Master Thesis

University of Kaiserslautern

Department of Computer Science
AG Datenbanken und Informationssysteme

Evaluation of fine-grained locking in XML databases

Author:
Martin Hiller

Supervisors:
M.Sc. Caetano Sauer
Prof. Dr.-Ing. Dr. h. c. Theo Härder

submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

March 3, 2014

Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema „Evaluation of fine-grained locking in XML databases“ selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

(Ort, Datum)	(Unterschrift)

Abstract

Concurrency is an important requirement for database systems, especially in the light of modern multicore processors. At the same time, concurrent data access must be properly regulated to achieve transaction isolation in accordance with the ACID paradigm. In this thesis, a fine-grained locking mechanism in the context of a native XML database is presented and evaluated by means of a highly contending benchmark workload. In the end, however, we will realize that increased concurrency offered by the respective locking protocol is not always accompanied by improved transaction throughputs.

Contents

1 Motivation and Introduction 1

2 BrackitDB Overview 2
2.1 File Layer and Buffer Manager . . . 2
2.2 Storage Layer and Locking . . . 4
2.3 XQuery engine . . . 5

3 Locking in BrackitDB 6
3.1 Basics of concurrency control . . . 6
3.1.1 Serializability . . . 7
3.1.2 Multi-user phenomena . . . 9
3.1.3 Isolation levels . . . 11
3.2 Introduction to Locking . . . 12
3.2.1 Basic RX Locking . . . 12
3.2.2 Lock Conversion . . . 13
3.2.3 Update Locks . . . 13
3.2.4 Locking Challenges . . . 13
3.3 Hierarchical Locking . . . 14
3.4 taDOM Isolation . . . 16
3.4.1 Node Locks . . . 17
3.4.2 Edge Locks . . . 20
3.4.3 Index Locks . . . 20
3.4.4 Lock Escalation . . . 21
3.5 Implementation Aspects . . . 23
3.5.1 Lock Table . . . 23
3.5.2 Edge Lock Internals . . . 24
3.5.3 Deadlock Detection . . . 25

4 Benchmark Architecture and Setup 26
4.1 The Benchmark Suite . . . 26
4.1.1 Terminology and Benchmark Layout . . . 26
4.1.2 XML Configuration . . . 28
4.1.3 BrackitDB-specific Extensions . . . 30
4.1.4 BenchSuite Architecture and Workflow . . . 33
4.2 Benchmark Document . . . 35
4.3 Benchmark Workload . . . 37
4.3.1 Architectural Placement . . . 37
4.3.2 Secondary Indexes . . . 38
4.3.3 Skewed Access . . . 39
4.3.4 Workload Transactions . . . 40
4.3.5 Workload Variants . . . 43
4.4 Hardware/Software Environment . . . 45

5 Benchmark Execution and Evaluation 46
5.1 BrackitDB System Parameters 46
5.2 Baseline Benchmark 48
5.3 Lock Depth and Lock Escalation 50
5.4 Workload Variants 53
5.5 I/O Edge Cases 56
5.6 Analyzing Throughput Bottlenecks 59

6 Conclusion 60

7 References 62

1 Motivation and Introduction

Concurrency control is one of the cornerstones in every database system aiming for online transactional processing. By focusing on the isolation goal of the well established ACