)
20      am.hotsetLRU() = am.hotsetLRU().amNext;
21  }
22  return v;
```

- **ARC** The overhead for a separate SBPX-based oversize simulation can be avoided, when the history LRU chains B_1 and B_2 are taken into account. Therefore, the page fix routine is slightly adapted towards ARC-awareness. In Listing 4.6, the fix() algorithm is aware that pages found in the buffer (lines 2–3) are either from a history LRU chain (lines 4–13) or a T_* chain (lines 14–20). We also added overflow and hotset counters for simulated misses and hits to illustrate the simple adaptation. Note, the refer() and victim() routines in line 21 and 23 are equal to the algorithms presented in [MM03].

Listing 4.6: ARC page fix

```
1 Frame fix(long pageNo() {
2   Page p = buffer.get(pageNo);
3   if (p != null) {
4     if (p.getList() == B1) {
5       if (!p.overflow())
6         overflowMissCnt++;
7       else
8         overflowHitCnt++;
9     } else if (p.getList() == B2) {
10      if (!p.overflow())
11        overflowMissCnt++;
12      else
13        overflowHitCnt++;
14    } else {
15      overflowHitCnt++;
16      if (!p.hotSet())
17        hotsetMissCnt++;
18      else
19        hotsetHitCnt++;
20    }
21    p.refer();
22  } else {
23    p = strategy.victim(buffer);
24    buffer.remove(p.pageNo);
25    ...           //  load page
26    buffer.put(pageNo, p);
27    overflowMissCnt++;
28    hotsetMissCnt++;
29  }
30  return p;
31 }
```

Listing 4.7: ARC simulation functions

```
1 boolean hotSet() {
2   int steps = t1.length;
3   if (list == T2) steps = t2.length;
4   steps -= steps * underSize / size;
5   int i = 0;
6   Page y = this;
7   while (i < steps && y.next != null) {
8     i++;
9     y = y.next;
10  }
11  return (i >= steps);
12 }
13
14 // same as hotSet() but for B* lists
15 boolean overflow() {
16  int steps = b1.length;
17  if (list == B2) steps = b2.length;
18  steps = steps * overSize / size;
19  int i = 0;
20  Page y = this;
21  while (i < steps && y.prev != null) {
22    i++;
23    y = y.prev;
24  }
25  return (i >= steps);
26 }
```

Simulation of different buffer sizes in ARC does not use flags to indicate hotset or overflow membership of a page. Instead, two functions calculate these properties on demand as shown in Listing 4.7. For hotset membership, we first estimate a valid hotset size depending on the current sizes of T_1 and T_2 (lines 2–4), which determines the upper bound of page links to follow (called steps, lines 7–10) towards the end of the T_* list. If the end can be reached within this bound, a page is not within the hotset range, otherwise it is.

The estimation of overflow membership is similar. Only the chain traverse order is different, because the steps calculated to find the beginning of a B_* list set the upper bound. Usually, the hotset size is close to the actual size as well as the SBPX size. Therefore, following next and previous pointers into the direction of the real buffer size is favorable. Note, this way of simulation works for SBPX sizes smaller than the history chains $B_1 + B_2$, i.e., SPBX size has to be less or equal to the cache size, which allows to forecast extensions of up to 100% of the current size.

4.4 Dynamic Buffer Pool Management

Even experienced database administrators with a deep knowledge of the workload and the database buffers, rely on the assistance of sophisticated monitoring tools to prevent negative effects of their tuning decisions. Often, they also run several observe-analyze-adjust cycles with reference workloads on dedicated test systems beforehand. Of course, this is time-consuming and expensive. Built-in self-monitoring and tuning components can ease this dilemma and reduce the risk of wrong decisions through rather small but continuous and incremental adjustments. In dynamic environments, however, those mechanisms may react too slowly to keep up with the rate of workload shifts or short-term resource allocation for higher-level tuning decisions like auto-indexing. Therefore, we aim towards a reformulation of the central question of automatic tuning from "Which adjustment *certainly will* give the greatest performance benefit?" to "Which adjustment *most likely will* give a performance benefit, *but will certainly not* result in a performance penalty?". In other words, when we know that our reconfigurations will not harm, we get the freedom to react quicker and may apply more aggressive tuning.

In general, the total amount of buffer memory is limited and, therefore, the decision to assign more memory to a certain buffer is directly coupled with the decision of taking this memory from one or several others. Fortunately, the performance optimization heuristics for IO-saving buffers (e.g. data pages, sorting) is straightforward: The more main memory can be used the better. Even an oversized buffer, i.e., a buffer larger than the actual data to be buffered, is less likely to become a performance bottleneck due to bookkeeping overhead; it is just a waste of main memory. Downsizing a buffer, however, comes along with severe risks: the buffer's locality may drastically decrease and even turn into thrashing causing excessive IO, which also influences the throughput of other buffers. Accordingly, we concentrate on the forecast of negative effects of memory reallocations and base our tuning decisions not only, as common, on the estimated *benefits*, but also on vindicable forecasts of additional *costs*.

4.4.1 Cost Model

Automatic tuning needs to derive costs from the current system state or from system behavior to quantify the quality of the current configuration. Additionally, it needs to estimate the costs of alternative configurations to allow for comparison. Ideally, these costs comprise all performance-relevant aspects including complex dependencies between system components and future workload demands in a single number to allow for perfect decisions. Clearly, such a perfect cost model does not exist in practice. Instead, costs are typically derived from a mixture of cheaply accounted runtime indicators and heuristics-based or experience-based weight factors. The goal is to reflect at least the correct relationship between alternative setups w.r.t. performance. The more precise this much weaker requirement can be met, the easier we can identify hazardous tuning decisions before they boomerang on the system.

In contrast to computational costs of a specific algorithm, costs expressing the quality of a buffer are inherently dependent on the current workload. Buffering 5% of the underlying data, for example, can be an optimal use of main memory at one moment, but become completely useless a few moments later. Therefore, each cost value is a snapshot over a window at a certain

point in time with limited expressiveness for at most few periods in the future. We define the *general goal function* for our tuning component as follows:

At a given point in time t with a configuration c, find a configuration c' that has less accumulated IO costs over the next n periods.

The optimal window size and the number of forecast periods again depend on the actual workload; slowly changing workloads enable more precise cost estimations for longer periods, while rapidly changing workloads also decrease accuracy of future costs.

For simplicity, our cost model only considers buffer service time, i.e., the time needed to handle a page fix request. Of course, costs assigned to a specific buffer are primarily determined by the number of IOs performed. On a buffer miss (denoted m), a victim page has to be selected for replacement and flushed, if necessary, before the requested page is fetched from disk. Accordingly, a buffer miss causes at least one read operation, but may also cause several writes for flushing the log and the victim page. The ratio between reads and synchronous writes is reflected by a weight factor f_{dirty} that may vary over time and from buffer to buffer.

Depending on the characteristics of the underlying devices or blocking times under concurrent access, IO times can also vary between buffers. Hence, the costs of all buffers must be normalized to a common base to become comparable. We use here a second weight factor w_{buffer} for each buffer. As the time needed for a single IO operation is easy to measure, these factors can be derived and adjusted at runtime causing low overhead. Finally, the cost of a buffer at the end of time period t is expressed as:

$$c_{\text{buffer}}(t) = w_{\text{buffer}}(t) \cdot (1 + f_{\text{dirty}}(t)) \cdot m(t)$$

Note, we assume that CPU costs can be safely ignored, either because they are independent of whether an operation can be performed on buffered data or requires additional IO, or because additional CPU cycles for search routines in larger buffers are negligible compared to an IO operation. In the remainder of this chapter, we assume that read and write operations have symmetric costs and a low variance. However, it should be evident that the presented model can be easily extended to take asymmetric read/write costs (e.g. for solid-state drives), different costs for random and sequential IO, and also the apportionment of preparatory, asynchronous flushes of dirty pages into account.

4.4.2 Decision Model

Our approach of buffer balancing is using the cost model presented in Section 4.4.1. In regular intervals, a buffer configuration is analyzed and optimized if reallocations of main memory promise lower IO costs for the entire system. In Listing 4.8, the main algorithm responsible for buffer balancing is shown.

At the end of a monitoring period, we calculate a *save* and a *rise* ranking for all buffer pools based on their cost estimations. The higher a buffer pool is ranked in the *save* list, the more costs can be saved (i.e., it provides a higher benefit) when its size is increased according to the simulated oversize. Similarly, the *rise* list ranks buffers by the cost estimations for undersize figures, where the minimum cost increase is ranked top (cf. Listing 4.8, lines 2–30).

Listing 4.8: Balance algorithm

```
void balance() {
  RankList<Buffer> save = new RankList<Buffer>()
  RankList<Buffer> rise = new RankList<Buffer>()
  for (Buffer b : buffers) {
    int saveIO = b.missCnt - b.overflowMissCnt;
    int riseIO = b.hotsetMissCnt - b.missCnt;
    boolean added = false;
    for (Buffer sb : save) {
      int cmp = sb.missCnt - sb.overflowMissCnt;
      if (!added && saveIO > cmp) {
        // insert into rank at current position:
        sb.insertIntoRank(b);
        added=true;
      }
    }
    if (!added)
      save.add(b);  // buffer is added at the end

    added = false;
    for (Buffer rb : save) {
      int cmp = rb.hotsetMissCnt - rb.missCnt;
      if (!added && riseIO < cmp) {
        // insert into rank at current position:
        rb.insertIntoRank(b);
        added = true;
      }
    }
    if (!added)
      rise.add(b);  // buffer is added at the end
  }

  Buffer i = null;   // buffer to (i)ncrease
  Buffer d = null;   // buffer to (d)ecrease
  while (true) {
   i = null; d = null;
   for (Buffer b : save) {
     for (Buffer b2 : rise) {
       if (d == null && b != b2) {
         i = buf; d = buf2;
         if ((d.hotsetMissCnt - d.missCnt) >
             (i.missCnt() - i.overflowMissCnt))
           red = null;
       }
     }
   }
   if (d == null)
     break;

   // amount depends on i's and d's simulation ranges
   i.increase(amount);
   d.decrease(amount);
   save.remove(i); rise.remove(i);
   save.remove(d); rise.remove(d);
  }
}
```

With a Greedy algorithm, buffer pool pairs are now picked from the top of both lists as long as the cost reduction on the *save* list is higher than the increase on the *rise* list (lines 32–45). A buffer may end up in both lists indicating a critical "jump" size that is easily recognized this way (line 38). A resize mechanism performs then the actual memory "shift": The selected buffer from the *save* list is allowed to keep more page frames and references in the cache (line 50), while the buffer from the *rise* list is shrunk (line 51). For this, we repeatedly choose a victim page, flush it to disk, if necessary, and deallocate the page frame. Finally, the resized buffers are removed from both ranking lists (lines 52–53) to avoid thrashing. Note, we could compute an optimal solution for the resizing, but the Greedy pairwise resizing is much cheaper and delivers good results.

Of course, the simulated undersize and oversize areas of a buffer have to be adjusted as well. This is similar to a "regular" buffer resize. For instance, the number of hotset pages is reduced by selecting victims out of this subset and by switching their flags. Obviously, oversize areas can be kept or resized as desired. As oversize and undersize simulations for several buffer pools do not necessarily have the same size in bytes, gradual reallocations may be become necessary. For that, we must extrapolate the buffer scaling behavior between the real size and the simulated sizes.

Obviously, buffer resizing is a potentially expensive operation, because it may require a forced flush of dirty pages when one buffer is shrunk in favor of another one. However, the expected benefits justify this temporal overhead in general. If desired, the resize penalty for dirty pages could also be included in the cost model to fully avoid this temporal negative effect [BS11, SB11].

Important for buffer resizing is its time horizon, e.g., [CW05] recommend to manage the system memory in "near real-time", that enables quick reaction to *workload shifts*. Only the costs for buffer decrease, as explained before, have to be justified. An increase usually comes at almost zero costs. Thus, we can expand our approach to handle major workload shifts affecting all buffer pools as well as to deal with short-term peak loads, as the following section will show.

4.4.3 Integrating Short-term Memory Consumers

Depending on the assigned memory, the algorithmic effort for short-term memory consumers like (merge) sort or join operators can be determined [OS11]. For certain memory sizes, an accurate cost estimation of IO operations based on common database statistics is possible. By estimating IO costs for feasible memory sizes, characteristics of such operations are disclosed showing their scalability in terms of main memory resources.

To integrate short-term memory consumers in buffer tuning, we need to extend our analysis of buffer statistics. Because sorts may last multiple tuning periods (or less than one), their costs have to be adapted to enable data buffer pool comparisons. By collecting IO statistics, we roughly know how many (weighted) reads and writes can be performed by the system in one tuning period. There are rather active buffers serving a lot of IO requests and, in turn, less active ones. Of course, a sort buffer behaves like a very active one. That means, if the system performs x IOs in a time interval (under load) and the most active buffer served y requests, the number of IOs the sort can achieve during a tuning period is approximated by $\frac{y}{x+y} \cdot x$. This

temporarily adds a separate buffer to the cost model (i.e., the memory assigned for a short-term memory consumer is considered as a buffer), which can be balanced to the remaining data buffers and the existing mechanism.

When short-term memory consumers are finished, the decision, which data buffer gets how much of the freed memory, has to be made. There are various alternatives possible: (1) The memory is returned to the original buffer it was taken from, which ignores current benefit simulations. (2) The memory is distributed according to the current (upsize) benefit estimations, which may result in an oversized buffer, because the simulation size is usually different. (3) A step-by-step approach, reassigning only memory portions according to current simulation sizes for individual buffers in each period, observes benefit estimations but may take considerably more time (i.e., spanning several periods). (4) Distributing the freed memory to multiple buffers promising a performance gain is another option. We simply trigger the buffer balance logic and artificially "down-rank" the sort buffer to have negative benefit in case of upsizing and positive benefit in case of downsizing.

4.4.4 Read-ahead

Read-ahead mechanisms, also called *prefetching*, were introduced by the operating system community [Smi78]. Various cache replacement algorithms and their interplay with prefetching led to different prefetch classes. The distinction between on-demand fetching and prefetching was made, but also the downside of prefetching, i.e., the overhead, if unused data is brought into the cache, was studied. In [Sto81], a first step towards DBMS-aware prefetching was presented by reusing operating system concepts. However, access patterns in a DBMS are logically ordered (e.g., sequential access of data pages in a B-tree) and not necessarily physically clustered. Thus, a DBMS needs to specify its own read-ahead logic on top of the operating system's cache mechanism [EH84, Sto81]. Later, hints were added to perform cost-benefit analyses for future IO [TPG97]. IO forecasts may help to adjust the prefetch size online [PZ91].

There are many challenges when applying *read-ahead* in DBMSs. In [VL00], many of them were addressed. The most important issues are (1) a timely, useful, and little overhead causing implementation, (2) secondary effects such as cache pollution and increased memory requirements must be taken into consideration, and (3) improvement of runtime by overlapping computation with memory accesses. The authors also identify the problems that occur when adaptive prefetching is done such as lower miss ratios may be partially annihilated by the associated overhead of increased memory traffic and contention.

We want to exploit the benefits of read-ahead in our buffer, too. The read-ahead should be independent of the replacement algorithm and work together with other self-tuning measures. Similar to common techniques in replacement algorithms, the access history is a good indicator for future accesses. As long as mechanical hard drives dominantly appear as external storage equipment, sequential access patterns seem to be the most favorable ones to be identified. Note, we concentrate on spatial locality on disk and not necessarily on logical data structures. Because operating system's prefetching does similar optimizations in detecting sequential accesses, we need to exploit them instead of ignoring them. The sheer call stack when reading data, from the user query down to the bits and bytes through certain DMBS layers (Sec-

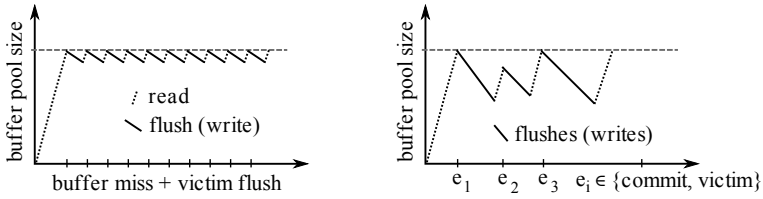


Figure 4.4: From single-page flush (left) to grouped flush (right)

tion 2.2), causes a lot of overhead, context switches, etc. However, physical IO is decoupled from logical one via the buffer, e.g., multiple transactions (sequentially) access data located at different areas in parallel. Due to scheduling effects, here, the raw IO may result in a see-saw hindering the identification of sequential access patterns. Therefore, the transaction-aware interplay of buffer accesses and external IO calls is important.

Our read-ahead implementation in XTC is straightforward. The x most recent page requests that led to a physical read are remembered and y pages can be prefetched, i.e., sequential pages to be read ahead. Both sizes x and y are adjusted online. Each page request leading to a miss is compared to the x history of requests. If the distance between at least one of these history pages is below two, a read-ahead is initiated. The read-ahead loads y consecutive pages. At the moment, XTC does not support prefetching for different sequential reads per buffer pool at the same time, i.e., all y pages are ordered consecutively. At least $y/2$ physical reads are necessary to switch to another "scan", i.e., a different physical page sequence. If read-ahead is beneficial and followed by more sequential access for this "scan", y can be increased if buffer size and free capacity allow that. Accordingly, y is decremented as soon as read-ahead is unfavorable. Although x can be changed online, our tests have not shown any noticeable benefit.

4.4.5 Sequential Writes (Buffer Flushes)

According to Section 4.4.4, nowadays, sequential access leads to higher throughput rates compared to random access. When a buffer page needs to be flushed because it is chosen by the replacement policy, a single (page) write to external disk occurs. Furthermore, depending on the log strategy, a kind of before-image or pointer switch needs to be performed synchronously (i.e., blocking IO). With the advent of log-structured file systems [RO92], single page modifications are solely kept in the sequential log. But those pages may belong to different database objects and moreover have been modified by different transactions. Hence, the physical order gets shuffled.

To overcome the single page-at-a-time flush (see Figure 4.4, left) causing a lot of random IO, because the replacement page (i.e., newly fetched) may be located somewhere else, grouped and sequential flushes are beneficial (see Figure 4.4, right). This is achieved by collecting sequences of pages from the same transaction without waiting for an explicit commit or by flushing any set of pages at-a-time. This set can contain a logical sequence of pages or the top-k pages to be flushed. Optionally the set of pages can be sorted before being flushed.

Although, multiple page flushes reduce the number of context switches, they do not guarantee to operate the buffer permanently at its capacity. In Figure 4.4, the impact on the buffer pool usage is visible. Another drawback exists when data is flushed too early, i.e., the assumption that eagerly flushing a (modified) page that will not be used again turned out to be wrong.

When using read-ahead (cf. Section 4.4.4) and grouped flushes, the underutilization of a buffer is compensated by the additional pages read and, thus, fast sequential reads easily fill the buffer again that was (partially) flushed before. Both techniques are based on the concept of sequential IO and their interplay is exploited.

4.4.6 Implementation Aspects

The entire buffer self-tuning is implemented within the propagation layer of XTC. Only some external interfaces are available to provide monitoring for performance evaluation and high-level control facilities.

Basic buffer *monitoring* is extended to collect additional statistics. That requires to account three hit/miss counters for the hotset, overflow, and real buffer size. Other metrics are calculated on demand or acquired by sampling. For instance, the *average response time*, which is used for goal-oriented tuning, is fairly costly to determine and usually does not change, for which reason sampling is used.

Each buffer page contains the following metadata:

```
Page implements Frame {
  byte[] data;           // size depends on page size
  int containerNo;       // physical data container
  PageID pageID;         // logical page identifier (8 Byte)
  Latch latch;           // lightweight lock
  boolean safe;          // transactional property
  AtomicReference<Object> cache; // cache for decompression
}
```

For the simulated overflow and history areas (e.g., B^* lists in ARC), a reduced page type is used that only contains the *pageID*. Pages are managed in several ways. Each buffer contains a hash map for fast *pageID*-based lookups and the internal data structures (e.g., LRU chains) of each replacement algorithm reference the same pages or their reduced type.

To integrate self-tuning capabilities for a buffer, basically the following interface *ResizableBuffer* needs to be implemented.

```
interface ResizableBuffer extends Buffer {
  increaseSize(int noFrames);
  decreaseSize(int noFrames, boolean immediateFlush);
  getMinSize();
  int getMaxRemovablePages();
  double getAverageResponseTime();             // goal-based tuning
  double getGoalResponseTime();                // goal-based tuning
  setGoalResponseTime(double goalResponeTime); // goal-based tuning
  resetResponseTimeCounters();                 // goal-based tuning
  long getSBPXSavedTime();                     // simulation-based tuning
  setSBPXsize(int size);                       // simulation-based tuning
  resetSBPX();                                 // simulation-based tuning
}
```

Therefore, a buffer carries some metadata, as shown in Listing 4.9.

Listing 4.9: Buffer metadata for self-tuning

```
1 Buffer implements ResizableBuffer {
2   // general buffer properties
3   HashMap<PageID, Frame> pageNoToFrame; // mapping page ID to buffer position
4   int bufferSize;  // number of maintained buffer positions
5   int pageSize;    // size of a buffer page in bytes
6   Policy policy;  // replacement policy for the buffer
7
8   // self-tuning properties:
9   boolean groupedFlush;
10  boolean groupedFlushSort;
11  HashMap<PageID, Frame> framesToFlush;
12  int minSize; // minimum size of this buffer
13  int costPR;  // cost for physical read
14  int costPW;  // cost for physical write
15
16  // goal-oriented
17  int noSW;    // synchronous writes counter
18  int noPR;    // physical reads counter
19  int noLR;    // logical reads counter
20  double goalResponseTime;    // goal response time
21
22  // simulation-based
23  HashMap<PageID, ReducedFrame> sbpx;  // SBPX mapping
24  Policy policySPBX;  // replacement policy for the overflow area
25  ...
26 }
```

The most obvious differences exist for the various self-tuning approaches. For instance, the goal-oriented tuning relies on simple counters, but is hard to control, i.e., to define a proper goal response time. In contrast, SBPX-based tuning requires an additional hash map and a separate instance of a replacement policy. In both cases, the hotset simulation is optional because it is integrated into the replacement policy that is present anyway.

For both, replacement algorithm and buffer, appropriate resize functionality is required. However, as Listing 4.10 shows, they are fairly simple and straightforward to implement.

The increased functionality first calls the replacement policy to add new buffer slots (line 3), before memory is allocated for them in the buffer (line 5).

Decreasing the size of a buffer has to ensure that no concurrent access interferes with the potential "mass" flush that requires to retrieve the list of victims synchronized (lines 13–27). After some sanity checks (lines 14–16), the replacement policy selects a set of victim pages (line 18), before those pages are either removed immediately or delayed (lines 21–25). In case of a delayed flush, the synchronized requirement is released, which allows concurrent buffer access, but may temporarily cause too much memory being assigned for buffers until all frames are flushed (lines 33–36).

As presented in Section 3.6.5, a simple STR call creates appropriate monitoring events (lines 8 and 29).

4.5 Evaluation

We assess the performance of our buffer extensions and optimizations with a generated set of benchmark workloads. A brief introduction of workload generation is followed by a description of our workloads we used to imitate various (common) access patterns including random

Listing 4.10: Resize functionality for buffer self-tuning

```
increaseSize(int noFrames) {
  synchronized {
   List<Frame> newFrames = policy.increaseSize(noFrames);  // call policy
   for (Frame frame : newFrames)
     frame.setPage(new byte[pageSize]);   // allocate memory
   bufferSize += noFrames;                // increase size
   // monitoring and reporting:
   STR.addEvent(new ReportEvent<int[]>(ReportComponent.BUFFER,"INCSIZE",stats));
  }
}

decreaseSize(int noFrames, boolean immediateFlush) {
  synchronized {
   if ((bufferSize <= noFrames) || (bufferSize - minSize < noFrames) ||
     (getMaxRemovablePages() < noFrames)) // see next method
     return InvalidOperation;

   List<Frame> victims = policy.decreaseSize(noFrames); // policy selects victims
   for (Frame frame : victims) {
     if (!frame.isDeleted()) {
       pageNoToFrame.remove(frame.getPageID());  // remove from buffer mapping
       if (immediateFlush)  //variant 1: immediateFlush
         flush(frame);
       else                 //variant 2: flush pages later
         framesToFlush.put(frame.getPageID(), frame);
       }
   }
   this.bufferSize -= noFrames;  // decrease size
   STR.addEvent(new ReportEvent<int[]>(ReportComponent.BUFFER,"DECSIZE",stats));
  }
  // not synchronized anymore:
  if (!immediateFlush) {  //variant 2 cont'd
    for (Frame frame : framesToFlush;
    {
      flush(frame);
      framesToFlush.remove(frame);
    }
  }
}

int getMaxRemovablePages() {        // retrieve number of non-fixed pages
  int currentfixCnt = 0;
  for (Frame frame: pageNoToFrame)
    if (frame.isFixed())            // cannot be flushed
      currentfixCnt++;
  int maxRemovablePages = bufferSize - currentfixCnt;
  if (minSize - currentfixCnt > 0)  // observe minimum buffer size requirement
    maxRemovablePages -= (minSize - currentfixCnt);
  return maxRemovablePages;
}
```

and sequential accesses of varying sizes. Thereafter, we evaluate the accuracy of our forecast mechanisms, which will be applied for buffer balance and workload analyses. Important aspects like overhead, integration of short-term memory consumers, and IO optimizations will conclude this evaluation part.

4.5.1 Workload

Buffer mechanisms have to deliver a reliable performance under various and sometimes exceptional access patterns. We built a workload generator that allows imitating realistic, artificial, and unusual access behavior. This allows us to analyze our optimizations in a structured way, i.e., varying different characteristics of the workload, and reliefs the pain of searching for adequate high-level workloads such as XQuery statements and suitable documents. Nevertheless, we also used the latter one to verify some of the results, but this is far from a structured "full" validation.

Generator

The basic unit of our generator is simply delivering a buffer request on each call, for instance, a page fix request parameterized by its logical page number. A basic unit is initialized to either generate the requests on demand or to use an already generated file containing the sequence of requests[6]. Multiple parameters are possible for initialization, such as page ranges and access type (e.g., sequential, random, or Zipf). By combining multiple of these basic units, we can easily assemble a specific workload scenario. Each generator is weighted within the workload to address varying shares of certain access types and ranges.

Buffer Characteristics

In our benchmark, we evaluate the buffer performance for different sizes and algorithms. A sample was already shown in Figure 4.1 on page 61. Typically, for each workload and each replacement strategy, the characteristics of buffer performance vary. Based on these characteristics, we choose interesting buffer sizes for further inspection, e.g., jump points, plateaus, and linear trends. We will see that different replacement strategies perform totally different for identical buffer sizes, but that does not necessarily hold for the whole range of possible buffer sizes. To justify our extensions and optimizations, we compare their effects for multiple algorithms.

Workloads

In Figures 4.5(a)–4.5(d), we analyze the critical buffer size ranges for various access patterns whose characteristics are summarized in Table 4.1. Note, the total number of database pages in a scenario is equal to the object size in the first column of the table. The only uniformly scaling buffer is measured for workloads dominated by random IO, see Figure 4.5(a), where

[6] These files can be used to repeat evaluations or to avoid costly live computations such as for generating a Zipf distribution. Simple scans and random access patterns are always generated on demand.

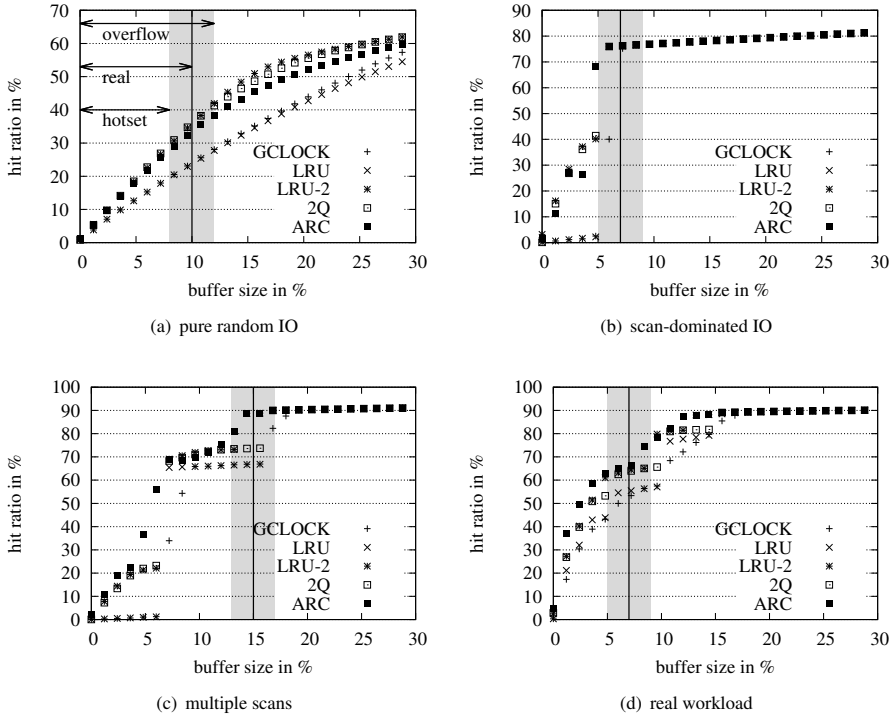


Figure 4.5: Buffer scalability for various workloads and replacement algorithms

the overall hit ratio is – as expected – quite low. In this case, resizing extrapolations will work properly, but such an access behavior is unusual in databases. Dominating scans mixed with random accesses are modeled and measured in Figure 4.5(b). Although some of the replacement algorithms are *scan resistant*, a dominant sequential access pattern easily provokes a "jump" in the buffer performance. In such cases, the buffer hit rate dramatically increases as soon as a frequent scan fits entirely into the buffer. Such "jumps" remain undetected if monitoring happens only on the opposite side. The third workload shown in Figure 4.5(c) is a mixture of multiple scans and random accesses in a single buffer. This scenario may represent a more typical buffer usage pattern that exhibits a realistic buffer scaling. In Figure 4.5(c), several areas can be identified having different slopes, where each area boundary may cause uncertainty for extrapolations. In the last sample workload, shown in Figure 4.5(d), we have a mixture of high-locality scans and some noise generated by random accesses. This typical workload scenario causes several (small) jumps resulting in a stair-case pattern. In this case,

Table 4.1: Workload characteristics

Workload	Figure 4.5(a) (random)		Figure 4.5(b) (scan)		Figure 4.5(c) (jumps)		
Request share in %	50	50	25	75	10	65	25
∑object size (pages)	150k	22k	150k	7k	150k	7k	13k
Access type	rnd	rnd	rnd	seq	rnd	seq	seq

Workload	Figure 4.5(d) (real)							
Request share in %	10	10	10	20	10	20	10	10
∑object size (pages)	250k	5k	10k	10k	500	500	1k	2k
Access type	rnd	rnd	seq	seq	seq	seq	seq	seq

fine-grained extrapolations necessary for buffer tuning may quickly fail, although the slope in the average is quite similar.

In the following sections, we investigate whether or not our algorithms are capable of identifying and handling all of these (more or less) typical workload scenarios.

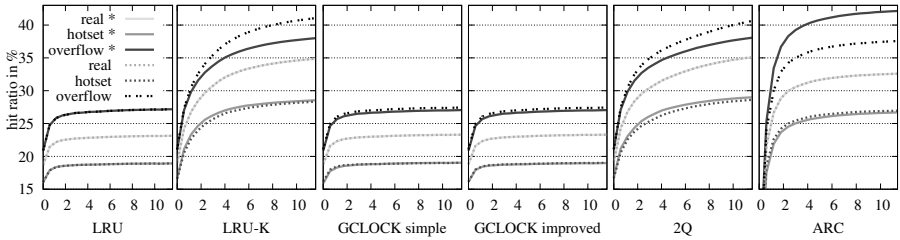
4.5.2 Forecast Accuracy

The quality of buffer self-tuning is based on the estimation accuracy of our extended buffer algorithms. Therefore, we need to evaluate it for the differing workloads. For the following experiments, the gray-shaded areas in Figures 4.5(a)–4.5(d) specify the simulated ranges centered around the actual buffer sizes indicated by the black lines (i.e., "real"). We choose a typical buffer tuning range of $\pm 2\,\%$ of the total DB size for upsizing and downsizing simulations. For each workload, we measure the accumulated "hotset" and "oversize" estimation accuracy. Each of the Figures 4.6(a)–4.6(d) contains the results of the six extended algorithms using the same workload and up to 1.2 Mio buffer calls. The lines marked with an asterisk (*) illustrate the simulation-based hit ratios and, to enable comparison, the others show those of real buffers having the same sizes.

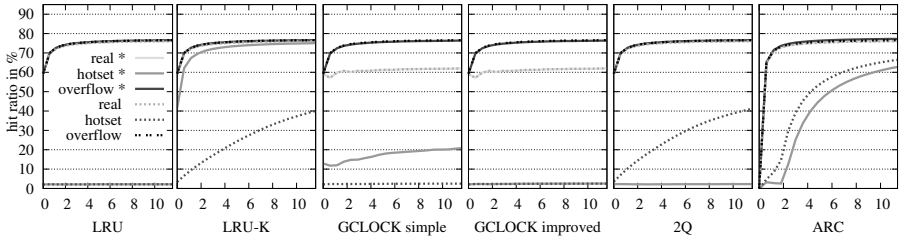
In each figure the first graph shows the standard LRU behavior that is always delivering perfect estimation accuracy; however, its hit ratio performance is not superior. But its lightweight simulation is definitely a plus. In contrast, the LRU-K results (second graphs) constantly indicate top hit ratios but show weaknesses w.r.t. forecast quality. Especially the downsize simulation of the *scan* workload fails with a dramatic overestimation.

The results for GCLOCK in Figures 4.6(b) and 4.6(d) (third graphs) reveal its sensitivity to page order and clock-hand position for hotset simulations. By adding a second clock hand and forward pointers to simulate a separate clock for the hotset pages, we achieve considerably better accuracy (fourth graphs), but its performance is always behind all other strategies.

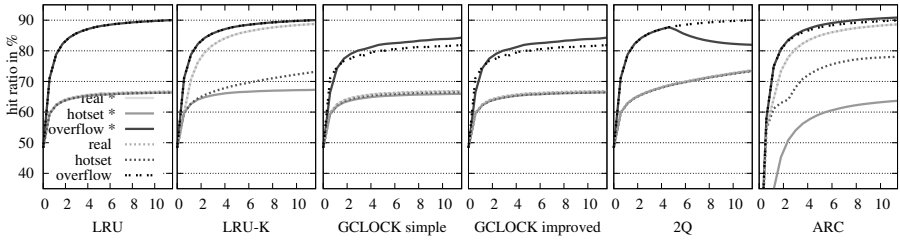
Forecast quality provided by the simplified 2Q algorithms is revealed, too (fifth graphs). In all scenarios, 2Q delivers excellent buffer results while only requiring low maintenance overhead. However, forecast quality is disappointing in some scenarios. Similar to LRU-K, it fails for workload *scan*, but in the opposite direction with underestimation. Further, we observe a suddenly degrading forecast quality for the workloads *jumps* and *real*. Moreover, oversize estimations and undersize estimations are negatively affected. Even the use of a separate policy for the oversize buffer does not lead to better results, which, in turn discloses the weaknesses



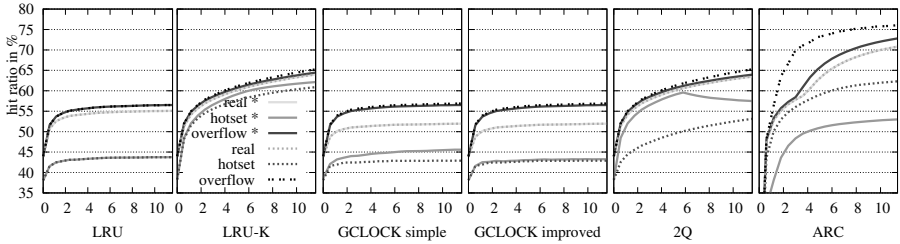
(a) Estimation accuracy for workload *random* (buffer calls ×100.000 on x-axis)



(b) Estimation accuracy for workload *scan* (buffer calls ×100.000 on x-axis)



(c) Estimation accuracy for workload *jumps* (buffer calls ×100.000 on x-axis)



(d) Estimation accuracy for workload *real* (buffer calls ×100,000 on x-axis)

Figure 4.6: Evaluation of hotset and oversize estimation accuracy

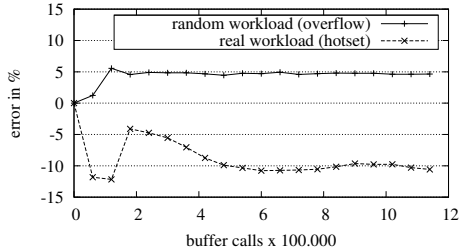


Figure 4.7: Simulation error analysis explaining hit ratio drift

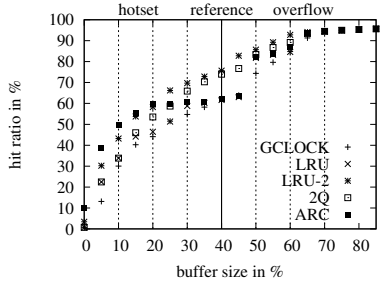


Figure 4.8: Workload characteristics used for the analysis of simulation ranges

of the SBPX approach.

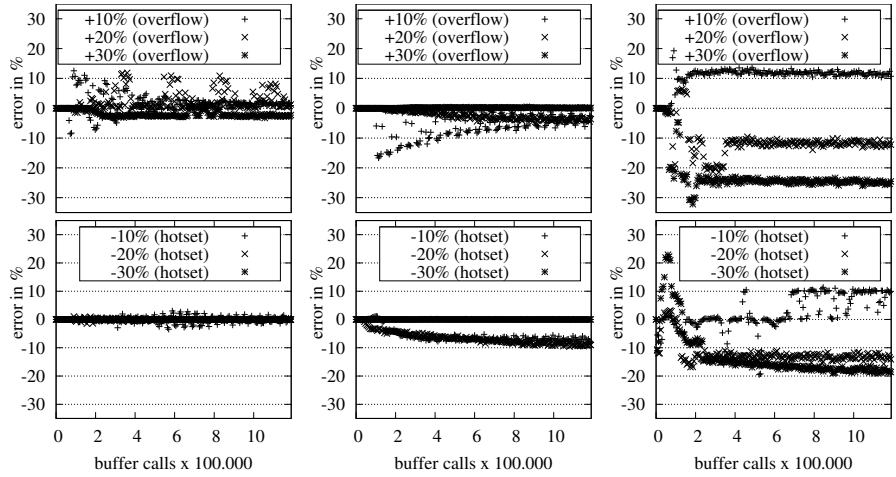
The performance of ARC and its forecast accuracy is shown on the right-hand side. Besides always good hit ratios, ARC's forecast for downsizing underestimates the actual performance except for *random* workloads. Although the visual distance between estimation and actual hit ratio seems to be huge sometimes (i.e., $< 10\%$), this forecast error solely evolves from the algorithm's warm up. In the following paragraph *Drift*, we show that the permanent error is quite low, as the trend of the cumulative hit ratio forecast already indicates.

The experiments reveal that our simulations based on the locality principle lead to trustworthy estimations in many cases. On one side, simple algorithms like LRU and GCLOCK fit well into our framework. On the other side, more advanced algorithms such as LRU-K, ARC, and 2Q also allow lightweight estimations, but suffer from unpredictable estimation errors in some scenarios. The reasons are built-in mechanisms to achieve scan resistance, which are hard to simulate. Further, these algorithms do not allow logical composition of individual buffers.

Drift

Some of the results shown in Section 4.5.2 reveal severe drifts leading to an apparent bad estimation quality. The simulations of ARC, for example, show dramatic drifts for oversize and hotset estimation in Figure 4.6(a) and Figure 4.6(d), respectively. Note, the results show cumulated hit ratios, where even small but constant estimation errors sum up. Therefore, we analyzed the estimation error for the hit ratios of simulated buffer sizes for both extreme drift situations in Figure 4.7. After a short warmup period, the error ratio settles between 5% for the overflow estimation of workload *random* and 10% for the hotset estimation of workload *real*.

Obviously, a good choice for the simulation interval helps to reduce the estimation error rate. However, this is another tuning parameter which may also be adjusted during runtime.



(a) GClock optimized (b) 2Q (c) ARC

Figure 4.9: Simulation error for various simulation sizes (reference size 40%)

Simulation Range

The size of the simulation range is an important parameter. In this test, we want to examine whether the forecast accuracy decreases if the simulation range is increased. Therefore, we step-wise increased the overflow area and decreased the hotset size. For the corresponding workload, depicted in Figure 4.8, we "stretched" the *real* workload from Figure 4.6(d), i.e., the gradient is spanning a wider buffer size range before saturation effects arise. This enables us, for a reference size of 40% (black line), to analyze hotset sizes and overflow sizes of $\pm$10%, $\pm$20%, and $\pm$30% (dotted lines).

The results are illustrated in Figure 4.9 for three interesting replacement algorithms, namely our optimized GClock in 4.9(a), 2Q in 4.9(b), and ARC in 4.9(c). Results for LRU-K and LRU are omitted because the former delivered similar results as we got for 2Q and the latter always delivered accurate simulations without any error.

As we expected, the GClock-based simulations are fairly accurate for any simulation size and the error always is below 10% but in most cases even below 2%. The 2Q simulations disclose an interesting effect: Larger simulation sizes (i.e., $\pm$30%) deliver nearly perfect accuracy with errors below 1%, while closer to the reference size (i.e., real buffer size) the error increases. In those cases, the accuracy of overflow simulation (i.e., SBPX) seems to improve during the course down to an error below 5%, while the simulation error for hotset marginally increases and settles at 10%. The results for ARC show a mixed situation. In the beginning, the error of all simulation ranges is oscillating. Except for the 30% overflow simulation, the

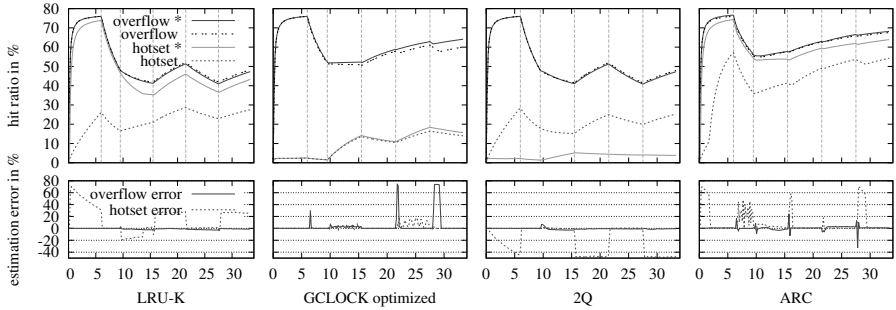


Figure 4.10: Shifting workload analysis (buffer calls x 100.000 on x-axis)

errors are always close to 10% in average. We believe that those little error margins, even for those large simulation sizes, still justify the applicability of our extensions.

4.5.3 Workload Shifts

Based on our accuracy and drift analysis, we examined *workload shifts*. To guarantee that we hit "interesting" buffer (simulation) sizes for these shifts, we extended our predefined workloads from Section 4.5.1. Each result in Figure 4.10 shows for 5 workload shifts the cumulative hit ratios for hotset and overflow simulations as well as their estimation errors. The vertical lines in the upper illustrations indicate the workload shifts, which (as expected) come along with kinks in the graphs. We omitted LRU simulations here because they always deliver fully accurate results.

On the one hand, we have the results for LRU-K and 2Q, which disclose serious problems for certain workloads, while hit ratio estimations for others are nearly perfect. For example, scan-dominated corner cases such as the third and fifth workload shift cause hotset errors of up to 25% and 40% until the next shift. On the other hand, estimations for GCLOCK optimized and ARC seem to be more reliable. Some workload shifts cause short peaks, where the simulation needs to adapt itself. However, most of the time, only marginal estimation errors below 2% are observably.

The workload shift analysis shows that for many cases, estimation errors are quite low. Certain corner cases, i.e., extreme workloads causing a kind of thrashing for the simulation ranges, are hard to estimate and may lead to weak estimations. Because those corner cases are rather exceptional, we believe that they do not affect the benefit of our algorithmic extensions for resize simulations in general.

4.5.4 Buffer Balance

For the balance benchmark, we let the self-tuning mechanism presented in Section 4.4.2 automatically tune two buffers. The results in Figure 4.11 show the hit ratio characteristics and the according buffer size changes for two buffers in each diagram. In the left scenarios, buffer 0 was fed with *random* workload from Figure 4.5(a) and buffer 1 with *scans* shown in Figure 4.5(b). In our second setup on the right, we again use two buffers, one that is fed from the workload *jumps* and the other from the workload *real* as shown in Figure 4.5(c) and Figure 4.5(d). Buffer sizes (i.e., simulation and real) are chosen as described in Section 4.5.2.

GCLOCK optimized and *2Q* were chosen for this benchmark because they represent a mixture of the results we got for all five algorithms. For comparison, we used a fixed memory shift granularity of 2% of the database size. Triggered by the cost model, tuning and memory shifts were preceded by a warm-up period of 1.2 Mio buffer calls.

In Figure 4.11(a) (left), the *random* workload buffer was shrunken according to its hotset simulation, whereas buffer 1 was increased. Although the hit ratio of buffer 0 slightly descends, the overall IO performance improves, because the hit ratio of buffer 1 increases considerably. Comparing this result to 2Q's forecast and balance, as shown below in Figure 4.11(b) (left), its confirmed that the penalty for the *random* workload buffer is quite low but the increase for the other one is at a similar range ($\sim +20\%$). Although, in both cases, the buffer sizes were equal, the apparently different results are due to the elementary performance of *2Q*, which is better.

In both settings of our second scenario on the right, SBPX fails, because it does not recognize that the size of buffer 0 is close to a "jump" boundary. However, as indicated by Figures 4.11(a) (right) and 4.11(b), our hotset simulation detects the pitfall and prevents buffer performance penalties. For both algorithms, a slight performance improvement is gained through balancing.

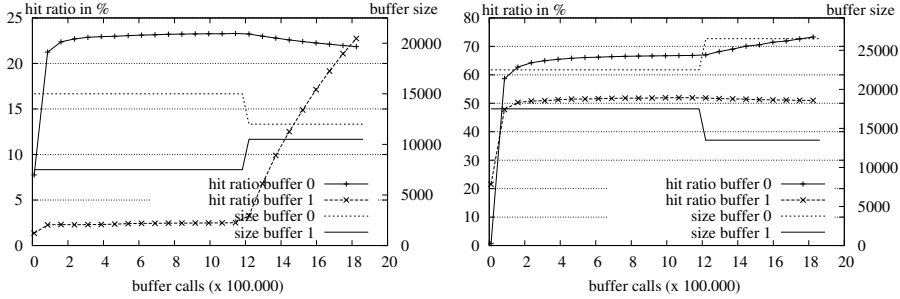
As explained above, resizing two buffers is fairly simple. Therefore, we combine both experiments in a single setup shown in Figure 4.12. The cut-out shows two memory shifts leading to minor descends of the hit ratio on the one side but clear improvements on the other side resulting in a steadily improved buffer performance.

In summary, we could experimentally prove that buffer balancing can be achieved at low cost, but it heavily depends on accurate and lightweight forecasts for both directions – upsize and downsize.

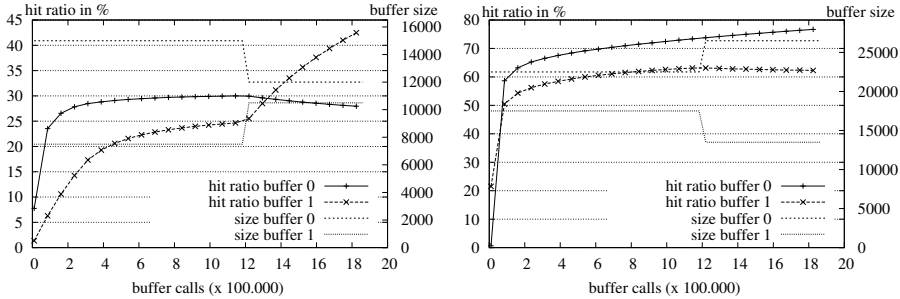
4.5.5 Overhead

Enabling our self-tuning mechanisms when operating on "normal" workload, i.e., database load causing physical IO, led to slight performance variations which could be attributed to measuring inaccuracies but not directly to the algorithmic overhead. Therefore, we extracted the algorithms and fed them directly with our generated workload to only measure the pure runtime overhead caused by hotset and SBPX simulations.

The algorithmic extensions are sensitive to the simulation size and the workload. The former defines the lookup size for "what-if" matches, whereas the latter determines the frequency how often the extensions are actually called. For both parameters, we measured the overhead for different configurations. On the one hand, we scaled the simulation size from $2\% - 15\%$. On



(a) Buffer balancing using *GClock optimized* (*random* and *scan* left; *jump* and *real* right)



(b) Buffer balancing using *2Q* (*random* and *scan* left; *jump* and *real* right)

Figure 4.11: Buffer balancing exemplified by *GClock optimized* and *2Q*

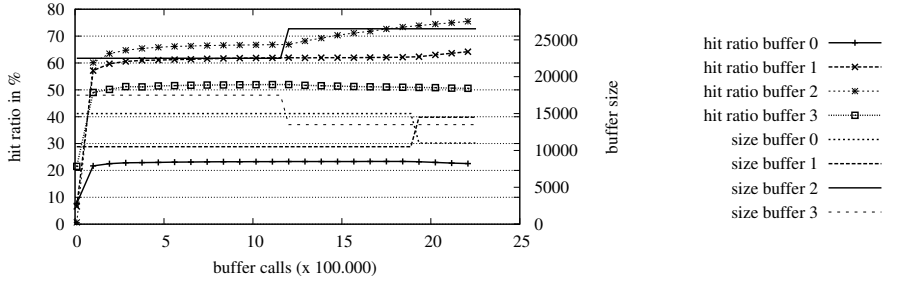


Figure 4.12: Balancing of four buffers under different workloads

Table 4.2: Overhead Analysis hit-dominated workload

Simulation size	LRU	LRU-K	GCLOCK	2Q	ARC
2%	0%	6.6%	0%	0%	68% (0.6%)*
5%	0.23%	28.3%	0.1%	1.8%	- - (2.4%)*
10%	4.33%	49.1%	2.68%	6.26%	- - (15.1%)*
15%	4.6%	58.8%	1.52%	6.9%	- - (16.0%)*

ARC simulation performance degraded too much (- -), instead a pointer-based version (*), delivered results comparable to 2Q.

Table 4.3: Overhead Analysis miss-dominated workload

Simulation size	LRU	LRU-K	GCLOCK	2Q	ARC
2%	22%	14.4%	8.5%	19.9%	10*x* (23.8%)*
5%	26%	35.6%	13.7%	24.9%	- - (31.5%)*
10%	27.7%	56.8%	11%	27.4%	- - (33.8%)*
15%	29.5%	67.4%	13.9%	28.6%	- - (33.9%)*

ARC simulation performance degraded too much (- -), instead a pointer-based version (*), delivered results comparable to 2Q.

the other hand, we evaluated a hit-dominated workload, more frequently calling the hotset simulation, and a miss-dominated workload, which more frequently calls the oversize simulation.

The results in Table 4.2 for the hit-dominated workload show that the algorithmic overhead is often negligible for small simulation sizes, but also relatively small for growing sizes. The pointer-based extensions, such as for LRU, GCLOCK, and 2Q, scale quite well and even cause only little overhead for large simulation sizes. In contrast, the original extensions for ARC and LRU-K require more efforts to (linearly) search in the history. Especially, ARC's overhead exceeded a reasonable limit of 200%[7] beginning from 5% simulation size. Therefore, we integrated the pointer-based approach into ARC, too, which delivered overhead figures that perform clearly better.

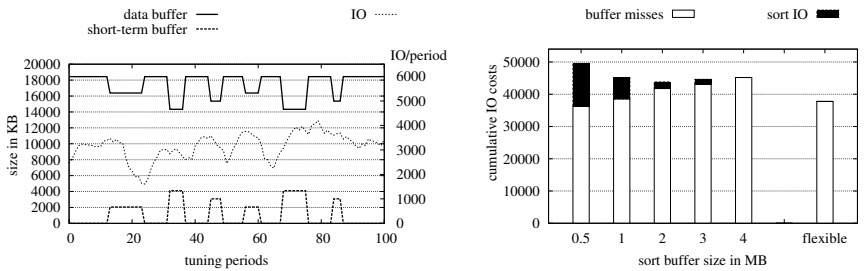
Although the results for the miss-dominated workload in Table 4.3 show that upsize simulations are more expensive than downsize simulations, in most cases, the overhead still pays off. Again, the original ARC approach can be substituted by a clearly cheaper pointer-based alternative.

In all cases, scalability solely depends on the algorithm and the more pointers or flags have to be maintained the more overhead is caused. But combining the hit/miss performance with pure algorithmic runtime figures and forecast quality, all algorithms except LRU-K have their specific advantages and show convincing results.

4.5.6 Integrating Short-term Memory Consumers

The effectiveness of on-demand buffer resizing to handle short-term memory consumers is demonstrated by the following test case.

[7] An overhead less than 200% is considered to be better than having 3 independent algorithms for the real, hotset, and SBPX size.



(a) Changing sizes and IO throughput

(b) Cumulative IO costs for static and dynamic buffer sizing

Figure 4.13: Integrating short-term memory consumers

There is one data buffer pool, on which multiple scans are repeatedly run causing an average miss ratio of 30%. Additionally, three sort operations are repeatedly performed at the same time. They have sizes of 2 MB, 3 MB, and 4 MB. The overall size of all buffer pools is 18 MB.

Due to the same mechanisms as for data buffer pool resizing, the on-demand integration of short-term memory consumers works as expected. The temporary increase of sort buffer sizes is illustrated in Figure 4.13(a). At the same time, we tracked the number of IO operations in a tuning period. The little oscillation is solely due to the duration of different buffer requests, e.g., misses imply costly page fetches. However, cumulative numbers of IO operations should be a valid indicator for the opportunities provided by our flexible adaptation. In Fig. 4.13(b), a similar workload containing two parallel clients and independent sort operations (third client) was executed while we used again a fixed upper limit of 18 MB for all buffers. This amount was either statically partitioned between sort buffer pools and data buffer pools or dynamically. The cumulative number of IO operations (i.e., buffer misses and additional sort IO[8]) necessary to process the same query and sort workload is shown. While an increased sort size also increases the buffer miss rate, i.e., buffer IO, it dramatically reduces the sort IO. On the right side, dynamic memory allocation avoided most of the sort IO while keeping the buffer miss rate at a very low level.

4.5.7 Read-ahead and Grouped Flush

Eventually, we measured the speed-up gained through our read-ahead and grouped (sequential) flush mechanisms. Results for both benchmarks are illustrated in Figure 4.14. For writing various workloads, i.e., storing XML documents in XTC, we got for the two different storage modes, elementless and full, speed-ups between 2% and 28%. This definitely results from grouped flushes exploiting sequential writing. For read-only workloads (i.e., a SAX scan), we got speed-ups between 4% and 35%. Here, the read-ahead mechanism takes effect, because

[8]The first read of sort items is attributed to the data buffer. Following merge runs are attributed to the *additional* sort IO.

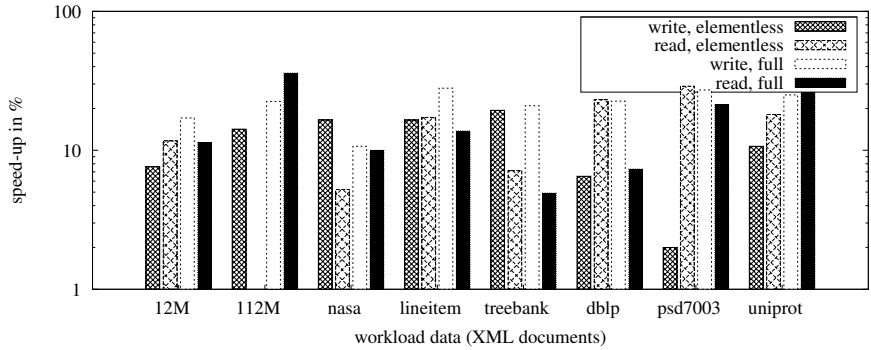


Figure 4.14: Read-ahead and grouped flush speed-ups

scans read all pages of a database object in consecutive order.

4.6 Conclusions

Buffer self-tuning is a critical aspect that incorporates multiple tuning techniques and areas. We have shown that individual replacement algorithms are extendable to not only *optimize* the current hit ratio, but also to *estimate* hit ratios for alternative buffer pool sizes. The accounting (i.e., *monitoring*) is permanently running, while the buffer performance *analysis* takes place in regular intervals. We have further shown how to improve the buffer configuration (i.e., *execute*) by changing the memory distribution and that the overhead for self-tuning pays off. As a buffer operates independently of a data model, our concepts are universal for all kind of DBMSs.

Chapter 5

Storage Self-Tuning for XDBMSs

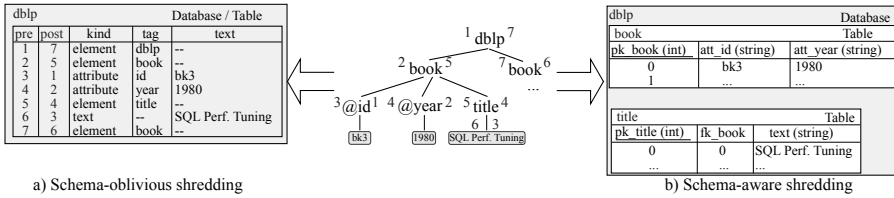
The large variety of XML data and XML processing possibilities makes it almost impossible to find a superior storage configuration for all of them. The workload (i.e., XML documents) may consist of up to millions of small configuration files, log messages, and (internet) application data (e.g., Web Services' WSDL, SOAP, XHTML) or large files such as protein databases [LDB+04], publication databases [Ley02], or text collections [DDD+09]. Their structure (i.e., depth, fan-out, and recursion) and amount of content (i.e., text length and share of text) may be totally different and thus needs to be considered when choosing an appropriate storage layout.

XML storage constitutes the foundation for DBMS-based XML processing. Although the storage layout often requires immutable design decisions such as XML mapping, data placement, or structure and content compression, knowledge gained through analysis or sampling of XML data or learning the effects of prior design decisions may allow for an autonomous configuration of those parameters.

In the world of XML processing (see: Section 2.4), different APIs and, therefore, different kinds of accesses to the XML store can occur. Here again, adjusted storage configurations may improve the performance for certain access patterns while downgrading the performance of others. Thus, design decisions need apriori knowledge about future usage. The most prominent categories of access patterns are *data-centric* and *document-centric* XML processing that are similar to their counterparts in the relational world, namely *OLTP* and *DSS*. Having a categorization for XML documents may help to guess their dominating usage pattern.

To reduce XML's redundancy and thereby processing costs, techniques to share common data structures such as the *Path Synopsis* or data indexes for multiple documents are available. However, the potential of savings has to be ensured before defining and building shared structures.

In this chapter, we will show how we can exploit XML compression techniques and workload analysis to tune the often static parameter selection for XML storage. Furthermore, we will analyze storage options for various kinds of XML processing and evaluate their performance. First, we introduce key concepts for native XML data mapping such as *node labeling* and different document mapping approaches, before we analyze optimization options and self-tuning abilities. Eventually, we evaluate the benefit of storage optimizations that can be achieved using a native XDBMS such as XTC. Note, due to the assumption that buffer tuning concepts, presented in Chapter 4, are working encapsulated within their layer and theoretically provide optimal IO performance for the underlying device equipment, this chapter only covers critical storage issues for XML data mapping on top of the buffer layer.



a) Schema-oblivious shredding

b) Schema-aware shredding

Figure 5.1: XML data shredding into relational tables

5.1 Native XML Storage

The physical representation of data is extremely important for the overall (X)DBMS performance, therefore dealing with XML data requires tailored storage mappings in the third layer (compare with Section 2.2, *access layer*). Not only space consumption is a critical aspect, but also read access and update performance. Therefore, a compact layout for the verbose XML structures is necessary, albeit tailored access operators need node-oriented and scan-oriented interfaces at the same time.

In [Mat09], an overview of different XML storage approaches can be found. In the following, we highlight the most important steps leading to state-of-the-art XML storage layouts.

In the beginning, so-called shredding solutions, i.e., mapping XML data to relational structures, emerged [STZ+99, CS01, TDCZ02]. Those approaches often require structural information in advance to instantiate an appropriate table structure and all the necessary referential constraints. To represent the data types for XML nodes, a different representation is required, e.g., mapping them to a numeric value. Quite a number of systems are based on shredding, either exploiting schema information such as [BFH+02, STZ+99] or without using schema information such as [YASU01, BGvK+06]. Figure 5.1 exemplarily depicts the shredding of XML data into schema-oblivious or schema-aware relations. This example shows that shredding may lead to a single table covering the entire XML document (e.g., Figure 5.1a) or to multiple tables (e.g., Figure 5.1b). However, case (a) requires a static pre/post numbering to address and identify XML entities. This prohibits efficient insertions of new nodes or subtrees, because the pre-/post-order numbers have to be updated as well, i.e., updating the entire table. In case (b), updates can be tracked easily, whereas preserving document order and handling schema changes require additional efforts[1].

While shredding is still a viable way to store XML, hybrid systems such as DB2 [NvdL05], Oracle 8i [BKKM00], and System RX [BCJ+05] allow to store XML data separately, i.e., using different containers for relational data and for XML data. Due to relational processing capabilities of the DBMS kernel, XML data and XML-aware operators need to be wrapped into relational counterparts. In Figure 5.2, the scheme of a hybrid DBMS engine (depicted as "Hybrid Compiler") is shown. Although the kernel needs to be extended for handling XML

[1] omitted here for simplicity

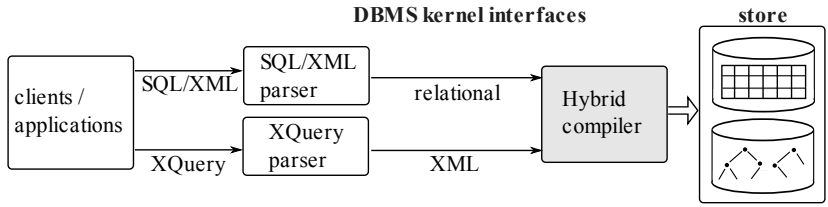


Figure 5.2: Hybrid XML storage

data, essential parts can be reused such as query optimization. Typically, tiny XML documents are inlined within a relational record as (C)LOBS to speed up processing. Because those systems anchor the XML data in a relation containing a column of type XML, the external XML storage lookup may not scale down for small XML documents. Thus, the classification of hybrid storage is once more eligible.

Another way of storing XML data exploits the tree structure of XML. In [FHK+02, BCJ+05], XML documents are divided and grouped by structurally similar subtrees. Because XML processing often requires path pattern matching, similar path targets (i.e., subtrees or nodes) are stored within the same container and thereby provide a high locality and better access performance. This deliberately chosen fragmentation alters document order and makes document reconstruction difficult. Moreover, navigational access, updating, or addressing larger parts of a document within a query requires access amongst several distributed containers (leading to random IO).

As mentioned already, native XDBMSs employ tailored techniques to natively store XML data. However, the more systems emerged the more differing ways of physical XML representation appeared. Most of the native XDBMSs represent XML data in a node-oriented manner or at least as tree structures such as Timber [JAKC+02], Sedna [FGK06], OrientX [XXM+06], eXist [Mei09], and our own prototype XTC [HH07]. Native storage systems provide native access methods (APIs) and operators. Furthermore, because data mapping is XML-aware, aspects such as page and index layout, log entries, lock protocols, etc. can be perfectly optimized for XML.

Eventually, a storage system for native XML data processing needs to provide the following properties:

- *Flexible operator support*. Streaming operators (SAX API) and navigational access operators (DOM API) are required at the same time. Moreover, declarative languages such as XQuery require both kinds of access options.

- *Efficiency*. Optional compression features and scalable data structures for XML mappings are required when storing and reconstructing XML documents. In particular, document-based systems such as messaging, logging, or archiving benefit from compression, because IO reductions lead to shorter runtimes and reduced logging overhead.

- *Modifiability*. Besides IUD for entire XML documents, fine-grained support for mod-

ifying individual nodes, the structure, and whole subtrees is required, especially for transactional multiuser environments.

- *Secondary access paths*. The storage subsystem must support indexes to allow tailored document access and processing.

- *Round-trip*. A storage system must define its level of round-trip guarantees it provides. Either by guaranteeing identity at the byte level when comparing the input document and output document, or by guaranteeing data consistency ignoring structural formatting issues (e.g., line breaks between element nodes[2]).

5.2 Node Labeling

Recent research [HHMW07] has shown that the node labeling mechanism plays an essential role for storage space consumption and efficient support of navigational and declarative query processing. Therefore, appropriate node labeling is not only needful to realize the aforementioned storage properties of Section 5.1, but also to enable flexibility and performance of the entire internal system behavior.

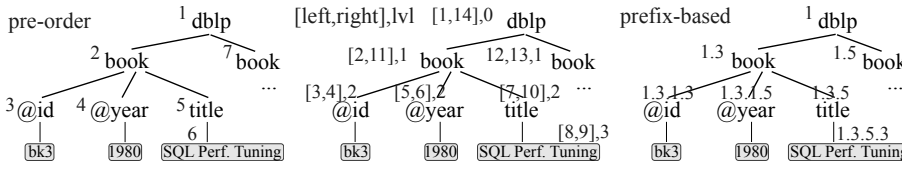
Various kinds of labeling schemes emerged throughout the research community, where different design goals led to totally different concepts. The only thing they have in common is the unique identification of XML nodes within a document. In Figure 5.3(a), ascending numbers are assigned to document nodes while traversing the XML tree in pre-order[3]. Thus, each node can be uniquely identified and addressed.

Based on the uniqueness of document nodes, additional properties are desirable by a node labeling. A very important aspect for XML processing is the determination of axis relationships for two nodes [DOM05]. Because XML is an ordered tree data structure, a label needs to identify the node's level and locate it within the document, thus preserving the document order. According to the flexibility and dynamics of XML (and XML schema definitions), a label is typically a value of variable length. Furthermore, concurrent document access requires transactional protection by acquiring locks on an XML document. The more fine-grained the lock protocol is defined the more parallelism is possible. Eventually, smart encoding and decoding for a compact physical representation is mandatory for storage interaction.

A lot of node labeling schemes can be found in the literature that address most of these issues [CKM02, CPST03, YLML05, HHMW07]. Almost all of them can be classified into range-based node labeling schemes and prefix-based ones. Throughout the entire work, we will often rely on the properties of node labeling, thus, we briefly introduce the two categories and show why a native XDBMS should be based on a prefix-based scheme.

[2] Byte-level identity requirements may expose unwanted behavior to the user because the DBMS system has to automatically process certain formattings. For instance, inserting a subtree between two sibling nodes, when 3 whitespace characters exist between them. The system has to keep them, but where – in front, post, fragmented?

[3] Attribute nodes comprise of the node name and value, text nodes such as *title* are split into an element node and a content node leading to individual node labels.



(a) Pre-order labeling (b) Range-based labeling (c) Prefix-based labeling

Figure 5.3: Comparison of different node labeling approaches.

5.2.1 Range-based Labeling

Assigning range-based node labels requires to traverse the XML tree in pre-order. Besides a pre-order number for each node, a second post-order number is assigned. For example, in Figure 5.3(b) the first *book* node is visited as second (in pre-order). After subtree traversal, the post-order visit leads to the range of $[2, 11]$. The two values span a numbering range containing all node labels of its subtree. Based on given range values, the descendant and ancestor relationship of two nodes can easily be decided by checking for containment. Even the following and preceding relationship can easily be decided, but for child, parent, and sibling relationships a third component of the label – characterizing the node's level in the XML tree – is necessary.

But range-based labeling has still some drawbacks when it comes to insertions. Even by leaving gaps while assigning the ascending numbers, insertions may exhaust them and cause expensive relabeling. Furthermore, relabeling would violate the consistency of secondary storage structures or other node references held by transactions i.e., locks.

5.2.2 Prefix-based Labeling

We recommend the use of prefix-based labeling schemes as sketched in the sample document in Figure 5.3(c). As its most prominent property, this labeling class annotates each node label with its parent node label as prefix. So far, a number of roughly equivalent prefix-based schemes are proposed in the literature. They do not only support all XPath axis operations and hierarchical locking schemes (because all nodes in the ancestor path can be easily derived from a node label), but also dynamic insertions without relabeling. Figure 5.4 exemplifies the insertion of a new node between existing and densely numbered nodes. Because only odd division numbers contribute to the level count, even numbers can be used to indicate overflows while preserving a correct logical order. Using DeweyIDs [Dew], as labeling mechanism, enables specialized attribute node mappings and – with a *dist* parameter used to increment division values and to leave gaps in the numbering space between consecutive labels – a kind of adjustment to expected update frequencies. Any prefix-based scheme such as OrdPaths [OOP+04], DeweyIDs [Dew], or DLNs [BR04] fulfills the desired properties mentioned in Section 5.1.

Table 5.1: Example Huffman code for DeweyID encoding (L_i fields).

Code	L_i	Value range of O_i
0	7	1–127
10	14	128–16,511
110	21	16,512–2,113,663
1110	28	2,113,664–270,549,119
1111	36	270,549,120– 2^{37}

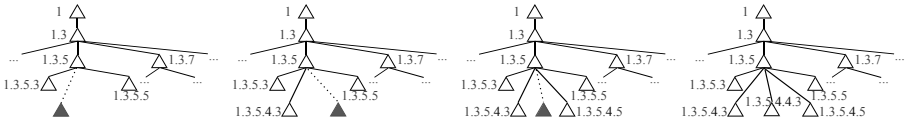


Figure 5.4: Overflow mechanisms for dynamic labeling using DeweyIDs

5.2.3 Conclusion

Based on the analysis of possible labeling alternatives, we will only use DeweyIDs throughout this work because of their advantages. They provide powerful mechanisms for query processing that are far beyond common row identifiers known from relational systems.

5.3 Node Labeling in XTC

As already mentioned in Section 5.2, prefix-based node labeling schemes provide the highest flexibility and stability and provide opportunities for compression. Therefore, XTC is using DeweyIDs for node labeling [HHMW07].

Due to the large variance of XML documents in number of levels and, even more, number of elements per level, we cannot design a (big enough) fixed-length storage scheme of DeweyIDs. For the sake of space economy and flexibility, the storage scheme must be dynamic, variable, and effective to capture tall/flat trees with varying fan-outs per node. At the same time, it must be efficient in storage usage, encoding/decoding, and value comparison at the bit/byte level.

Because DeweyIDs can become fairly long (i.e., a lot of (overflow) divisions) and the "natural" representation of dot-separated numbers is unhandy when processed automatically, we encode them in byte arrays using Huffman codes to compress them at the same time. Huffman codes are a general mechanism to serve byte-oriented compression and comparison requirements. For example, the i-th division value can be represented as a pair $C_i \mid O_i$ where C_i is a prefix-free code used to assign a length value to O_i via a mapping table (or an equivalent Huffman tree). Table 5.1 illustrates a particular mapping of variable-length division values where, in addition, byte alignment is observed for each individual division. Because the codes can be freely chosen – regarding the definition of a Huffman tree and a order-preserving comparability – and the assignment of length values in column L_i is independent from them, tailored mappings can be derived for a document. Even in the encoded form, comparison of two DeweyIDs

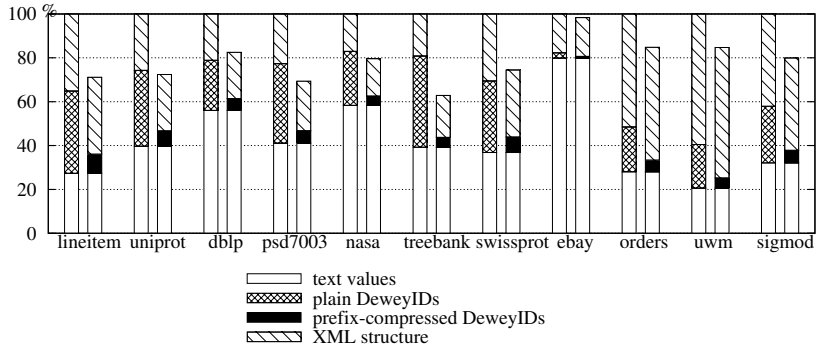


Figure 5.5: Prefix compression gain of DeweyIDs in XTC

or prefixes of them works at a level basis (and decides according to the document order). Therefore, we could refine our encoding mechanism and even choose Huffman encodings per level which could be adjusted to the node distribution of these levels. While such an optimization may save some space at the low percentage range, but may contribute to the implementation complexity, some heuristic encoding rules may be more effective. Frequently, large sets of nodes only occur at levels close to the document root, whereas the fan-out deeper in the tree is typically limited to a few and often to a single node.

For instance, our example code in Table 5.1 is applied down to a certain level, while at deeper levels an even more compact form of encoding is chosen, e.g., using code 0 and $L_i = 3$ allows the representation of values 1 – 7 with 4 bits, which is, however, not byte-aligned anymore.

Despite the claims in [LLH08], so-called quaternary codes or, more generally, use of separators [HHMW07] cannot provide fast bit-level comparisons and fail to support direct byte-level comparisons of encoded DeweyIDs.

When implementing Huffman-based prefix compression in XTC, the overhead of *DeweyIDs* is dramatically shrinked down to a reasonable size. In Figure 5.5, we compared the space consumption of DeweyIDs for a common set of XML documents [Mik], which was analyzed in detail in [SH07]. The share of text values and XML structure is always the same, independently of prefix compression. However, compression of DeweyIDs saves substantial storage space and can compete with relational RIDs typically having 4 bytes. Thus, the relational storage approaches, presented in Section 5.1, may consume the same or more amount of storage for node identification.

5.4 Full Storage Mapping

As already introduced in Section 5.1, documents are represented in XTC in a native way. That means, all XTC storage modes support declarative and navigational processing of XML documents as basic primitives. XML documents are stored in containers (i.e., files) that are

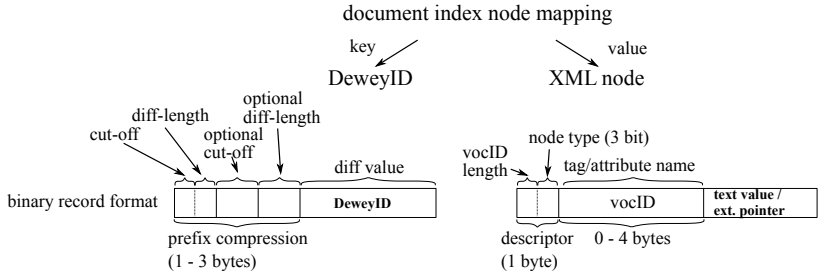


Figure 5.6: Physical record format (full storage mapping).

subdivided into equal-length units – logical pages. Page sizes typically vary from 4K to 64K bytes. XTC allows the assignment of several page types to enable the allocation of pages for documents, indexes, blobs, overflow records, etc. in the same container.

The *full* storage mapping is a fine-granular DOM-tree representation that easily preserves the so-called round-trip property when storing and reconstructing the document[4]. The *full* storage mapping is flexible enough to accommodate arbitrary insertions and deletions of nodes and subtrees, where the document storage structure is dynamically balanced. Storage-based node navigations to parent/child/sibling nodes are directly possible due to the DeweyID labeling.

We use a B*-tree as the fundamental storage structure. Because these trees easily maintain the document nodes and provide logarithmic access costs (based on B-trees [BM70, BM72]). Even node updates such as changing a tag name or value as well as node and subtree insertions or deletions are perfectly supported by the combination of DeweyIDs and B*-trees.

Node Mapping

Individual XML nodes are mapped to variable-length byte arrays, as shown in Figure 5.6. Because we use B*-trees, we require key/value pairs for index entries. Therefore, a node is split into its key part – the node label (DeweyID) – and its value part, which is type-dependent. As already introduced, DeweyIDs are perfectly suited for prefix compression, shown by the *cut-off* and *diff-length* fields. Using the Huffman encoding, DeweyIDs are easily transformed into byte-aligned byte arrays preserving their logical order at the byte level, too. Thus, two consecutively stored nodes carrying "neighbor" labels only need to store their differing parts. That means, the first *cut-off* bytes are equal and need not to be repeated while the differing part – *diff-length* – is stored in the *diff-value*. Usually, 4 bits are enough to encode the two length fields, but in case of overflows (i.e., both bits are set to 1), an additional byte can be used to encode the length values.

The value part of an index entry is encoded as follows: For XML tag names, a vocabulary is maintained, as shown in Figure 5.7, to translate strings into simple (short) integer values.

[4]Round-trip property means that the identical document must be delivered back to the client which was stored before. However, performance considerations weaken this requirement a little bit, while document structure and content are preserved, formatting features may be stripped.

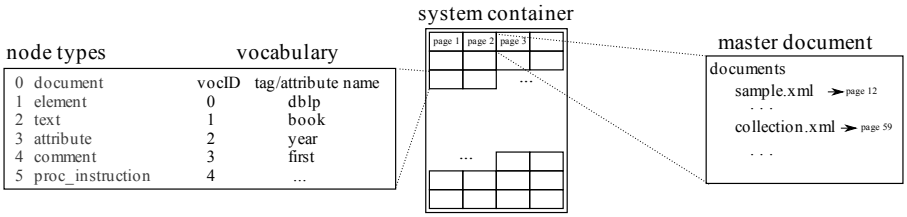


Figure 5.7: Database metadata structures.

Moreover, the figure shows our node type encoding requiring 3 bits only. In a descriptor byte – shown in Figure 5.6 – vocabulary ID length and node type are encoded. In case of an attribute or element node type, the following bytes are used to encode the vocabulary ID itself. The byte array carries content as payload (i.e., texts, comments, or instructions).

Document Mapping

A central role in storage mapping plays the *master* document (internally called "_master.xml"). It is stored in a separate system container, depicted in Figure 5.7, together with the mapping tables for node type encoding and vocabulary encoding. Because the master document itself is a normal XML document in the database, it uses the same mechanisms for storing, retrieval, querying, and manipulation.

The master document takes care of referencing database objects such as XML documents (including itself), collections, indexes, statistics, and blobs. Only the entry point of XTC, in form of a page number has to be specified from the outside to get the system started.

Each document or collection in XTC is employing its individual *document index*, shown in Figure 5.8. Typically, user documents reside in so-called data containers. An XML document is serialized into a sequence of nodes that are consecutively stored in a B*-tree structure.

The document index is separated into two logical parts. Key/pointer pairs (DeweyID, page pointers) are used in the first part (shown as triangle) for indexing the first node in each page of the *data container*. The second part consists of a set of double-chained pages carrying the actual document nodes. For convenience, nodes of the sample document in Figure 5.8 are depicted in the data pages in a simplified form.

The value part of an index entry (i.e., node value) is materialized (stored inline) up to a parameter *max-val-size* together with the node's metadata. When the content size exceeds *max-val-size*, it is stored in referenced mode (i.e., *ext. pointer* in Figure 5.6).

Document Access

According to the requirements presented in Section 5.1, documents stored in XTC can be processed at the storage level in various ways. The most common access primitives comprise *scans* and *navigation*. Full document scans are fairly easy to perform because the entry point to

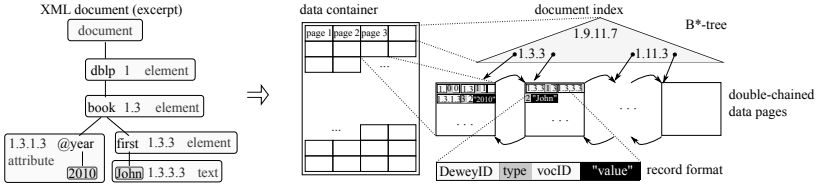


Figure 5.8: XML document mapping (nodes-to-page serialization).

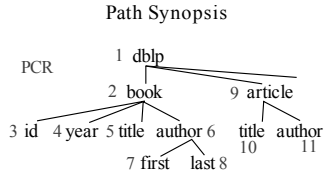


Figure 5.9: Path synopsis for sample XML document

the double-chained node sequence is referenced in the master document, i.e., the page number containing the document root node. Various access operators are based on that scan, which may automatically apply simple filtering and complex predicate evaluation.

Scanning a document without any filtering can be used to reconstruct the original XML document for retrieval or external use.

On the other hand, navigational primitives based on DOM are possible by exploiting the node labels; in our case, DeweyIDs. They allow to easily traverse the document store or jump through the document index' B*-tree part exploiting the concept of divisions and levels.[5]

Any kind of document manipulations can directly be performed at the storage level. Besides primitives for node IUD operations, XTC provides means to operate on subtree instances for insertion, deletion, or replacement. Here, the variable-length record format, prefix compression, and B-tree-based storage structure permit high-performance document changes.

5.5 Path Synopsis

While optimizing native XML storage structures for XTC, we developed a new storage concept that will be presented in the following section [SH07]. The cornerstone of this storage concept is our so-called *path synopsis* (or short PS) that is also exploited for several other optimizations in the field of storage, indexing, querying, and self-tuning throughout this work. Therefore, we want to emphasize its importance for many of the presented concepts.

XML documents usually have a high degree of redundancy in their structural part, i.e., they contain many paths having identical sequences of element/attribute names. Such identical path

[5] A cache mechanism can be used to keep the nodes of a decoded page in an object-oriented format, thereby allowing binary search within a page.

Table 5.2: Node and path statistics for exemplary XML documents

Document	Decription	Size	Nodes			Paths
			elem	attr	content	
lineitem	TPC-H data	32 MB	1 Mio	8	1 Mio	17
uniprot	Universal protein resource	1.8 GB	36 Mio	46 Mio	53 Mio	121
dblp	Computer science index	330 MB	7.5 Mio	1.5 Mio	8 Mio	153
treebank	Wall street journal records	86 MB	2.4 Mio	1	1.4 Mio	220k
psd7003	DB of protein sequences	716 MB	21.3 Mio	1.3 Mio	17.2 Mio	76
nasa	astronomic data	25.8 MB	476.646	56.317	371.593	73
XMark	Artificial auction data	12 MB	167.864	38266	156.407	439
		112 MB	1.68 Mio	381.870	1.5 Mio	451
account	TPoX benchmark documents for accounts, orders, and securities	∼6 KB	∼130	∼20	∼100	∼100
order		∼2 KB	∼10	∼80	∼80	∼83
security		∼6 KB	∼50	∼5	∼35	∼53

instances are represented uniquely as a path class in our PS. This kind of path summary can be captured in a small main-memory data structure; the basic idea is similar to that of a DataGuide which, however, is used as a structure overview for the user, for storing statistical document information, and, thus, enabling query optimization [GW97]. In addition to that, our primary use of a path synopsis is for structure virtualization, hierarchical locking, and empowering indexing and query processing [HH07]. For all these usages, we have equipped every node in the path synopsis with a path class reference (PCR) number, as illustrated for the sample document in Figure 5.9. In our system, a path synopsis obtains its full expressiveness by the interplay of PCRs and DeweyIDs: a DeweyID delivers all DeweyIDs of its ancestors while a PCR connected to a DeweyID identifies the path class a DeweyID-identified node belongs to. For example, starting from an arbitrary content, attribute, or element node – whose unique position in the document is identified by its DeweyID – that is associated with a reference to its path class, it is easy to reconstruct the specific instance of the path class it belongs to. This usage of the path synopsis indicates its central role in all structural references and operations. To increase its flexibility, we provide indexed access via PCRs and hash access using leaf node names. Additional links between vocabulary IDs (vocIDs) and their occurrences in the path synopsis offer direct entry points for further navigational steps and matching/searching operations starting at non-leaf nodes.

The path synopsis has to ensure that (1) each path in the document is represented and (2) that each path of the synopsis actually exists in the document. However, the second requirement may be softened while not violating consistency requirements for XML processing, as we will see later in this work. Several individual documents or a collection of documents may also share a single path synopsis. Document updates that introduce new paths have to ensure that they also update the path synopsis structure.

5.6 Elementless Storage Mapping

Structural redundancy of XML is one of the main issues in database-oriented XML document storage. Therefore, a novel technique of structure virtualization was described and evaluated in

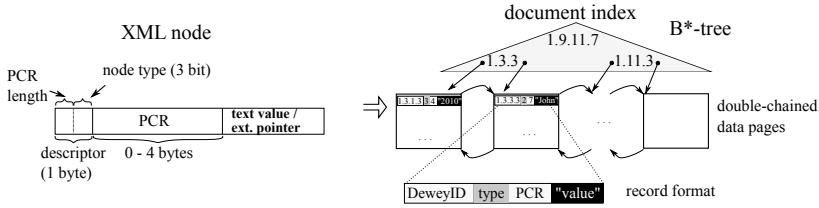


Figure 5.10: Elementless node mapping

[SH07, HMS07]. The so-called elementless document storage mapping does not contain any structure node (element nodes) in its physical representation, i.e., the document container only stores the content (leaf) nodes, each equipped with a DeweyID and a PCR. For our sample documents from [Mik, NKS07], we summarized important figures such as node instances and path instances in Table 5.2. Moreover, Figure 5.10 shows the physical record format for the value part as well as the layout of document index entries. Note that the remaining storage structures are equal to that of the full storage mapping, presented in Section 5.4.

Document Access

As we have already shown in Section 5.5, it is easy to reconstruct all paths and nodes on demand, e.g., when referenced during the evaluation of an XPath/XQuery expression, and it is even possible to perform navigation on this virtualized structure. Thus working with a document or collection stored in *elementless* fashion guarantees fast and cheap main-memory access to the path synopsis.

All kinds of document manipulations, which are possible in full storage mode are applicable in the elementless mode, too. However, some peculiarities may occur. For instance, the term "elementless" has to be refined when storing an empty element node (e.g., the document or subtree: `<xml></xml>`) without any attributes or child nodes. In that case, we use a placeholder for the element node carrying no content. Another important difference to the full storage mode may occur when a node is renamed. Because the node's PCR value changes, to an already existing one or a new one, all PCRs of descendant nodes have to be updated as well, which may cause a lot of updates and IO.

5.7 Document Collections

Multiple XML documents may be stored in a *collection*, a logical compound that makes documents, typically originating from the same domain, available under a common address. Collections have two major objectives, namely joint query processing and joint storage structures. In XTC, an artificial root node, to which all documents are attached, serves as proxy for collection processing. In contrast to the order-preserving requirement of the XML specification, in a collection, each individual document is only a subtree of the artificial root node – without

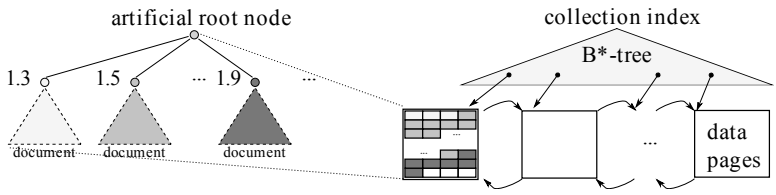


Figure 5.11: Collection mapping in XTC

a specific order.

Although a collection is independent of the used storage mapping, we favor elementless storage in XTC. As path synopses in a collection are shared between the documents, the benefits for joint query processing, statistics management, and indexing are essential. Moreover, storage savings for (tiny) documents may become a determining aspect for collections due to the shared data structures, i.e., B*-tree indexes. Figure 5.11 shows the logical view of a collection store and its documents. Each document subtree (depicted as triangles) is stored consecutively in the data pages of the B*-tree.

Access to document collections is gained through the same mechanism that is used for single documents. Especially scan-based access operators exploit the notion of node sequences, according to the XDM. Therefore, XTC only needs to provide all document root nodes in a single sequence. The other way around, a single document in XTC that is not part of a collection, is accessed through its root node wrapped in a node sequence, making document and collection access uniform for all kinds of processing.

5.8 Self-Tuning for XML Storage Configurations

Not only XML itself provides flexibility but also the native storage concepts shown in the last sections. Besides a storage management purely tailored to reduce space consumption, which seems to favor the *elementless* storage mapping all the time, we require usage-dependent optimizations for XML processing, too. Available parameters have to be adjusted to the expected workload. If applied, computational overhead for structural and content compression or building document collections has to pay off. APIs such as SAX and DOM behave differently and favor different configurations. Thus, together with the underlying buffer, the storage layout and configuration are the most important IO performance drivers. Because many storage parameters are immutable (for a distinct document or collection), they need to be chosen in advance. Knowledge about future processing is typically not available, but similar database objects may already exist. Efficient identification of those similar objects and comparison may help to at least guess future usage and thereby adjust the configuration accordingly. However, large documents may slow down database processing if being analyzed in advance. Sampling techniques or offloading may reduce these efforts. To enable an effective workload-dependent storage management, i.e., configuration, the database may exploit a monitoring, analysis, decision framework covering the usage and performance of existing objects.

In this section, we show several techniques for content compression, which are orthogonal to the structural compression presented in Section 5.6. Furthermore, we show how to gather meaningful statistics supporting document classification and providing decision support for storage options.

5.8.1 Compression

Many XML-specific tuning options are directly related to the verbosity of XML. Therefore, many existing compression techniques have been adapted to the XML data model (i.e., distinction between structure and content) or new compressors have been tailored to certain data types, XML domains, or XSD-awareness.

Although almost all of these techniques address the same issue – verbosity – not all of them are applicable in a DBMS environment. We distinguish between three different kinds of compression techniques for XML.

1. Document-oriented compression
   Those compressors "squeeze" the entire XML document into a binary format that does not allow for further data model-aware processing, i.e., only decompressing the entire document at once. Thus, document modifications result in full decompression and compression cycles. The handling is comparable to zip-style compression tools.

2. Block-oriented compression
   The original XML document is split into chunks for further processing. However, in favor for a high compression rate, the chunks are typically too coarse and, thus, the overhead for decompression and compression is unacceptable for database processing. Furthermore, the chance of skipping a compressed block to avoid further inspection during document processing heavily depends on data distribution (e.g., selectivity, clustering) and detailed knowledge of the block's content may be necessary, which is often impossible due to compression.

3. Node-oriented compression
   Individual nodes are compressed one at a time. Especially for database-centric XML processing, preserving the logic of XML nodes and content seems to be inevitable to allow for declarative query processing based on XML node sequences as in XQuery/X-Update. However, the node-wise compression overhead needs to be considered.

There are more properties to characterize a compression technique such as compression ratio, overhead for compression or decompression, and IO and CPU impact in a multiuser environment, which will be addressed during the evaluation in Section 5.10.5. An overview of existing XML compression approaches and reasons why they are not applicable in an XDBMS can be found in Appendix B.5.

5.8.2 Document Statistics

Detailed knowledge about the structure and content of XML documents can help to fine-tune storage parameters. According to the flexible storage concepts presented in Section 5.4 and

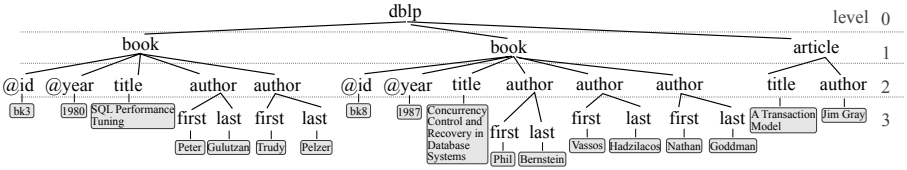


Figure 5.12: XML sample document for statistics

Section 5.6, various document statistics can be exploited when storing new XML data [SH07]. Here, we will give an overview of all the document statistics that are meaningful for the following sections.

Statistics can be distinguished between structure-relevant and content-relevant. Hence, we explore both kinds of statistics for our sample document given in Figure 5.12.

Structure Statistics

The structure of an XML document is (often) used to model application data. Thus, the storage and processing of XML is also influenced by the model and, therefore, by the structure of the document.

Beginning at the "external" side, i.e., XML files, file size is an important parameter. However, in Section 5.1, we have shown that (X)DBMSs typically avoid the verbosity of XML and, thus, the file size is actually an indicator for the expected storage size and time[6].

Basic XML statistics cover the number of (*element*, *attribute*, *text*, *) *nodes*, maximum depth of a document (see *level* in Figure 5.12), the *average depth* of a node, number of distinct paths ($max(PCR)$ in Figure 5.13), and number of distinct element/attribute names (*vocabulary size*). Note, for simplicity we do not care about other node types such as processing instructions or comments because they do not occur very often and, therefore, are negligible for the "big picture".

More sophisticated statistics include instance counters for path classes, average and maximum *fan-out* numbers of element nodes as well as information about mixed content occurrences and subpath recursions.

Content Statistics

In our work, content is related to text nodes and attribute values, where each sibling text node in a mixed content children set of nodes is considered independently. Basic statistics cover the total amount of text (typically in *bytes*) and the *average length* of a text value.

More sophisticated statistics include level-wise content information (i.e., vertical content distribution), horizontal content distribution, and q-gram or word-based counters of distinct

[6]Note, in really unusual cases this limit may be exceeded, too, and we do not consider dynamic imports such as referenced XSD or DTD files.

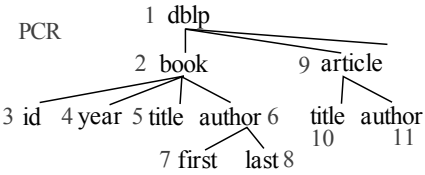


Figure 5.13: Path synopsis for sample document

structure

file size [MB]	**1.2**	distinct paths	**11**
elements	**23**	vocabulary size	**9**
attributes	**4**	path class instances	
text nodes	**18**	pcr|1|2|3|4|5|6|7|8|9|10|11	
max/average depth	**3/2.18**	# |1|2|2|2|2|5|5|5|1|1 |1	

content

amount of text [bytes]	**78**
average length of text value	**78/18 = 4.3**
wordbook	**'1980', '1987', 'A', 'bk3', 'bk8', ...**

Figure 5.14: Structure and content statistics for sample document

values. We do not consider datatypes in this work, although it may slightly improve compression capabilities, but at the same time, increase its recognition and calculation overhead.

According to [SH07], Figure 5.14 sums up the most meaningful statistics exemplified for our sample document from Figure 5.12.

In the following sections, we will explore different possibilities to gain good (enough) statistics and see how to exploit them for an *ASM - Adaptive Storage Manager* [SH07].

5.8.3 Classification of Documents

When working with XML documents in a (X)DBMS environment, it is beneficial to know before what kind of XML data is processed and what kind of operations can be expected. This may allow sharing common metadata and secondary data structures (e.g., compression indexes) between several XML documents. Note, research proposed to use XML-defining documents but, if at all, limited itself to DTD-based classifiers or similarity metrics [NJ02, KSH02]. Although XSDs offer more flexibility and should be the preferred method to define XML data, our approach does not require the presence of any DTD or XSD. Thereby, we can classify all kinds of XML data in the same way[7], which is also supported by [MBV03], where the authors conclude that 52% of the XML documents in the Web are schemaless.

For native XML storage two classification categories are meaningful:

1. Document-centric or data-centric XML processing
2. Similarity of XML documents (compared to existing database objects)

In the following, we look at both classification approaches.

Document-centric or Data-centric XML

Besides traditional workload analysis (offline and online), the initial process of storing an XML document needs to "guess" the prospective usage or requires some help by the user. There are

[7] A schema may improve the classification, but it is only declaring possible document shapes instead of the actual one. Furthermore, schema evolution and dynamic schemas make it more impractical.

several metrics how to differentiate usages. For instance, by evaluating the share of read and write operations, where secondary structures (e.g., indexes) need to be maintained as well. Furthermore, access patterns may serve as distinguishing feature such as access frequency, value range, and processed data volume. Thus, in the first place, we start to classify the workloads present in native XML databases. In doing that, we identify two major kinds of usage patterns for XML documents – collections of small (tiny) documents often processed in a document-centric way and single big (huge) documents usually processed in a data-centric way.

- **Document-centric usage**
  XML database access for document-centric processing primarily retrieves and stores a document as a whole. This is often the case for tiny documents with storage sizes less than a database disk page or for documents having only a small structure part which often results from automatic data transformations into the XML representation, e.g, "binary" data, articles, or books. Although support by structural indexing is possible, the additional IO of secondary indexes often does not pay off. Text collections and systems exchanging XML documents (Web services, messages) benefit from downloading or retrieving entire documents to process them at the client side. Even if documents are evaluated or modified at the server side, e.g., using XPath or XQuery, tiny documents (with a size less than a (few) disk page(s)) benefit from being processed as a whole. As a result, they fit into a single (or a few) buffer page(s) and only require marginal main-memory space. Furthermore, content-related queries requiring (keyword) search often dominate the workload for these kinds of XML documents, where an additional full-text index is usually oversized or too expensive.

- **Data-centric usage**
  For example, benchmarks like TPoX [NKS07], XML stream processing, and (semi-) structured data sets embedded into XML documents (e.g., relational data) use query languages that rely on index support to optimize selective data access [BBON06]. Often, tiny document fractions, aggregated data, or a few element nodes constitute the final query result. In such cases, exploiting indexes to minimize disk IO is essential for huge documents.

The concepts developed in this thesis are applicable for both kinds of XML usage, however, some fine-tuning measures only target at data-centric domains, where highly selective access to huge amounts of data is essential. This kind of data access and manipulation is comparable to OLTP (Online Transactional Processing).

Similarity of XML Documents as Database Objects

When storing XML documents, similarity identification helps to avoid redundant work. For instance, wordbooks, path synopses, metadata catalogs, which exist for stored documents, do not need to be recreated for new documents if their similarity is high enough and thereby a joint use is reasonable. Moreover, the evaluation of a storage configuration is often restricted to a point in time when further adjustments are impossible. Thus, a knowledge base for the

gains of prior document configurations and a similarity measure for new documents may be useful to increase the quality of storage parameter selection.

There exist a lot of different similarity measures for the XML domain, albeit the majority is based on structural properties using a variation of the so-called *tree edit distance* [ZSS92]. Those measures calculate the *distance* between two XML documents by counting the number of insert, delete, or update operations necessary to transform one tree into the other one [Bil05]. The shorter the *distance*, the more similar are two documents. Several algorithms emerged to speed up similarity detection, but do not take the content into account [FMM+05, Hel07, ABG10]. This issue is better addressed by the IR (information retrieval) community [DG07], where structure and content is analyzed to categorize and cluster XML documents. Inspired by XML's structural power, semantic clustering [TG10] is also possible and tailored similarity operations (e.g., joins) for XML [ABG10] are developed.

Because categorization and clustering will be done during the storage process, we restrict the analysis to storage-relevant similarity measures and pay attention to their overhead.

As several comparison features are possible, we will briefly introduce the most important approaches:

1. **Counters**
   Based on *file size*, number of *elements, attributes, text* nodes, *amount of text*, and *average text value length*, sketched in Figure 5.14, similarities and differences at a high abstraction level are visible. For instance, it is fairly uncommon that GB-sized documents will be processed (and therefore stored) the same way as KB-sized ones, or that documents having no text nodes are from the same domain as documents that are dominated by text (nodes).

2. **Structure**
   Exploiting again the path synopsis from Section 5.5, sets of path classes can be directly compared. A completely calculated tree edit distance is not meaningful, because we do not want to transform a document. Moreover, it is not meaningful to operate on an abstract summary (e.g., path synopsis) where, for instance, ordering is negligible. For storage purposes, it is enough to know the number of common path classes and different path classes which would have to be added (i.e., merged into the existing path synopsis). However, instance counters for path classes allow weighting the importance of path classes and, thus, further supporting structural similarity measures.

3. **Wordbook**
   It is very typical that similar documents employ similar wordbooks (i.e., XML tag and attribute names, or documents from the same domain or based on the same XSD/DTD). Comparing two wordbooks is simple but (often) provides sound criteria for similarity evaluation[8].

4. **Content**
   Independent of structural similarity, the content of XML documents can be compared

[8] In case of an XSD or DTD description available, the wordbook comparison may be done based on them without losing to much accuracy.

as well. Many approaches in this field originate from the *information retrieval* community, where edit distances (e.g., Levenshtein), subword matching (e.g., n-gram splitting), or exact (unordered) word matches are used to compute the degree of similarity[9]. However, the simple technique, exact word matches, is sufficient for us. In contrast to structural comparisons, frequencies will not be so important here. Later, we will see how this approach is extended to character-based comparisons (e.g., documents of the same language have character-wise similar frequency values, which will be exploited for compression options).

5.8.4 Analysis Options

Analyzing documents before they are actually stored in the database is necessary to setup immutable storage parameters. Those parameters can only be changed if the document(s) is removed and stored again, which however is often not reasonable because of running transactions, secondary data structures (i.e., indexes), and last but not least the overhead (i.e., IO costs). Details about storage costs can be found in Section 5.10.2.

The goal of an analysis step is to get detailed knowledge about the XML structure and content (Sections 5.8.2 and 5.8.3). The incoming document can be matched with existing documents and the best-matching collection or document can be identified[10]. In case of a collection, an insertion of the incoming document into the collection can be examined, too. When a document was found to be a "close enough" match, its storage parameters can be applied to new ones. Note, although used before, these storage parameters are not necessarily advantageous. Therefore, feedback about the benefit of former storage decisions is required.

During an analysis step, certain storage options can be tested (i.e., trial and error). As long as their overhead is low enough, such a search may be beneficial. For instance, a subset of the document can be transformed into a database's internal representation and different text compressors may be applied. However, a good knowledge base containing information about former storage decisions seems to be a (more) viable solution.

Document analysis can be done in two ways, either by analyzing the entire document before it is actually processed again for storage or by sampling on a representative subset of the document:

Pre-Analysis

If the entire XML document is present, so-called "block-mode" arrival, a full scan of them is possible to derive exact statistics. While performing a SAX scan, a vocabulary and the path synopsis can be dynamically constructed and kept in main memory as well as depth information and text content characteristics can be captured. In addition, this information can be reused for the second parse. Although the full XML document(s) has to be parsed before it is actually

[9] Information retrieval is extremely mature in calculating fine-grained similarity because of ranking possibilities. Moreover, a certain degree of uncertainty is tolerated. As we do not need any ranking in the first place and typically avoid any uncertainty, we can restrict our content comparison to simple, coarse, and fast calculations.

[10] Such an evaluation is actually similar to the ranking of *information retrieval*, although we do not require and use ranking of documents elsewhere.

Table 5.3: Document analysis options compared

Property	Pre-Analysis	Sampling
scan range	full document (100 %)	head of document ($\ll$ 100 %)
precision	100 %	$\ll$ 100 %
runtime	(very) long	short (upper bound)
buffer effects	good	best
block mode	yes	yes
stream mode	(no)[11]	yes
validation (XSD)	yes	no
reusing structures	yes	(no)[12]

stored in the database, the overhead is (often) quite low. Due to buffer effects, the second parse is reading the document(s) faster. Furthermore, an optional validation may take place and in case of violations the actual storage process can be stopped. Needless to say that this is an exceptional but quite common case in the world of XML. Thus, the actual storage process (i.e., preparing disk pages, synchronously writing data to disk, etc.) is dominating the entire process. Moreover, such an analysis is typically done only once for each document, while it is (later) queried multiple times.

Sampling

While being expensive, a full document analysis always delivers accurate statistics and thereby an optimal parameter selection for the storage configuration. But for large document(s) or collection(s), sampling may be appropriate to conceivably reduce the analysis efforts.

Sampling only allows to approximate important auxiliary structures and statistics such as vocabulary, path synopsis, content size per node, fan-out (in upper levels), and document depth. Certainly, a good reason for sampling are so-called stream-mode documents, that is, the document enters the database as a stream of nodes without any information about its size or stream length.

When sampling on block-mode or stream-mode documents, only initial fragments may be exploited for reasonable estimations. Sampling inner parts would require to "jump" into the middle of the document thereby losing location awareness and context information for determining levels, paths, and other structure information. Despite auxiliary knowledge about document size and structure for block-mode documents available, sampling proceeds the same way in both cases.

Neither *sampling* nor *pre-analysis* is superior for all kinds of applications. In Table 5.3, the main issues are compared. Note, the *precision* of the analysis depends on the sampling buffer size and directly correlates to the (unknown) document size. However, typically sampling, as the name indicates, does not cover the full document and thus a lower precision is expected. On the other hand, *runtime* is definitely the strong side of sampling because it will never exceed a certain limit (i.e., an upper bound for overhead estimations is available). Comparing the *buffer effects* reveals an ambiguous situation. As long as the document(s) fit into the (free) buffer, the disadvantage of a full pre-analysis is fairly low. However, sampling may use the same buffer as the storage engine itself and nothing needs to be reloaded from (external) disks. Accordingly,

another property – *stream-mode* analysis – is clearly better supported by sampling. However, an optional *validation* cannot be advanced to the analysis step. Eventually, *reusing* data structures collected during the analysis is easier when doing pre-analysis, because the structures are complete and can be optimized (e.g., encoding or compression).

Statistical Inference

A document analysis may back up configuration decisions, because they can be tailored to the characteristics of a single document or collection. However, most of the possible options are already presented in Section 2.5, but so far have to be configured with default values or chosen by the user (or administrator). For a self-tuning approach, the following observations can be exploited:

- **Node labels**: Pre-analysis or sampling deliver statistics about fan-out and density characteristics of a document. This information can be used to backup *dist* parameter decisions, even for individual levels of the document. When taking similar documents' statistics (of the same collection) or the new document's statistics into account, it helps to leave tailored gaps that may limit the overhead of division overflows in case of document updates. Moreover, customized Huffman codes can be applied for different divisions of a DeweyID adjusted to the expected average number of siblings. These fine-tuning measures become necessary as soon as space consumption becomes a critical issue.

- **Encoding**: Similar to the node label encoding, a document's individual vocabulary or path synopsis (in case of elementless storage) encoding may exploit statistics gathered by pre-analysis or sampling. For instance, a vocabulary may use a flexible encoding for a variable range of values or a fixed-length encoding[13] which requires a frequency distribution of tag names. The total number of different names as well as the expected extent rate can be used to safely choose the best, i.e., most compact form of encoding.

  Path synopsis encoding, especially for PCRs, is similar to that of a vocabulary. Besides the decision between fixed-length and variable-length encoding, the order of codes for PCR numbers may exploit instance statistics of paths. Thereby, frequent PCRs get smaller code words compared to rare PCRs, e.g., the root node exists only once and may get a large PCR or no code word assigned. Special considerations are required when thinking about the physical node instances because the elementless document store only contains leaf nodes, where the size of intermediate (e.g., element node) PCR codes does not matter. But secondary indexes may contain these "large" PCR code words, which is typically unknown while storing new documents, except when storing into a collection that already contains indexes.

[11] By temporarily storing the stream, it is possible but considerably more expensive.

[12] Accepting additional work, the reuse of structures gained through sampling may be possible.

[13] An example – a vocabulary comprising 200 different tag names is encoded as follows: (1) fixed-length means an upper bound of possible entries (i.e., 1 byte - max 255 entries) (2) variable-length means to employ "overflow" encoding reducing the number of possible values (i.e., the first byte encodes up to 128 different entries and uses a bit to indicate that a second byte is needed.

- **Structural compression**: This basically means to decide whether to store in full storage mode or elementless mode. Having only tiny document(s), each fitting into a few pages may better make use of the full storage mode. Because then, document access only requires to process (e.g., scan) these (often consecutively stored) pages. Also flat documents, i.e., less depth and nearly all nodes are leaf nodes may better make use of the full storage mode. Because the main advantage of the elementless mode is to avoid the physical representation of pure structural information, which is very low in such flat documents. On the other hand, space savings through elementless storage can easily add up to a huge amount. As long as the number of distinct paths (i.e., size of the path synopsis) is below a reasonable threshold and the path synopsis can therefore be kept in main memory, the additional functionalities (i.e., index types, query processing support) advocate the usage of elementless storage in most cases. Moreover, statistics about distinct paths, instance counters, average content sizes, depth, and leaf information are exploited.

- **Content compression**: Compressing text values does not always pays off, because the size reduction is too small or the algorithmic overhead harms the actual processing. Therefore, analyzing or sampling the document content is not necessarily limited to the text value length. It can also estimate the compression gains. For instance, character-wise encodings such as Huffman may simply account the frequency for (some or all) different characters. Together with the average content length, the compression gains can be estimated. Even for wordbook-based compressors an estimation of compression gains is possible. Therefore, typically a subset of (long) content values is compressed on-the-fly. Whether to analyze the full document's content or only a subset is a performance-critical issue, which is independent from the possibilities of statistical inference examined here. Often, long content values stemming from a certain domain (e.g., language, numbers, research area) heavily benefit from wordbook-based compressors.

- **Page size**: The concept of pages requires that records are aligned to the page size, i.e., records have to fit completely into a page or they are divided into multiple pages and are referenced. Moreover, the page concept leads to cutting losses as soon as the remaining free space is not enough for a new (or enlarged) record in that page. Regarding total space consumption of a document, it is important to keep the free space below a certain limit. Either by controlling the *max-value-size* of records or by exploiting statistics about frequent node value lengths are viable options. Thus, document-centric XML may require larger page sizes than data-centric XML.

  Together with structure compression, it may be beneficial to choose the page size according to the (estimated and/or expected) total document size.[14]

[14] We do not consider the concept of subpages in this work, which may solve some of the problems introduced by tiny documents or additional (tiny) data structures such as path synopsis or vocabularies.

5.8.5 Workload-Dependency

Classification of documents, as shown in Section 5.8.3, is used to distinguish document types and for tailoring the storage configuration accordingly. However, when more usage information is available, such as typical access patterns or access frequencies, value ranges, data volumes, and query types, configuration refinements are possible and may even be necessary.

There are other metrics to differentiate potential usages. For instance, the share of read and write operations, whereby secondary structures (e.g., indexes) need to be maintained as well.

First of all, we classify document operations in the following three categories:

- **document-based** – storing and reconstructing complete documents (SAX API), obviously favoring document-centric XML data.
- **index-based** – point and range queries often benefit from indexes (query optimization), however, the penalty of index creation (which implies a document scan) has to be amortized by frequent search operations; for data-centric XML, indexes are essential.
- **fragment-based** – the most complex and varying operations refer to node/subtree lookups, modifications, and deletions; both, document-centric and data-centric documents may be accessed in this way.

Furthermore, large documents typically require selective access when XPath/XQuery predicates are evaluated or subtrees are inserted or modified. In contrast, usually small documents are units of processing, i.e., they are entirely fetched, processed in memory, and, when modified, completely restored to disk. Anticipated operations are specified by a mix of simple queries for searching and modifying data [NKS07]. In contrast, large XML documents vastly benefit from additional indexes when processed as DB objects.

Our second distinguishing metric is the share of read/write operations. Even in a transactional environment, *read-only* workloads may not only benefit from adjusted lock protocols but also from a tailored storage layout. For instance, the size of disk pages is changeable, which affects the transfer unit from disk to buffer. Redundant data structures such as element indexes or space-consuming content indexes are supportive because no contention and updates are expected. The storage container selection favors a read-optimized solution such as flash devices or certain RAID types. In contrast, a write-dominated workload (e.g., log files, backup side) is more sensitive to updates on secondary data structures or the write performance of buffer and the underlying physical device. Anyway, real scenarios contain mixed workloads of read and write operations. This causes the real challenge for workload-dependent storage configurations, because the oppositional optimizations for read and write loads have to be accommodated for each document or collection.

5.8.6 Autonomous Collection Building

Document collections share data structures such as path synopses and may further increase the filling degree for data pages (compare Section 5.7). For self-tuning storage management,

the collection building can be done autonomously, too. Using the document similarity analysis from Section 5.8.3, the overhead that accounts the costs for joining a new document with an existing collection is computable. Depending on document and collection size (e.g., number of documents and storage occupation), it is fairly easy to evaluate the merge. In case of minor adjustments due to similar structure, wordbook, and (single document) size, a join is recommended. However, a single document may be regarded as a collection and, therefore, a merge of two documents can create a new collection in the database. Note, this decision is independent from indexing and query workload, which may disapprove the join. Thus, the join decision driven by storage space savings and structure optimization has to be supported by something like a domain similarity factor. That means, anticipated usage (see Section 5.8.5) is equally important. Fortunately, in real world scenarios, similar documents often belong to the same domain and, therefore, reveal the same characteristics in storing and querying.

Besides autonomously joining documents to collections, another requirement for a flexible self-tuning storage system is to separate documents from a collection when necessary. Because XQuery handles documents in a collection anonymously (i.e., they have no name, ID, or order), the anonymization has to be hidden somehow.

5.8.7 Data Placement

The issue of data placement contains two major aspects. On the one hand, the increase of physical storage alternatives offered different IO performances for different-sized media and prices. On the other hand, data values can be distributed and duplicated to parallelize IO.

In [MD97], disk impact was studied to analyze the benefit of parallel or sequential data access using a simulator. Furthermore, the skew in query and join predicates was considered, just like relation size and index effects. Even IO scheduler alternatives have been studied to improve the processing of various workload compositions. Here, the effects of data clustering and disk physics (e.g., spin-up time, rotation speed) are important. More sophisticated scenarios consider different RAID configurations or SSD devices as in [CMB+09], where object-wise read/write statistics were used to move objects from disk to (the faster but smaller) SSD. In such a mixed disk environment, the knapsack problem (cf. Section 3.1.2) needs to be solved.

Even newer technologies such as MicroElectroMechanical Systems (MEMS) [YAA07] are used instead of disks to allow for column-wise and row-wise data retrieval. Therefore, a new page layout (FRM – flexible retrieval mode) was developed because the well-known PAX, DSM, and NSM did not sufficiently match all requirements. Besides the page layout, even data structures can be adjusted for data placement such as the adaptive B^+-tree in [LKO+00] focusing on data migration.

Although most of the approaches are for the relational data model, to some extend, they can be adjusted to the XML data model, because the separation of data and indexes or log files is fairly the same. When considering XML documents as entire DB objects, similar to a relation, the object-wise placement may work, too. However, access patterns are different and, therefore, viable options have to be investigated.

5.8.8 Shifting Load to the Client-side

In today's computing environments, the classical "client/server" principle is often mitigated because peer-to-peer computing, cloud computing, or GRID, and commodity-class server farms are ubiquitous. Even in centralized environments, often oversized and underutilized client machines are available awaiting response from an overloaded server.

In order to scale with (peak) workloads, the server-side is upgraded with more hardware – KIWI (kill it with iron). We propose an approach to balance the loads by exploiting the processing capabilities of the client engines, too. Certain isolated (i.e., only visible for the client) computation-intensive processing loads may be sourced out to the client. For instance, the analysis step (Section 5.8.4) or compression (Section 5.8.1) can be done at the client-side before data is actually transferred to the server.

However, client and server resource capabilities and load situations have to recommend such a processing shift. Furthermore, the problems of code shipping, trustworthiness, and transactional protection have to be addressed as well.

Especially the initial data loading may benefit from preprocessing (i.e., compression and validation) XML data at the client-side. Furthermore, document retrieval may exploit a similar technique. Due to the network bottleneck, a compressed stream of document information may reconstruct the document at the client-side.

5.9 Realization in XTC

In this section, we highlight the most important algorithms and data structures we implemented in XTC to realize and demonstrate storage-related self-tuning concepts.

5.9.1 Statistics

There are three kinds of document or index statistics available in XTC. Either global configuration flags or online parameter settings that control the actual statistics gathering while documents are stored, manipulated, or indexes materialized. Let us have a look at them:

1. Basic *B-tree* statistics cover the height, cardinality, size in bytes, number of pages, and pointers as well as the number of leaf pages. These statistics can individually be gathered for each document index as well as for all secondary indexes during materialization. Note, we do not guarantee consistency for these kinds of statistics, i.e., either an explicit statistics run is required to update them or document updates causing changes of the underlying B-tree structure may not be reflected.

2. *Structural* or so-called *Path Synopsis* statistics are only available in elementless storage mode, because they are developed as an extension to the path synopsis. They cover statistics about the number of path class instances, the average content length, and a modification count.

3. *Fine-granular structural* statistics can optionally be created to indicate DOM relationships for elements and attributes. The fundamental concept is based on the EXsum approach [AMFH08] for XML statistics.

Because B-tree statistics can be easily gathered by traversing an index and, for instance, counting the various page types, we do not discuss them in more detail. However, the path synopsis statistics are more important for self-tuning, especially, when we discuss the index tuning in Chapter 7. Therefore, we will have a closer look on how to create and maintain them. Because the gain in XTC of using EXsum statistics is similar to the gain of using the *cheaper* path synopsis statistics, we confine ourselves to a size and performance comparison during the evaluation in Section 5.10.9.

Path Synopsis Extension

As already mentioned, the elementless storage mapping can easily be extended with some basic and cheap statistic values. Listing 5.1 shows the local variables a path synopsis node carries, where lines 17–19 represent the (added) statistic counters XTC is aware of. Compared to the remaining metadata, shown in lines 3–10 and 13–14, it has a rather small footprint.

Listing 5.1: Path Synopsis node data structure

```
1 class PathSynopsisNode implements PsNode {
2   /* consistent metadata */
3     final byte nodeType;
4     final int pcr;
5     final int vocId;                // vocabulary ID
6     final String name;
7     final PathSynopsisNode parent;  // parent pointer
8     final PathSynopsis ps;          // path synopsis pointer
9     PathSynopsisNode[] children;    // children pointer
10    boolean stored, visible = false;  // transactional properties
11
12  /* lazy evaluated metadata, for path synopsis processing */
13    final int level;
14    Path<String> vocIdPath;
15
16  /* statistics */
17    int count = 0;                  // instance counter
18    int updateCount = 0;            // IUD counter
19    double avgCntLength = 0;        // average content length
20 }
```

Statistics Gathering

Path synopsis statistics are gathered while a document is initially stored or via an explicit statistics run. Therefore, XTC provides a listener pattern for scan-oriented document processing. Its interface is close to the SAX specification (e.g., startElement(), endElement(), text(), attribute(), etc.). For B-tree statistics and path synopsis statistics, corresponding listeners are implemented and attached to the initial document store method. Each time an XML entity is parsed, the statistic callback is executed. Besides cheap counter maintenance, which is always incremented, the average content length (i.e., *avgCntLength*) can be computed in two ways:

(1) for each callback re-compute the average content length based on its old value and the new (increased by one) count value or (2) sum up the content length values and, in case of an endDocument() or endSubtree() call, divide it by the current count value. The overhead caused by path synopsis statistic gathering will be analyzed in the evaluation of Section 5.10.9.

Statistics Maintenance

Although we do not enforce path synopsis statistics to be fully consistent, we can enable automatic maintenance for them to keep them up to date. That is, each committed document modification is reflected by the count, *updateCount* and, if necessary, the *avgCntLength* values. Note, intermediate states, i.e., concurrent transactions modifying and reading the same statistical information, may not be isolated according to their actual transactional contexts – this type of weak isolation is only valid for statistic values.

The modification accounting is fairly similar to the initial statistics gathering. The same listener can be used, but its mode options are not limited to inserts anymore, because update and deletes are allowed, too. For simplicity, XTC implements document updates based on a delete followed by an insert, which reduces the complexity of path synopsis statistic maintenance. For instance, Listing 5.2 shows the "SAX-styled" attribute callback method, which switches between *INSERT* and *DELETE* mode to update statistical values on an individual path synopsis node identified by its PCR.

Listing 5.2: Path Synopsis statistic update sample (PathSynopsisStatisticsListener)

```
1 void attribute(TX transaction, ElNode node) throws DocumentException {
2     int pcr = node.getPCR();
3     String value = node.getValue(transaction);
4     if (mode == ListenMode.INSERT)
5         pathSynopsis.increaseCounter(transaction, pcr, 1, value.length());
6     else if (mode == ListenMode.DELETE) {
7         pathSynopsis.increaseCounter(transaction, pcr, -1, value.length());
8         pathSynopsis.increaseUpdateCounter(transaction, pcr, 1);
9     }
10 }
```

5.9.2 Statistics Gathering by Sampling

Because sampling the statistics is a kind of subset of statistics gathering presented in Section 5.9.1, we only show their differences. Again, we use a document listener following the SAX API by extending the *DefaultHandler*. Note, sampling analyzes the head of an XML document (or stream) to extrapolate its statistics based on its raw document size. That means, the file or stream size should be known in advance, otherwise, certain statistics are hard to estimate. The sample size is specified by a predefined amount of bytes (absolute or in percentage of the document size) or by the number of XML entities that need at least to be processed. As soon as one of these conditions or the *endDocument()* callback is reached, sampling stops.

Sampling can be done in three different ways:

1. **Server-side** sampling before the document is actually stored. This process is inherited from the pre-analysis step of XTC's document storage process. Therefore, the document

is fully available at the server side, i.e., in a temporary storage. While the actual storing takes place, the overhead of statistics gathering can be avoided.

2. **Client-side** sampling can be used to source out the entire sampling efforts and statistics gathering. The client's document stream may be extended by the sampling results to make them available in the server. Note, such a sampling step is exclusive, which means no transactional properties have to be preserved.

3. **Server-side buffered** sampling avoids the re-read of the document "head". Therefore, the desired sampling size has to ensure that the sampling buffer is large enough to keep the document data in the buffer. After sampling finishes, the SAX parser is reset to run without statistics gathering, i.e., first the buffered data is consumed before the input is transparently switched to the remaining (external) document data.

During the evaluation in Section 5.10.6, overhead and accuracy results of sampling are examined.

5.9.3 Compression

To support text (i.e., content values) compression in XTC, we implemented various techniques that achieve a content-wise encoding. That means, each content value (i.e., XML node value) is encoded and decoded separately. We developed a set of character-based compression modes using Huffman codes and a wordbook-based compression mode, which is similar to common database compression approaches.

The interface each text compressor has to implement is partially shown in Listing 5.3.

Listing 5.3: Text compressor interface

```
1 interface TextCompressor {
2     PageID getCompressionID();        // for persistent storage
3     void learnText(String value);     // learning phase
4     void createCompression();        // calculates mapping
5     String decode(byte[] encodedValue);
6     byte[] encode(String value);
7     void setCharset(String newcharset);  // character encoding support
8     String getCharset();
9   /* iterator methods for materialization  */
10    boolean next();
11    byte[] getKey();
12    byte[] getValue();
13 }
```

Our integration of the compression steps, while (new) XML documents or collections are stored, is illustrated in Figure 5.15. The flow chart shows the effects of certain decisions that can be made regarding text compression. For instance, an existing compressor (called domain compressor) can be chosen or a new one is created by *learning* the content values (*learnText(String value)* method). Buffer support by exploiting the *sampling buffer* idea avoids multiple *SAX parser* runs of the same input document(s). Subsequently, the compression mapping is created and stored via an iterator interface (*next(), getKey(), getValue()*) into a B-tree index.

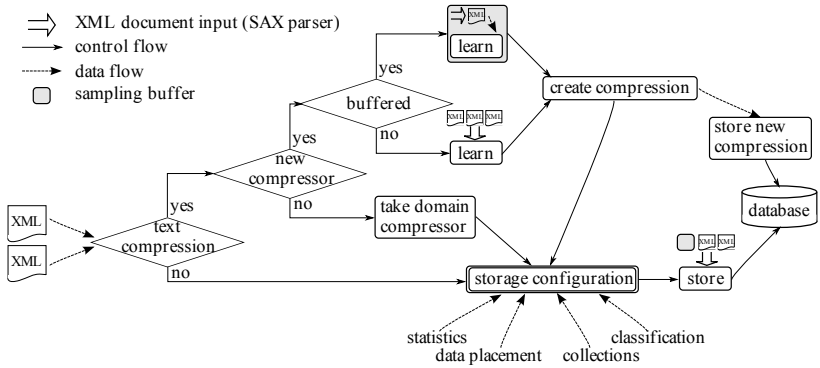


Figure 5.15: Document storage process including compression choices

The double-framed box – *storage configuration* – represents all configuration decisions that can be made during document storage in XTC. Later, we are going to explore some of them in more detail. But before, we present XTC's compression alternatives.

Character-based Compression Modes

Depending on the character distribution and frequencies, which may lead to varying compression gains, XTC employs the following alternatives based on Huffman encoding.

- *Fixed Huffman* (M1): During a pre-analysis run, the character frequencies were collected on a document basis, for which a Huffman tree is then constructed. To adjust the code for later document modifications, encoding was provided for all 256 possible characters.
- *Flexible Choice* (M2): Depending on the characteristics of a content node, it is either encoded by a document-wide *Fixed Huffman* or a tailor-made node-specific Huffman. Therefore, each content value is analyzed and, if favorable, a tailored encoding is derived solely on the frequency distribution of existing characters. If chosen, the tailored Huffman tree (typically < 40 nodes) is stored together with the encoded node's content.
- *Selective Encoding* (M3): This method optimizes the runtime of M2 by calculating and applying tailored Huffmans only to longer text/attribute values; smaller text values are encoded by the *Fixed Huffman* of the document.
- *Domain Encoding* (M4): Especially applicable to small documents, an overall encoding constructed from a domain-related character distribution base is used to reduce space requirements and to speed up compression time.

Let us have a look at some internals for Huffman encoding in XTC. Each entry in the Huffman tree (or table) has the lightweight structure shown in Listing 5.4, including its frequency, its value, and up to two child pointers.

Listing 5.4: Huffman Node

```
1 class HuffmanNode {
2     HuffmanNode one, zero;  // symbolic for left/right
3     double frequency;
4     int value;
5     HuffmanNode(int initialValue) {
6         value = initialValue;
7     }
8     HuffmanNode(HuffmanNode child1, HuffmanNode child2) {
9         one = child1;
10        zero = child2;
11        frequency = child1.frequency + child2.frequency;
12    }
13 ...
14 }
```

Depending on the encoding scheme (M1 – M4), up to one descriptor byte is reserved per entry, but often only one or two bits. Note, the encoding cannot be changed after a document is stored, i.e., document (or collection) updates must use the same encoding scheme. The only way to change an encoding scheme is to re-store the entire document or collection.

Keep in mind that character-wise compression of content seems to be favorable for data-oriented XML, but not for document-oriented XML, for which the second option – wordbook-based compression – was added to XTC.

Wordbook-based Compression Mode

As a common interface for all compressors is used in XTC, any compressor can be substituted with anyone of the others. Thus, the API for wordbook compression is equal to the character-based one presented before.

Hence, we only present some high-level parameters, which are (up to now) cannot be self-tuned. Each wordbook meets the following limits:

- MAX_BOOKSIZE: Limits the maximum size in bytes a wordbook can use on disk, the default value in XTC is 10 MB.
- BIT_LENGTH: This value defines the maximum number of bits a single word in the wordbook is encoded with. The default value of 19 bits leads to 524,288 words being possible.
- MAX_ENTRIES: Depending on the bit length, the maximum number of wordbook entries is calculated as follows:

$$\text{MAX\_ENTRIES} = (int)Math.pow(2, \text{WordBook.BIT\_LENGTH})$$

- MIN_OCCURRENCE: So that a word occurs in the wordbook, it has at least to appear MIN_OCCURRENCE times in the learning phase. XTC's default value is set to two.
- MIN_WORD_LENGTH: Each word's original string representation has to reach this minimum length in byte to be considered for the wordbook. It prevents codes that are longer than the actual (tiny) word.

Storage space consumption is further reduced by frequency-dependent code words. After a wordbook was created, new words can be added (or rejected automatically), which, however, implies a slightly different adherence to MIN_OCCURRENCE, which can be violated now.

Main data structures to create and use a wordbook are fairly simple, as shown next:

```
HashMap<String, Integer>  words
HashMap<String, Integer>  code
ArrayList<String>         reverseCode
```

In the *words* hash map, frequency statistics are collected during the initial creation; it can be dropped as soon as the wordbook is materialized. The *code* hash map ensures quick code lookup for encoding, while the *reverseCode* array list is used for fast decoding using the code word itself as array position number. Storing wordbooks is done by implementing the same iterator interface as for the character-based compressors. Moreover, wordbooks can also easily be shared for multiple documents or collection.

Eventually, the compression support in XTC allows for several alternatives, where only some of the parameters are chosen automatically. Additionally, we implemented a faster and customized bit set class to speed up our space-efficient encoding of text values and decoding of byte arrays, respectively.

5.9.4 Structural Classification of Documents

Document statistics, introduced in Section 5.8.2, are used to classify (new) documents and their similarity to existing documents. Note, we do not explicitly refer to collections because we do not distinguish between a single document and a collection throughout the following, except when especially mentioned. Two important metrics are used to analyze similarity: (1) a wordbook comparison (weak similarity) and (2) structural comparison (strong similarity).

Wordbook Analysis

Basically two wordbooks are compared, namely the existing wordbook W_e and the new one W_n. Each entry $w \in W_n$ is searched in W_e. A naive approach based on the unordered wordbooks in XTC[15], produces $O(|W_e| \cdot |W_n|)$ costs. However, active wordbooks, i.e., cached in main memory, provide hash-based access causing $O(1)$ lookup costs for each $w \in W_n$. In order to ensure that most entries of both wordbooks are conforming and, thereby, synergy effects can be exploited, the following two conditions need to be fulfilled:

1. **Overlap**: W_n has to overlap for $k\%$ of its entries with W_e, i.e., at least $\lceil |W_n| \cdot k \rceil$ words of W_n already exist in W_e.
2. **Growth**: The wordbook W_e is limited to grow by factor j.

Having chosen meaningful values for j and k, they can be used to specify upper boundaries to speed up wordbook comparisons. To avoid unnecessary comparisons at all, the new wordbook is only analyzed if:

$$|W_n| \leq (j \cdot |W_e| + |W_e|)$$

[15] Wordbooks in XTC are unordered because their position number is used as unique identifier, which is not allowed to be changed in case of wordbook insertions.

Table 5.4: Comparing sample wordbook vectors for $j = 0.5$ and $k = 0.7$

	V_n	V_{e1}	V_{e2}
A	0	5	4
B	6	1	3
C	5	4	4
Σ	11	10	11
new words		6	4
condition 1 (overlap)		no, $6 \not\leq \lfloor(1-0.7)\cdot 11\rfloor = 3$	no, $6 \not\leq \lfloor 0.5\cdot 10\rfloor = 5$
condition 2 (growth)		no, $4 \not\leq \lfloor(1-0.7)\cdot 11\rfloor = 3$	yes, $4 \leq \lfloor 0.5\cdot 11\rfloor = 5$

The growth factor j is used to define an upper boundary $C_g = \lfloor j \cdot |\mathrm{W}_e| \rfloor$. Similarly, k is used to specify the upper limit of not matching words by defining the boundary C_o as follows:

$$C_o = \lfloor (1-k) \cdot |\mathrm{W}_n| \rfloor \text{ , because } 0 \leq k \leq 1$$

Premature termination of comparisons is possible as soon as C_{min} non-matches occurred; $C_{min} = min\{C_g, C_o\}$. Another optimization can transform inactive wordbooks (i.e., not cached already in main memory), which are unordered, into an ordered sequence allowing binary search or map them into a hash map for cheaper lookups.

Because complete wordbook comparisons may become expensive, a kind of compressed pre-filtering was developed, too. Here, a wordbook is represented by an (optionally run-length encoded) vector indicating if a valid XML entity character is the first character of at least one entity. Having these vectors of stored documents, i.e., existing wordbooks, makes it easy to avoid unnecessary comparisons. Let us look at the example in Table 5.4. A new document wordbook vector V_n is compared with two existing vectors V_{e1} and V_{e2}. For simplicity, the vectors to be compared consist of three entity groups, namely A, B, and C. We can easily calculate the minimal number of new words, and if one of the conditions is definitely violated. That means, false positives are still possible and a detailed comparison of words instead of grouped vectors is necessary.

Structural Analysis

Structural classification of XML documents aims for exploiting collection storage. Because XTC relies on the path synopsis to evaluate similarities, only the elementless storage is supported. According to the wordbook analysis, path-synopsis-based comparisons have to follow the same conditions. For a new path synopsis P_n and an existing path synopsis P_e, we specify:

1. **Overlap**: P_n has to overlap for $k\%$ of its path classes with P_e, i.e., at least $\lceil |P_n| \cdot k \rceil$ path classes of P_n exist already in P_e.

2. **Growth**: The target path synopsis P_e is limited to grow by factor j.

Basically, we can use the tree edit distance measure to evaluate similarity. Note, we only require to simulate *inserts*, because we want to extend the existing path synopsis with "missing"

path classes, i.e., no deletion or renaming of existing path classes is allowed[16]. Therefore, all paths p are extracted that hold for:

$$p \in P_n \wedge p \notin P_e \text{ , for all } p \in P_n$$

Because modifications close to the root node may yield significantly more impact compared to leaf node modifications, [ZCZ03] introduced a cost metric taking node levels into account:

$$cost(v) = \frac{1}{(1+level(v))^k}$$

The relevance factor $k \geq 1$ is not explained in detail. However, we extended this vertical cost metric with a horizontal weight taking the bushiness of the insertion place into account. That led to the following cost formula, when inserting a new node v:

$$cost(v) = \frac{1}{max\{r,q\}+(1+level(v))} + \sum_{i=1}^{n} cost(c_i)$$

Our horizontal relevance factor is given by r and q that represent the number of nodes in the insertion level and parent level. Thereby, huge fanouts or deep levels do not attribute high costs to insertions. The second sum aggregates the costs for all the child nodes c of the inserted node v.

Using the cost formula, we can define a cost limit for an existing path synopsis P_e as follows:

$$cost_{limit}(P_e) = j \cdot \sum_{i=1}^{level_{max}} r_i \cdot \frac{1}{max\{r_i, r_{i-1}\}+(1+i)}$$

The formula simply reuses the cost formula introduced before to account for the "extend" of the existing path synopsis. With r, the number of nodes in a level is considered and j is used to weight the limit, according to the conditions.

Reducing the efforts of comparing path synopses is possible by checking root node names first. That means, path synopses having different root tags (i.e, XML element names) will not be compared at all. Although XTC allows to have multiple root nodes (and thereby multiple path synopses) in a collection, it is not exploited by the classification process. Another condition, which is easily checked, targets at the total number of new paths. Thus, path synopses violating this condition are rejected: $|Pn| > j \cdot |P_e| + |P_e|$.

The actual comparison performs several steps. First, the cost limit for P_e is calculated. Second, starting in level one, each node of P_n is looked up in P_e, i.e., same (tag) name and node type is required. Third, all nodes not found in P_e are put on a stack called *newNodes*, their costs are calculated and added up, and their descendants are ignored for further lookups because they are already accounted by the costs. The entire comparison is aborted as soon as the cost limit is exceeded. Otherwise, the overlap and growth conditions are checked afterwards.

[16]Only in a special maintenance mode, path synopses are cleaned from unused paths, i.e., paths having no instances.

5.9.5 Storage Decision Process – Document Processing

In XTC, many storage options that are available can be chosen by the user on demand, i.e., individually for each new document or collection. However, we extended XTC to support the user in making meaningful storage design decisions. Depending on certain conditions such as workload, statistics, and classification, XTC tries to adjust as many as possible storage parameters automatically. Because some storage decisions have implications for the remaining parameters, the decision process follows a specific procedure.

1. **Load balancing**: Depending on the computational capacity of the client and current server load, the server can recommend that a client prepares documents before transmitting them to the server. A client should observe this, but is not enforced to do so.

2. **Document analysis**: During the document analysis phase, XTC is analyzing the gathered statistics, i.e., data volume, structural complexity, and content characteristics. These statistics may result from a pre-scan (even at the client-side) or sampling. A so-called *document descriptor* is assembled that describes all storage-relevant aspects.

3. **Classification**: In this phase, the incoming document is classified. Here, XTC simply takes processing knowledge of existing documents into account. For instance, the share of content compared to the structural complexity (i.e., document-centric and data-centric) is an indicator for its future usage, like the type of XQueries, index requirements, or statistic maintenance. Note, the recommendations are solely based on the experience, gained by monitoring the usage of existing documents and collections, or based on user instructions such as default values. Especially the kind of expected workload, i.e., scan-based or node-based, is a classification metric.

4. **Collection building**: Each new document's descriptor can be used to measure its similarity to existing collections. Therefore, path synopsis matching is optionally followed by dictionary comparison to calculate the similarity. Either the first match (i.e., a predefined threshold is met) is taken for merging the new document or, after all existing documents and collections are compared and ranked, the best match is chosen.

5. **Data placement**: Before physical data structures are assigned, the target container is selected. Here, the favored cluster size of the workload class is an important aspect. We do not yet consider the load on certain containers (or devices), which may further improve the data placement decision. But we can automatically assign different containers for data and support structures such as path synopses, compression dictionaries, and secondary indexes. However, this is, at least at the moment, a system-wide configuration.

The document (stream) is then materialized into a physical representation, i.e., B^*-tree. Data structures like path synopses or compression dictionaries are stored afterwards. The new document is also registered in the metadata ("_master.xml") and optionally in the collection index.

We do not automatically match compression dictionaries or code tables, which is not necessary in case of collection storage, because collections do not support a mixture of content-compressed and uncompressed documents. Only a single compressor is allowed for the whole collection, which makes collection processing easier and faster.

5.10 Evaluation

So far, this chapter presented the fundamentals of XTC's XML storage together with many self-tuning approaches. Before we can evaluate their benefits and characteristics, we will present our benchmark setup containing several real-world datasets and some artificial datasets, as well as the hardware configuration.

Hardware Configurations

We have chosen three different hardware configurations for benchmarking, because this allows us to determine the impact of them in terms of CPU power, CPU cores, main memory size and speed, and disk configurations. All hosts ran Ubuntu Linux and Oracle Java 1.6.0_17.

- **Configuration 1**: Pentium IV, 3.2GHz HT, 1GB Ram, 2x 80GB SATA drive 7200rpm 3.5" (kernel: 2.6.24-16-server)
- **Configuration 2**: Xeon Quad Core 2.66GHz, 4GB Ram, 2x 500GB SATA drive 7200rpm 3.5" (kernel: 2.6.24-16-server)
- **Configuration 3**: Core 2 Duo 2.53GHz, 3GB Ram, 250GB SATA drive 5400rpm 2.5" (kernel: 2.6.27-14-generic)

5.10.1 Datasets

Benchmarking storage options requires a rich set of XML data providing manifold characteristics. Therefore, we evaluate our (self-)tuning techniques on a set of *real-world* datasets and a set of *artificial* datasets, which will be described in the following. In Table 5.5, an extended view of Table 5.2 is showing interesting document characteristics such as average and maximum depth, the vocabulary size (i.e., number of distinct XML tag and attribute names) as well as the average content length (i.e., text node's content and attribute node's content).

Real-world Datasets

Most of our real-word datasets are derived from the repository that can be found under [Mik]. Sometimes, we use a more recent or different version of a document than provided there.

The range of document characteristics is quite diverse, especially for document depths and vocabulary sizes.

Artificial Datasets

Artificial datasets, mostly generated by benchmark tools, help to investigate certain storage aspects. For instance, scalability issues can easily be assessed by generating various document sizes.

For document generation, we exploit two of the most popular XML benchmark projects, namely *XMark* [SWK+02] and *TPoX* [NKS07]. XMark provides artificial auction data in a

Table 5.5: Extended documents statistics

Document	Depth		Vocabulary size	Average content length
	max	avg		
Real-word datasets				
dblp	6	3.4	41	17.0
unirot	6	4.5	89	24.0
psd7003	8	5.68	70	17.0
treebank	37	8.44	251	33.4
nasa	9	6.08	70	20.9
Artificial datasets				
XMark (any size)	13	5.5	77	52.78
lineitem (TPC-H)	4	3.0	19	6.5
account (TPoX)	8	4.7	< 88	10.6
order (TPoX)	5	2.6	< 139	8.1 (attributes)
security (TPoX)	6	3.5	< 64	92.2

single, but scalable document ranging from the KB-level up to the GB-level. TPoX is a more recent proposal from IBM that can be used to evaluate transactional processing over XML (TPoX). Note, TPoX is referring to a real-world XML schema (FixML) and defines three types of documents having rather small sizes (≤ 26 KB). The dataset size can be scaled from several GB to one PB. Our third artificial dataset is an XML representation of a relation from the frequently used TPC-H benchmark.

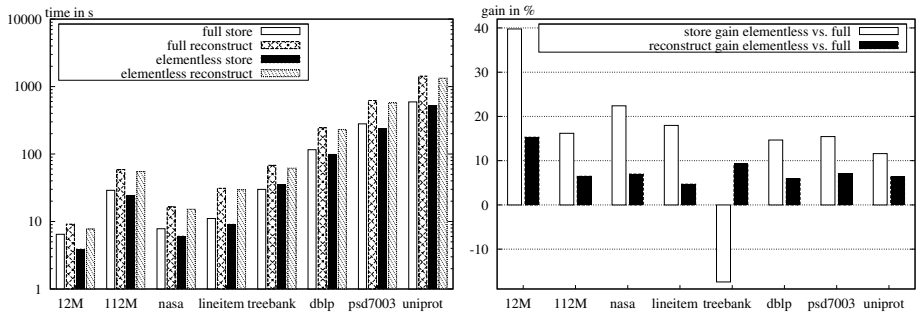
Our mix of artificial datasets provides a wide range of content lengths, vocabulary sizes, and node type occurrences. Moreover, the structural part of these documents is quite different compared to our real-word documents in terms of clustered subtree types (e.g., XMark documents) or fully regular structures (e.g., TPC-H table lineitem). The huge number of tiny documents in TPoX is another challenge for storage optimization, which is not covered by our real-word examples.

5.10.2 Access Performance

For a native XML database, it is important that the storage system is not only XML-aware but also delivers high document throughput and fast access to individual nodes. In the following series of benchmarks, we evaluate the benefits of our methods to optimize the storage and reconstruction of entire documents as well as the performance of fine-grained accesses like DOM operations.

Storage and Reconstruction

A first indicator for efficient processing is the overhead for an incoming document in its external format ("plain") to transform and save it and, in turn, to reconstruct it again. For our set of benchmark documents, we have measured the storage and reconstruction times and show the results for configuration 1 in Figure 5.16(a) (Figure B.1(a) in the appendix for configuration 2, respectively). Obviously, processing times are more or less linearly dependent on the document size. Because the size of our documents differ almost by two orders of magnitude, we



(a) Storage and reconstruction times

(b) Gain elementless vs. full

Figure 5.16: Storage and reconstruction figures comparing full and elementless storage mapping (hardware configuration 1)

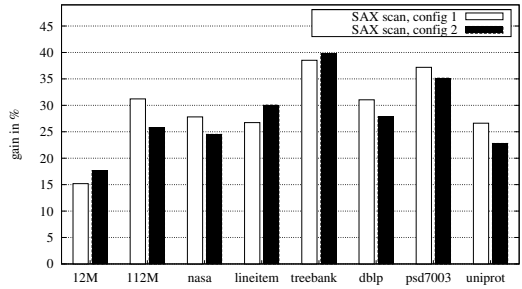


Figure 5.17: Scan-based access gain for elementless vs. full storage mapping

also refer to normalized gains defined as $gain = (\text{time}_{\text{full}} - \text{time}_{\text{elementless}} * 100\%)$. These gains are illustrated in Figure 5.16(b) for configuration 1 (Figure B.1(a) for configuration 2, respectively). Except for the treebank document, all storage and reconstruction processes run faster by $\sim$ 10%–20% using the *elementless* storage. The relative speed-up marginally depends on the hardware. For instance, configuration 1 seems to benefit more from the *elementless* storage when compared to configuration 2.

Scan-based Access Performance

In this test, we evaluated the scan-based access performance of our storage models. We compared the duration of a full SAX scan for *elementless* and *full* storage. In Figure 5.17, the speed-ups gained through *elementless* storage for our benchmark documents are shown. For

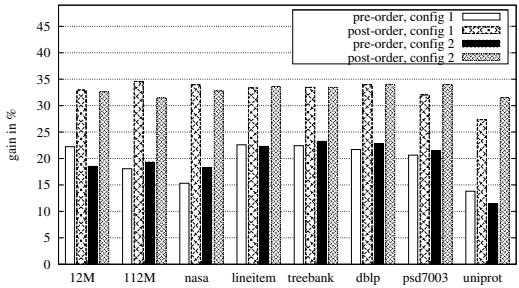


Figure 5.18: DOM traversal gain for elementless vs. full storage mapping

both configurations 1 and 2, the SAX performance on all documents is clearly improved, making use of the *elementless* storage mapping.

Another typical use case for XML processing concerns simple XPath evaluations, where the elementless mapping exploits the path synopsis by pre-filtering a set of valid PCRs, before comparing leaf nodes via PCR matching. This technique dramatically reduces processing time for large documents. In contrast, for small documents the secondary path synopsis structure needs to be loaded and, thus, disproportionally increases the number of algorithmic steps.

Navigation

Again, we sketch the relative performance gain of the *elementless* mapping w.r.t. the *full* mapping. Because navigation is context-dependent and the execution time of single navigational operations is not very expressive, we have designed a benchmark consisting of a pre-order traversal and a post-order traversal of our benchmark documents. These traversals start from the root node and apply the operations *first_child() / next_sibling()* respectively *last_child() / previous_sibling()* along the document structure. In case of *elementless*, the root node and the entire inner structure is computed on demand while the traversal is proceeding. The results in Figure 5.18 demonstrate that *elementless* processing is faster for all kinds of documents or configurations. In fact, in all cases we achieved improvements of 12%–34% due to less IO for the more compact *elementless* documents.

To decide which model outperforms the other one, we started with simple node lookups and XPath evaluations. Figure 5.19 reveals, independent of the buffer state (cold or hot), one-time lookups are usually faster using *full* storage mapping. Due to increased caching effects for growing repetitions, the performance of the *elementless* storage mapping is positively affected. That means, having a high share of locality in the access patterns, computational costs for on-demand reconstruction of *elementless* are negligible.

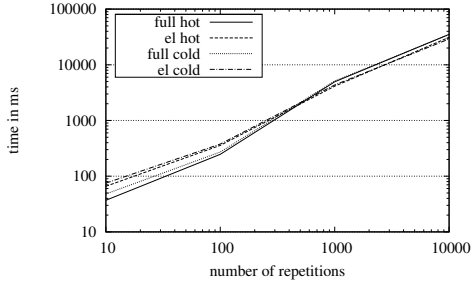


Figure 5.19: Scalability of node access

Table 5.6: Scalability of modifications

Ordering	Selectivity	Time (ms)	
		full	elementless
ordered	1%	22843	25431
ordered	0.1%	1638	1659
random	1%	30935	29600
random	0.1%	3533	3476

Update

Another group of experiments examined content node updates, which embody, besides read-only queries, typical operations on large documents. Here, we focus on ordered and randomized distributions of the modified nodes, as this seems to be realistic for single- and multi-user accesses. Table 5.6 shows that, on average, updating content nodes is dominated by transactional costs and, therefore, both major models perform similar. However, modifying nodes in document order seems to favor the use of the full storage mapping.

5.10.3 Space Consumption

The results of our space consumption benchmarks can be found in Figure 5.20. All results are compared to the *gross* format (100%), i.e., the plain XML document as received by the user. Because XTC encodes all nodes as variable-length records containing vocabulary information (vocID) and PCR numbers, some overhead (*encoding overhead*) is caused. This also includes overhead for byte alignment. The *naive* scheme represents the uncompressed storage of XML documents, i.e., node labels are not prefix compressed. The remaining two, *full* and *elementless*, are the storage mappings we normally use in XTC. In this test, we focus on the relative saving regarding the structure part only, because content compression is orthogonal and will be evaluated separately in this section. An interesting aspect is that the *naive* approach does not always achieve a storage space reduction (e.g., for XMark documents and psd7003). Here the saving from vocID usage is compensated by the need for node labels. In general, space saving of *naive* seems to be less than $\sim 35\%$ compared to *gross*. However, storage gain from *naive* to *full* and *naive* to *elementless* is substantial. The lion's share of this saving is due to prefix compression of the DeweyIDs (black-colored fractions) which reduces the storage space needed for node labels in all cases to less than 25% of its original size[17]. For all documents, the structural compression of *full* ranges from $\sim 40\%$ to $\sim 50\%$ and from $\sim 70\%$ to $\sim 80\%$

[17] Despite the "obvious length" of DeweyIDs, range-based or sequential labeling schemes would consume more storage, because they do not lend themselves to compression

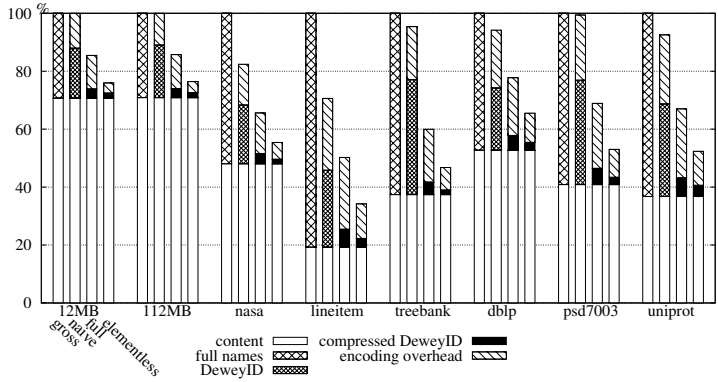


Figure 5.20: Storage space analysis for storage mappings

for *elementless*, respectively. In terms of space consumption, the combination of *elementless* storage and prefix-compressed DeweyIDs delivers competitive savings.

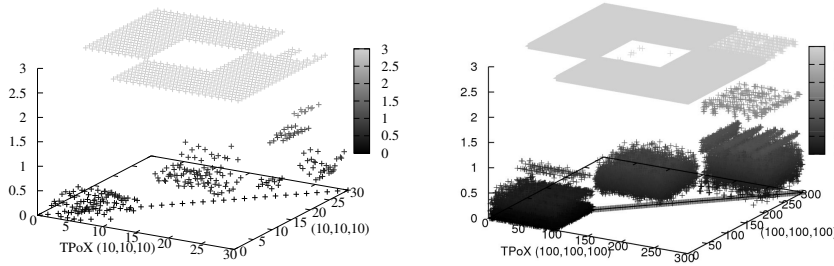
5.10.4 Structural Similarity

Based on the elementless storage mapping, structural similarity is evaluated with the help of path synopses and document wordbooks. According to our document classification scheme presented in Section 5.9.4, we choose the following overlap and growth parameters for our evaluations:

- **wordbook** overlap $k = 0.7$, growth $j = 0.5$
- **path synopsis** overlap $k = 0.7$, growth $j = 0.5$

Manipulation costs for attribute insertion amount to half of the costs accounted for element insertions.

Fortunately, the TPoX benchmark already provides three document classes with varying characteristics. For our evaluation, we draw a random selection of documents out of these classes. The results presented in Figure 5.21, are gained by comparing all documents with each other (i.e., like a cross product). In each case, the x-axis and y-axis represent the documents, grouped by their actual class membership. For instance, in Figure 5.21(a), documents between 1–10 are randomly chosen from the *custacc* class, while 11–20 are randomly chosen from the *order* class, and 21–30 from the *security* class, respectively. The z value is split into two areas. Everything below the value 2 is indicating the costs, i.e., path synopsis operations and wordbook extensions that are necessary to put the documents into the same class. The costs are normalized to range between 0, i.e., a perfect match or same documents, and 2. In contrast, non-matches are simply raised to a z value of 3, which makes them clearly distinguishable.



(a) TPoX documents (10x custacc/order/security)

(b) TPoX documents (100x custacc/order/security)

Figure 5.21: Similarity measurements and cost analysis for *moderate* parameter selection

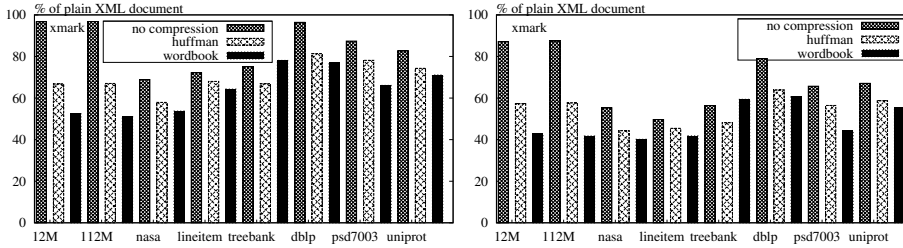
As we can see, the moderate parameter selection yields almost perfect results. Mostly document comparisons from different classes result in z values of 3, whereas the others cavort at the bottom of the cost scale. Increasing the number of randomly selected documents per class to 100 confirms these findings, as shown in Figure 5.21(b). Sometimes the comparison of *security* class documents seems to result in "wrong" conclusions. But because their number of attributes is varying strongly, which may account to path synopsis extensions by nearly 100%, this result is correct, too. In the Appendix B.2, more evaluations are shown to shed light on the impact of varying parameters and to justify our similarity-based classification framework.

5.10.5 Content Compression

Content compression is solely focusing on storage space savings, but its computational overhead for compression and decompression has to be justified. First, we analyze the space reduction gained by our proposed compression techniques. After this, we look at the processing overhead (i.e., additional CPU costs and runtime) caused by compression for our hardware configurations. This includes a costly pre-analysis of the documents before the actual storage process takes place. Therefore, the last benchmark assesses the performance for compression sharing, i.e., a collection or documents of the same domain share a common compression dictionary or code table. Note, sometimes, we provide supplementary results in the appendix.

Space Reduction

Figure 5.22 reveals the potential space reduction due to content compression. As expected, the combination of elementless storage mapping and wordbook compressors is nearly always the best performing configuration, see black bars in Figure 5.22(b). Space reductions of up to



(a) Compression gain with full storage mapping

(b) Compression gain with elementless storage mapping

Figure 5.22: Relative document size of Huffman and wordbook compressors compared to external document sizes (*plain*)

60% are achieved. Only the "exotic" treebank document is best compressed using a Huffman compressor.

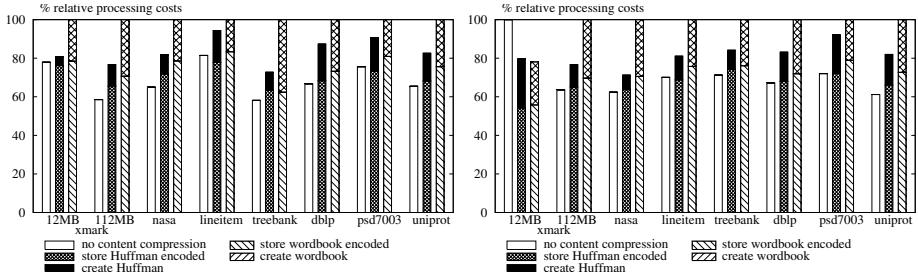
The results further indicate that content compression for document-centric XML yields more space reduction compared to variants that do not perform content compression. For instance, the XMark documents profit more from content compression than the data-centric lineitem document, independent of the chosen storage mode.

More results, especially for our variations of Huffman and wordbook compressors, can be found in the Appendix B.4.

Processing Overhead

The computational overhead of our compression techniques is measured using the following benchmarks. We compared the document storage times for the most effective Huffman and wordbook variants with the uncompressed version. In Figure 5.23, the results are shown for configuration 1 that comprise an evaluation for *full* storage (a) and one for *elementless* storage (b). Each diagram shows the relative time to store an uncompressed document (white bars) compared to the time to store a compressed version of the same document. Compression times are further distinguished between times to create the compression data structure (i.e., *create Huffman* and *create wordbook*) and times to actually store the compressed document (i.e., *store Huffman encoded* and *store wordbook encoded*). Note, the second time portion is equal to the time necessary to store a document using an existing compression dictionary. This means for our benchmark results that in most cases, the computational overhead for on-the-fly compression is negligible, because their storage time is only slightly higher and even sometimes better (e.g., lineitem or psd7003 in Figure 5.23(a) and 5.23(b)).

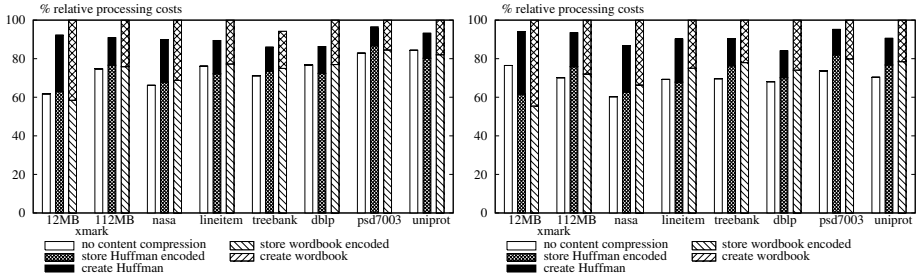
These findings are approved by the results we gained for configuration 2. Accordingly, results for *elementless*-based and *full*-based compression figures are shown in Figure 5.24. Eventually, content compression, that preserves the full fine-grained processing capabilities, is effective in terms of space consumption and time overhead. Because documents are typically



(a) Compression time with full storage mapping

(b) Compression time with elementless storage mapping

Figure 5.23: Relative compression time (configuration 1)



(a) Compression time with full storage mapping

(b) Compression time with elementless storage mapping

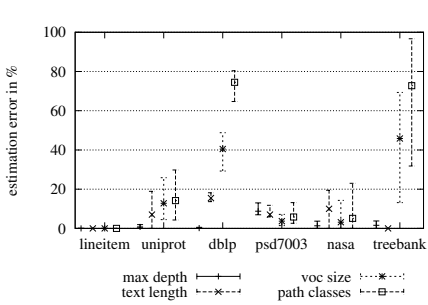
Figure 5.24: Relative compression time (configuration 2)

stored only once and processed repetitively, the initial overhead definitely pays off.

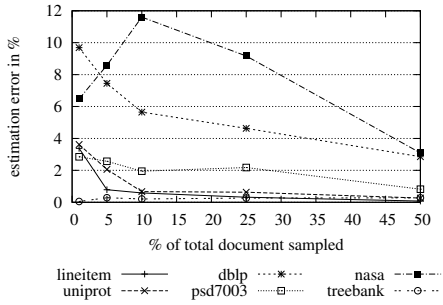
Although the wordbook-based compression method allows certain parameter variations, such as wordbook size and minimum occurrences, their computational overhead for initial creation is always higher compared to the "simple" Huffman encoding. But their space reduction is significantly better for common XML documents.

5.10.6 Sampling

Sampling experiments were run for a subset of our benchmark documents. We have determined – for storage-critical document parameters – the ranges of expected estimation errors. In Figure 5.25(a), the graphical symbols depict the average estimation error per document, where the max/min of the error range correspond to estimations computed when a sampling buffer was filled with 1 and 50 MB, respectively. These results highlight one of the most fundamental



(a) Ranges of sampling errors on XML documents



(b) Extrapolated sampling errors when doc size is known

Figure 5.25: Relative estimation error of sampling

problems in sampling small pieces of documents with heterogeneous or skewed structures. As indicated for dblp and treebank in Figure 5.25(a), parameters such as vocabulary size and number of path classes may cause the selection of an unfit representation model. When we start building the document with such "wrong guesses", we may get suboptimal structures or may be enforced to revise our design decision. In general, however, the parameters for max/avg depth, average text size, and fan-out are accurate and stable, even for tiny fractions of 1 MB samples. Hence, we can use stable parameters for decisions concerning DeweyID encoding and page size tuning. Of course, a Huffman code table can be derived by sampling, too. Because character distribution and their frequencies typically are domain dependent, nearly optimal encodings can be expected.

When sampling on block-mode or stream-mode documents, outliers for specific parameter values or a skewed document structure may lead for both arrival modes to wrong decisions, only reloading or dynamic reconfiguration may guarantee optimal configurations. Having complete documents and precise analysis available, the mismatch is most likely even lower. Stream-mode documents necessarily enforce our adaptive storage management to configure storage structures with less than perfect parameter knowledge, as characterized in Figure 5.25(a). Because file size information is available for block-mode documents, extrapolation of some parameters using the size of the entire document is applicable. To show the precision of a sampling step instead of a full scan for size information (number of attribute/element/text nodes), Figure 5.25(b) exhibits the relative estimation errors for various sample sizes. Surprisingly, our results reveal that, even with only a sample of 1%, an error of not more than $\sim 10\%$ may be expected. Of course, larger sample sizes improve this error margin. Figure 5.25(b) also shows that there exist "simply structured" documents where sampling delivers perfect knowledge of size parameters even for very small samples. In summary, sampling often delivers accurate-enough parameters to plan the physical configuration of an XML document.

5.10.7 Usage-driven Storage Structures

Using the following benchmarks, we show the potential of customized storage configurations, especially the different kinds of workloads identified in Section 5.8.5 (i.e., document-based, index-based, and fragment-based).

Exploiting document characteristics, either gathered through sampling or full analysis, we show for our storage models how suitable parameters can be estimated, what storage and processing gain can be achieved as compared to the *standard* representation and how various operations scale on theses models.

As it is our aim to enable XTC to handle any kind of XML document arriving at the DBMS, the storage manager applies a basic set of storage parameters (including page size, DeweyID compression, content compression, and storage model), which normally cannot be changed afterwards. While future access behavior is not considered, all kinds of documents can be stored using the following set of *standard* parameters, enabling a maximum degree of flexibility:

- *16 KB page size:* allows to store large documents, because the addressable storage space is bounded to the maximum page number.
- *No DeweyID compression:* avoids overhead of compression encoding and offers direct DeweyID access (no need to compute predecessors).
- *No content compression:* avoids encoding maintenance; neither undue compression overhead is paid for small documents nor IO savings are gained for large documents.
- *Full-storage model:* does not require an additional path synopsis for structure virtualization, which may blow up memory consumption when dealing with too many path classes; no path encoding is required (PCRs).

The substantial complexity of all measures reducing storage consumption or query processing does not allow simple design decisions. An important optimization is the use of DeweyID schemes and encodings adjusted to the document characteristics, which was applied in all experiments.

Configurating Single Documents

To illustrate the various storage configurations, we assembled a set of predefined configurations. Note, we confine them to the most promising aspects and combinations, i.e., parameters and configuration combinations not presented here may also improve (some) of the conducted test cases, but their benefit is too small to be evaluated and to be presented in detail.

- *Standard Storage* Configuration (default)
- *Full Storage*, DeweyID compression, 4/8/16/32/64 KB pages, optional content compression, shared/distinct vocabulary encoding
- *Elementless Storage*, DeweyID compression, 4/8/16/32/64 KB pages, fixed or adjusted PCR encoding, optional content compression, shared or distinct vocabulary encoding

Workload Scenarios

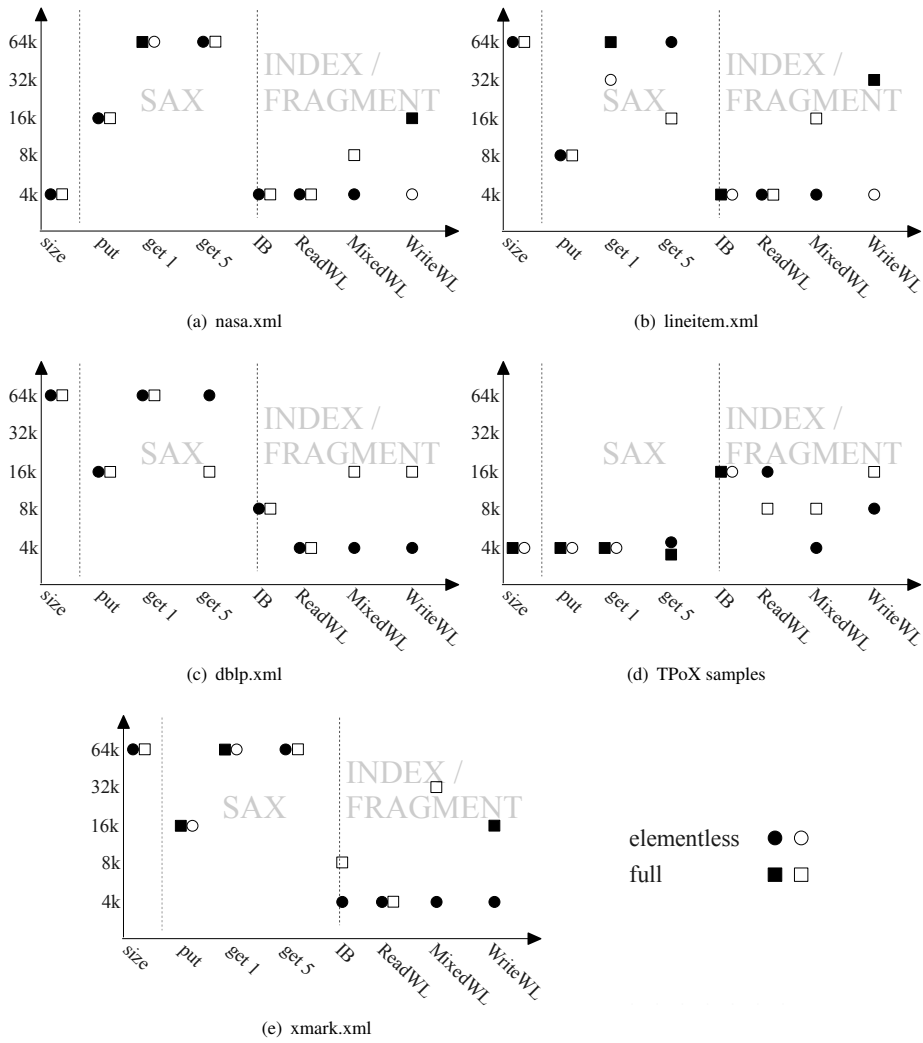
To represent different document usages, we defined the following specific workload scenarios:

- **Storage space consumption** – indirectly matters when processing takes place
  - ***size***: space consumption of documents (including path synopsis when stored in elementless mode)
- **SAX-based** – for document-based access; read/write of the entire XML document (SAX API)
  - ***put***: storing the XML document once
  - ***get 1***: reading the entire XML document once
  - ***get 5***: reading the entire XML document multiple times (5 times)
- **IB** – index building costs (only evaluated for a simple index that contains all element nodes)
- **Index-based or fragment-based**
  - ***ReadWorkload***: read-only access; a mixture of XPath expressions and/or DOM navigational steps
  - ***MixedWorkload***: a mix of read/write and navigational operations: XPath, simple XQuery, DOM-based units of work, updates, and deletions
  - ***WriteWorkload***: write-only workload; inserts, updates, and deletes on nodes and/or subtrees

Parameter Space Analysis

In the first part, we evaluate what kind of storage model in combination with varying page sizes leverages specific workload types for certain document types. In Figure 5.26, we assembled five representative diagrams for a selection of artificial and real-world documents. The vertical axis scales the page size used and the other axis identifies the workloads analyzed. The storage models compared (elementless and full storage) are illustrated by dots and rectangles, respectively. A filled symbol means that this storage mode is the best one for the given workload (e.g., Figure 5.26(a) shows that the fastest storing of the *nasa* document is done in elementless mode using 16KB pages), whereas an unfilled symbol means that this storage mode is inferior but, when enforced, would suit that page size at best (e.g., Figure 5.26(a) shows that storing the *nasa* document in full-storage mode is worse than in elementless mode, but would be best using 16KB pages).

We can further see that disk utilization (labeled *size*) directly depends on document size and structural complexity. Hence, simply-structured and uniform documents (e.g., *lineitem*, *dblp*, and *xmark*) do prefer larger page sizes, whereas complex-structured documents (e.g., *nasa*) or tiny documents (e.g., *TPoX*) better utilize small pages of 4 KB.



(a) nasa.xml

(b) lineitem.xml

(c) dblp.xml

(d) TPoX samples

(e) xmark.xml

Figure 5.26: Workload vs. storage model benefits for selected documents

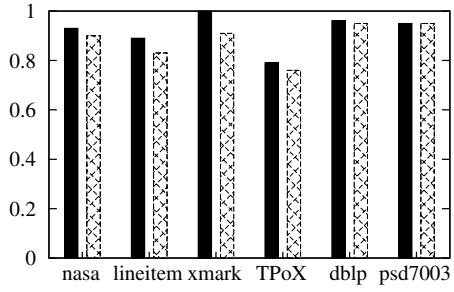
The middle part of each diagram depicts so-called SAX-based workloads for storing and reconstructing XML documents. Document storage (labeled *put*) performs best on medium-sized pages and in elementless mode for large documents, whereas tiny documents prefer the full storage and the smallest page size covering the entire document. For SAX-based reconstruction, small and medium-sized (e.g., *nasa*) documents and very simply-structured documents (e.g., *lineitem*) benefit from the full-storage mode. This benefit becomes a drawback, when a document needs to be reconstructed (labeled *get 5*) repeatedly. Here, all kinds of documents favor the elementless storage, even though the storage model is almost irrelevant for tiny documents. Index-building costs (labeled *IB*) are a mixture of a SAX scan on an already stored document and the actual index creation. The benchmark results show that simply-structured and small documents favor the full-storage mode, whereas all kinds of documents prefer small page sizes.

The right part shows node-based and subtree-based workload scenarios. The read-only workload (labeled *ReadWorkload*) reads tiny fractions of a document and, therefore, all kinds of documents favor small page sizes. Because every query of such a workload needs to repeatedly access the path synopsis, elementless document storage is preferable in all cases. This observation also holds for mixed workload scenarios applying read and write operations on XML documents. But for write-only scenarios (labeled *WriteWorkload*), sometimes (e.g., *nasa* and *lineitem*) the full-storage model is preferable, because the additional path synopsis structure required for the elementless mode needs to be maintained when subtree/node insertions refer to new path classes.

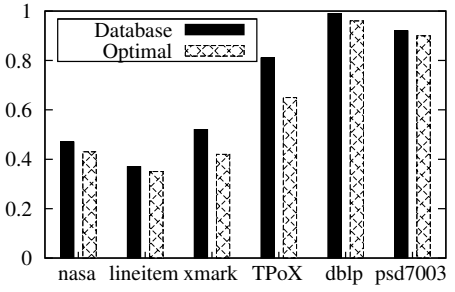
Automatic Storage Management

This benchmark compares the *standard* storage performance with an *autonomic* configuration chosen by the storage manager – based on our heuristics and experimentally evaluated rules – and a theoretically optimal setup. In Figure 5.27, we have assembled three benchmark results reflecting three different workload patterns. We further weighted the workload as shown in Table 5.27(d) to distinguish between typical usage patterns. The workload scenarios, we used before (sketched in Figure 5.26), were combined and weighted. In each diagram, the *standard* configuration is used as reference (y-axis gain is exactly 1). The solid bar shows the performance gains of the *autonomously* chosen setup.

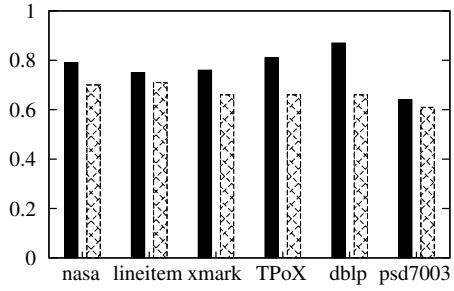
The second bar is the theoretical limit if each workload in a pattern is executed using an optimal configuration on its own. Unfortunately, such a proceeding is impractical, because most storage parameters cannot be changed easily during runtime. In Figure 5.27(a), we can see that a workload dominated by *SAX-based* operations does not benefit much from an optimized storage configuration. But especially for the *Read*-operation-dominated workload, depicted in Figure 5.27(b), an *autonomously* chosen setup is competitive and nearly as good as the theoretical optimum. Even the *Write*-dominated workload, depicted in Figure 5.27(c), can be improved by a respectable margin.



(a) SAX workload optimized



(b) Read workload optimized



(c) Write workload optimized

Workload	SAX	Read	Write
put	3	1	1
get 1	3	1	1
get 5	3	0	0
read	1	7	2
mixed	0	1	3
write	0	0	0

(d) Workload weights

Figure 5.27: Performance gains for autonomic and theoretically optimal configurations compared to *standard*

5.10.8 Load Balancing

Especially when new XML data is stored, the transfer of the (raw and verbose XML) documents from the client to the server and the server-side encoding of physical records causes long running transactions and server-side load. Using the following benchmark, we evaluate the potential of our load balancing mechanisms (cf. Section 5.8.8) by reducing transaction times for document storage through client-side preprocessing.

For this benchmark, we used the hardware configuration 2 for the server and the client, which are connected via high-speed LAN (1 Gb), as well as XMark documents of varying sizes between 12 MB and 500 MB. In Figure 5.28, the results show the storage times necessary to *put* a document on the server and to *load* a document to the server. Loading figures are separated into client-side loading time, i.e., analysis, compression, and transfer, and server-side loading time, i.e., persisting the prepared document stream. Results on the left-hand side are gained while advanced buffer mechanisms were disabled. In contrast, the results on the right-hand side are gained while read ahead and sequential write were enabled.

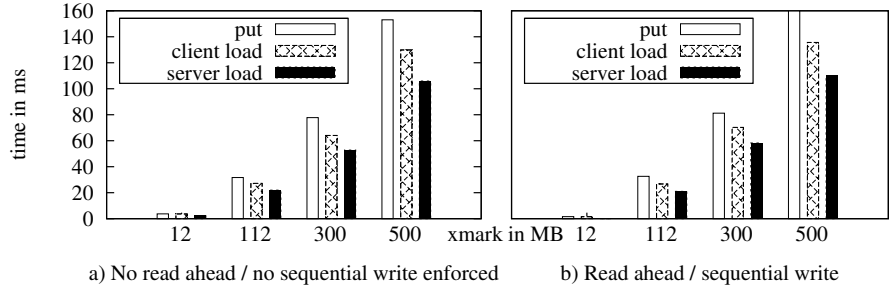


Figure 5.28: Storage performance for client-side loading vs. server-side encoding (powerful client)

In both scenarios, the reduced processing time for *loading*, compared to *putting* is clearly visible. But not only is the overall loading process faster, even more, the processing time at the server-side is reduced once again due to the client-side document preparation. This optimization is only possible, when the client is powerful enough or in case of weak network bandwidth or speed. The impact of our advanced buffer mechanisms is negligible, because document storage does not benefit from read ahead and, usually, the pages are already allocated in sequential order making sequential writes happen anyway.

Client-side preprocessing is heavily dependent on the client's power, i.e., CPU and IO capabilities. Therefore, we made a second run using configuration 1 as client (i.e., weak) and configuration 2 as server (powerful), which may reflect more commonly client/server scenarios. In Figure 5.29, the results indicate that the weak client needs to do more compared to the powerful client. For instance, the times for *load* and *put* are fairly the same at the client-side. But in case of loading, the efforts at the server side are still less, which advocates the usage of client-side preprocessing. Note, increasing the XML data volume results in an increase of the gain our load mechanism provides.

5.10.9 Statistics

The document and index statistics, we introduced in Section 5.8.2 and Section 5.9.1, are an integral part for many (self-)tuning features and decisions.

Therefore, the typical overhead of collecting statistics is important. For this reason, we stored our artificial and real-world benchmark documents and measured the additional time for storing and processing. Table 5.7 contains storage consumption figures for a representative document collection, their path synopses size(s), and, for the sake of comparison, storage figures needed by the full-fledged XML statistics framework EXsum [AMFH08]. These figures confirm a fairly small footprint of the path synopsis extension ($< 1\%$ of the actual document(s)).

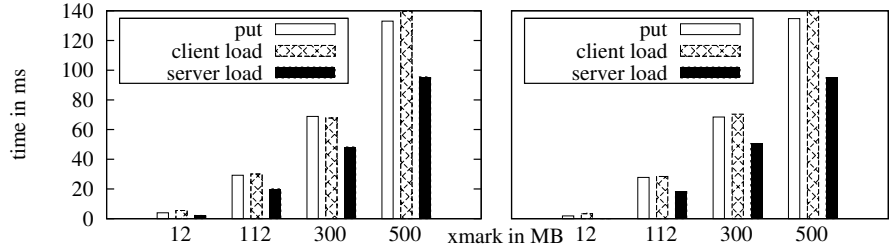


Figure 5.29: Storage performance for client-side loading vs. server-side encoding (weak client)

Table 5.7: Statistics space consumption figures for selected documents

Document(s)	Document / storage	Path synopsis	Ext. path synopsis	EXSum
XMark	12M / 9.5M	4K	11.2K	14.2K
	112M / 95M	4.1K	11.6K	14.4K
nasa	25M / 13M	0.8K	2.2K	10.5K
lineitem	32M / 13M	0.1K	0.4K	1.6K
treebank	90M / 46M	2.9M	7.3M	63K
dblp	330M / 233M	1.1K	3.3K	6.7K
TPoX collection (scale factor XXS)				
security	126M / 107M	0.7K	2K	11.4K
custacc	58M / 26M	0.7K	2.2K	14K
order	73M / 54M	1K	3.1K	4.6K

Figure 5.30 reveals the processing time overhead in percent. The first column indicates the amount of extra processing time consumed to gather the statistical values for the extended path synopsis. The second column adds the additional overhead for collecting the B-tree index statistics. Fortunately, in almost all cases, the maximum of both is below $\sim 5-6\%$ and, note, this overhead occurs only once for each document while mapping it to the DB storage representation. Thus, the overhead caused by the path synopsis extension is negligible compared to the potential future gain. In contrast, the overhead of EXsum [AMFH08] is much higher (column 3) and often would not pay off.

5.11 Conclusions

In this chapter, we presented concepts for tailoring and self-tuning the XML storage subsystem of a native XDBMS. Especially, optional compression techniques addressing structural redundancy and text content have demonstrated to be often beneficial in terms of space reduction and processing costs, i.e., IO operations. The many different XML APIs and varying work-

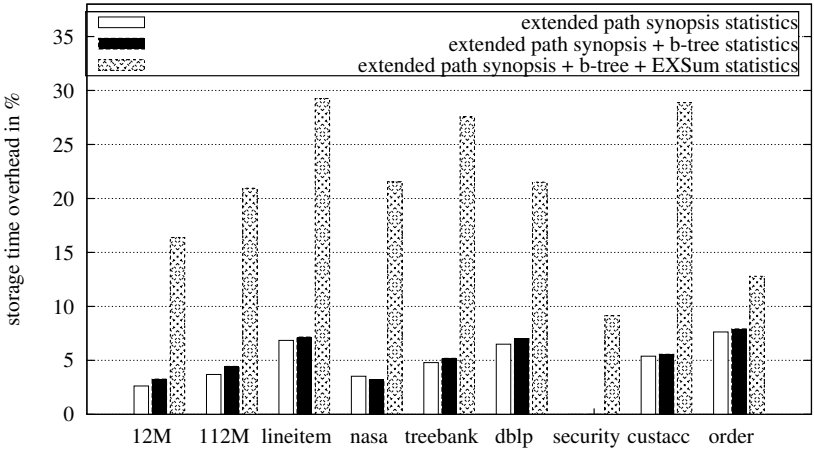


Figure 5.30: On-the-fly statistics maintenance: overhead analysis

loads underlined the importance of tailored XML storage configurations. Therefore, we also investigated different ways of XML analysis to get statistics causing low-overhead and to automatically decide storage options based on similarity measures and experience. Furthermore, client-side preprocessing and XML classifications helped to improve the performance of the entire storage subsystem.

Chapter 6

Index Options and Query Processing in XTC

Native XML databases require secondary access paths – so-called indexes – for the same reasons, relational systems do. Query processing typically exploits those indexes to avoid costly (document) scans. The larger the XML documents become, the longer it takes to scan them, especially when the query semantics dictate multiple scans of the same document while processing a single query[1]. Indexes not only reduce the amount of nodes that need to be processed or even temporarily stored for intermediate results but also the number of locks acquired in a concurrent user scenario. That is, fine-grained locking mechanisms (i.e., node-based or axis-based) can avoid document-wide locks and thereby increase the number of concurrent transactions operating on the same document. Furthermore, selective read access caused by query predicates (i.e., `where`-clauses, XPath predicates) may exploit indexes if the index' scope covers the queried document parts and thus less physical reads are necessary compared to a full document scan. Another benefit of secondary indexes is that they support uniqueness properties as well as a certain clustering or order (i.e., index key order) of its indexed elements. Hence, queries that include an *order by* specification may get the (often required) node order for free if an appropriate index can be used.

Besides all the benefits indexing provides, some severe negative aspects exist, too. Because indexing always duplicates data, the consistency requirement for database systems enforces their transactional maintenance during IUD operations. Those index maintenance tasks heavily produce random IO and may also slow down system performance. Moreover, having a rich set of indexes available, each of them needs to be checked in case of updates whether it is affected, which also produces computational overhead.

In this chapter, we present current index approaches tailored to the XML domain, before we present our flexible index options developed in the context of XTC. To demonstrate the usage of indexes, we give a brief introduction into the basics of XTC's query processor as well as the integration of index usage into it.

6.1 Related Work

Current DBMSs offer two major types of XML index support, (1) pure text-oriented indexes, so called *content indexes*, and (2) structure-oriented indexes, so-called *path indexes* [MS99]. A third group of hybrid indexes, which is only available for the XML world, combines structural and content predicates to address the indexed parts.

[1] Available main-memory sizes limit the size of documents that can be processed in main memory only based on a single scan, and thus scan-based processing easily requires multiple scans.

In recent years, research has proposed many different index structures that are especially tailored to the XML data model. There are simple element indexes [LM01, JAKC+02], structure-aware indexes, i.e., path indexes [MS99, WJW+05], dynamic indexes such as [CMS02], and hybrid indexes [CSF+01], but most of the proposals do not address the update problem and its costs and, moreover, do not show how they can be integrated into an XDBMS. In particular, the flexibility and variability of XML index structures is the strength for an optimal query support but, at the same time, it is the weakness concerning the *Index Selection Problem* (ISP) [PS83, Com78].

6.2 Indexing in XTC

Index support in XTC focuses on two critical issues, namely *query support* and *storage integration*.

A query optimizer typically relies on the availability of eligible indexes to avoid document scans. Furthermore, the complexity and flexibility of XML requires an adequate set of indexes to avoid heavy uncertainty, if no statistics are available. Based on the classification given in the literature, we propose to have at least four major index types as follows: (1) element index, (2) content index, (3) path index, and (4) CAS index, which means *C*ontent *A*nd *S*tructure.

Integrating index features into XTC follows certain design guidelines. Inter alia, indexing has to provide the following properties: *optional use*, *expressiveness*, *selectivity*, *updates*, *applicability*, and *result computation*; a detailed description can be found in [MHS09]. To support all these properties for all kinds of indexes, we have to establish indexing in XTC based on elementless storage. As we will see, the powerful PCR concept, DeweyIDs, and B-trees are the cornerstones for optimizing index options. Most of our index types provide various kinds of clustering and allow for efficient prefix-based key compression. In the following, we present our four index types.

6.2.1 Element Index

The logical structure of an element is comparable to a 1-index [MS99], where each distinct XML name tag is associated with a list of node labels carrying this tag. Figure 6.1 gives an example for an element index covering three different tag names (i.e., vocabulary entries) and thereby indexing multiple path classes automatically. Furthermore, the *elementless* storage allows for two different clusterings of the element index, namely, DeweyID-based clustering and PCR-based clustering. These clusterings influence access costs and are determined by the index covering.

- **DeweyID clustering** The index key is composed by concatenating the DeweyID, a separator, and the PCR. All index entries are stored in document order and false-positive filtering is needed if a subset of indexed path classes is requested.

- **PCR clustering** The index key is composed by concatenating the PCR, a separator, and the DeweyID. To build this index, the entries have to be grouped by their PCR and,

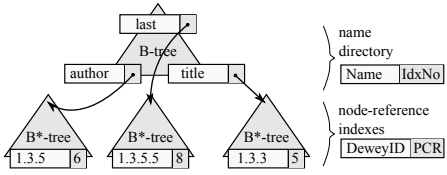


Figure 6.1: Sample element index

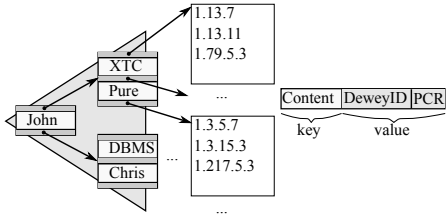


Figure 6.2: Sample content index

within each path class, they retain their document order. Index access may need to scan scattered areas of the index if non-consecutive-stored PCRs are requested.

Definition 1 *An element index $I_{E(V,clust)}$ preserves document order and covers a set of vocabulary entries V whereby $V \neq \emptyset$ and $V = \{v | v \in Vocabulary\}$ and $clust = \{DeweyID|PCR\}$. V is the set of index keys and the values are defined as follows:*

$$value(I_E) = \begin{cases} DeweyID + PCR & \text{if } clust \text{ is DeweyID (default)}, \\ PCR + DeweyID & \text{if } clust \text{ is PCR}. \end{cases}.$$

Element index access: Typically, named element streams are delivered by an element index. If path filtering is required, only the elementless storage provides such built-in information. But storing those path information within the element index leads to additional storage costs.

6.2.2 Content Index

Content indexes, as shown in Figure 6.2, typically index the whole content of an XML document, where they distinguish between text and attribute nodes and optionally operate on typed values (e.g., integer, double, string, etc.).

Definition 2 *A content index $I_{C(vType,nType)}$ only preserves document order for content nodes having the same value and covers all content nodes of the given value type vType and node type nType. vType can be undefined or a valid data type for content nodes (e.g., integer, string, etc.).*

$$nType = \begin{cases} \emptyset & \text{attribute + element (default)}, \\ attribute & \text{only attribute nodes}, \\ element & \text{only element nodes}. \end{cases}.$$

Content index access: Value predicates are typically evaluated using content indexes. Only using elementless storage, the content index includes path information, i.e., PCRs, which increases the index size, but may avoid document accesses.

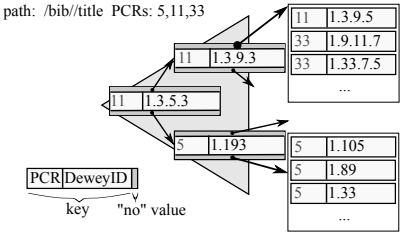


Figure 6.3: Sample path index

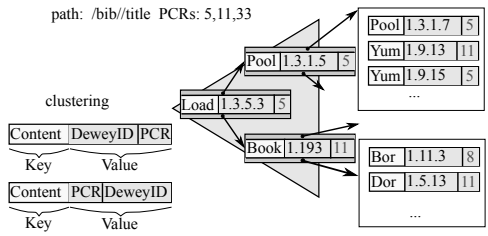


Figure 6.4: Sample CAS index

6.2.3 Path Index

This index type is only available for the elementless storage mode and typically used for the evaluation of complex path expressions within an XQuery. A path index is specified by one or more XPath-like expressions[2]. All nodes addressed by such an expression are indexed independently if they are attribute or element nodes. A sample path index, defined by $/bib//title$ is shown in Figure 6.3. By exploiting the path synopsis, it is very easy to map an XPath expression to a set of PCRs. Here again, *elementless* storage allows for two different clusterings of the path index, DeweyID-based and PCR-based, and key composition is equal as for the element index. An attribute index is a special path index having an attribute step in its path definition.

Definition 3 *A path index $I_{P(E)}$ preserves the document order and covers a set of XPath expressions E where $E \neq \emptyset$ and $E = \{e_1 \vee e_2 \vee ... \vee e_n\}$. Each e_i evaluates to a set of PCRs P_i and the index covers the union of all P_i in P where $P = \bigcup_{1<i<n} P(e_i)$.*

Path index access: Given a set of PCRs, a path index either delivers the whole set of referenced nodes, a subset, or none of them. That means, a query expression, i.e., path expression, is evaluated using the index definition. In case of requesting multiple path classes, it depends on the clustering whether false-positives have to be filtered or distributed index entries need to be accessed. A path index may imitate an element index when being defined as $//*$[3].

6.2.4 CAS Index

A content and structure index (CAS) is a combination of XPath expressions and value-based indexing to support complex XQuery evaluation. A sample for the $/bib//title$ expression is shown in Figure 6.4. Together with *elementless* storage, the following options are allowed: clustering, typing, and uniqueness.

[2] We do only accept forward axis steps, name tests, wildcards, and their disjunction

[3] Nevertheless, an element index is always primarily clustered by its name directory, which only happens by chance for a path index.

Table 6.1: Index feature comparison

Index	Query-support features				PCR-based	Coverage
	clustering	typed	axis	unique		
element	yes	-	-	-	optional	$\leq$ 100% of element nodes
content	no	optional	-/attribute	-	no	$\leq$ 100% of text nodes
path	yes	-	yes	-	yes	$\leq$ 100% of path instances
CAS	yes	optional	yes	yes	yes	$\leq$ 100% content
document	fixed	-	-	(DeweyID)	optional	100%

Definition 4 *A CAS index $I_{CAS(E,clust,type,unique)}$ is limited to preserve document order only for very certain cases. The index covers a set of content nodes for the given XPath expressions E where $E \neq \emptyset$ and $E = \{e_1 \vee e_2 \vee ... \vee e_n\}$ and $clust = \{DeweyID|PCR\}$. Each e_i evaluates to a set of PCRs P_i and the index covers the union of all content nodes belonging to these P_i in P where $P = \bigcup_{1<i<n} P(e_i)$. The type attribute is equal to that of the I_C index in Definition 2. Optional uniqueness can be enforced by setting $unique = \{true|false\}$. Content of the matching nodes forms the set of index keys and the values are defined as follows:*

$$value(I_{CAS}) = \begin{cases} DeweyID + PCR & \text{if } clust \text{ is DeweyID (default)}, \\ PCR + DeweyID & \text{if } clust \text{ is PCR}. \end{cases}.$$

CAS index access: Due to the content-based clustering, CAS index usage is primarily targeting at queries containing a content predicate. Otherwise, filtering and (often) ordering would dramatically increase the access costs. Additional path filtering based on PCR sets makes this index one of the most powerful and flexible index options. A CAS index defined for the whole content, i.e., $//*$ is identical to a content index.

Index Type Overview

We summarized the feature list of our indexes in Table 6.1. It shows that all of them have different characteristics in terms of query support or XML entities being indexed. The *simple* index types – element and content – are available for both, elementless and full storage modes. In contrast, the *advanced* index types – path and CAS – are strictly based on the elementless storage features such as PCRs.

Another type of indexes, so-called *attribute indexes*, is represented by a path index in XTC containing the attribute XPath axis in its specification. This makes a separate index type obsolete. Although XTC allows to define attribute indexes, which are simply mapped into appropriate path indexes, and, if attribute content has to be indexed by CAS indexes, respectively, we rarely make use of this distinction.

This overview indicates that the flexibility of index types and configurations is definitely a strength of XTC to provide tailored query support.

6.3 Query Processing in XTC

We want to give a brief introduction into XTC's query processing, which is necessary to enable the reader to interpret query plans and to understand the cost-based optimizer decisions. Because the internal representation is based on the QGM (cf. Sections 2.4.2 and 2.5), we introduce its basics and core operators. Some aspects of query optimization will be presented, before we show how to integrate available indexes into the query planning process.

6.3.1 XQGM and Query Plan Operators

Note, this introduction is only addressing the *optimizer* part (cf. Figure 2.4), which selects the "optimal" query execution plan (QEP), i.e., the plan found in a reasonable time that seems to be the most efficient one based on the current cost model for IO and CPU processing.

1. **Algebraic rewriting** So-called transformation rules are applied to transform the initial query graph into semantically equivalent plans, which have different logical plan operators or a differing operator combination and order. These rules are applied in the specified order based on their priority, i.e., cycles of rule application are avoided and rule dependencies observed.

2. **Plan generation** For each alternative plan that was generated before, costs for IO and CPU are estimated taking all plan operators and selectivities as well as intermediate result sizes into account. Eventually, the plans are ranked by their expected costs and the cheapest one is translated into a physical plan to be executed (QEP), i.e., logical operators are translated into their physical counterparts.

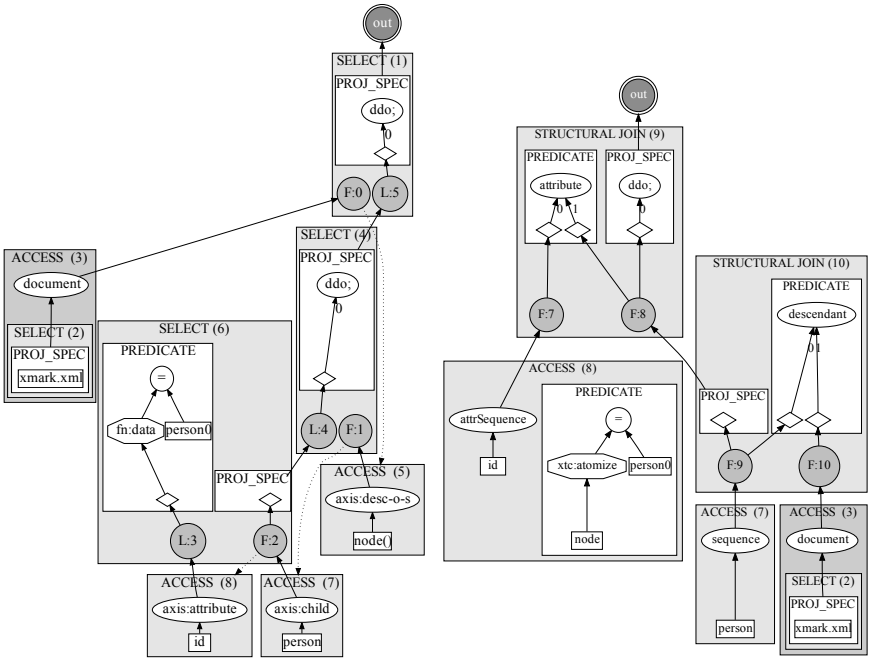
XQGM

XTC applies an extended version of the query graph model (QGM) introduced by [HFLP89] for relational query processing. The requirement for an extension was driven by [Mit95] and realized by XQGM in [MWHH08, Mat09].

An XQGM is an operator graph representing the query in a procedural way based on logical operators (boxes) and their data flow (arrows). Each operator typically produces a sequence of output tuples (according to the XDM) based on its input tuple sequence. For instance, a simple XQuery (similar to XMark query 1 [SWK+02]):

$$doc(\text{"}xmark.xml\text{"})//person[@id = \text{"}person0\text{"}]$$

returning all *person* subtrees having an *id* attribute with the value *person*0, is internally represented by the XQGM shown in Figure 6.5(a). This plan contains several *select* and *access* operators. Operator inputs are operating on so-called *tuple variables* (*L* and *F* circles), which control the data flow. Furthermore, an operator may house *predicates* for certain conditions, filtering, or order expressions. Usually, a *projection* is specified to define the operator output. A *tuple variable* can be *let*-quantified (L) or *for*-quantified (F). A *F*-quantified input is operating on a single tuple at-a-time, evaluates assigned predicates, and in parallel, may provide



(a) Initial XQGM for sample query

(b) Algebraic rewritten "final" XQGM

Figure 6.5: XQGMs for sample query: $doc(\text{"}xmark.xml\text{"})//person[@id = \text{"}person0\text{"}]$

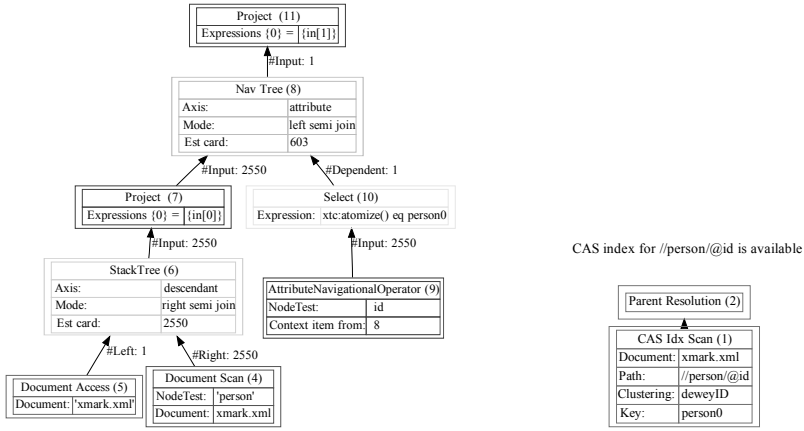
the evaluation context for a nested subexpression (dotted lines), too. In contrast, *L*-quantified inputs do not iterate over the input tuples, but instead operate on the complete tuple sequence.

The first optimization step *Algebraic rewriting* of our initial XQGM for our sample query results in the final XQGM shown in Figure 6.5(b). The *descendant-or-self* filtering of the *access* operator 5 in the initial graph is transformed into a *structural join* (10 in the final XQGM). Also the attribute navigation is transformed into the *structural join* 9. Because predicate pushdown is also available for XQGMs, the equal predicate (i.e., "person0" comparison) is pushed down into the *access operator* 8.

Operators

For this work, the following logical operators of XTC's XQGM are important:

- **ACCESS**: Three access types are available: *document access* delivering the root node(s) of a document (or collection), *node access* providing access to one or more nodes, and *sequence access* providing multiple accesses to a single node.



(a) QEP without secondary indexes

(b) QEP with secondary indexes

Figure 6.6: Query execution plans for sample query

- **SELECT**: A select operator is a primitive operator, initially generated for predicate evaluation, projections, or to *join* input sequences, which makes the naming appearing a little bit awkward. Many rewriting rules are based on this operator.

- **GROUP_BY**: Standard grouping and aggregation functionality based on a given predicate.

- **UNION, INTERSECT, EXCEPT**: These operators operate on *sets* of ordered tuple sequences. Their semantics are similar to their relational counterparts.

- **SORT**: Sorting and duplicate elimination.

- **JOIN**: Basically n-way structural join operators are evaluated in XTC. Therefore, this core operator is constructed during the algebraic rewriting and includes a join predicate as well as a projection specification. Note, a special twig operator [Mat09] was implemented as well.

6.3.2 Optimization

Many concepts within XTC's current query optimizer are tailored to the features of the elementless storage, especially to the extended (i.e., statistics-carrying) path synopsis. Nevertheless, the basic concepts are independent of the storage model, but additional efforts may be required to achieve the same quality in optimization.

Physical operators in XTC implement the ONC protocol (open, next, close). Besides a basic set of navigational, scan, and structural join operators, many set-oriented and node-oriented operators exist for special purposes such as sorting, unnesting, or value-based joining.

Possible QEPs for our sample query are depicted in Figure 6.6. If no indexes are available, the final (and hence the cheapest) QEP, shown in Figure 6.6(a), includes a *scan* (operator 4), two *structural joins* (stack tree operator 6 and navigation tree operator 8), and an attribute *navigation* (operator 9). In contrast, having tailored CAS index support for that query may result in the QEP shown in Figure 6.6(b), which is obviously smaller and solely consists of a single *index scan* and a so-called *parent resolution* operator that evaluates parent nodes, which comes at no charge for our elementless storage (cf. Section 5.6).

Furthermore, the left QEP is annotated with cost estimations for the structural join operators (called *estimated cardinality*) and with the actual runtime statistics attached to the data flow arrows. Cost estimations are based on the extended path synopsis and XTC's cost weights for IO and CPU [Wei11].

Special Operators

We extended XTC with two additional types of physical scan operators that are tightly coupled with XTC's internals.

- **PCR scan**: This operator is using a PCR filter, which is first evaluated against the path synopsis, to scan a document or collection for matching nodes. Its high throughput is achieved by avoiding the full decoding of physical records and directly filtering the node stream at the B-tree level.

- **Shared scan**: Because the integration of this operator is only experimental, we do not yet include it into our cost-based optimization framework. The shared scan only works for read-only transactions that require document scans on the same document. A kind of *window*, which behaves like a buffer, is cycling through the document as long as listeners, i.e., scan operators, are connected.

Another optimization is addressing operator matching and query rewriting. We allow to promote predicates, which are embedded in a path expression to be separated into *WHERE* expressions and vice versa. This semantic-preserving query reformulation enables XTC to find tailored operators and to evaluate the predicate(s) more efficiently.

6.3.3 Construction of Index Access Alternatives

Before the optimizer actually selects an index to be used, all possible indexes have to be identified for the given query. During the query translation phase of an XQGM, all document and collection references are collected. Only the index descriptions for these references are made available to the optimizer. The optimizer basically has two mechanisms to search for indexes and to "branch" the query plan for those operators that may be evaluated by alternative index use. In Figure 6.7, a sample path query's XQGM (a) and two possible QEPs (b and c) are shown. The first mechanism, called *access operator alternatives*, creates query (sub)plan

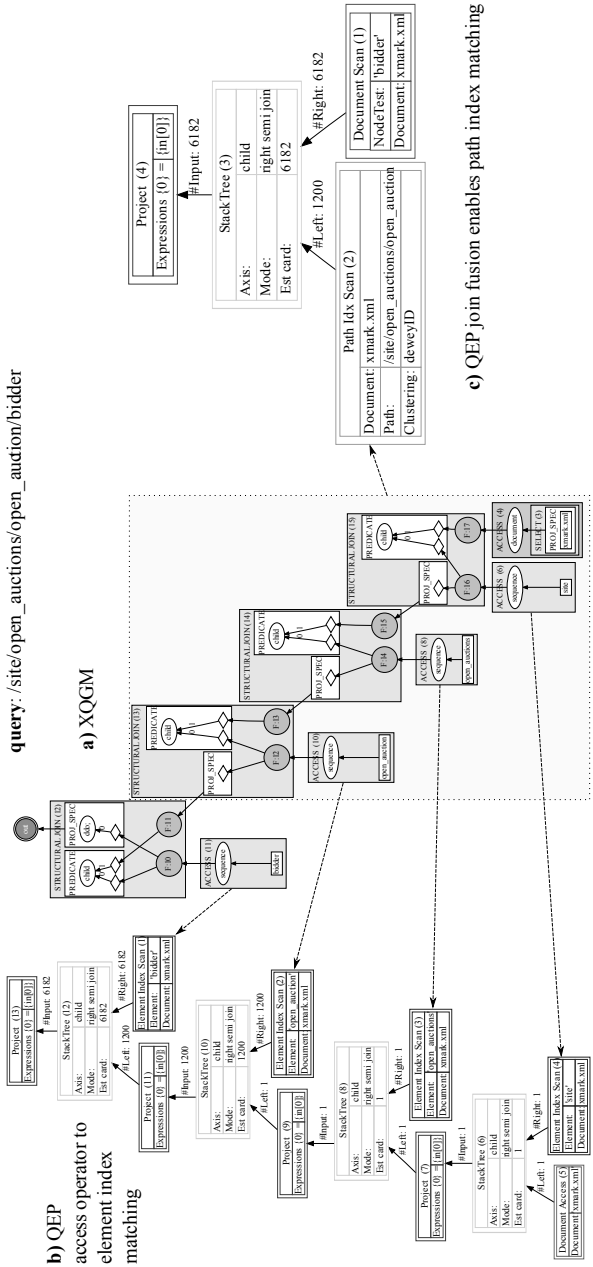


Figure 6.7: XQGM operator to physical index operator mapping sample

alternatives containing element index or path index operators. On the left, the sample QEP indicates that all logical access operators are translated into physical element accesses. The second mechanism is trying to *fuse path steps*, i.e., structural joins in the XQGM, in order to apply path or CAS indexes for these paths. The sample on the right reduces the join cascade by substituting three joins by one path index access. Even (value) predicate expressions are observed by the index search. However, the combination variety of paths increases exponentially with each new path step, i.e., structural join, whereas the complexity is often reduced if predicates are present, because they can make certain index types inapplicable.

During the plan cost evaluation phase, index operator costs are estimated based on current document and index statistics as well as their clustering and order-preserving properties.

6.4 Index Use

All index types introduced in Section 6.2, serve certain XML query specifics. For instance, depending on the availability of such an index variety, XMark query 01 [SWK+02]

```
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person[@id = "person0"]
return $b/name/text()
```

can be processed with increasing sophistication:

- *Without secondary index*: Query evaluation plan (QEP) in Figure 6.8a) indicates that several scans (*Document Scan*) over the entire document index are necessary to join (*Structural Join*) the XPath steps. Expensive navigations (*Attribute Navigation*) are needed to retrieve all "@id" attribute values.

- *+Content Index*: This option may replace the navigation operator by a *Content Index Scan* to reduce document index access (see Figure 6.8b)).

- *+Element index*: In Figure 6.8c), the QEP can replace three document scans by an *Element Index Scans*. Note, the resulting node streams may contain nodes originating from different paths to be filtered out.

- *+Path index*: The QEP in Figure 6.8d) illustrates the use of a *Path Index Scan* that covers, at least, the path $/site/people/person$. Exploiting this path index allows to remove the left QEP part, avoiding two element index scans and an access to the document root node.

- *+CAS index*: Such indexes have higher selectivity compared to generic content indexes. Hence, an additional *CAS Index Scan* (in Figure 6.8e)) can be used to substitute the content index scan. The CAS index result needs to be sorted by their labels (document order).

Even this simple query example impressively illustrates the variability of XML index support. Apparently, it is fairly difficult for a DB administrator to define suitable indexes beyond

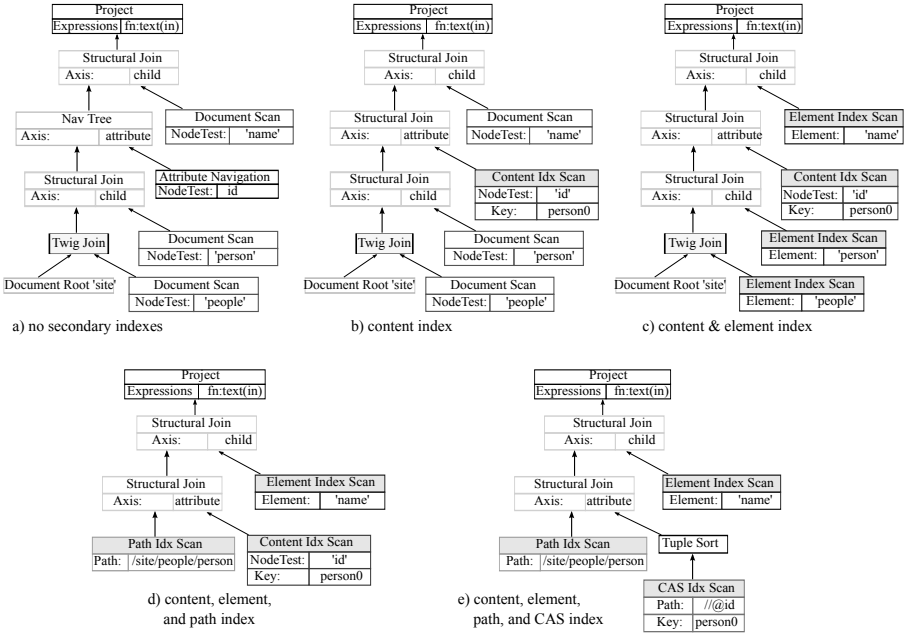


Figure 6.8: XMark query 01: Index-driven alternatives in query evaluation plans, visualized using a simplified version of XTCcmp [MWHH08].

the simple element or content index. Moreover, the variety and flexibility of XML structures and, in turn, query predicates (e.g., wildcards (*), descendant axis (//)) make it impossible to specify static index configurations facing ad-hoc queries.

Index Size

Even for our simple sample query, slight index variations lead to varying index sizes. Table 6.2 gives an overview what to expect from these different setups in terms of additional space requirements in relation to index coverage. We compared *minimal* and *standard* index definitions for each scenario. The *standard* definitions are easy to specify and offer a wider scope of query support, but require considerably more space. In contrast, the *minimal* definitions are perfectly tailored to the scenarios and require minimal additional space. All definitions from (1) to (9) can be found in Appendix B.6. Note, content indexes are automatically translated into appropriate CAS indexes because elementless storage was used. This usage of elementless storage, furthermore, implicates that, especially in *standard* definitions, the coverage may excess 100%, because virtualized inner element nodes need to be indexed explicitly.

Table 6.2: Index configurations for sample query evaluation plans

Index use scenario	Index type	Index minimal			Index standard		
		space in %	coverage in %	def	space in %	coverage in %	def
a)	document	100	100	-	100	100	-
b)	content	+8	24.35	(1)	+42.97	99.19	(2)
c)	content	+8	24.35	(1)	+42.97	99.19	(2)
	element	+0.86	4.7	(3)	+20.21	106.25	(4)
d)	content	+8	24.35	(1)	+42.97	99.19	(2)
	element	+0.58	3.08	(5)	+20.21	106.25	(4)
	path	+0.27	1.63	(6)	+0.27	1.63	(7)
e)	content	+8	24.35	(1)	+42.97	99.19	(2)
	element	+0.58	3.08	(5)	+20.21	106.25	(4)
	cas	+1.23	3.84	(8)	+34.96	74.84	(9)

6.5 Index Selection Problem

Besides the flexibility of XML, another major challenge comes up when space restrictions are to be met and/or insert, update, and delete (IUD) operations are present. Thus, the well-known index selection problem (from the relational world) is substantially more complex for XML indexing. The evaluation of a path expression containing multiple path steps may exploit various index types in various combinations (to avoid unwanted document scans). As a typical example, Figure 6.9 reveals considerable complexity to only search for a proper index type – a base task of a query optimizer. Although the listed queries are fairly similar, a heuristics-based query optimizer would identify the query properties and usually look for indexes in differing orders given in the illustration. Because a full set of indexes can never be maintained, index selection typically favors those serving different queries. Furthermore, this simplified decision tree hides the necessity that a cost-based selection usually accounts for size (height) and cluster properties of an index. Moreover, a combination of different indexes and types is possible as well, thus considerably increasing the search space.

Containment Problem

Digging a little deeper in the search space of alternatives reveals the *containment* problem for XML indexes. Related to the introductory example of Section 6.4, it is difficult to strictly distinguish index uses as long as they are overlapping, contained in each other, or identical. For the user or DB administrator, containment may not be visible during index definition. For instance, an element index specified for "*person*" nodes may have the identical scope as a path index defined on "$//people/person \vee //show/person$". Therefore, an index configuration may contain redundancy to be maintained during updates, too. In contrast, the system may prefer one of these index alternatives due to cost effectiveness resulting from a different clustering or compression overhead. Due to the expressiveness of XPath, it is easy to define indexes containing a subset or being a superset of an existing one. For instance, the set of nodes addressed by the path $/a/b/c/d$ are $\subseteq$ compared to the path $/a//d$. Adding wildcard steps and different index types amplifies the containment problem.

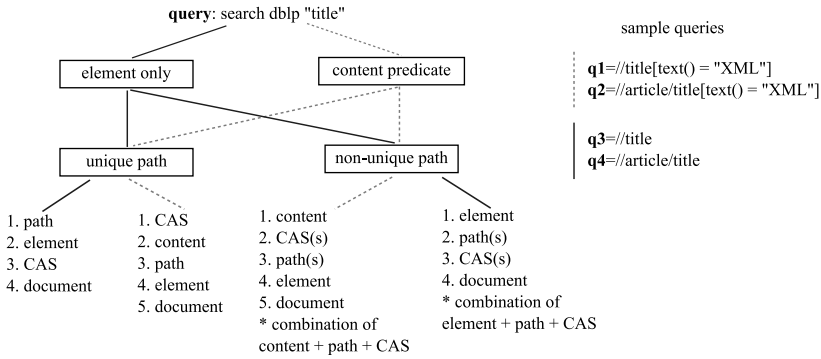


Figure 6.9: Index-type decision tree

Generalized Indexes

XML indexing allows for tailor-made index definitions favoring specific queries, e.g., a full path containing only child axes. To support as many queries as reasonable, specialized indexes should be combined to more general indexes to share physical structures and, at the same time, improve buffer usage. On the other hand, such a shared usage would provoke increased contention produced by parallel IUD queries, which, in turn, would again advocate the use of more specific indexes. Apparently, this flexibility for indexing and storing XML has to be identified and exploited when selecting a set of query-supporting XML indexes.

6.6 Summary

In this chapter, we presented XTC's set of secondary index options, their properties, and flexibility. Although some index options are based on the elementless storage mapping and, therefore, rely on the path synopsis concept, their general application is not impaired.

Challenging for query processing is the integration of secondary access paths. We presented some insight how the cost-based query processor of XTC actually selects indexes. We identified the problem of accurate statistics (monitoring) and cost metrics (decision/model) to generate valid QEPs. Although the query optimizer itself is a kind of self-tuner for access path selection, it heavily relies on statistics and indexes to choose from, which leads to further problems. Besides the prominent index selection problem (ISP), XML indexing causes additional problems such as the containment problem, type flexibility, and generalization options. Thus, the resulting huge search space of alternatives needs to be analyzed and evaluated automatically, which is the goal of our next chapter.

Chapter 7

Index Self-Tuning for XDBMSs

The previous chapter disclosed the opportunities provided by sophisticated indexing options within an XDBMS and the challenges that arise out of that. Finding an optimal index configuration is an NP-complete problem due to the large number of possible combinations [PS83]. Therefore, indexing decisions must be workload-driven to find a suitable setup, i.e., queries need to be analyzed by administrators and indexes must be created by hand. The workload dependency becomes especially critical when query patterns change and the current index configuration is outdated. Simply creating new indexes can easily degrade system performance, because they may require maintenance and their materialization causes additional IO and contention. Hence, changing an index configuration is a sensitive task and a cost estimation is required to support such a decision on-the-fly. In this chapter, we propose an integrated and autonomous indexing framework for a native XDBMS. This cost-based index configuration management is utilizing various index types and the XQuery compiler. We will discuss related work, before we describe the framework, its integration into XTC, and its performance gains.

7.1 Related Work

Numerous approaches for *index tuning* have been proposed in the relational world. Almost all commercial systems are equipped with some sort of *Index Advisor*, which basically analyzes workloads, i.e., queries, data, and query frequencies to estimate query processing costs for hypothetical index configurations. The *Index Tuning Wizard* by Microsoft and their AutoAdmin project use "what-if" index recommendations [CN97, CN98]. Oracle ships an *Index Tuning Wizard* [DRS+05], too. In the *SMART* project [LL02] of IBM, the query optimizer is exploited for index recommendations [VZZ+00, ZZL+04]. Those kinds of tuning tools operate offline and do not change the index configuration at all. They only inform the DB administrator about alternative index configurations.

Recommending tailored XML indexes is still an open research topic. However, the integration into (X)DBMSs is mostly substituted by relational realizations. Only some offline approaches, such as XIST [RPBP04] and KeyX [HKL05], are capable of identifying XML paths and recommending simple path indexes. But they are not fully integrated into the cost-based query optimizer. In the context of DB2, a "tight optimizer coupling" for XML index recommendations was investigated [EAZ+08b, EAZ+08a]. Thus, the cost-based decision and selection of indexes through the query optimizer itself leads to index recommendations that are guaranteed to be used by the system. Unfortunately, the authors do not indicate at which state index candidates are generated and how they are evaluated.

The next steps towards autonomous index management are solely developed in research projects. As a kind of add-on, the external tool *QUIET* [SGS03] monitors and analyzes index candidates before instructing the DBMS to change its index configuration, i.e., materialize or remove individual indexes. This approach requires a query proxy for external query inspection, which is not necessarily compliant with the DBMS kernel. An integration into PostgreSQL was done within the COLT approach [SAMP06]. The strength of COLT is to adjust the overhead for index analysis to the current load and to change the index configuration online in case of workload shifts. Each index candidate is profiled and its potential benefit for the current workload is evaluated. During a reorganization phase, indexes are materialized and removed. However, the authors do not disclose how they actually identify index candidates. Another PostgreSQL integration is using *soft indexes* to evaluate "what-if" index candidates [LSSS07]. Moreover, indexes can be built while a table scan is taking place and a *SwitchPlan* operator is used to switch a scan operator into an index access operator on-the-fly. Although these approaches are only available for relational data and indexes, they served as model for our XML indexing framework in XTC.

Instead of reconfiguring the set of indexes, certain approaches try to adjust a single XML index according to the query workload. Inspired by DataGuides and similarity-based path matching, APEX [CMS02] was introduced to be workload-aware. Compared to our path index, it basically adjusts its definitions, i.e., set of PCRs, which enables us to fully imitate APEX as well. Another approach, named D(k)-Index [CLO03], is based on the A(k)-Index [KSBG02] and 1-Index [MS99]. Again, index-defining path expressions are refined according to frequent query patterns, which are identified using a similarity measure. The minimum distance of paths is denoted by "k", of which A(k)-Index uses a fixed value for k, and D(k)-Index uses a dynamic one. Both approaches are solely isolated indexing approaches, i.e., without the integration into a DBMS. The authors only touch the topics index update and query support. Moreover, the indexes seem to operate fully autonomous and provide no central mechanism for a DBA to control them or a DBMS to coordinate multiple indexes.

7.2 Autonomous Indexing Framework

An architectural overview of our autonomous indexing (AI) framework is sketched in Figure 7.1. The AI framework is hooked into the existing database server. The metadata management is extended to handle new index attributes such as *virtual* or *managed*. The framework itself follows the *monitoring, decision, action* approach to autonomously manage the index configuration of a database.

Monitoring To propose effective index definitions for a dynamic workload, an incoming query is monitored and index candidates are derived for that specific query. Furthermore, index usage statistics are collected to account for costs and benefits.

Decision Supported by a cost model for index access, update, creation, and deletion, the *index manager* regularly triggers the *index advisor* to calculate the optimal index set and to update system-wide index usage statistics. During this optimization step, the *ISP* is addressed.

Action Depending on the *AI* mode (i.e., lazy, eager, deferred), new indexes may be materialized or existing indexes may be removed.

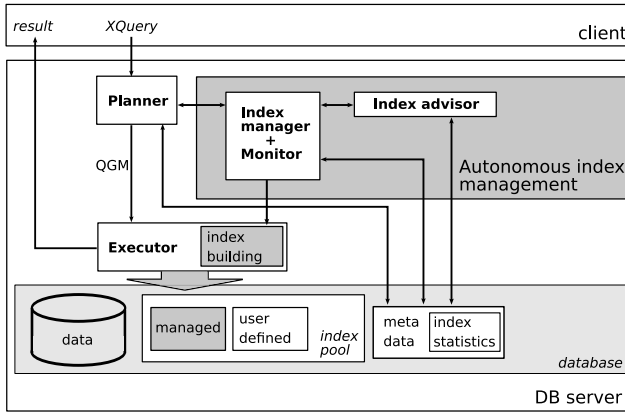


Figure 7.1: Autonomous indexing framework

The *AI* management is working aside the user-defined index set. That means, user-defined indexes have a higher priority and should not be touched but observed when usage calculation takes place. As long as an index is *virtual*, the executer does not know this index but the query planner can be injected to include both *virtual* and materialized indexes. Therefore, the *AI* has to estimate index access costs to support query planning for all kinds of indexes in a "what-if" query processing style.

The different *AI* modes may schedule index materialization, wait for beneficial load situations, or exploit an autonomous index in the same query the candidate is derived from. Therefore, we propose the following *AI* modes:

Eager Each query may generate index candidates which can be materialized before or during query evaluation to directly exploit this index. This mode focuses on local optimization.

Lazy Index candidates are materialized after query execution but may observe a global cost-benefit calculation beforehand. This mode focuses on query runtime which should not be penalized by index materialization overhead.

Scheduled The most comprehensive approach is to defer index materialization according to a certain schedule or to a certain point in time when the system is operating at a low load. The main focus for that mode is to globally optimize index candidates for multiple queries. Although this approach may miss an intermediate optimal index configuration, it reduces management overhead and temporary beneficial index candidates.

7.2.1 Virtual Indexes

The concept of virtual indexes [LSSS07, SH10] is necessary to exploit the DBMS query optimization capabilities. If a prospective index candidate is identified, a virtual index is created. It consists only of a definition and is not materialized, i.e., it requires zero disk space but cannot

yet support query processing, nor causes maintenance overhead. However, instrumented optimizers can take them into account for query planning, i.e., search for alternative query plans. Furthermore, statistics are generated and optionally collected for virtual index use, which constitute the starting point for *self* decisions of the AI system. First, candidates have to be derived or generated and, second, these candidates have to be evaluated. We will sketch both steps in the following.

Candidate generation During query planning, either the internal query structure is analyzed [SH10] or optimizer calls, searching for appropriate indexes, are collected [EAZ+08b].

Candidate evaluation The optimizer-based process of index candidate evaluation is quite simple (in Section 7.3.4, we will present more details):

1. Execute queries Q using all materialized indexes I_m and track cost estimations $cost(Q)$ and real costs for their QEPs.

2. Optionally replan (selected) queries Q_e, $Q_e \subseteq Q$ with several index configurations, containing virtual indexes I_v and materialized indexes I_m. Analyze the QEPs – actually without execution – and collect the cost estimations $cost(Q_e)$.

3. Calculate the expected $benefit$, while observing maintenance costs mc for indexes[1]:

$$benefit = cost(Q_e) - cost(Q) - (mc(I_v) + mc(I_m))$$

7.2.2 Index Configuration Self-Tuning

Our framework uses a cost-benefit model that addresses costs for index creation and updates as well as benefits for query processing (i.e., XQuery). To observe space restrictions, dynamic programming or Greedy search is used to select the (new) index configuration. Based on the notation from Section 7.2.1, a new configuration is achieved by dropping indexes I_{drop} with $I_{drop} \subseteq I_m$ and materializing indexes I_{create} with $I_{create} \subseteq I_v$.

An existing index configuration permanently has to keep track of its usage figures, i.e., cost-benefit numbers for each virtual and each materialized index. Frequent analysis of the current configuration is necessary, because workload shifts may quickly degrade an index configuration and cause unnecessary penalties for query processing. Yet, the overhead for maintaining and analyzing virtual indexes is not negligible. Besides transactional properties and space consumption, algorithmic overhead has to be taken into account, too.

7.2.3 Update Issues

Autonomous indexing typically relies on the identification of access paths for each kind of query, i.e., XQuery statements (read-only) and XQuery Update (modifications) [AYBB+08]. Thus, the mapping between (candidate) indexes and query parts is easily achieved and corresponding cost-benefit statistics are updated, too. That means, a query optimizer integrates

[1] Although we do not address the benefit distribution problem in this work, we will show how we can approach this issue using statistics-based cost/benefit estimations.

all the index information into a query plan (QEP) that are necessary to maintain their consistency. But analyzing a workload containing modification statements, i.e., XQuery Update expressions, may lead to severe problems. For instance, insert, update, and rename queries can introduce new paths or on-demand assembled paths, which are covered by an index definition but are unknown during query planning. The costs for those unforeseen index maintenance operations depend on selectivities and optional elements, making their estimation even harder.

A simple solution is to ignore the (possible small fraction of) unidentifiable virtual indexes to keep the overhead low. But retaining full consistency for virtual indexes requires to attach index listeners for them in case of XQuery Update statements. This, however, requires having a reasonable small set of virtual indexes at hand.

7.2.4 Local Optimization Issue

A workload consists of several queries that occurred during a monitoring period. For that period, the set of indexes is incrementally analyzed. That means, as soon as a query enters the system, appropriate candidates are generated for it (i.e., *locally*). Henceforth, costs are tracked for candidates that turned into virtual indexes. However, former queries of the same monitoring period could have caused costs, too, that are not accounted and lead to low accuracy.

Usually, only the candidate indexes selected by the optimizer to build the cheapest plan are considered for virtual indexes. Such a behavior may impede *globally* better configurations. Let us give an example comprising only two queries: Query one is favoring index a and query two is favoring index b. For both queries, an index c is used in the second best plan, instead of a or b. Thus, the costs for a and b could be larger than for c alone (materialization), making this a suboptimal solution. Moreover, the given size budget for indexes is violated by $a+b$ but not by c, which would result in only having a or b, and leaving one query unoptimized at all. Therefore, it is important to have a lightweight and query-crossing candidate generation and evaluation.

7.3 AI in XTC

The tight integration of AI into XTC primarily affects the query processing pipeline (cf. Section 2.4.2). For your convenience, we sketch the pipeline again in Figure 7.2, with an emphasis on the AI components. Basically, the AI evaluation (called AI step 1 and 2) is optionally added to normal query processing. After a query was processed (normal pipeline) and before the result is actually returned to the client, virtual indexes are generated within *AI step 1*. The XTC optimizer is now exploited to optimize the initial XQGM again for virtual, candidate, and existing indexes. Its result is analyzed in *AI step 2*, for cost and time accounting, before the actual query result is delivered to the client. The optional AI techniques will be addressed throughout the following paragraphs.

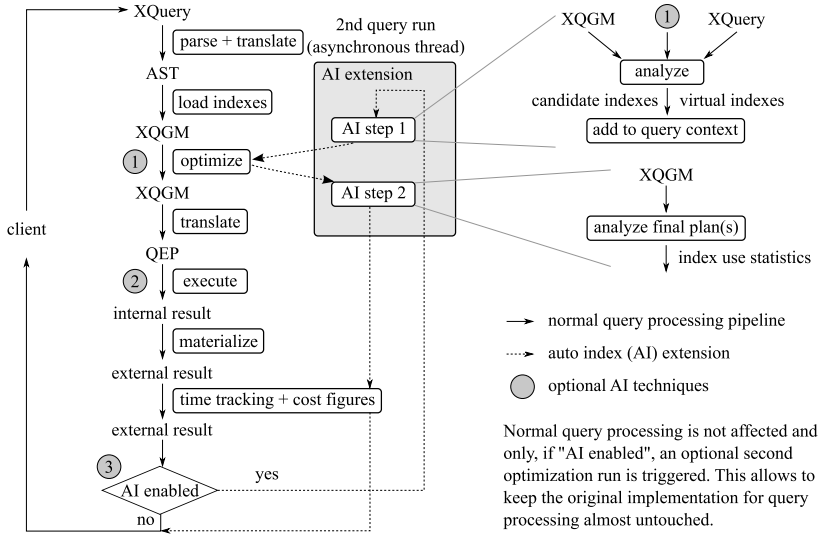


Figure 7.2: Query processing pipeline and AI extension

7.3.1 Index Management

The lifecycle of an index is sketched in Figure 7.3. Without AI enabled, XTC requires the user to specify indexes and also to drop them (a). In contrast, having AI enabled allows the user and the system to specify indexes cooperatively, although a fully autonomous management without user interaction is possible, too (b). This leads to the three index states – candidate, virtual, and materialized. Virtual and automatically materialized indexes are collectively called *auto indexes*. The necessary metadata to manage indexes autonomously is quite small. A state attribute and some counters for IUD, queries answered, tuples returned, cost, benefit figures, and AI statistics are sufficient. This usually amounts to less than 100 bytes of extra statistics for each AI-controlled index.

Fortunately, XTC manages its metadata in an XML document (i.e., _master.xml). Therefore, it is easy to adjust the metadata scheme without violating existing structures. In Appendix C.1, a sample containing new index attributes and AI structures is shown.

The issue of *update accounting* is addressed in XTC by the extended path synopsis carrying IUD statistics for each node, i.e., PCR-identified path classes. This optional feature (cf. Figure 7.2 optional feature 2) does a precise accounting and, therefore, allows to accurately calculate index maintenance costs.

Another issue for XML databases is *index matching* [LNF09]. The identification of suitable indexes (i.e., its definition to the current query) is again done via PCR matching. We only need to match an (ordered) PCR set derived from the query pattern with possible indexes. Each index remembers the document's largest PCR that existed when it was last evaluated against the path

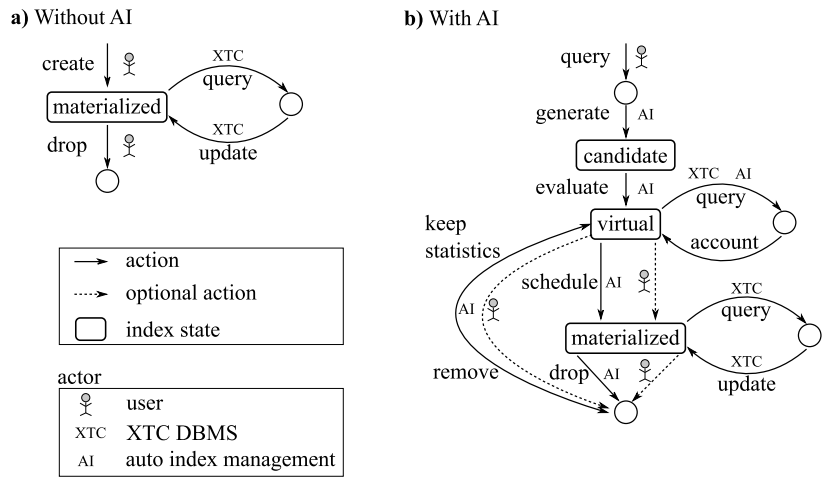


Figure 7.3: Index lifecycles in XTC (without and with AI)

synopsis. This ensures that, in case of new path classes, indexes that need to be reevaluated are easy to detect.

7.3.2 Candidate Generation

In our framework, we allow different ways of generating index candidates. In Figure 7.2 they are embedded in *AI step 1*. A lightweight query string analysis is fast but error-prone, whereas the query graph analysis is complex but delivers highly accurate candidates. A third approach, similar to [EAZ+08b], exploits optimizer calls to identify candidates.

Query String Analysis

Based on the input string (XQuery), regular expressions are used to extract document/collection names, path expressions, predicate expressions, let and for clauses, and element or attribute expressions. Heuristics help to "assemble" identified expressions and form meaningful index candidates. For instance, predicates (identified by enclosing brackets) hint at CAS index candidates. For and let clauses are used to generate additional paths by combining their surrounding path expressions. However, those concatenations may lead to meaningless candidates imposing additional overhead to the candidate evaluation phase. All element or attribute names, identified by their enclosing path step expressions, are collected into element index and path index candidates. Although this approach is fairly simple and fast, its quality is either impaired by multiple document references, awkward XQuery notations, or the generation of too many meaningless candidates and unnormalized query strings.

Query Graph Traversal

By traversing the *query graph* bottom-up, path information, join predicates, and value predicates are collected [SH10]. Our rule set to identify candidates can be found in Appendix C.2. Note, the identification logic is similar to the optimizer logic for index application. That means, even complex predicates can be analyzed and path information suitable for index candidates are extracted. Moreover, predicate types, iterations, multiple occurrences in the same query, and join participation are considered to improve the index candidate generation. Because this also requires a lot of analysis effort, we implemented a second, simplified method that reduces the set of predicate identification rules. This improved the search speed, made the approach more general, and kept the quality of candidates almost identical.

Injecting Query Optimizer

Recent work in [EAZ+08b], based on the idea of [VZZ+00], exploits the optimizer's index matching to generate candidates. We adopted this idea and collect the index lookup calls from the optimizer. The necessary DBMS extension is quite simple and delivers good results. In contrast to the query graph traversal, we could not infer quality aspects from the search order, neither could we identify how an index was planned to be used, e.g., point access, range scan, predicate evaluation, etc. Although this approach has still some limitations, it will never recommend unusable candidates[2].

Note, this approach should be embedded in the first query run (cf. Figure 7.2 optional technique 1), otherwise a third optimizer invocation would become necessary. Of course, this always causes a little bit overhead to the actual query processing, even if the AI extensions are processed by a separate thread.

Listing 7.1: Data structure to capture *path* expressions

```
enum Axes {CHILD, PARENT, DESCENDANT, ANCESTOR, ATTRIBUTE, DESC_ATTRIBUTE,
        DOCUMENT, FOLLOWIMG_SIBLING, FOLLOWING, PRECEDING, PRECEDING_SIBLING};
Step {
  Axes axis;
  String value;     // tag or attribute name or wildcard
}

Path {
  List<Step> steps;
  ...
}
```

Extending the Candidate Set

Before we show how to extend the set of candidate indexes, we introduce two basic data structures required for candidate index handling, namely *Path* sketched in Listing 7.1 and

[2]Other approaches have to verify the index candidate definition, which can easily be done exploiting the path synopsis and PCR set evaluation.

AIpathElement in Listing 7.2, respectively.

Listing 7.2: AI *path element* data structure

```
1  // Capture paths and indicate if they are "stoppers" i.e., end on attribute or
2  // text() functions => CAS candidates
3  AIpathElement {
4   Path path;
5   boolean attribute, CAS; // indicate the axis/predicate type
6   boolean alternative;  // indicates if a prior identified CAS index was transformed
        into a path index
7                         // because at least a non-leaf PCR is included
8   int count;  // how often was this path found in the query
9   boolean combination;  // is true if this path expression is part of at least a single
        concatenation
10  private Set<Integer> pcrs;
11
12  AIpathElement(Path path, boolean attrib, boolean cas) {
13    ...
14    // fix: remove final step of cas index path e.g., /homepage/*  <CAS> -> /homepage <
          CAS>
15   if (cas && path.getLastStep().getElementName().equals("*"))
16     ... // remove last step
17  }
18 }
19 AIpath {
20  ...
21  List<AIpathElement> paths;
22  boolean lastStepIsCAS;
23  boolean lastStepIsAttribute;
24 }
```

More index candidates are generated by combining (simple) path expressions. For instance, the query $Q_1 = //library/author[address/state =' TX']$ may result in two identified path expressions $//library/author$ and $address/state$, which are internally represented as *AIpathElement*s. By keeping the text predicate for the second path in mind (i.e., set *lastStepIsCAS* to true), the algorithm in Listing 7.3 tries to concatenate meaningful paths and predicates to extend the candidate set by adding more specific index candidates. While candidates are generated, their definition is matched against the path synopsis and "search paths" are stopped in case of non-matching definitions. Moreover, the data types of predicates are checked to tailor the index definition by specifying numeric or string typing.

7.3.3 Candidate Size Estimation

The evaluation of index candidates heavily depends on the simulated statistics, which are required by the query optimizer. Fortunately, the extended path synopsis statistics, presented in Section 5.9.1, are sufficient to accurately estimate B-tree statistics for all kinds of virtual indexes.

For each index candidate generated, an index size (*IdxStats*) estimation is calculated as shown in Listing 7.4:

Because this estimation needs to be done for all index candidates, the matching in line 4 exploits a path cache to speed up the generation of *PSNode* lists. Structural changes of the path

Listing 7.3: Combine paths algorithm

```
1 combinePaths(AIpath p, List<AIpath> exist) {
2   if (p.firstAxis() == CHILD) tryDescendant = true;
3   Set<Integer> pcrs = pathSynopsis.getPCRs(p);  // get PCRs for path p
4   if (pcrs.isEmpty() && tryDescendant) {
5     ... // set first axis of p to descedant
6     pcrs = pathSynopsis.getPCRs(p);  // get PCRs for changed path p
7   }
8   if (!pcrs.isEmpty()) {
9     AIpath test = p.getCopy();
10    for (AIpath e : exist) {
11      test.append(e);              // concatenates two AI path elements
12      if (test is CAS) {
13         if (pathSynopsis.includeNonLeafPCR(pcrs)) {
14           test.CAS = false;       // only leaf PCRs are allowed
15           test.alternative = true; // remember this type switch
16         }
17      }
18    }
19    if (tryDescendant) {
20      exist.add(test.getCopy());
21      // switch axis in p back to child]
22      ...                                  // restore original properties
23    }
24    else
25      exist.add(test.setPCRs(pcrs));  // add combined path and set PCRs
26  }
27  else
28   // invalid path, do not try to combine nor add to existing paths
29 }
```

synopsis, i.e., creation of new paths, or bulk updates and deletions in the document, empty the look-up cache to improve estimation accuracy. The *calcIndexStructure* method in line 11 uses heuristics, gained through experiments, to approximate the index size. The heuristics themselves take cluster-dependent key compression ratios, average descriptor overhead, and typical B-tree occupancy into account. The resulting *IdxStats* object contains all metrics the query optimizer might be interested in.

Listing 7.4: Estimate index size

```
1 IdxStats estimateSize(IndexDef ic, PathSynopsis ps) {
2   // generate path class-based set of synopsis nodes
3   IdxStats stats = new IdxStats(ic);
4   List<PSNode> nodes = match(ic, ps);
5   int size = 0; count = 0, updCnt = 0;
6   for (PSNode node : nodes) {
7     size += node.getInstances() * node.getAvgLength();
8     count += count; updCnt += node.getUpdateCnt();
9   }
10  // estimate index-type-dependent height and #leaves
11  calcIndexStructure(size, stats, count, updCount);
12  return stats;
13 }
```

A further optimization to reduce *IdxStats* estimation efforts is possible by incrementally

estimating index candidates, i.e., during bottom-up traversal of the query graph, where each step creates new and often more selective index candidates. Moreover, for each materialized index, accurate statistics are gathered during its creation, which are used to adjust the heuristics, especially for conten and CAS indexes.

7.3.4 Cost Benefit Calculation

Similar to [LSSS07], index candidates are frequently ranked by their cost-benefit ratio gained through the accounting of QEP benefit and index maintenance cost. Because this process may become too costly, the ranking frequency is adjusted each time to the recent success of index tuning and overhead. A Greedy algorithm observing space restrictions marks indexes according to their rank for the new configuration. Finally, indexes marked for deletion are removed and virtual indexes marked for materialization enqueued to be built asynchronously to normal query processing.

Index Usage Tracking

During query processing, potential index application is tracked as follows:

1. Determine *what-if* index set for each analyzed query. This set consists of all kind of indexes (i.e., candidates, virtual, and automatically materialized) that were used in the what-if QEP but not in the executed QEP. Note, different usage numbers of the same index may occur, which is accounted as well.

2. Estimated cost difference of both QEPs is equally attributed to indexes of the index set.

3. Query count statistics of the indexes are updated.

Cost Metrics for Index Ranking

Two metrics for index ranking are possible. One is based on pure *gain* figures, and the other one on *gain*/*size* figures. To rank all *auto indexes* I_x, we calculate their *gain* as follows:

$$gain(I_x) = benefit(I_x) - maintenance(I_x) - \gamma \times cost(I_x)$$

Complementarily, *maintenance* costs are approximated by:

$$maintenance(I_x) = (deletes(I_x) + updates(I_x) + inserts(I_x)) \times IO_{cost} \times 2$$

All the necessary counters are contained in the index statistics, which makes maintenance cost estimation fairly easy. Materialization cost estimation is based on the number of index pages and the number of document pages that need to be filled and scanned, respectively. Furthermore, optional sorting increases the cost estimation, which are calculated as follows:

$$cost(I_x) = (Doc_{pages} + I_{pages} \times 2) \times PageIO_{cost} + (I_{tuples} + Doc_{tuples}) \times PredEval_{cost}$$

$$\text{if (PCR clustering) } cost \; = cost + I_{tuples} \times SORT_{cost}$$

$$\text{if (ELEMENT index) } cost = cost + I_{tuples} \times SORT_{cost} \times 2$$

Note, materialization costs are weighted by a parameter γ, which controls the degree of required amortization; typically γ is chosen between 0.5 and 1. We do not yet adjust this value online, which is, however, straightforward and emphasizes the self-tuning characteristics.

7.3.5 Index Selection

Based on index ranking, the most promising virtual indexes are selected for materialization. To meet space restrictions, the system can reclaim index space by removing enough indexes having a negative gain or considered less beneficial than the new one(s).

The algorithm in Listing 7.5 shows the space-dependent and gain-dependent selection of new indexes. Several space constraints must be observed to correctly plan index materializations. For instance, when multiple candidates are planned at once, their (estimated) space consumptions have to be considered all the time, which is accounted by *cleared*. Moreover, space requirements for already scheduled index materialization jobs have to be tracked, too, e.g., *AI.ScheduledCreationJobsSpace* (line 13). The current space occupation by *auto indexes* is reflected by *used* (line 12). All these counters are used to determine the minimum space (i.e., *required* line 19) that is required for the new candidate. Note, in case of (additional) space restrictions that depend on the database size (lines 21–30), *required* is calculated by querying the total size of XML documents and collections from the metadata (lines 23–24).

If space needs to be reclaimed (lines 33–41), materialized *auto indexes* are ranked by their current gain and a simple Greedy search is used to find a minimum set of "bad" indexes that can be dropped in favor of the new one (lines 46–53). If enough space can be reclaimed (line 55), the indexes are dropped (line 59) and the statistics are updated (line 56) as well as a monitoring event is emitted (line 60).

Listing 7.5: Index selection observing space restrictions

```
1  indexSelection() {
2    List candidates;  // candidates that are selected for materialization
3    long cleared = 0;  // required in case of space restrictions
4    for (RankingEntry e : ranking) {
5      if (spaceCleared(e, cleared)) {
6        candidates.add(e);
7        cleared+=e.IdxSize;
8      }
9    }
10 }
11 boolean spaceCleared(RankingEntry entry, long cleared) {
12   long candidateSize = entry.IdxSize;
13   long used = AI.CurrentSpaceOccupation;
14   long reserved = AI.ScheduledCreationJobsSpace;
15   long required = 0; // min space required to create index
16
17   // first, check if enough space is available
18   if (AI.indexSpace > 0) {  // absolute space restrictions for AI
19      if ((used+candidateSize+cleared+reserved) > AI.indexSpace)
20        required = used+candidateSize+cleared+reserved-AI.indexSpace;
21    }
22    double dbSize = 0;
23    if (AI.indexSpacePercent > 0) {  // relative space restrictions
24      String sizeQuery = "fn:sum(doc(\"_master.xml\")//doc/statistics/@size)";
25      Result result = executeXQuery(sizeQuery, ResultType.STRING);
```

```
26      dbSize = Double.valueOf(result.getStringResult());
27      double required2 = dbSize * AI.indexSpacePercent / 100;
28      if (required2 < (used+candidateSize+cleared+reserved))
29        required2 = (used+candidateSize+cleared+reserved) - required2;
30      required = max(required, required2);  // more restrictive condition
31    }
32
33    // try to remove existing indexes having a bad gain
34    if (required > 0) {
35      Ranking existing;
36      for (Index idx : getExisting()) {
37        AIStats stats = idx.AIStats;
38        double maintenanceCost = (stats.DeleteCnt+stats.UpdateCnt+stats.InsertCnt)*
              INDEX_PAGE_COST*2; // fetch & store
39        double gain = stats.Benefit - maintenanceCost;
40        // optional: gain = gain/size
41        existing.rank(idx, gain);
42    } }
43    long reclaimed = 0;  // what is actually reclaimed by removing these indexes
44    long gainReduced = 0;  // can be negative
45    RankingEntry toBeRemoved = existing.pollLast;
46    List removeMe;   // remember auto indexes to be removed
47    while (toBeRemoved != null) {
48      if (reclaimed < required || toBeRemoved.gain < 0) {
49        if (entry.gain > toBeRemoved.gain) { // at least the gain of this index should
              not be larger than the gain for the new one
50          reclaimed += toBeRemoved.IdxSize; // add potential space savings
51          removeMe.add(toBeRemoved);
52        }
53      }
54      toBeRemoved = ranking.pollLast();
55    }
56    if (reclaimed >= required) {
57      // remove indexes & update  space occupation statistic
58      for (RankingEntry rm : removeMe) {
59        AI.free(rm.IdxSize);
60        dropIndex(rm);
61        STReporterImpl.getInstance().addEvent(new ReportEvent<long[]>(ReportComponent.AI
              ,"GenerateTop.DropIndex",...));
62      }
63    }
64    else
65        return false;
66  return true;
67  }
```

7.3.6 Optimizations

The set of indexes, the AI management, and the search for better configurations offer several starting points for improvement, mostly related to the overhead, i.e., processing costs. We implemented a set of optimization rules to shrink and optimize the set of candidate indexes, which can be found in Appendix C.3. Furthermore, in the following, we want to sketch a few quality and performance improvements we added to our AI framework.

Statistics Aging

In the course of time, statistics of index use may outdate and, thus, “block” better performing index configurations. Hence, it becomes beneficial to replace indexes due to their disadvantageous recent use in favor of new indexes that provide immediate gain. Therefore, we allow

certain statistics to age. During each analysis cycle, path synopsis statistics for updates and queries are reset to enable current candidate index estimations. Except for initial materialization costs, all prior statistic numbers are simply decreased to achieve aging. In contrast, index statistics could be kept with differential numbers giving a better indicator for recent usages, i.e., during the last tuning period.

Identification of Frequent Query Types

Reducing the number of queries to be analyzed by the AI framework is a strong demand. Typically, certain query types occur frequently due to prepared statements. We want to exploit this fact, but XTC does not (yet) support prepared statements. Therefore, we *identify similar queries* to save the costs of benefit estimations. The analysis is based on the raw query string. For each query, whitespaces are minimized (i.e., sequences of whitespaces are condensed to one), literals are removed, and a hash value for that string is calculated. The cost benefit statistics for individual virtual indexes are remembered and stored together with the hash signature. Using a query cache containing these hash values, it is simple to identify similar queries and to reapply the same cost benefit numbers again.

Pruning

To avoid the investigation of inferior alternatives, our AI framework contains several *pruning* steps; most of them are integrated as optional features (cf. Figure 7.2 optional feature 3).

The number of queries to be investigated is controlled through a query runtime threshold and an operator threshold. If processing time for a query is below a certain threshold, AI analysis is skipped. This threshold is increased, if no document scan operators are present. Note, query processing times, AI overhead, and AI cost-benefit statistics of each query are used to permanently adjust these thresholds. We use a (default) confidence tolerance of 10% to analyze some of the fast queries and, if necessary, to adjust the threshold. Queries without scan operators may increase this threshold only by (default) 25%, because the potential savings through a different index configuration (i.e., AI analysis is triggered) is usually smaller compared to scans replacing indexes.

Another pruning step is integrated into the search for alternative index candidates. The optional *CAS to path* index cast (and vice versa) can dramatically reduce the number of index alternatives without losing expressiveness. Index candidates, not picked by the optimizer, but frequently identified using XQGM, may also be transformed into virtual indexes if they occurred n times. Here, n is another pruning threshold. Index type preferences and index key type preferences can be used to prefer more general indexes, which may automatically support unseen (future) queries instead of tailored indexes currently present. Our *top-n plan inspection* threshold allows to analyze multiple plans of a query – enabling global cross-query optimization – and, thereby, the quality of AI is improved.

Index Space

Usually, the storage space for secondary indexes is limited to a fixed size. In our AI framework, we also allow a limit that depends on the actual data volume. For instance, the user may allow that up to 30% of additional space can be occupied by secondary indexes, optionally including the ones specified by users. This specification is always aligned to the actual data volume.

Parallel Index Materialization

Index materialization has a huge impact on DBMS performance. Besides transactional issues, exclusive resource consumption can cause additional penalties for query processing. For instance, index building often requires costly sort operations. Therefore, our framework keeps control of the number of parallel jobs building indexes.

7.4 Evaluation

Our benchmarks were performed on a Pentium IV computer (2x 3.2 GHz, 1 GB of main memory, 160 GB of external memory, Java Sun JDK 1.6), the same setup as configuration 1 in Section 5.10. The benchmark data originates from the benchmarks XMark [SWK+02] and TPoX [NKS07], and the XML repository [Mik].

Because XTC does not support XQuery Update natively, we implemented the TPoX update queries through the procedure concept of XTC. In Appendix C.4, a description of the TPoX implementation can be found. Basically, these queries are split into a read-only XQuery part using XTC's query processor and an update part using the DOM and SAX interfaces.

7.4.1 Index Estimation Accuracy

As mentioned before, index tuning heavily depends on cost estimation accuracy. Accordingly, the available statistics need to be used properly to estimate index access costs. In Figure 7.4, we show the estimation error of all materialized indexes generated for a subset of 10 different XMark queries in a workload. The total error E_t is the weighted sum of the cardinality error E_c, page number error E_p, height error E_h, and size error E_s (using the ratios of $\frac{estimated}{real}$ values):

$$E_t = \alpha * E_c + \beta * E_p + \gamma * E_h + \delta * E_s, \sum_{\alpha}^{\delta} = 1$$

To augment cost-based query optimization, the weights are adjusted to their expected cost. The cardinality error α and the height error γ are weighted by 0.3 each, because the query optimizer's cost estimation heavily depends on them. The remaining weight of 0.4 is equally distributed to β and δ.

The results in Figure 7.4 reveal that index estimation is fairly accurate in almost all cases (black dots equal $E_t \leq 12\%$). The minimum and maximum estimation error is shown via the error bars; clearly, the larger the workload (document size), the lower the min, max, and weighted error. Although this experiment did no lead to content indexes, for comparison, we

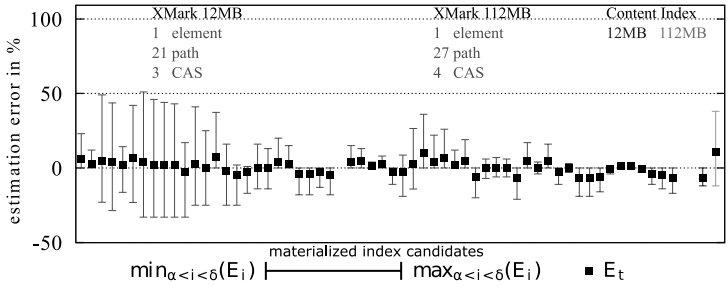


Figure 7.4: Estimating index characteristics: error margins

show their estimation error on the right-hand side. Apparently, the tiny overhead caused for statistics gathering is easily amortized by the value of accurate index estimations.

7.4.2 Index Candidate Generation Aspects

The different approaches for index candidate generation are subject to the next experiment. We assembled three query workloads, two containing each a subset of XMark queries and one containing the TPoX queries. We did not limit the index space. The results in Table 7.1 show that for simple queries, i.e., a single document reference and simple predicates, even the query string analysis results in almost perfect index recommendations. Only the complexity of TPoX queries is hard to capture on a string base.

In terms of query speed-up, both QGM-based approaches deliver almost similar results. However, the simpler version is noticeably faster, even with the *combine path* algorithm, which also led to results requiring less index space. Interesting is the difference for the TPoX workload, because the complex version recommends space-consuming path indexes of different clusterings instead of a cheaper and combined element index as the simple version does. The difference in query speed-up is negligible. In the following evaluations, we used therefore always the simple QGM$^+$ generation approach, because it constantly delivered a high quality in terms of indexes and space consumption.

Note, the table only lists indexes that were actually used and therefore materialized. The number of "useless" candidates is not included, because it is hard to distinguish between slightly worse and effectively useless candidates. In addition to that, the path-synopsis-based validation early removes impossible candidates.

7.4.3 Self-Tuning Quality

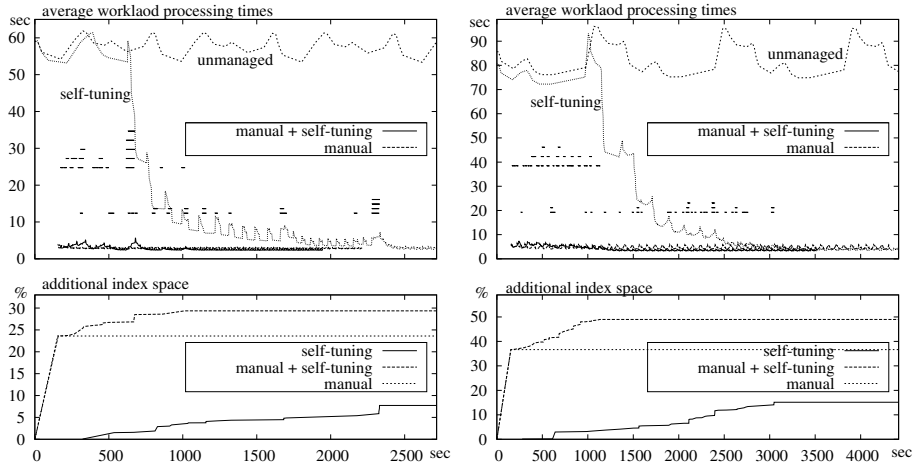
The quality of index self-tuning is a crucial aspect. Therefore, we designed four scenarios reflecting possible applications of index (self-)tuning.

- **Unmanaged**: No indexes were predefined or automatically created.

Table 7.1: Comparison of index candidate generation approaches

Candidate Search	XMark set 1	XMark set 2	TPoX
XQuery string	16*	15	10**
QGM (complex)	15	16	31***
QGM (simple)$^{+}$	17	15	22

$^{+}$ the algorithm *combine path* to extend the candidate set was applied, too
* only a CAS index was "missing"
** problems in determining descendant axes and multiple collections
*** path indexes instead of combined element index (similar coverage)



(a) XMark query set 1

(b) XMark query set 2

Figure 7.5: Workload processing times, index materialization (horizontal bars), and index space

- **Manual**: Only a content index and an element index covering the whole document are created, because they are straightforward to define. Furthermore, this setup usually delivers an acceptable performance and serves as baseline.
- **Manual + Self-tuning**: Based on *manual* consisting of an element and a content index, AI begins to tune the index configuration automatically.
- **Self-tuning**: No predefined indexes are present, solely AI is responsible for the index configuration.

The results in Figure 7.5 show the average workload performance (upper graph) and the index space consumption (lower graph), for two different XMark query sets. Additionally, the

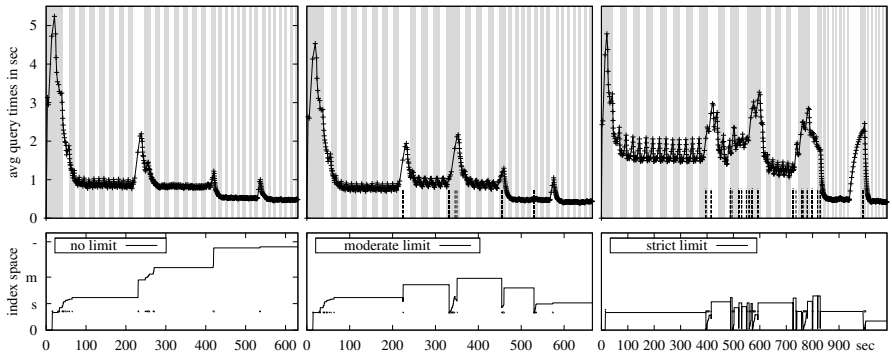


Figure 7.6: AI performance under workload shifts with varying space restrictions

horizontal bars, in the upper graphs, indicate index materializations in the self-tuning scenarios (i.e., scenarios 3 and 4). These bars correlate to the increased index space consumption in the lower graphs.

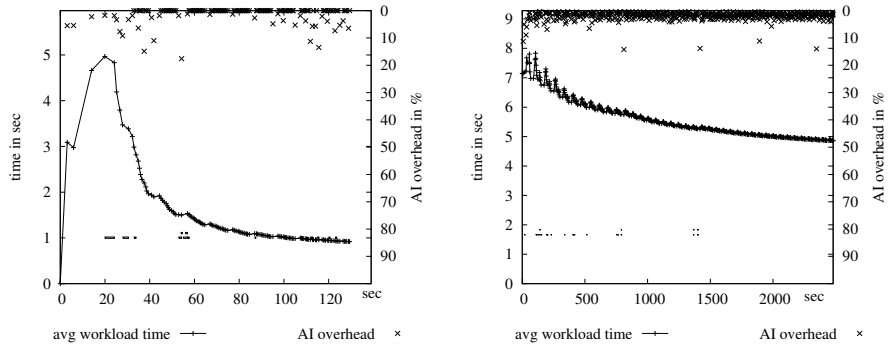
Although the scales of Figure 7.5(a) and Figure 7.5(b) are different, their findings are similar. The workload performance in the manual scenarios is satisfactory, but they require a lot of index space. Yet, the combined scenarios increase index space overhead while providing only marginal performance improvements. In the self-tuning scenarios, it takes some time until the first indexes become materialized. But AI continuously improves the performance down to the level (or even $\sim$ 10% better) of the manual versions, yet consuming clearly less index space. In sum, these results prove the effectiveness of the cost-based index tuning.

7.4.4 Workload Shifts

Changing workloads easily degrade query performance and may require adjustments of the index configuration. Therefore, we analyzed the reaction of our AI framework in case of workload shifts and varying space restrictions. The upper part of Figure 7.6 shows the average query performances as well as index deletions (vertical lines). The vertical gray and white bars show the runtimes of workload runs – achieved through the changing set of indexes. The results below show index space consumption and materialization (horizontal bars). We varied space restrictions from *unlimited* (-) to *moderate* (m) and *strict* (s). The workload consists of four major shifts that are best depicted by the results for the moderate space restriction (middle) – wide workload runs and performance peaks indicate the shifts.

The *unlimited* variant is the fastest one, but requires the largest amount of index space. Each workload shift is quickly adapted by creating additional indexes – no indexes need to be removed.

In our second variant *moderate*, AI automatically removes weak indexes in case of workload



(a) XMark 12MB (b) XMark 112MB

Figure 7.7: Workload processing times, index materialization (horizontal bars), and AI overhead

shifts to stay beyond the given space limit. Although less space is consumed due to the cost-based index selection, the query performance decreases only by a small margin for a very short time. This is illustrated by the higher peaks compared to the *unlimited* variant.

A *strict* space limit clearly causes a declined workload performance due to index thrashing (i.e., frequent materializations and deletions of indexes).

Eventually, space restrictions have a huge impact on the possible benefit through AI. However, workload shifts are quickly handled by the AI framework due to statistics aging.

7.4.5 AI Overhead

Self-tuning overhead is always a critical issue that requires justification. After we have shown the pure performance speed-up of self-tuning indexes, we evaluate the overhead caused by the AI framework. Therefore, we used the XMark workload for a small 12MB document and a larger 112MB document – results are shown in Figure 7.7(a) and Figure 7.7(b), respectively. As before, we focus on workload performance and index materialization. Additionly, the y-scale on the right-hand side is used to plot the overhead. For your convenience, the scale is in reverse direction, meaning zero overhead on the top of the graph.

The results show several findings. The overhead is always fairly low, mostly less than one percent but never higher than 15 percent. Actually, the outliers, i.e., high overhead values, mostly correlate with parallel index materialization jobs. That means, we observed interference of index metadata access and index materialization, instead of real AI overhead. Thus, the pure and permanent overhead of query analyses, i.e., even in situations when no self-tuning actions take place, is negligible. Because the self-tuning frequency adjusts itself, too, the overhead almost disappears.

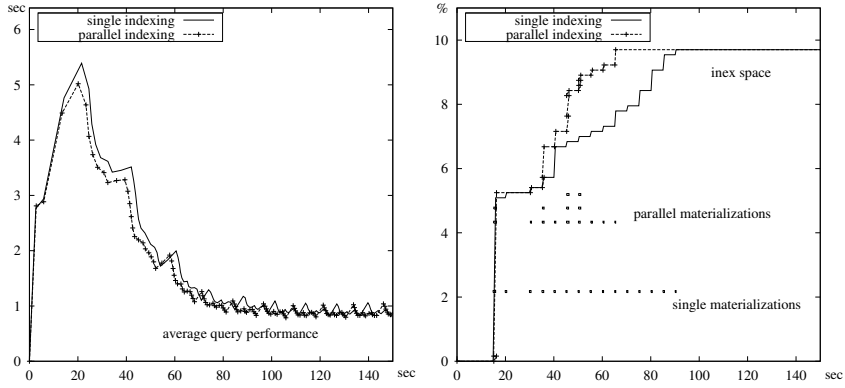


Figure 7.8: Impact of parallel index building

Virtual Index Listener and Update Accounting

To improve the accuracy of all index statistics, virtual indexes listen for updates, too. So-called virtual index listeners track IUD operations that would have caused index maintenance costs. Due to the fast PCR-based index matching, these in-memory counters are maintained without noticeable costs. We used a TPoX collection of ∼500 MB and created the same virtual indexes multiple times. Because the TPoX IUD queries contain a separate update logic, we could avoid to present all of these redundant indexes to the read-only query part. Thereby, only the overhead for the virtual index updates are accounted. Even when adding a hundred times more virtual indexes than real indexes – all of them matching the TPoX-based IUD operations –, the accounting overhead was less than one percent of the query processing time. Note, in real scenarios, only the set of matching index listeners are attached to the IUD transaction, which is typically a small subset of all available *auto indexes*.

Parallelism

The impact of parallel indexing, that is creating multiple indexes concurrently on the same collection, is analyzed for some XMark-based benchmarks. In Figure 7.8, the workload runtimes (left) and index characteristics (right) for different degrees of parallelization are shown. In a *single indexing* setup, indexes are materialized sequentially, one at a time. For comparison, we increased the parallelism, for which the results *parallel indexing* are shown, too.

The margins between the workload runtimes (left figure) are fairly small, but, nevertheless, they also indicate that parallel indexing may have a positive influence on workload runtimes. As expected, the total growth of the index space (right figure) is independent of the degree of parallelism. Eventually, at least for our tested single-user scenarios, parallel indexing does not have any negative impact at all.

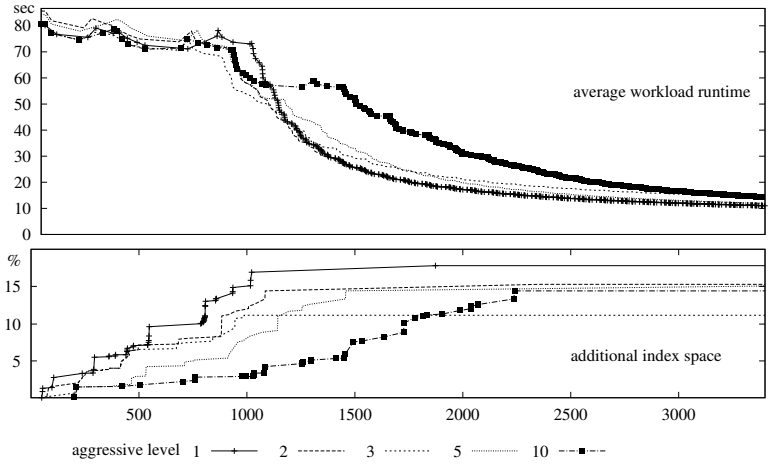


Figure 7.9: Varying aggressiveness of index building

Aggressiveness of Index Building

Important for the index tuning quality and its reaction time is the cost-benefit amortization. We explored various levels of "aggressiveness". Figure 7.9 shows declining workload runtimes (top) and the corresponding index space consumptions (bottom) for various thresholds controlling index materialization aggressiveness. Especially, the index containment problem is automatically addressed by analyzing several queries/workloads to exploit synergy effects of index candidates. This is clearly visible for aggressive levels > 3. In terms of space overhead, moderate policies are the most efficient ones. However, nearly all setups increase their throughput in the same order. Thus, it is possible to adjust the threshold to specify whether fast adaptation to changing workloads or conservative space occupation is preferred.

7.5 Conclusions

This chapter presented our AI framework that is based on the cycle of *monitoring*, *decision*, and *action*. Our framework employs several ways of generating index candidates and is based on a lightweight statistics accounting. These statistics are required for the concept of virtual indexes and for using the query optimizer for index selection. Emphasis was put on AI overhead, index space restrictions as well as changing workloads and updates.

Our results proof that index self-tuning is a complex task that – supported by our AI framework – easily exceeds manual tuning efforts. Note, manually tuning the index configuration typically requires more time to react and the user is confronted with a huge and confusing number of index options.

Although the framework is tightly coupled with XTC, the general approach can easily be adopted to other XDBMSs, as long as comparable index types, statistics, and a cost-based query processor are available.

Chapter 8

Interplay of Self-Tuning Components

In the course of this thesis, we have shown that our (self-)tuning mechanisms are effective and improve the performance of XML processing. So far, the self-tuning approaches were presented component by component. The evaluation of individual components and self-tuning techniques is crucial to systematically draw meaningful conclusions from their effects [SBH09], but for real-world scenarios, their interplay is important as well. Due to component dependencies and contradictory tuning goals, self-tuning techniques may interfere with each other. But also static tuning options can interfere with self-tuning techniques, when they target the same resources or tuning goals. Such effects are hard to track and, in the worst case, may lead to configuration thrashing, i.e., the tuning mechanisms alternately change the configuration according to their individual goal.

This chapter looks at the impact on query processing and DB configuration when major tuning mechanisms are enabled at the same time. Therefore, we analyze the interplay of autonomous buffer and index tuning in case of changing workloads. Based on this analysis, we explore the interplay of various combinations of (self-)tuning techniques including optional content compression. Thereby, we concentrate on query processing performance and resource usage.

8.1 Workload and Environment

For this benchmark series, we use again hardware configuration 3 from Section 5.10 and the XMark dataset to populate multiple databases of 12MB size and 112MB size. The workload consists of complex XMark queries, a set of newly created update queries tailored to the XMark documents, and SAX scans evaluating simple XPath expressions. Except for the SAX scans, all queries may be supported with various indexes. Each container file consists of one DB instance and is assigned to an individual buffer. The total size limit of all buffers is aligned to the size of a DB instance. That means, the data volume that can be kept in main memory is slightly higher than that of a single DB instance. There exist no predefined indexes, if not stated otherwise.

The kernel cache is bypassed using direct IO to emphasize the self-tuning effects and to avoid that caching techniques of the operating system might mislead the result interpretation. For workloads on larger DB instances, which exceed main memory capacities, we would not require this restriction.

8.2 Interplay of Buffer and Index Self-Tuning

The results of the benchmarks are shown in Figure 8.1 for the databases of 12MB size and in Figure 8.2 for the databases of 112MB size, respectively. For each benchmark, the query workload, shown in the upper part, is identical. There are different kinds of workload patterns, SAX scans (dark gray bars), XMark queries (light gray bars), and update queries (black bars). The bars also indicate which of the three databases are utilized. Below the workload markers, white boxes indicate index use, distinguished by read and write access, where write access means index building or index maintenance. For your convenience, we divided the workload into 13 sections reflecting workload shifts and also marked their boundaries in the measurements of *average query time*, *index space*, and *buffer size*. This alignment enables us to interpret the results regarding buffer and index self-tuning. For instance, during the processing of section 3 (SAX load in Figure 8.1), the average query processing times drop from $\sim$1.3 seconds to $\sim$0.3 seconds, because the buffer configuration changes. The size of buffer 1 decreased and the size of buffer 2 is increased. During this workload period, no indexes were accessed and the index configuration is not changed.

In the following, we look at the individual workloads and shifts presented in Figure 8.1. For your convenience, we do not explicitly distinguish between a database instance and its corresponding buffer:

1. The initial buffer configuration is quickly adjusted to the SAX load on database 1. As soon as the entire database fits into the buffer, the query times drop considerably.

2. The shift towards an XQuery workload leads to an increase in query processing times. This increase is immediately compensated by index materializations. The buffer configuration does not need to be changed.

3. The target database changes and queries are processed using buffer 2. The SAX load causes a buffer reconfiguration in favor of buffer 2. Subsequently, the query times drop again. Note, although this workload is similar to workload 1, but using a different database, the buffer size changes are even more drastically.

4. The set of XMark queries on database 2 is comparable to workload 2, except for the target database. Here, no initial peak of query times is observed, because the size of buffer 2 is still large enough and necessary indexes are materialized in container 0. Due to the following heavy use of indexes, the size of buffer 0 is slightly increased. Because document scans are not necessary anymore, the corresponding downsize of buffer 2 has no negative impact.

5. The SAX load from workload 3 is applied again. Due to the former index use during workload 4 and the downsize of buffer 2, optimal speed-up is impeded. It takes some time to reverse the most recent tuning decision and to bring back the prior buffer configuration. Note, this behavior is accepted in favor of non-thrashing configurations. The following query speed-up meets the expectations.

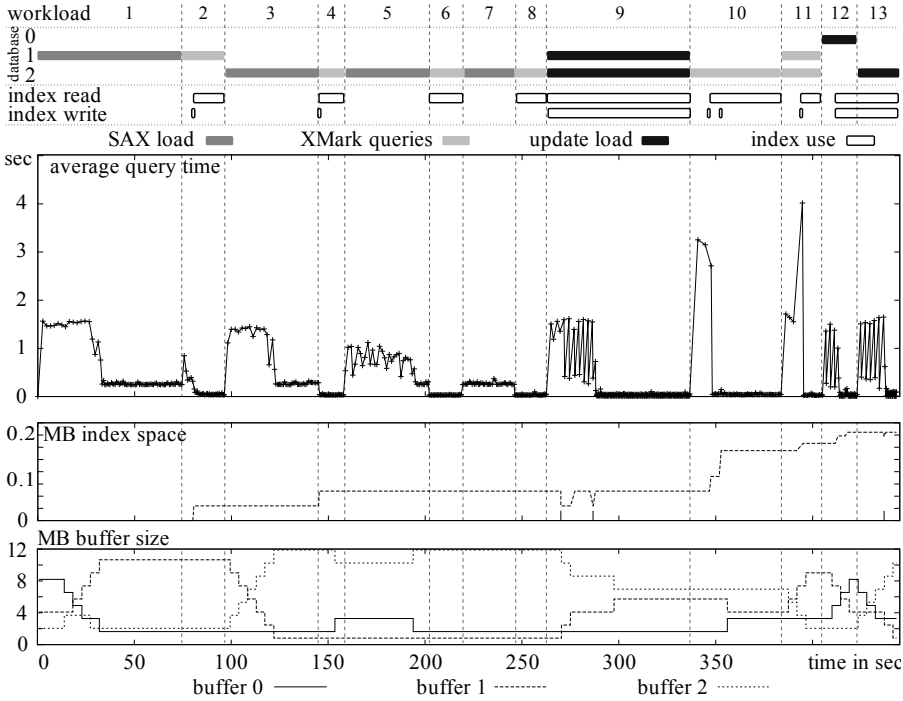


Figure 8.1: Self-tuning effects for 12MB XMark databases and shifting workloads

6.–8. Again, sections of XMark queries and SAX loads alternate on database 2. But now, query times do not increase, because the buffer configuration is settled to serve index requests with a small buffer 0 and scan requests by the large buffer 2. Note, no additional indexes are created, which means that the index configuration already achieved its optimum during the very first XMark workload.

9. The workload now consists of update queries on databases 1 and 2, which affects the buffer configuration and the index setup. Some indexes are dropped due to their maintenance costs and some new indexes are created to support the query part of the XQuery Update procedures. After the index configuration is changed and the buffers 1 and 2 are resized, query times drop drastically. Note, in the middle of the reconfiguration phase, queries on database 2 are processed faster compared to database 1. Due to the initial conditions for this workload shift, i.e., the large buffer 2 and the existing indexes on database 2, different query speed-ups can be observed. Finally, query processing on both databases is almost only performed by using indexes. The following buffer resiz-

ings show no further effect, but are aligned to the buffer's current load.

10. A mix of XMark queries on database 2 requires new indexes to achieve the expected query speed-up. The query performance slightly improves again after buffer 0 (containing the indexes) is increased again and the underutilized buffer 1 is decreased. Note, again there is no immediate increase of the size of buffer 2 observable, because this would cause configuration thrashing. The parallel index adjustments make those changes needless anyway.

11. The XMark query load is evenly spread to databases 1 and 2. Because database 1 has no appropriate indexes, document scans are required until new indexes are created. This also requires that its buffer size grows larger than the buffer size of database 2.

12. A major workload shift towards update queries on database 0 happens. It has no index support yet, but new indexes are created and the size of buffer 0 is increased immediately, too. The final query performance settles at the expected level.

13. The update load moves to database 2 causing high maintenance costs for existing indexes. The index configuration and the buffer configuration are adjusted to significantly improve the query performance.

To show that the behavior of our self-tuning techniques is independent of the database size, we processed the same sequence of workloads for a setup with ten-times larger databases and a ten-times larger buffer limit. The results are shown in Figure 8.2. Except for workload 9, only marginal differences are observed. During the update queries on two large databases in workload 9, the buffer tuning seems to overreact twice before it is settled. Because index materializations take now more time compared to the small database scenario, buffer tuning actions take place while indexes are built. In this phase, index builders on database 1 cause document scans, which drive the rapid buffer increase. Later, the indexes on database 2 were dropped followed by new index creations, now causing on buffer 2 a high scan load. However, the ideal buffer configuration adjustment is also recognized as thrashing, which is why it takes a short time to increase the size of buffer 2 again. Finally, the index configuration is stabilized and the buffer configuration is settled, too.

In both scenarios, the interplay of index tuning and buffer tuning works very well to improve the query processing performance. Index tuning is purely query-driven and buffer tuning follows the IO load, even for short-term index use or creation. The remaining question is, how does further IO-related tuning, like content compression, affects this interplay. Inter alia, this will be answered by our next benchmark series.

8.3 Varying Combinations of (Self-)Tuning Features

The last benchmarks have proven that buffer self-tuning and auto indexing play well together. However, to prove that they really benefit from each other, we need to evaluate their individual application for the same benchmark series. Moreover, additional tuning features may be added, while the performance is expected to increase again. Such an additional feature like content

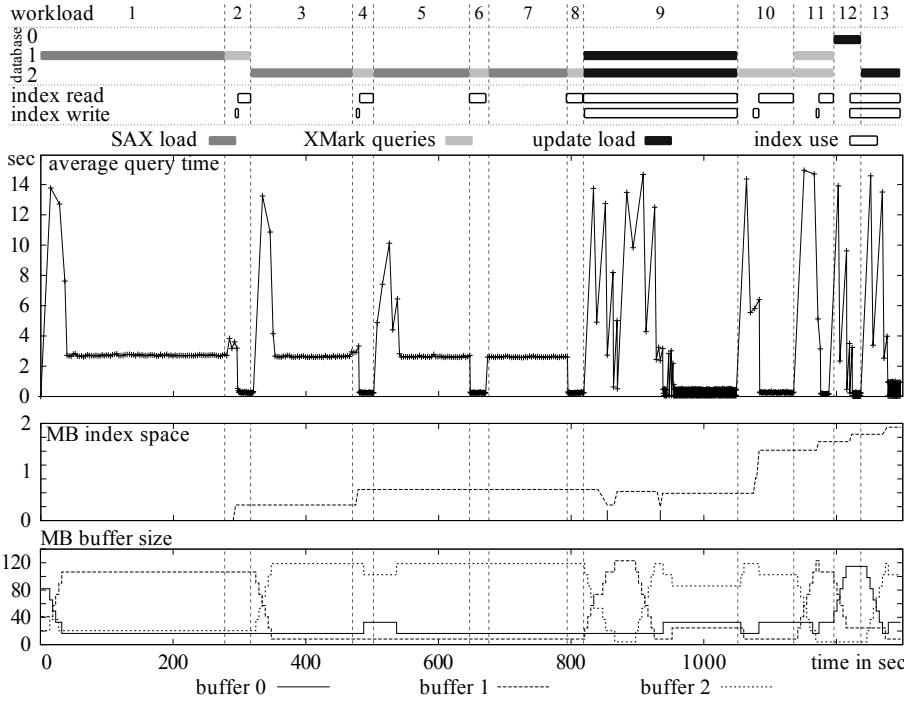
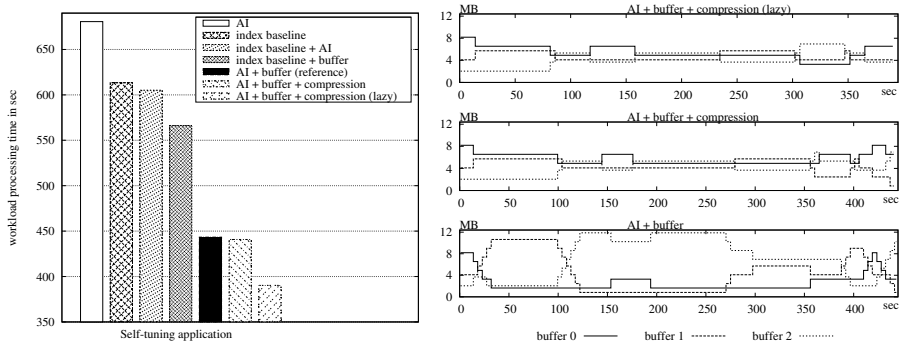


Figure 8.2: Self-tuning effects for 112MB XMark databases and shifting workloads

compression focuses on the same tuning goal (i.e., IO reduction) as buffer tuning. In contrast, auto indexing increased the storage space consumption, while content compression focuses on its reduction, making scan-based access more attractive again, which, in turn, affects buffer tuning. The results in Figure 8.3 compare different (step-wise) applications of our self-tuning features. Obviously, the more features are applied the faster is the entire benchmark processed.

For comparison, we added results for a run without self-tuning. However, this run, denoted as *index baseline*, includes a user-defined element index. Otherwise, the processing times would be out of scale. The individual usage of buffer self-tuning and auto indexing are denoted as *AI* and *index baseline + buffer*, respectively. Because buffer self-tuning without indexes results in extremely long runtimes, we added the element index as baseline, too. The reference results are identical to the results of our last section. Moreover, we have two runs where content compression was enabled, denoted as *AI + buffer + compression* and *AI + buffer + compression (lazy)*, respectively. Figure 8.3(a) shows the absolute runtimes for the benchmark runs using 12MB databases, whereas Figure 8.3(b) shows buffer characteristics for the last three runs.



(a) Self-tuning combinations

(b) Buffer tuning with/without content compression

Figure 8.3: Comparison of self-tuning application (12MB XMark databases)

Simply adding content compression (i.e., run *AI + buffer + compression*), for such a mixed benchmark of SAX, XQuery, and updates workload, only leads to a marginal speed-up of approximately 1%, compared to the reference result. Because the overhead for encoding and decoding almost outweigh the storage space savings, we enabled our lightweight PCR filter technique[1]. SAX workload noticeably benefits from the "lazy" decoding of content values, but even XQuery processing is improved, which leads to the clear performance speed-up.

The buffer tuning results, in Figure 8.3(b), reveal the effects due to content compression. For your convenience, the reference result from the last section is depicted together with the results using content compression. As expected, the amplitude of buffer sizes is smaller when content compression is enabled. The steps of buffer resizing are mostly larger and often only a single tuning interval is required to adjust the buffer configuration. This is possible, because the buffer tuning simulation fully covers important buffer characteristics and precisely predicts its performance.

As index characteristics are almost the same for runs having AI enabled, we do not present additional charts for them. Note, *baseline* runs require clearly more storage space compared to pure *AI* runs. As baseline, we chose the element index, because most update queries affect content values, which minimizes maintenance costs for the baseline, while delivering almost equal query support, compared to tailored CAS indexes.

The results show that individual self-tuning application is worse compared to their combined use. Although all techniques focus on the same goal, namely IO reduction, their measurements and resource usages are different. Even resource sharing (i.e., CPU and main memory) does not impede (self-)tuning.

[1] Instead of fully decoding binary stored content and attribute nodes during their first access, the PCR filter is applied first to avoid unnecessary decoding of compressed content values. In case of value access, for instance to evaluate a comparison predicate, the value is *lazily* decoded.

8.4 Analysis of Self-Tuning Effects

In our complex workload scenarios, certain effects can be observed. As the classification from [CW05] in Section 1.2 already indicated, cache and memory tuning may happen in near-real time, whereas physical design tuning happens less frequently. This can be observed in our scenarios, too. The buffer tuning is fairly cheap and happens frequently, but the costly index tuning is less aggressive. Moreover, index deletions are rare events, which need a clear justification first.

Another interesting effect is that a sequence of index uses leads to the following: For XQuery statements containing value predicates, CAS indexes are build. The subsequent update statements cause high maintenance costs and lead to the deletion of these CAS indexes, although the query parts of the update statements exploit them, too. As a reaction, the system automatically creates the second best index instead, a path index that supports the query part but does not require maintenance for the update part. Although we do not consider update queries as a single query instance, tuning does this automatically due to our implementation as a procedure. This is only possible, because the n best plans are considered for index self-tuning.

Due to data placement decisions, buffer tuning needs to quickly switch between buffers for index use or document use. This happens whenever queries on database 1 or 2 use indexes from container 0 or cause their creation in container 0. Buffer tuning then increases the buffer for container 0 to improve the performance of index building and use. Finally, as soon as index-only query processing becomes possible, buffer tuning adjusts the sizes of all buffers again to accommodate all index IO in the buffers. When document scans or frequent document access are still required, buffer tuning reduces the size of buffer 0 once the indexes are built.

Resource usage is another interesting aspect. Our second benchmark series using content compression saved storage space (and IO), but the CPU load was slightly higher compared to the runs without content compression. However, having more concurrent transactions may change this pattern. The reduced buffer size demands (for a single data container) allowed larger index sorts to be performed in memory, i.e., in our scenario short-term memory consumers do indirectly benefit from content compression. As index sorts are rare events, the memory limit for buffer pools may be lowered while other critical areas (e.g., lock space, connection pool) are increased. Note, indexes are not carrying compressed content values, because usually the content value is the key and needs to be ordered and comparable. Thus, space savings only originate from the database files. The specification for an index space limit for AI needs to observe this, especially when this limit depends on the database size.

8.5 Conclusions

To get a more complete picture of database self-tuning, the effects of multiple tuning decisions, made at the same time, have shown that index tuning and buffer tuning play well together and do not lead to configuration thrashing. We have also shown that the tuning capabilities are independent of the database size, even in the case of several workload shifts. Furthermore, only a couple of queries and tuning periods are required to quickly achieve a noticeable speed-up of query processing. Adding more (self-)tuning such as content compression or lazy evaluation

led to improved query processing times. Although the usage of resources and tuning decisions are different, the performance was always improved. These results give evidence that individual tuning decisions of distinct tuning components can be beneficial for each other, although no explicit link between them exists.

The obvious next step for self-tuning, we derived out of our experiments, is to make these links explicit and exploit them. In the outlook chapter, we will discuss some ideas how to achieve this.

Chapter 9

Conclusions and Outlook

This chapter concludes our work by giving a short summary of the problems addressed and the techniques we developed to solve them. Finally, we give some insight into potential future research directions that were either untouched by this work or arose out of it.

9.1 Conclusions

This work presented self-tuning concepts for native XDBMSs. We introduced the research challenges in the area of database self-tuning, presented current achievements for self-tuning in the relational world, and took the next step towards tailored self-tuning concepts for XDBMSs.

Various existing concepts for self-tuning have been considered when we created our novel approaches for XML-related self-tuning. Specifics of native XDBMSs such as the query languages, the storage model, and the APIs have been considered, when we developed tailored (self-)tuning mechanisms for the entire XDBMS stack. Fortunately, having our own prototype XTC, it was possible to implement and evaluate all of the techniques, developed in the course of this thesis, into a full-fledged XDBMS, while taking the MAPE paradigm for online self-tuning as template.

By following the layered architecture of a DBMS, from the bottom layers of XML storage and buffer management to the top layer of query processing, we identified bottlenecks and (self-)tuning options in almost all layers.

First, we examined the buffer management, which is actually not XML-specific, but requires efficient fine-tuning to cope with (frequently) changing load. We have shown that state-of-the-art techniques to predict buffer performance are not sufficient to correctly decide buffer resizing. Our hotset simulation is a lightweight extension, applicable for many existing replacement algorithms, that considerably improves the forecast accuracy, especially in case of discontinuous buffer scaling. Together with the dynamic management of buffer pools, fast and cheap adjustments of the buffer configuration become possible. Our benchmarks have proven that the overhead for buffer self-tuning always pays off. However, complex replacement algorithms such as LRU-k are unfavorable for the hotset simulation, although the achieved forecast accuracy is very good. Furthermore, our results have shown that the combination of multiple buffer pools, with individual replacement decisions such as the existing SBPX approach, may result in low forecast accuracy for buffer upsizing.

When it comes to XML storage there are several performance-critical aspects like data transfer to the server, data transformation into an internal representation, and access primitives. We

developed specific XML storage concepts that make use of structural and content compression techniques. Based on our path synopsis, we designed a novel XML mapping approach – elementless storage. The path synopsis is also used to apply XML similarity measures during data placement decisions as well as to support indexing, statistics management, and query processing. Eventually, we have demonstrated that our native XML storage concepts perfectly serve for fast navigational access and scan-based access. The use of B-tree, vocabulary, and path synopsis makes XML storage very efficient in terms of space consumption and concurrent access. Furthermore, we have shown that the efficiency of content compression depends on the compressor and the XML data. Indeed, we developed character-based and wordbook-based compressors, because existing XML content compressors do not support query processing through fine-grained access and manipulations. Hence, content compression is regarded as an optional storage feature. Although not all configuration parameters of a native XDBMS store can be tuned autonomously, we have shown that, at least for some of them, self-tuning delivers convincing results.

Query processing performance is obviously as critical as storage in XDBMSs. Typically, a cost-based query optimizer is used to find the cheapest QEP. Especially, document access can drastically be sped up by having adequate indexes instead. To deliver versatile query processing support, we developed a unique setup of index options for XDBMSs. Most of the index flexibility is achieved through the use of our elementless storage model. However, this variety of index use also increases the already-hard-to-solve ISP.

Indexing has to observe many aspects. Besides query support, indexes require maintenance in case of modifications causing additional overhead to query processing. Often, a system has space limitations, which may prohibit simply creating all useful indexes. The cost and benefit of indexes need to be permanently tracked. Our so-called AI framework is a fully server-integrated index tuning approach for native XDBMSs. We have shown how to exploit query plans and the query optimizer to account usage, benefit, and cost statistics for index candidates as well as existing indexes. Using a lightweight "what-if" mode of query processing, we generated and used virtual indexes during the search for the best QEP. The AI framework observes space restrictions and manages the index configuration by creating and dropping indexes automatically. Further XML-specific aspects such as index containment, overlapping, and substitution have been addressed in our work. Consequently, the increased complexity of XML index tuning can only be handled by autonomous tool support such as our AI framework.

The effectiveness of individual self-tuning techniques is successfully demonstrated with benchmarks and tailored workloads. However, important for their usability within a DBMS is their interplay. In a special scenario, we analyzed the effects of applying our index and buffer self-tuning techniques as well as additional content compression at the same time. The results have successfully proven that both of them – improvement of query performance and fast convergence of a stable configuration – were achieved. The self-tuning mechanisms also coped with multiple workload shifts and varying database sizes.

With the help of extensive experiments, we have shown that self-tuning of an XDBMS is possible, but often requires XML-specific consideration. The focus of self-tuning techniques is to deliver fast and cheap mechanisms that are capable to autonomously improve DBMS performance. That means, the DB administrator is relieved of the burden doing configuration

changes online for a confusing large number of options. It is still questionable, whether even a skilled administrator is capable of performing similar tuning results with the same reaction time and quality.

The research of self-tuning databases is still an active and important topic. Of course, this thesis could not cover all aspects of self-tuning, let alone XDBMS self-tuning. Therefore, we provide the following ideas for future research in this field.

9.2 Outlook

In this work, we examined certain DBMS areas related to native XML processing. We identified self-tuning options and presented solutions for them. But there is always room for further research. Aside from the specific open problems mentioned in the individual chapters, we briefly describe those ideas we identified as good candidates for future research.

9.2.1 Determine Simulation Parameters

For now, some simulation parameters of our self-tuning algorithms are pre-specified. For instance, the AI framework uses a fixed number of query plans to inspect or buffer tuning needs initial sizes for upsize and downsize simulations. Some thresholds of our simulation-based algorithms are given by the user through a configuration file.

To achieve sophisticated self-tuning, it is necessary to avoid as many pre-specified parameters as possible. The open questions are: (1) How to learn them faster? (2) How to determine good starting values for them? and (3) Can those parameters be removed?

Taking a step back and reconsidering the simulation approach leads to the following idea. Actually, the "iterative" style of comparing simulated configurations with the current one does not only require multiple simulation rounds, but may also end up in a local optimum. This, however, depends on the simulation parameters. The research challenge is to find the ideal configuration with the minimal search effort, i.e., number of simulations. Let us give some more detailed examples:

Ideal Buffer Pool Configuration

An ideal memory distribution for buffer pools may be possible after a single simulation or analysis interval. Even the performance of unknown (i.e., not yet simulated) buffer sizes may be evaluated (i.e., estimated or calculated), which is useful to determine a fine-grained buffer scaling function. Having such a function for each buffer pool requires the solution of an (optional linearized) system of equations to calculate the ideal memory distribution.

Ideal Parameters for Index Self-Tuning

For many parameters (or thresholds) used by the AI framework, the question is whether changes of them improve quality or decrease overhead. Usually, both goals are contradictory. An ideal "balance" of those parameters is desirable, which can be regarded as pareto optimum (cf. Section 3.2.6). For instance, the number of query plans that should be investigated for a single

query can be raised, but, at the same time, the number of index candidates may be unnecessarily increased. Moreover, we do not exploit plan differences in terms of cost estimation and index use. It may be beneficial to specify the number of plans to be inspected dependent on their differences. Parameters controlling the aggressiveness of AI such as the cost-benefit ratio of virtual indexes or the query runtime threshold should be aligned to each other.

Ideal Cost Parameters

Our current approach of cost parameter adaptation is straightforward. We take the difference of expected costs and measured costs to adapt the cost models. The next step is to take workload shifts, peak loads, and concurrency into account. This may help to prevent that outliers (e.g., peak situation) lead to disproportional parameter adjustments. The query planning cost model, index cost model for index use, maintenance, and materialization, as well as the buffer IO cost model may deliver better estimation quality when tuned for stability – at least in the average case.

9.2.2 Index Self-Tuning

The field of index self-tuning still offers a lot of challenges, especially in the world of XML indexing. In contrast, our approach of benefit/usage accounting may be brought to the relational world, too. The following two aspects are interesting research challenges.

Index Dependencies

There is some work for relational index tuning that takes index dependencies into account. The question is whether dependencies in XML indexing are different due to XML index characteristics such as overlapping, containment, and typing. In the relational world, the choice between index application and table scan is often “boolean”. We have shown that XML indexing often provides various indexing alternatives for the same query, which makes dependency modeling even harder. A first step we made to tackle this problem is the option to inspect multiple query plans instead of only the cheapest one.

Incremental Indexing

Today, research on incremental indexing, often called database cracking or adaptive merging [IMKG11], incrementally refines indexes. That is, only database partitions addressed by queries were actually indexed. Furthermore, indexing performance may benefit from delayed sorting and merging of those partitions. Anyway, the XML world already has similar approaches for structural indexes such as APEX or D(k)-Index. The open issues are to investigate content carrying indexes and to analyze if partitioning of XML documents is possible. Note, our approach to optimize candidate and virtual indexes is doing a kind of incremental indexing based on PCR sets.

9.2.3 Modern Hardware

To take full advantage of new hardware developments, self-tuning has not only to consider the "size" (i.e., number of CPUs, size of main memory and storage), but also the structure of the hardware architecture. For instance, communication is an important aspect, even between certain CPU cores and caches. Tailored algorithms for computational tasks become more and more available. Thus, self-tuning may select and deploy algorithms dependent on the load and available hardware. The trend for new volatile and non-volatile storage technologies should be exploited by self-tuning. On the one side, the bandwidth and IO rates within the memory hierarchy are permanently changing. On the other side, affordable capacities, volatility aspects, and unbalanced IO costs increase the complexity finding a suitable memory setup. Especially flash memory is a promising technology for database use, as we have shown in [SHKR08, SOH09, HSOB09]. But also multi-(core/node), cloud-based, and main-memory architectures are hot hardware trends that need to be covered by the self-tuning community. Besides individual query or workload tuning, management of those environments benefits from self-tuning, too.

9.2.4 Next Generation of Tuning Goals

Database tuning mainly focuses on transaction throughput and query response times. Driven by another major trend – green computing –, energy consumption is becoming a hot topic. From hardware selection (e.g., solid-state drives instead of hard disks) to algorithmic adjustments, self-tuning may become energy-aware. This may require to extend the query optimizer and its cost model to estimate energy consumption for a certain QEP. Through the use of energy-efficient hardware components and algorithms, the DBMS performance may slow down, but, in any case the overall efficiency (i.e., transactions per joule) may increase. Eventually, usability is important so that the user should only prioritize certain tuning goals and verifying their fulfillment.

9.2.5 Evaluation of Self-tuning

Several aspects of performance evaluation are challenging, especially, when self-tuning techniques depend on varying conditions such as load or cost functions. But also different evaluation platforms (i.e., DBMSs) prevent direct comparisons. Often the exact repeatability of experiments is hard and slight variations have to be accepted. For instance, asynchronous tuning jobs such as index tuning have a considerable impact on parallel running transactions. However, the research community mostly confines performance evaluations to single-user scenarios. This ensures to get repeatable and explainable results. Yet, DBMSs are typically deployed to multiuser environments containing concurrent transactions. Here, self-tuning techniques increase the problem of performance analysis due to the problematic repeatability.

To find a suitable way of evaluating self-tuning techniques, a kind of self-tuning benchmark, explicitly modeling workload shifts and changing resource capacities, is an interesting mission. Such a benchmark may allow to compare the performance of different self-tuning techniques,

even if they are developed for different DBMSs. Interesting aspects of such a benchmark may be how tuning goals are defined or stability requirements are fulfilled.

9.2.6 Towards a System Model

As soon as all tunable components of a DBMS have been made self-tunable, a holistic self-tuning approach may become viable. Such an approach should be based on a *system model* describing physical resources, monitoring spots and tuning knobs, component dependencies, and their correlations. The model is used to estimate (or simulate) state changes, parameter changes, and tuning impact.

In [WMHZ02], the authors recommend a model based on economic principles to allocate resources in DBMSs. We adapted this idea and recommend to form *functional groups* [Sch09]. Functional groups help to create a hierarchical system model, where well-known tuning techniques can be locally applied within a functional group, before their individual state (using common metrics) is propagated to the system-wide and coarser component model. The resulting component model should also serve as visualization model, indicating data hot spots, data flow, bottlenecks, and resource usage.

As our experiments in Chapter 8 advocate, making the links of self-tuning techniques explicit may lead to a transparent system model, which allows for a better estimation and evaluation of self-tuning. For instance, resource usage should be comparable between self-tuning components. Thereby, expected gain-cost ratios (measured in time and resource use) are comparable, too. Because resources are limited, self-tuning may concentrate on the most promising areas, i.e., delivering the best gain-cost ratios.

Finally, the consolidation of individual self-tuning components may result in a “100 % view” of DBMS self-tuning enabling the application of the same self-tuning techniques on a DBMS-wide level – self-tune the self-tuner.

Appendix A

Architectures of Native XDBMSs

Here, we give a brief overview of existing XDBMSs and their architectures. We want to show that our tuning techniques are applicable to a wide range of systems, because almost all of them use a similar layered or component architecture as our XTC system. Therefore, the basic self-tuning components developed in this work can be integrated in these systems, too.

Lore

One of the first and perhaps the most influential work regarding native XDBMSs research is Lore, the *L*ightweight *O*bject *RE*pository [MAG+97, QWG+96].

Lore's architecture distinguishes between two major layers, namely the *data engine* for storage and physical operators and query processing within the *query compilation* pipeline, as shown in Figure A.1. The object manager is an OEM store[1] for practical all kind of (Web) objects. It is also responsible for caching. Lore's *IndexMgr* supports various kinds of indexing mechanisms for content and structure. Besides value indexes (Vindex), path indexes, and link indexes (Lindex for parent pointers), it supports the probably most cited index structure for XML – the *DataGuide* [GW97].

Query processing in Lore is similar to the compilation pipeline in XTC. Lore was the first XDBMS applying basic optimization rules for query plans, although its language support is restricted to Lorel – its own query language. However, updates to objects have been possible from the beginning. Internal XML node labeling is simply based on integer values.

Tamino

One of the first commercial systems available was Tamino – *Transaction Architecture for the Management of Internet Objects* [Sch01]. In Figure A.2, its most current architecture is shown.

The highlights of Tamino are its high-performance XML parser, native XML data store, server extensions (*X-Tension*) for custom functionality, indexing, and a schema repository (*data map*). Tamino supports many communication features, for instance, to map data to RDBMSs and to call Web services or business level services. The central component *X-Machine* is responsible for XML parsing, XML object processing, query interpretation, and object composition, i.e., XML document storage, retrieval, and querying. Web server integration is employed via *X-Port* and application programmers can use the X-Tension feature, essentially Tamino's

[1] OEM - *O*bject *E*xchange *M*odel is designed for semi-structured data representing a labeled directed graph. No fixed schema is required making the data (structure) self-describing based on its edges.

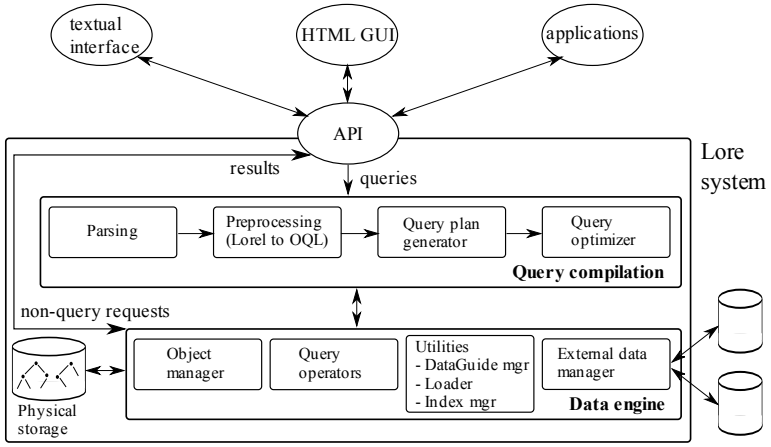


Figure A.1: Lore architecture overview [MAG+97]

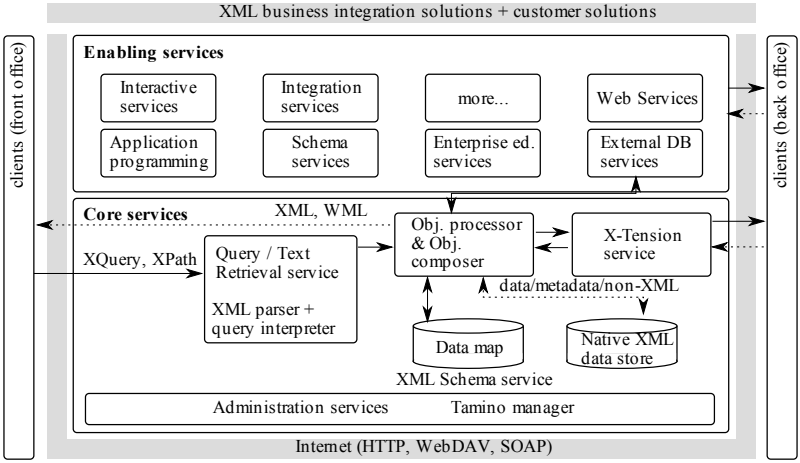


Figure A.2: Tamino architecture overview (source: `www.softwareag.com`)

variant of a stored-procedure language, to implement application functionality within the server by working directly on its internal data model.

Metadata, DTDs, and XML transformation guidelines are stored within the data map, i.e., Tamino's catalog. For query processing, structural and value indexes can be defined on elements and attributes.

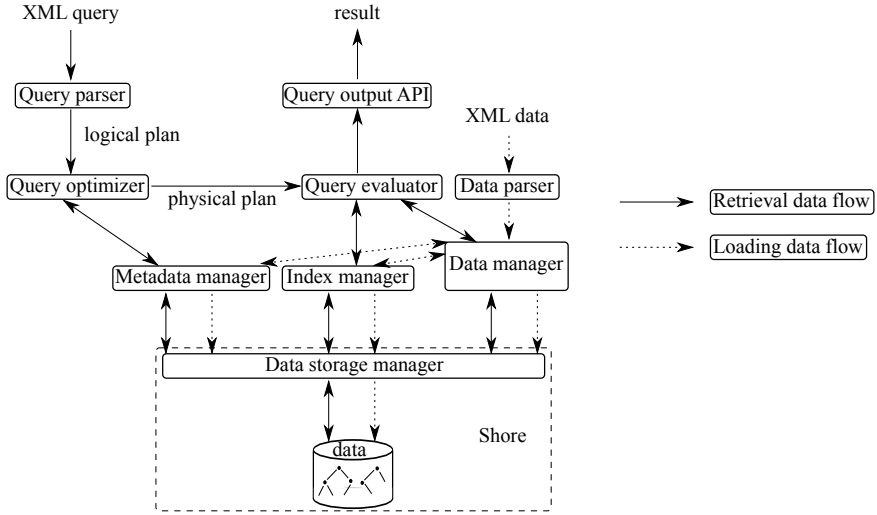


Figure A.3: TIMBER architecture overview [JAKC+02]

TIMBER

Developed at the University of Michigan, Timber was influenced by the Niagara system [JAKC+02] and employs the TAX algebra [JLST02] to process bulks of hierarchical data in a set-at-a-time manner. It furthermore has a node-oriented storage and makes extensive use of structural joins for pattern matching. For simplicity, Timber physically represents attribute nodes and text nodes as child nodes of their parent element instead of considering their different semantics.

Storage is based on *Shore*[2] [CDF+94], which is responsible for disk management, caching, and concurrency control. In Figure A.3, Timber's architecture and data flow during query processing is shown. All managers for XML data (*Data Manager*), index data (*Index Manager*), and metadata (*Metadata Manager*) are build on Shore. As query languages, Timber supports XQuery and Xupdate[3]. A notable feature of Timber is its labeling scheme using double values to encode a node's pre-order number, post-order number, and level, i.e., so-called (start, end, level) labels. To reduce costly relabeling, Timber leaves gaps in the labeling scheme.

Natix

The layered architecture of Natix [FHK+02], shown in Figure A.4, is similar to that of XTC. The storage layer contains a buffer manager that handles pages, which are grouped in segments. These segments are then stored on disk. A new aspect in Natix is the representation of XML in

[2] SHORE – Scalable Heterogeneous Object REpository – is a persistent object system under development at the University of Wisconsin

[3] `http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html` (working draft since 2000)

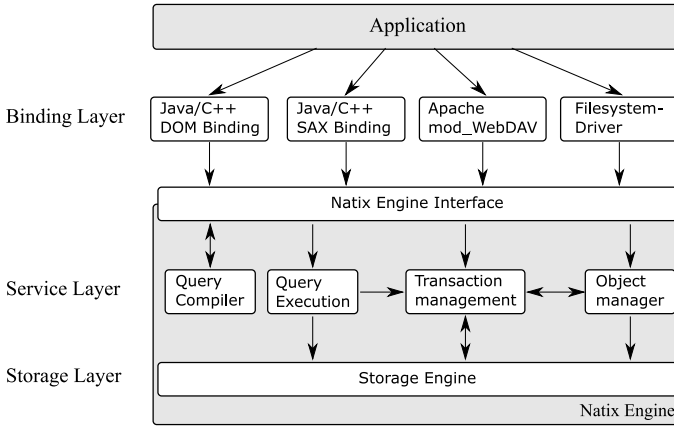


Figure A.4: Natix architecture overview [FHK+02]

so-called *segments*. Here, entire subtrees are stored together according to a given *split matrix*. The split matrix is used to express application or user preferences regarding the clustering of nodes. It specifies whether a node (type) is stored *standalone*, *clustered*, or within the *same* record as its parent. So-called *proxy nodes* are used to reconnect the subtrees forming the full document again.

Natix includes full-text index support and a new index structure called eXtended Access Support Relation (XASR), which preserves some XPath relationships. Natix uses a depth-first traversal to label XML nodes. Similar to the approach used by Timber, gaps are introduced to better cope with updates.

Galax

The Galax system has its focus on query processing and does not provide index support or any transactional features – it is more an XQuery compiler and processor than a full-fledged XDBMS [FSC+03]. In research projects, Galax is often used for comparative evaluation due to its focus on query processing. As the architecture shown in Figure A.5 illustrates, the makers of Galax emphasize on the XML processing pipeline. Thus, they used Objective Caml for implementing Galax, because "its algebraic types and higher-order functions simplify the symbolic manipulation that is central to the query transformation, analysis, and optimization".

Although Galax operates navigation-based on the input documents, it is capable of handling and validating XML Schema. Similar to XTC, queries are parsed into an AST before being normalized into XQuery Core expressions.

Due to Galax' main-memory representations of documents, optimizations are necessary to process GB-sized documents as well. Basically, large documents can be *projected* into smaller ones, so that Galax evaluates query-relevant paths beforehand and assembles a (smaller) pro-

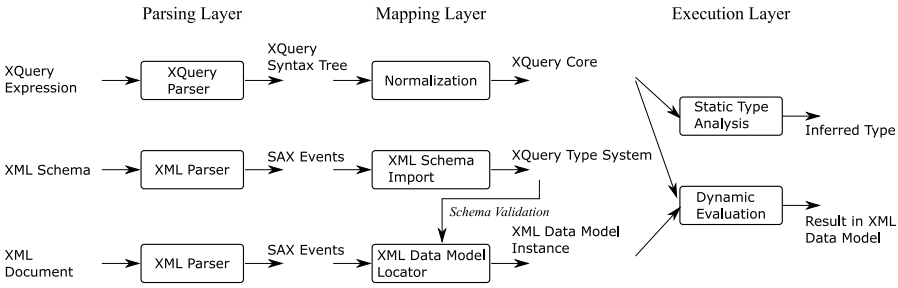


Figure A.5: Architecture of Galax [FSC+03]

jected document.

Typing rules, query transformation, and evaluation patterns are implemented literally making their correlations from definition to implementation easier.

OrientX

The OrientX system uses and adapted many ideas from existing XDBMSs such as Timber, Natix, and Lore [XXM+06]. Its strength is the efficient schema-based storage, either the Clustering Element-Based (CEB) or the Clustering Subtree-Based mapping. However, schema-independent storage is supported as well, either Lore-like (element-based) or Natix-like (subtree-based). In Figure A.6, the architecture is shown, which is following the five-layer reference design.

At the bottom, the *File Manager* operates on files of fixed sizes such as 8 MB. On top, the *Storage Manager* maps the physical fixed-size pages of 8 KB to logical pages. Two buffer managers are employed in OrientX, one for pages using LRU replacement and one for records caching tree structures. A uniform access to data manager, schema manager, and index manager is realized by the *Access Manager* hiding underlying details.

The strong schema-dependency of OrientX controls (or sometimes also restricts) not only data and queries but also updates, indexing, type checking, and user access. Physical storage uses variable-length records carrying a unique object id and pointers to its parent record or sibling records. Index capabilities are provided by a schema-guided path index named SUPEX. Moreover, a histogram-based query optimizer called HISTOPER is developed within OrientX.

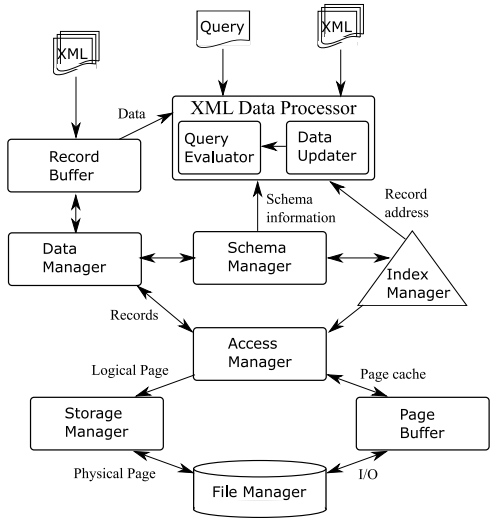


Figure A.6: OrientX architecture [XXM+06]

Appendix B

Storage

B.1 Storage Gains for Elementless

These results extend the evaluation of our storage mappings from Section 5.10.2. In Figure B.1, we show the gains of *elementless* storage for a typical reference set of large and individual XML documents. In almost all cases, storage and reconstruction of documents is faster using the elementless storage mapping. Only exceptional cases, such as the highly complex structure of the treebank document, favor the full storage mapping for large individual documents.

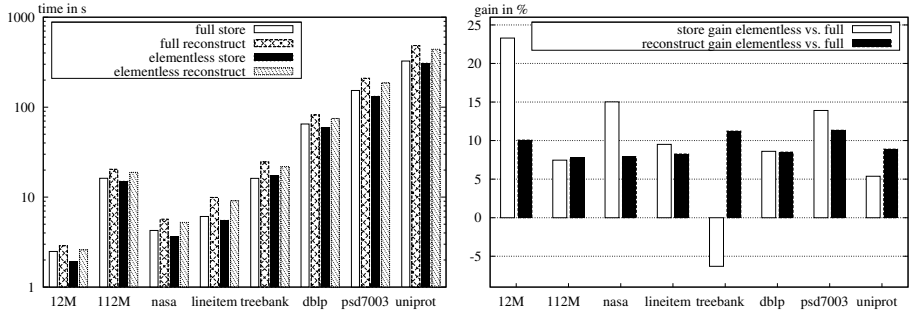
B.2 Storage Self-Tuning Similarity Findings

Here, we continue the analysis of our similarity-based classification of XML documents from Section 5.10.4. The cost and weight parameter selection for similarity-based classifications can be used to control the degree of congruency required for adding new documents to existing classes or documents. Therefore, we provide, based on the TPoX benchmark, more results for alternative parameter configurations shown in Table B.1.

The following parameters and plots are equivalent to the results presented in Section 5.10.4 (denoted as *moderate* configuration), we only changed the parameter weights to evaluate other typical configurations. The first configuration *attribute* shows the impact for raising the attribute cost weight to 0.9 compared to the cost for an element modification. The results are shown in Figure B.2. Matching costs are increased for document classes such as *order* and *security*, because they typically require the addition of new attributes.

Table B.1: Alternative parameter configurations for structural similarity

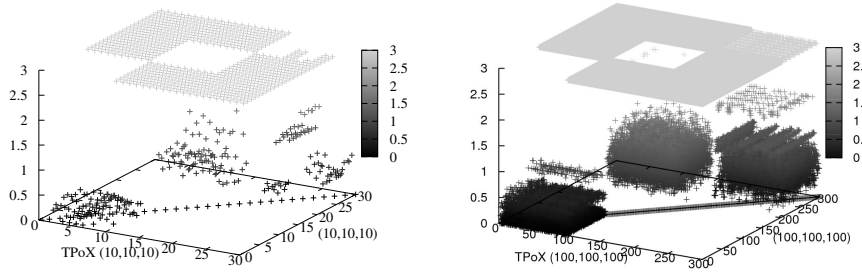
Parameter	Configurations			
	moderate	**attribute**	**strict**	**relaxed**
Wordbook overlapp	0.7	0.7	**0.9**	**0.1**
Wordbook growth	0.5	0.5	**0.2**	**0.9**
Path synopsis overlapp	0.7	0.7	**0.9**	**0.1**
Path synopis growth	0.5	0.5	**0.2**	**0.9**
Attribute weight	0.5	**0.9**	0.5	0.5
Figure	5.21	B.2	B.3	B.4



(a) Storage and reconstruction times

(b) Gain elementless vs. full

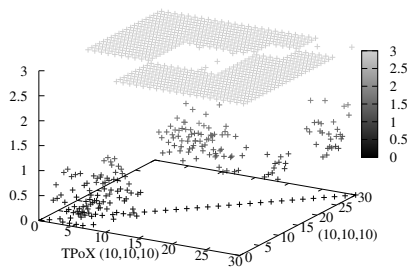
Figure B.1: Storage and reconstruction figures comparing full and elementless storage mapping (hardware configuration 2)



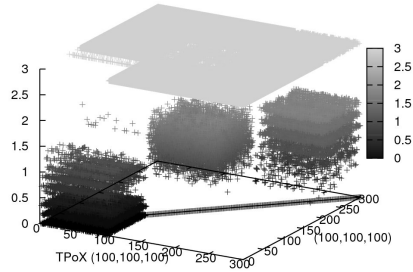
(a) TPoX documents (10x custacc/order/security)

(b) TPoX documents (100x custacc/order/security)

Figure B.2: Similarity measurements and cost analysis for increased *attribute* costs

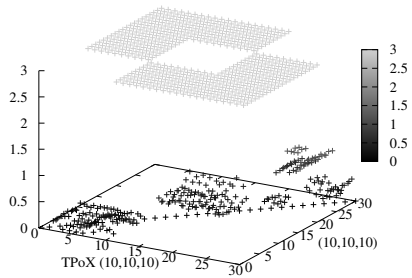


(a) TPoX documents (10x custacc/order/security)

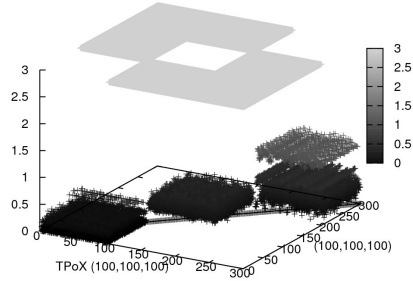


(b) TPoX documents (100x custacc/order/security)

Figure B.3: Similarity measurements and cost analysis for *strict* parameter selection



(a) TPoX documents (10x custacc/order/security)



(b) TPoX documents (100x custacc/order/security)

Figure B.4: Similarity measurements and cost analysis for *relaxed* parameter selection

In a *strict* configuration, shown in Figure B.3, minimal growth (i.e., 20%) of path synopsis and wordbook structures are allowed while a huge share of structural overlapping (i.e., 90%) is required to match documents. Because fewer documents can be joined, especially from different classes, the average cost value (i.e., z-axis value) increases dramatically.

The counterpart, a *relaxed* configuration, is presented in Figure B.4. Because maximal growth (i.e., 90%) is allowed and only marginal overlap (i.e., 10%) enforced, all documents of the same class are matched – indicated by the empty squares at z value 3.

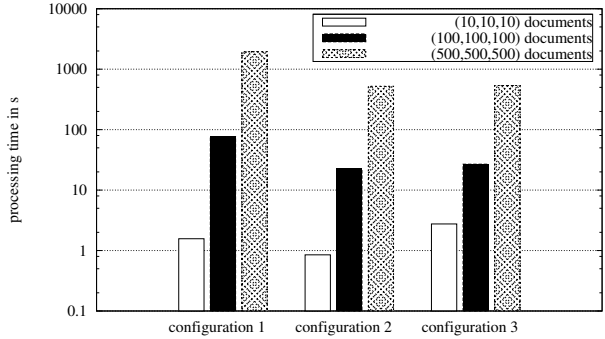


Figure B.5: Similarity matching performance

B.3 Similarity Matching Performance

Matching performance is a crucial point for the applicability of similarity-based classification. We used the same sets of documents as in Sections 5.10.4 and B.2. We only scaled the number of documents for each TPoX class from 10 to 500, which resulted in 30×30, 300×300, and 1500×1500 comparisons. The results are shown in Figure B.5. During each run, all documents were analyzed and cross-wise matched. Note, in a real DBMS, already existing classes of documents are represented and matched only once for each new document. However, we wanted to measure the overhead of document matching, which seems to be fairly small and nearly independent of the hardware configuration. The results show that – as expected – only the CPU power influences the processing time.

B.4 Storage Compression Gains

According to Section 5.9.3, we compared all content compression techniques for our set of XML benchmark documents. The results in Figure B.6 and Figure B.7 show the space savings dependent on the storage mapping. In both cases, the *optimized wordbook* compressor delivers the best results, except for the treebank document. In contrast, the Huffman-based compressors M1 (fixed), M2 (flexible choice), and M3 (selective encoding) deliver almost equal compression ratios, although M1 is marginally better.

Besides compression ratio, we also evaluated the runtime for our content compressors. The results for hardware configurations 1 and 2 are given in Figure B.8 and Figure B.9, respectively. Note, the time scale (y-axis) is logarithmic. Nevertheless, the results show that the more complex the algorithm the more CPU processing, i.e., runtime, is required. Thus, the simple Huffman encoding M1 is the fastest compressor, except for the small 12MB XMark document. Furthermore, the optimized wordbook compressor is slightly slower, especially on hardware configuration 1.

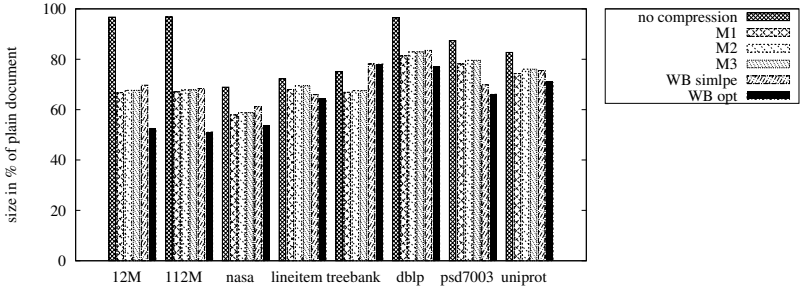


Figure B.6: Full storage mapping

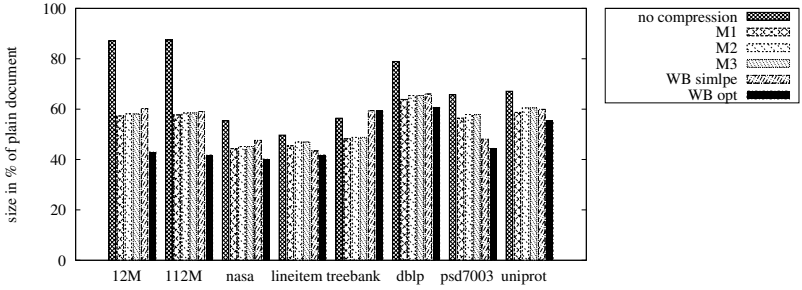


Figure B.7: Elementless storage mapping

Eventually, there seems to be a correlation between compression complexity and its gains. The simple algorithms do not necessarily deliver best compression ratios but deliver often the best runtimes. The hardware impact is negligible when selecting a compressor, but document size and domain are important factors.

B.5 Alternative XML Text Compressors

There exists a plethora of XML-related compression approaches. However, we want to show for some of the best performing compressors to what extent they are applicable for a native XDBMS, especially, with regard to XTC's already built-in compression techniques.

The XML text compressor XMill [LS00] is one of the first and probably the most influential one. Unfortunately, it does not provide query processing capabilities on compressed documents, because XMill physically separates structure and content to achieve maximum compression ratios without retaining homomorphism. The whole document (or chunk by chunk if split into smaller parts) needs to be decompressed for each query and re-compressed in case of

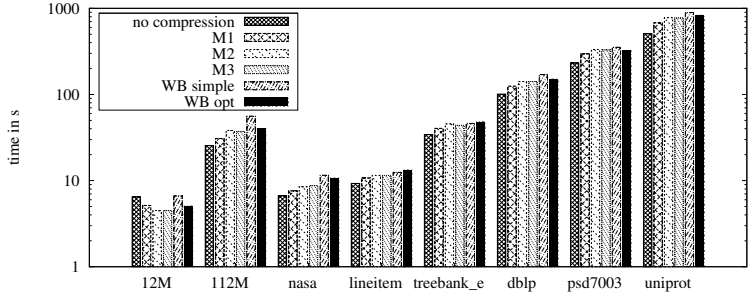


Figure B.8: Hardware configuration 1

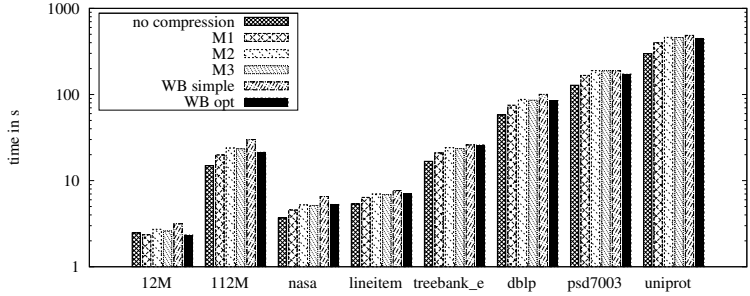


Figure B.9: Hardware configuration 2

modifications, making XMill unacceptable for native XDBMSs.

Another compressor XGRIND [TH02] supports certain query types but provides a lower compression ratio than XMill. XGRIND strictly requires a DTD to enable data-type-dependent compressions. This prohibits its usage in an XDBMS when document modifications cause schema changes. The XPRESS [MPC03] compressor is not applicable within transactional query processing, because it uses a fixed element-frequency-based encoding for paths (reverse arithmetic encoding). In addition, XPRESS retains document homomorphism and uses a "type inference engine" to adjust the compression method to the content. This kind of compression is also fixed, which causes complete document re-compression cycles in case of document modifications. Both compressors do not support set-based query evaluation as required for structural joins and top-down query evaluation.

Many XML compressors, developed throughout the recent years, either require complete decompressions at the document level or provide limited query support without updates at all. Therefore, in the context of XTC, we developed our own content compression techniques working on a node-oriented basis.

B.6 Index Definitions for Sample Query Evaluation Plans

The following indexes are used in Section 6.4 (SPLID is a synonym for Stable Path Label IDentifier, which corresponds to DeweyID in XTC):

(1) `ON 112M CREATE CONTENT INDEX FOR ATTRIBUTE CONTENT` is internally translated to:
`ON 112M CREATE CAS INDEX PATHS //@* OF TYPE STRING WITH SPLID CLUSTERING`

(2) `ON 112M CREATE CONTENT INDEX` is internally translated to:
`ON 112M CREATE CAS INDEX PATHS //@*,//* OF TYPE STRING WITH SPLID CLUSTERING`

(3) `ON 112M CREATE ELEMENT INDEX (INCLUDING people,person,name) WITH SPLID CLUSTERING`

(4) `ON 112M CREATE ELEMENT INDEX WITH SPLID CLUSTERING`

(5) `ON 112M CREATE ELEMENT INDEX (INCLUDING name) WITH SPLID CLUSTERING`

(6) `ON 112M CREATE PATH INDEX PATHS /site/people/person WITH SPLID CLUSTERING`

(7) `ON 112M CREATE PATH INDEX PATHS //person WITH SPLID CLUSTERING`

(8) `ON 112M CREATE CAS INDEX PATHS //@id OF TYPE STRING WITH SPLID CLUSTERING`

(9) `ON 112M CREATE CAS INDEX PATHS //* OF TYPE STRING WITH SPLID CLUSTERING`

A detailed description for the *create index* command can be found by using “help on” in the command line processor of XTC.

Appendix C

Indexing

C.1 Excerpt of AI Metadata in XTC

The following listing exemplifies the AI extensions (`<aistats>` element and the `<ai>` subtree) of the metadata document "_master.xml".

```
<?xml version="1.0" encoding="utf-8"?>
<xtc>
 <dir name="/">
   <doc id="2" name="_master.xml" collection="false" pathSynopsis="3">
       ...
   </doc>
   <doc id="16" name="/auction.xml" collection="false" pathSynopsis="17">
     <indexes>
       <index type="ELEMENT" keyType="STRING" id="13340" clustering="SPLID" collectionID="16"
           >
         <including> 70@SPLID,71@SPLID,3@SPLID, ... </including>
         <statistics size="2416640" card="204733" height="3" leaves="295" ... />
         <aistats benefit="11000086" avgBenefit="95944" cost="212395" sequence="false"
               qgmusage="202" palusage="0" inserts="0" updates="0" deletes="0" sizechange="0"
                reads="1893268" queries="114" />
       </index>
       <index type="CAS" keyType="STRING" id="13600" clustering="SPLID" collectionID="16"
             isAttributeIndex="false">
         <path> /site/closed_auctions/closed_auction/annotation/description/parlist/listitem/
               parlist/listitem/text/emph/keyword </path>
         <statistics size="8192" card ... minKey="a..." maxKey="y..." />
         <aistats benefit="680298" avgBenefit="340149" cost="30521" sequence="false" qgmusage
               ="0" palusage="0" inserts="20" updates="10" deletes="10" sizechange="424"
               reads="6" queries="2" />
       </index>
       <index type="PATH" keyType="STRING" id="13458" ... >
         <path> //annotation </path>
         <statistics size="262144" card="21750" height="2" leaves="32" ... />
         <aistats benefit="79950" avgBenefit="19987" cost="32357" sequence="false" qgmusage="
               0" palusage="0" inserts="0" updates="0" deletes="0" sizechange="0" reads="
               21750" queries="3" />
       </index>
         ...
     </indexes>
     <ai>
       <index type="PATH" keyType="STRING" id="-1" clustering="SPLID">
         <path> //item </path>
         <statistics size="238190" card="21750" height="2" leaves="29" ... />
         <aistats benefit="147561" cost="32357" sequence="false" qgmusage="0" palusage="0" />
         <statistics size="24576" card="2175" height="2" leaves="5" ... />
         <aistats benefit="45" cost="4" sequence="false" qgmusage="0" palusage="0" />
       </index>
       <index type="PATH" keyType="STRING" id="-2" clustering="SPLID">
         <path> //closed_auctions/closed_auction </path>
         <statistics size="106440" card="9750" height="2" leaves="13" ... />
         <aistats benefit="10669" cost="31179" sequence="true" qgmusage="42" palusage="0" />
       </index>
```

```
39          ...
40        </ai>
41        <documentStatistics summaryPage="1310" />
42        <statistics size="101892096" card="1568362" height="3" leaves="12415" ... />
43      </doc>
44    </dir>
45</xtc>
```

C.2 Query Graph Traversal Rules

Brief overview of query graph traversal (QGT) rules for index candidate generation in XTC:

C.2.1 Access Operator

This operator represents a physical XML node access including node name and node type information. When no appropriate index is available, typically a document scan is necessary. Furthermore, during query planning, this operator is configured to access a single node (one-time) or a sequence of nodes (multiple times).

QGT Rule 1 *Each QG node of type access leads to a direct candidate for* element index *or* attribute index. *Depending on the access type (*node *or* sequence*), the usage counter is adjusted accordingly.*

QGT Rule 2 *If a predicate exists that consists of a binary operator between the operator's value and an atomic value, index candidates for both* CAS index *and* content index *are generated. Additional schema information is used to infer the index type.*

Further path steps on top of the access operator may be used to assemble entire paths for *structural indexes*.

QGT Rule 3 *Multiple axis steps are composed to generate a* path index *candidate for access operators without predicates. For access operators with predicates, more specific* CAS index *candidates are generated as well.*

C.2.2 Join Operator

The join operator in the XQGM of XTC is used to represent a *join*, but also to represent a simple *projection* or a *merge* operator. Thus, a join operator may have up to two inputs.

QGT Rule 4 *If only one input is present, index candidates can be derived directly by exploiting the operator's step and/or predicate (e.g., a path, CAS, content, or element index).*

QGT Rule 5 *If two inputs are present of which one uses the context of the other one (i.e., a correlation to a descendant path step), their paths can be combined.*

QGT Rule 6 *If two inputs are present and the join predicate is an axis test, a* path index *candidate is generated by concatenation of the two input paths.*

During the QGT, the output of a projection can be used for new index candidates only if the projected value is still a valid path expression and not a scalar value.

C.2.3 Join Operator with Sort

The optional sort flag allows for duplicate elimination and sorting of at least one operator input.

QGT Rule 7 *If at least one input is set to be sorted, QGT Rule 5 is not applicable.*

C.3 AI Optimization Rules

Optimization Rule 1 *The path expression of a candidate index is validated using the path synopsis. A non-existing path causes the candidate to be dropped.*

Optimization Rule 2 *Remove candidates that overlap with existing user-defined indexes.*

Because user-defined indexes have a higher priority, equivalent candidate indexes are dropped. Only an element index candidate is reduced by removing element identifiers that are covered by existing (user-defined) element indexes.

Optimization Rule 3 *Merge candidates with existing virtual indexes or auto indexes.*

Virtual indexes can be merged, i.e., new path expressions or PCRs can be added, although this is only done for element and content indexes. An already materialized index that is controlled by the AI framework may gets the statistics merged, at least for the overlapping parts.

Optimization Rule 4 *Multiple index candidates of type element or content can be merged.*

Optimization Rule 5 *Remove candidates that are not chosen by the optimizer.*

This rule ensures that only those index candidates are kept, the optimizer would select. The AI framework allows to evaluate the top n plans for index selection and, therefore, index candidates not occurring in the cheapest plan are kept, too. Furthermore, the use counter of candidate indexes during optimization indicates their frequency of identification, which is used to let frequent candidates survive, too.

Optimization Rule 6 *Multiple path indexes sharing a common (ancestor) path prefix can be generalized to a path index for that ancestor path.*

For instance, the two path expressions *//bib/book/title* and *//bib/book/author* may be generalized to *//bib/book*. Regarding processing costs and storage space consumption, this generalized index may be beneficial for answering both path expressions.

Optimization Rule 7 *Virtual indexes are removed if they can be substituted by cheaper indexes supporting the same use.*

XML indexing allows to index the same XML nodes multiple times, even in indexes of different types. Therefore, we identify index-type-crossing overlaps and remove expensive index candidates.

C.4 TPoX Update Query Integration

According to the TPoX[1] benchmark specification, we implemented a *workload driver* for XTC. Our implementation enables the TPoX benchmark suite to run on top of XTC. We provide support for multiple parallel connections and support all transaction templates. The only limitation is the missing support for prepared statements, which, however, does not limit the application of the benchmark. All performance metrics are collected by the benchmark suite. Because queries are based on templates, we emulate prepared statements by our stored procedures.

Let us give an example. For the TPoX *update query 3* "price change", shown in Listing C.1, the corresponding implementation in XTC is shown in Listing C.2

Listing C.1: Update query 3 in XQuery (TPoX)

```
1  -- The price of a security changes [simple value update]
2  -- For a given security symbol, replace the values of the following elements
3  -- in the corresponding security document: LastTrade, Ask, Bid.
4  UPDATE security
5  SET sdoc = XMLQUERY('declare default element namespace "http://tpox-
6  benchmark.com/security";
7    transform
8      copy $secdoc := $doc
9     modify
10          let $price := $secdoc/Security/Price
11          let $newlasttrade := $price/PriceToday/Open*0.95
12          return (
13          do replace value of
14                  $price/LastTrade with $newlasttrade,
15          do replace value of
16                  $price/Ask with $newlasttrade*1.01,
17          do replace value of
18                  $price/Bid with $newlasttrade*0.99)
19    return $secdoc'
20    PASSING sdoc AS "doc")
21 WHERE XMLEXISTS
22 ('declare default element namespace "http://tpox-benchmark.com/security";
23  $sdoc/Security[Symbol="$symbol"]' PASSING sdoc AS "sdoc"
24 )
```

The XTC procedure first calculates the *newLastTrade* price (lines 2–3). Then the subtree to be updated is resolved (lines 5–6) and iterated (lines 7–18). During the child iterations, DOM operations are used to update the corresponding values.

[1] http://tpox.svn.sourceforge.net/viewvc/*checkout*/tpox/TPoX/documentation/TPoX_BenchmarkProposal_v2.0.pdf

Listing C.2: Update query 3 in XTC

```
security(tpoxCollection tp, String symbol) {
 String query = "doc(\"%s\")/Security[Symbol=\"%s\"]/Price/PriceToday/Open*0.95";
 XQueryResult result = executeXQuery(String.format(query, tp, symbol));
 double newLastTrade = result.getDouble();
 query = "doc(\"%s\")/Security[Symbol=\"%s\"]/Price";
 result = executeXQuery(String.format(query, tp, symbol));
 List<Node> nodes = result.getNodeList();
 for(Node node : nodes) {
   Stream<Node> children = node.getChildren();
   while(children.hasNext()) {
     Node child = children.next();
     if (child.getName().equals("LastTrade"))
       child.setValue(String.valueOf(newLastTrade));
     else if (child.getName().equals("Ask"))
       child.setValue(String.valueOf(newLastTrade*1.01));
     else if (child.getName().equals("Bid"))
       child.setValue(String.valueOf(newLastTrade*0.99));
   }
   children.close();
} }
```

Using a mixture of XQuery statements, node-oriented DOM operations, and subtree-oriented operations (i.e., delete, insert, scan), all TPoX benchmark queries can be processed in XTC.

Bibliography

[ABG10] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. The pq-gram distance between ordered labeled trees. *ACM Trans. Database Syst.*, 35(1):1–36, 2010.

[Abi97] Serge Abiteboul. Querying Semi-Structured Data. In *Proc. ICDT*, pages 1–18, 1997.

[ACN06] Sanjay Agrawal, Eric Chu, and Vivek Narasayya. Automatic physical design tuning: workload as a sequence. In *Proc. SIGMOD*, pages 683–694, 2006.

[AHK+04] A. Aboulnaga, P. Haas, M. Kandil, S. Lightstone, G. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated statistics collection in DB2 UDB. In *Proc. VLDB*, pages 1158–1169, 2004.

[AKJK+02] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE*, pages 141–152, 2002.

[AMFH08] José de Aguiar Moraes Filho and Theo Härder. EXsum: an XML summarization framework. In *Proc. IDEAS*, pages 139–148, 2008.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason Mchugh, Jennifer Widom, and Janet Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1:68–88, 1997.

[AYBB+08] Sihem Amer-Yahia, Chavdar Botev, Stephen Buxton, Pat Case, Jochen Doerre, Mary Holstege, Jim Melton, and Microsoft Jayavel Shanmugasundaram Michael Rys. XQuery Update Facility 1.0. W3C Candidate Recommendation. `http://www.w3.org/TR/xquery-update-10/`, 2008.

[Bac09] Charles W. Bachman. The Origin of the Integrated Data Store (IDS): The First Direct-Access DBMS. *IEEE Ann. Hist. Comput.*, 31:42–54, October 2009.

[BBC+07] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernndez, M. Kay, J. Robie, and J Simon. XML Path Language (XPath) 2.0. W3C Recommendation. `http://www.w3.org/TR/xpath`, 2007.

[BBON06] Andrey Balmin, Kevin S. Beyer, Fatma Özcan, and Matthias Nicola. On the path to efficient XML queries. In *Proc. VLDB*, pages 1117–1128, 2006.

[BC05] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *Proc. SIGMOD*, pages 227–238, 2005.

[BC07] Nicolas Bruno and Surajit Chaudhuri. Physical design refinement: The merge-reduce approach. *ACM Trans. Database Syst.*, 32, November 2007.

[BC08] Nicolas Bruno and Surajit Chaudhuri. Constrained physical design tuning. *Proc. VLDB Endow.*, 1(1):4–15, 2008.

[BCF+07] S. Boag, D. Chamberlin, M. Fernanadez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language. W3C Recommendation. `http://www.w3.org/TR/xquery`, 2007.

[BCJ+05] Kevin Beyer, Roberta J. Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy Lohman, Bob Lyle, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Tuong Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang. System RX: one part relational, one part XML. In *Proc. SIGMOD*, pages 347–358, 2005.

[BCL93] Kurt P. Brown, Michael J. Carey, and Miron Livny. Managing Memory to Meet Multiclass Workload Response Time Goals. In *Proc. VLDB*, pages 328–341, 1993.

[BCL96] Kurt P. Brown, Michael J. Carey, and Miron Livny. Goal-oriented buffer management revisited. In *Proc. SIGMOD*, pages 353–364, 1996.

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5:78–101, June 1966.

[BFH+02] Philip Bohannon, Juliana Freire, Jayant R. Haritsa, Prasan Roy, and Jérôme Siméon. LegoDB: customizing relational storage for XML documents. In *Proc. VLDB*, pages 1091–1094, 2002.

[BGJM04] R. J. Bayardo, D. Gruhl, V. Josifovski, and J. Myllymaki. An evaluation of binary XML encoding optimizations for fast stream based XML processing. In *Proc. WWW*, pages 345–354, 2004.

[BGvK+06] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. SIGMOD*, pages 479–490, 2006.

[Bil05] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[BKKM00] Eepan Banerjee, Vishu Krishnamurthy, Muralidhar Krishnaprasad, and Ravi Murthy. Oracle8i - The XML Enabled Data Management System. In *Proc. ICDE*, pages 561–568, 2000.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proc. SIGMOD*, pages 310–321, 2002.

[BM70] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indexes. In *SIGFIDET Workshop*, pages 107–141, 1970.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, (1):173–189, 1972.

[BM04] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 187–200, 2004.

[BMM+04] Dharini Balasubramaniam, Ron Morrison, Kath Mickan, Graham Kirby, Brian Warboys, Ian Robertson, Bob Snowdon, R. Mark Greenwood, and Wykeen Seet. Support for feedback and change in self-adaptive systems. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems (WOSS)*, pages 18–22, 2004.

[BR03] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35:268–308, September 2003.

[BR04] Timo Böhme and Erhard Rahm. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In *DIWeb*, pages 70–81, 2004.

[BS11] Sebastian Bächle and Karsten Schmidt. Lightweight Performance Forecasts for Buffer Algorithms. In *Proc. BTW*, pages 147–166, 2011.

[CC03] Zhimin Chen and C Zhimin Chen. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proc. VLDB*, pages 237–248, 2003.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proc. SIGMOD*, pages 383–394, 1994.

[CGJ97] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. *Approximation algorithms for bin packing: a survey*, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[CKM02] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling dynamic XML trees. In *Proc. PODS*, pages 271–281, 2002.

[CLL05] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proc. SIGMOD*, pages 455–466, 2005.

[CLO03] Qun Chen, Andrew Lim, and Kian Win Ong. D(k)-index: an adaptive structural summary for graph-structured data. In *Proc. SIGMOD*, pages 134–144, 2003.

[CLT+06] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proc. VLDB*, pages 283–294, 2006.

[CMB+09] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. An object placement advisor for db2 using solid state storage. *Proc. VLDB Endow.*, 2(2):1318–1329, 2009.

[CMS02] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: an adaptive path index for XML data. In *Proc. SIGMOD*, pages 121–132, 2002.

[CN97] Surajit Chaudhuri and Vivek R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proc. VLDB*, pages 146–155, 1997.

[CN98] Surajit Chaudhuri and Vivek R. Narasayya. AutoAdmin 'What-if' Index Analysis Utility. In *Proc. SIGMOD*, pages 367–378, 1998.

[CN01] Surajit Chaudhuri and Vivek Narasayya. Automating Statistics Management for Query Optimizers. *IEEE Trans. on Knowl. and Data Eng.*, 13(1):7–20, 2001.

[CN07] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proc. VLDB*, pages 3–14, 2007.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[Coe00] Carlos A. Coello. An updated survey of GA-based multiobjective optimization techniques. *ACM Comput. Surv.*, 32:109–143, June 2000.

[Com78] Douglas Comer. The Difficulty of Optimum Index Selection. *ACM Trans. Database Syst.*, 3(4):440–445, 1978.

[COWL02] Jamieson M. Cobleigh, Leon J. Osterweil, Alexander Wise, and Barbara Staudt Lerner. Containment units: a hierarchically composable architecture for adaptive systems. *SIGSOFT Softw. Eng. Notes*, 27:159–165, November 2002.

[CPST03] Vassilis Christophides, Dimitris Plexousakis, Michel Scholl, and Sotirios Tourtounis. On labeling schemes for the semantic web. In *Proc. WWW*, pages 544–555, 2003.

[CS01] Surajit Chaudhuri and Kyuseok Shim. Storage and Retrieval of XML Data Using Relational Databases. In *Proc. VLDB*, page 730, 2001.

[CSF+01] Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proc. VLDB*, pages 341–350, 2001.

[CVZ+02] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed XML documents. In *Proc. VLDB*, pages 263–274, 2002.

[CW05] Surajit Chaudhuri and Gerhard Weikum. Foundations of automated database tuning. In *Proc. SIGMOD*, page Tutorial Slides, 2005.

[DDD+09] Gianluca Demartin, Ludovic Denoye, Antoine Douce, Khairun Nisa Fachry, Patrick Gallinar, Shlomo Gev, Wei-Che Huang, Tereza Iofciu, Jaap Kamps, Gabriella Kazai, Marijn Koolen, Monica Landoni, Ragnar Nordlie, Nils Pharo, Ralf Schenkel, Martin Theobald, Andrew Trotman, Arjen P. de Vries, Alan Woodley, and Jianhan Zhu. Report on INEX 2008. *SIGIR Forum*, 43:17–36, June 2009.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder Policy Specification Language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 18–38, 2001.

[Dew] M. Dewey. Dewey Decimal Classification System. `http://frank.mtsu.edu/~vvesper/dewey2.htm`.

[DG07] Ludovic Denoyer and Patrick Gallinari. Report on the XML mining track at INEX 2005 and INEX 2006: categorization and clustering of XML documents. *SIGIR Forum*, 41(1):79–90, 2007.

[DOM05] W3C DOM. Document Object Model. `http://www.w3.org/DOM/`, 2005.

[DRS+05] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic Performance Diagnosis and Tuning in Oracle. In *Proc. CIDR*, pages 84–94, 2005.

[DT90] Asit Dan and Don Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. *SIGMETRICS Perform. Eval. Rev.*, 18(1):143–152, 1990.

[DTB09] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with iTuned. *Proc. VLDB Endow.*, 2:1246–1257, August 2009.

[DYC93] Asit Dan, Philip S. Yu, and Jen-Yao Chung. Database Access Characterization for Buffer Hit Prediction. In *Proc. ICDE*, pages 134–143, 1993.

[EAZ+08a] Iman Elghandour, Ashraf Aboulnaga, Daniel C. Zilio, Fei Chiang, Andrey Balmin, Kevin S. Beyer, and Calisto Zuzarte. An XML index advisor for DB2. In *Proc. SIGMOD*, pages 1267–1270, 2008.

[EAZ+08b] Iman Elghandour, Ashraf Aboulnaga, Daniel C. Zilio, Fei Chiang, Andrey Balmin, Kevin S. Beyer, and Calisto Zuzarte. XML Index Recommendation with Tight Optimizer Coupling. In *Proc. ICDE*, pages 833–842, 2008.

[EH84] Wolfgang Effelsberg and Theo Härder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, 1984.

[EM09] Said Elnaffar and Patrick Martin. The Psychic-Skeptic Prediction framework for effective monitoring of DBMS workloads. *Data Knowl. Eng.*, 68:393–414, April 2009.

[FGK06] Andrey Fomichev, Maxim Grinev, and Sergey Kuznetsov. Sedna: A Native XML DBMS. In *SOFSEM 2006: Theory and Practice of Computer Science*, volume 3831 of *LNCS*, pages 272–281. 2006.

[FHK+02] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4):292–314, 2002.

[FMM+05] Sergio Flesca, Giuseppe Manco, Elio Masciari, Luigi Pontieri, and Andrea Pugliese. Fast Detection of XML Structural Similarity. *IEEE Trans. on Knowl. and Data Eng.*, 17(2):160–175, 2005.

[FMM+07] Mary Fernndez, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model. W3C Recommendation. `http://www.w3.org/TR/xpath-datamodel`, 2007.

[FRK99] Daniela Florescu, Inria Roquencourt, and Donald Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22:27–34, 1999.

[Fry03] C. Fry. Streaming API for XML. JSR 173. `http://www.jcp.org/en/jsr/detail?id=173`, 2003.

[FSC+03] Mary Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing XQuery 1.0: the Galax experience. In *Proc. VLDB*, pages 1077–1080, 2003.

[Gar02] David Garlan. Model-based Adaptation for Self-Healing Systems. In *Proceedings of the first workshop on Self-healing systems*, pages 27–32, 2002.

[Geo05] John C. Georgas. Architectural runtime configuration management in support of dependable self-adaptive software. In *Workshop on Architecting Dependable Systems*, pages 1–6, 2005.

[GGU72] M. R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analysis of memory allocation algorithms. In *Proceedings of the fourth annual ACM symposium on Theory of computing (STOC)*, pages 143–150, 1972.

[GHS07] Christian Grün, Alexander Holupirek, and Marc H. Scholl. Visually Exploring and Querying XML with BaseX. In *Proc. BTW*, pages 629–632, 2007.

[GKW04] David Garlan, Jeff Kramer, and Alexander Wolf, editors. *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. ACM, 2004.

[Gra02] Mark Graves. *Designing XML Databases*. Prentice Hall, Upper Saddle River, NJ, USA, 2002.

[GST70] J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, 1970.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. VLDB*, pages 436–445, 1997.

[Hel07] Sven Helmer. Measuring the structural similarity of semistructured documents using entropy. In *Proc. VLDB*, pages 1022–1032, 2007.

[HFLP89] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. SIGMOD*, pages 377–388, 1989.

[HH07] Michael Peter Haustein and Theo Härder. An efficient infrastructure for native transactional XML processing. *Data Knowl. Eng.*, 61(3):500–523, 2007.

[HHL06] Michael P. Haustein, Theo Härder, and Konstantin Luttenberger. Contest of XML Lock Protocols. In *Proc. VLDB*, pages 1069–1080, 2006.

[HHMW07] Theo Härder, Michael Peter Haustein, Christian Mathis, and Markus Wagner. Node labeling schemes for dynamic XML documents reconsidered. *Data Knowl. Eng.*, 60(1):126–149, 2007.

[HKL05] Beda Christoph Hammerschmidt, Martin Kempa, and Volker Linnemann. Autonomous Index Optimization in XML Databases. In *ICDE Workshops*, page 1217, 2005.

[HLS05] Zhen He, Byung Suk Lee, and Robert Snapp. Self-tuning cost modeling of user-defined functions in an object-relational DBMS. *ACM Trans. Database Syst.*, 30:812–853, September 2005.

[HMS07] Theo Härder, Christian Mathis, and Karsten Schmidt. Comparison of Complete and Elementless Native Storage of XML Documents. In *Proc. IDEAS*, pages 102–113, 2007.

[Hor01] Paul Horn. Autonomic computing: IBM's perspective on the state of information technology. IBM Research Project, 2001.

[HR83a] Theo Härder and Andreas Reuter. Concepts for implementing a centralized database management system. In *International Computing Symposium on Application Systems Development*, pages 28–59. B.G. Teubner, 1983.

[HR83b] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15:287–317, December 1983.

[HR07] Marc Holze and Norbert Ritter. Towards workload shift detection and prediction for autonomic databases. In *Proceedings of the ACM first Ph.D. workshop in CIKM*, pages 109–116, 2007.

[HSOB09] Theo Härder, Karsten Schmidt, Yi Ou, and Sebastian Bächle. Towards Flash Disk Use in Databases - Keeping Performance While Saving Energy? In *Proc. BTW*, pages 167–186, 2009.

[IBM04] IBM. An architectural blueprint for autonomic computing. Technical Report, 2003 (revised: 2004).

[IMKG11] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9), 2011.

[JAKC+02] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *VLDB Journal*, 11(4):274–291, 2002.

[JCL90] R. Jauhari, M. J. Carey, and M. Livny. Priority-Hints: An Algorithm for Priority-Based Buffer Management. In *Proc. VLDB*, pages 708–721, 1990.

[JLST02] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In *International Workshop on Database Programming Languages*, pages 149–164, 2002.

[JLW03] Haifeng Jiang, Hongjun Lu, and Wei Wang. XR-Tree: Indexing XML data for efficient structural join. In *Proc. ICDE*, 2003.

[JS94] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. VLDB*, pages 439–450, 1994.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 190–201, 2000.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[Kep04] Stephan Kepser. A proof of the turing-completeness of XSLT and XQuery. In *Technical report SFB 441, Eberhard Karls Universität Tübingen*, 2004.

[KLS+03] Eva Kwan, Sam Lightstone, K. Bernhard Schiefer, Adam J. Storm, and Leanne Wu. Automatic Database Configuration for DB2 Universal Database: Compressing Years of Performance Expertise into Seconds of Execution. In *Proc. BTW*, pages 620–629, 2003.

[KM07] Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering (FOSE)*, pages 259–268, 2007.

[KRML05] Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. FiST: scalable XML document filtering by sequencing twig patterns. In *Proc. VLDB*, pages 217–228, 2005.

[KS89] James F. Kurose and Rahul Simha. A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems. *IEEE Trans. Comput.*, 38:705–717, May 1989.

[KSBG02] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proc. ICDE*, pages 129–140, 2002.

[KSH02] Meike Klettke, Lars Schneider, and Andreas Heuer. Metrics for XML Document Collections. In *EDBT Workshops*, pages 15–28, 2002.

[LCL05] Jiaheng Lu, Ting Chen, and Tok Wang Ling. TJFast: effective processing of XML twig pattern matching. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 1118–1119, 2005.

[LDB+04] Rasko Leinonen, Federico Garcia Diez, David Binns, Wolfgang Fleischmann, Rodrigo Lopez, and Rolf Apweiler. Uniprot archive. *Bioinformatics*, 20:3236–3237, 2004.

[Ley02] Michael Ley. The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, pages 1–10, 2002.

[LKO+00] Mong Li Lee, Masaru Kitsuregawa, Beng Chin Ooi, Kian-Lee Tan, and Anirban Mondal. Towards self-tuning data placement in parallel database systems. In *Proc. SIGMOD*, pages 225–236, 2000.

[LL02] Guy M. Lohman and Sam S. Lightstone. SMART: making DB2 (more) autonomic. In *Proc. VLDB*, pages 877–879, 2002.

[LLH08] Changqing Li, Tok Wang Ling, and Min Hu. Efficient updates in dynamic XML data: from binary string to quaternary string. *VLDB Journal*, 17(3):573–601, 2008.

[LM01] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. VLDB*, pages 361–370, 2001.

[LNF09] Henrik Loeser, Matthias Nicola, and Jana Fitzgerald. Index Challenges in Native XML Database Systems. In *Proc. BTW*, pages 508–525, 2009.

[LNPM98] Spyros Lalis, Christos Nikalaou, Dimitris Papadakis, and Manolis Marazakis. Market-driven service allocation in a QoS-capable environment. In *Proceedings of the first international conference on Information and computation economies*, pages 92–100, 1998.

[LS00] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. In *Proc. SIGMOD*, pages 153–164, 2000.

[LSSS07] Martin Lühring, Kai-Uwe Sattler, Karsten Schmidt, and Eike Schallehn. Autonomous Management of Soft Indexes. In *ICDE Workshops*, pages 450–458, 2007.

[LWF77] Tomás Lang, Christopher Wood, and Eduardo B. Fernández. Database buffer paging in virtual storage systems. *ACM Trans. Database Syst.*, 2(4):339–351, 1977.

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. Lore: a database management system for semistructured data. *SIGMOD Rec.*, 26(3):54–66, 1997.

[Mat09] Christian Mathis. *Storing, Indexing, and Querying XML Documents in Native Database Management Systems*. PhD thesis, University of Kaiserslautern, München, 7 2009.

[MBV03] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The XML web: a first study. In *Proc. WWW*, pages 500–510, 2003.

[MD97] Manish Mehta and David J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB Journal*, 6(1):53–72, 1997.

[Mei09] Wolfgang Meier. eXist: An Open Source Native XML Database. In *Web, Web-Services, and Database Systems*, LNCS, pages 169–183. 2009.

[MEW06] Pat Martin, Said Elnaffar, and Ted Wasserman. Workload Models for Autonomic Database Management Systems. In *Proc. ICAS*, page 10, 2006.

[MHS09] Christian Mathis, Theo Härder, and Karsten Schmidt. Storing and indexing XML documents upside down. *Computer Science - Research and Development*, 24:51–68, 2009.

[Mik] G. Miklau. XML Data Repository. `www.cs.washington.edu/research/xmldatasets`.

[Mit95] Bernhard Mitschang. *Anfrageverarbeitung in Datenbanksystemen - Entwurfs- und Implementierungskonzepte*. Vieweg, 1995.

[MLZ+00] Patrick Martin, Hoi-Ying Li, Min Zheng, Keri Romanufa, and Wendy Powley. Dynamic Reconfiguration Algorithm: Dynamically Tuning Multiple Buffer Pools. In *Proc. DEXA*, pages 92–101, 2000.

[MM03] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, 2003.

[MPC03] Jun-Ki Min, Myung-Jae Park, and Chin-Wan Chung. XPRESS: a queriable compression for XML data. In *Proc. SIGMOD*, pages 122–133, 2003.

[MRT99] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st international conference on Software engineering (ICSE)*, pages 44–53, 1999.

[MS99] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proc. ICDT*, pages 277–295, 1999.

[MW99] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proc. VLDB*, pages 315–326, 1999.

[MWHH08] Christian Mathis, Andreas Weiner, Theo Härder, and Caesar Ralf Franz Hoppen. XTCcmp: XQuery Compilation on XTC. In *Proc. VLDB (Demo Track)*, 2008.

[NDD92] Victor F. Nicola, Asit Dan, and Daniel M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *Proc. SIGMETRICS*, pages 35–46, 1992.

[Neu28] John Von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[NFS95] R. Ng, C. Faloutsos, and T. Sellis. Flexible and Adaptable Buffer Management Techniques for Database Management Systems. *IEEE Trans. Comput.*, 44(4):546–560, 1995.

[NJ02] Andrew Nierman and H. V. Jagadish. Evaluating Structural Similarity in XML Documents. In *WebDB Workshop*, pages 61–66, 2002.

[NKS07] Matthias Nicola, Irina Kogan, and Berni Schiefer. An XML transaction processing benchmark. In *Proc. SIGMOD*, pages 937–948, 2007.

[NTA05] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 239–248, 2005.

[Nue06] David Nuescheler. JSR 170: Content Repository for Java technology API (Release version 1.0.1), Apr 2006.

[NvdL05] Matthias Nicola and Bert van der Linden. Native XML support in DB2 universal database. In *Proc. VLDB*, pages 1164–1174, 2005.

[OHJ10] Yi Ou, Theo Härder, and Peiquan Jin. CFDC: a flash-aware buffer management algorithm for database systems. In *Proceedings of the 14th east European conference on Advances in databases and information systems*, pages 435–449, 2010.

[OOP+04] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: insert-friendly XML node labels. In *Proc. SIGMOD*, pages 903–908, 2004.

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *SIGMOD Rec.*, 22(2):297–306, 1993.

[OS11] Ludger Overbeck and Karsten Schmidt. Self-Tuning for Short-Term Memory Consumers. *Datenbank-Spektrum*, 11:37–41, 2011.

[PC03] Feng Peng and Sudarshan S. Chawathe. XPath queries on streaming data. In *Proc. SIGMOD*, pages 431–442, 2003.

[PCS+05] Shankar Pal, Istvan Cseri, Oliver Seeliger, Michael Rys, Gideon Schaller, Wei Yu, Dragan Tomic, Adrian Baras, Brandon Berg, Denis Churin, and Eugene Kogan. Xquery implementation in a relational database system. In *Proc. VLDB*, pages 1175–1186, 2005.

[PDA07] Stratos Papadomanolakis, Debabrata Dash, and Anastasia Ailamaki. Efficient use of the query optimizer for automated physical design. In *Proc. VLDB*, pages 1093–1104, 2007.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. SIGMOD*, pages 39–48, 1992.

[PJ05] Stelios Paparizos and H. V. Jagadish. Pattern tree algebras: sets or sequences? In *Proc. VLDB*, pages 349–360, 2005.

[PJK+06] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *Proc. CASES*, pages 234–241, 2006.

[Pre08] Sebastian Prehn. A Java Content Repository Backed by the Native XML Database System XTC, 7 2008.

[PS83] Gregory Piatetsky-Shapiro. The Optimal Selection of Secondary Indices is NP-Complete. *SIGMOD Record*, 13(2):72–75, 1983.

[PWLJ04] Stelios Paparizos, Yuqing Wu, Laks V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *Proc. SIGMOD*, pages 71–82, 2004.

[PZ91] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. Technical report, 1991. Brown University.

[QSF+07] Lin Qiao, Basuki Soetarman, Gene Fuh, Adarsh Pannu, Baoqiu Cui, Thomas Beavin, and William Kyu. A framework for enforcing application policies in database systems. In *Proc. SIGMOD*, pages 981–992, 2007.

[QWG+96] Dallan Quass, Jennifer Widom, Roy Goldman, Kevin Haas, Qingshan Luo, Jason McHugh, Svetlozar Nestorov, Anand Rajaraman, Hugo Rivero, Serge Abiteboul, Jeff Ullman, and Janet Wiener. LORE: a Lightweight Object REpository for semistructured data. In *Proc. SIGMOD*, page 549, 1996.

[Rao87] S.S. Rao. Game theory approach for multiobjective structural optimization. *Computers & Structures*, 25(1):119 – 127, 1987.

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[RP86] J. K. Reynolds and J. Postel. Official ARPA-Internet protocols, 1986.

[RPBP04] Kanda Runapongsa, Jignesh M. Patel, Rajesh Bordawekar, and Sriram Padmanabhan. XIST: An XML Index Selection Tool. In *Proc. XSym*, pages 219–234, 2004.

[RS91] Steve Rozen and Dennis Shasha. A Framework for Automating Physical Database Design. In *Proc. VLDB*, pages 401–411, 1991.

[SAL+96] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: a wide-area distributed database system. *VLDB Journal*, 5:048–063, 1996.

[SAMP06] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. COLT: continuous on-line tuning. In *Proc. SIGMOD*, pages 793–795. ACM, 2006.

[SAX04] Simple API for XML (SAX) 2.0.2. `http://sax.sourceforge.net`, 2004.

[SB11] Karsten Schmidt and Sebastian Bächle. Low-overhead decision support for dynamic buffer reallocation. *Computer Science - Research and Development*, pages 1–15, 2011.

[SBH09] Karsten Schmidt, Sebastian Bächle, and Theo Härder. *Benchmarking Performance-Critical Components in a Native XML Database System*, pages 64–78. Springer-Verlag, 2009.

[Sch01] Dr. Harald Schöning. Tamino - A DBMS Designed for XML. In *Proc. ICDE*, pages 149–, 2001.

[Sch09] Karsten Schmidt. Goal-Driven Autonomous Database Tuning Supported by a System Model. In *SIGMOD Workshop "Innovative Database Research (IDAR)"*, 2009.

[SGAL+06] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *Proc. VLDB*, pages 1081–1092, 2006.

[SGS03] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. QUIET: continuous query-driven index tuning. In *Proc. VLDB*, pages 1129–1132, 2003.

[SH07] Karsten Schmidt and Theo Härder. An Adaptive Storage Manager for XML Documents. In *BTW Workshops*, pages 317–328, 2007.

[SH08] Karsten Schmidt and Theo Härder. Usage-driven storage structures for native XML databases. In *Proc. IDEAS*, pages 169–178, 2008.

[SH10] Karsten Schmidt and Theo Härder. On the Use of Query-driven XML Auto-Indexing. In *SMDB Workshop*, pages 1–6, 2010.

[SHKR08] Karsten Schmidt, Theo Härder, Joachim Klein, and Steffen Reithermann. Green Computing - A Case for Data Caching and Flash Disks? In *Proc. ICEIS*, pages 535–540, 6 2008.

[SLMK01] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *Proc. VLDB*, pages 19–28, 2001.

[Smi78] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, 3(3):223–247, 1978.

[SOH09] Karsten Schmidt, Yi Ou, and Theo Härder. The Promise of Solid State Disks - Increasing Efficiency and Reducing Cost of DBMS Processing. In *C* Conference on Computer Science & Software Engineering (C3S2E-09)*, pages 35–41, 5 2009.

[Sto81] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, 1981.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. VLDB*, pages 302–314, 1999.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: a benchmark for XML data management. In *Proc. VLDB*, pages 974–985, 2002.

[TCW+04] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White. A Multi-Agent Systems Approach to Autonomic Computing. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, pages 464–471, 2004.

[TDCZ02] Feng Tian, David J. DeWitt, Jianjun Chen, and Chun Zhang. The design and performance evaluation of alternative XML storage strategies. *SIGMOD Rec.*, 31(1):5–10, 2002.

[Ten95] J. Teng. Goal-oriented dynamic buffer pool management for data base systems. In *Proceedings of the 1st International Conference on Engineering of Complex Computer Systems*, pages 191–, 1995.

[TG10] Andrea Tagarelli and Sergio Greco. Semantic clustering of XML documents. *ACM Trans. Inf. Syst.*, 28(1):1–56, 2010.

[TH02] Pankaj M. Tolani and Jayant R. Haritsa. XGRIND: A Query-Friendly XML Compressor. In *Proc. ICDE*, pages 225–234, 2002.

[THTT08] Dinh Nguyen Tran, Phung Chinh Huynh, Y. C. Tay, and Anthony K. H. Tung. A new approach to dynamic self-tuning of database buffers. *Trans. Storage*, 4(1):1–25, 2008.

[TPG97] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. In *Proc. SIGMETRICS*, pages 100–114, 1997.

[VL00] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.

[VZZ+00] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proc. ICDE*, pages 101–110, 2000.

[Wei11] Andreas M. Weiner. Advanced Cardinality Estimation in the XML Query Graph Model. In *Proc. BTW*, pages 207–226, 2011.

[WHH+92] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A Distributed Computational Economy. *IEEE Trans. Softw. Eng.*, 18:103–117, February 1992.

[WJW+05] Wei Wang, Haifeng Jiang, Hongzhi Wang, Xuemin Lin, Hongjun Lu, and Jianzhong Li. Efficient processing of XML path queries using the disk-based F&B Index. In *Proc. VLDB*, pages 145–156, 2005.

[WMHZ02] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Proc. VLDB*, pages 20–31, 2002.

[WRRA08] David Wiese, Gennadi Rabinovitch, Michael Reichert, and Stephan Arenswald. Autonomic tuning expert: a framework for best-practice oriented autonomic database tuning. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds (CASCON)*, pages 3:27–3:41, 2008.

[XMP02] Xiaoyi Xu, Patrick Martin, and Wendy Powley. Configuring buffer pools in DB2 UDB. In *Proc. CASCON*, page 13, 2002.

[XXM+06] Meng Xiaofeng, Wang Xiaofeng, Xie Min, Zhang Xin, and Zhou Junfeng. OrientX: An integrated, schema based native XML database system. *Wuhan University Journal of Natural Sciences*, 11:1192–1196, 2006.

[YAA07] Hailing Yu, Divyakant Agrawal, and Amr El Abbadi. Mems based storage architecture for relational databases. *VLDB Journal*, 16(2):251–268, 2007.

[YASU01] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. Internet Technol.*, 1(1):110–141, 2001.

[yCFW+95] Jen yao Chung, Donald Ferguson, George Wang, Christos Nikolaou, and Jim Teng. Goal Oriented Dynamic Buffer Pool Management for Data Base Systems. In *IBM Research Report RC19807*, pages 191–198, 1995.

[YLL+07] Chi Yang, Chengfei Liu, Jianxin Li, Jeffrey Xu Yu, and Junhu Wang. Semantics based buffer reduction for queries over XML data streams. In *Proceedings of the nineteenth conference on Australasian database*, pages 145–153, 2007.

[YLML05] Jeffrey Xu Yu, Daofeng Luo, Xiaofeng Meng, and Hongjun Lu. Dynamically Updating XML Data: Numbering Scheme Revisited. *World Wide Web*, 8(1):5–26, 2005.

[ZCZ03] Zhongping Zhang, Rong Li Shunliang Cao, and Yangyong Zhu. Similarity metric for XML documents. In *Proc. of Workshop on Knowledge and Experience Management*, 2003.

[ZND+01] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proc. SIGMOD*, pages 425–436, 2001.

[ZRL+04] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 design advisor: integrated automatic physical database design. In *Proc. VLDB*, pages 1087–1097, 2004.

[ZSS92] Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Inf. Process. Lett.*, 42(3):133–139, 1992.

[ZZL+04] Daniel C. Zilio, Calisto Zuzarte, Guy M. Lohman, Hamid Pirahesh, Jarek Gryz, Eric Alton, Dongming Liang, and Gary Valentin. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *Proc. ICAC*, pages 180–188, 2004.

curriculum vitae

Karsten Schmidt

Personal Information

Business address:	Gottlieb-Daimler-Str. 47 67663 Kaiserslautern Germany Phone: +49 (0) 631 205 3277 E-mail: kschmidt@cs.uni-kl.de
Date of birth:	July 26th 1979
Place of birth:	Brandenburg/Havel, Germany
Marital status:	Married
Nationality:	German

Education

06/2006 – 09/2011	Doctoral candidate, Database and Information Systems Group University of Kaiserslautern, Germany
10/1999 – 05/2006	Diploma in computer science (Dipl.-Inf.) Ilmenau University of Technology, Germany
09/1990 – 07/1998	Secondary school: Herzog-Georg-Gymnasium Bad Liebenstein

Working Experience

06/2006 – 09/2011	Scientific staff member, Database and Information Systems Group University of Kaiserslautern, Germany
10/1998 – 09/1999	Civilian service, county hospital, Bad Salzungen, Germany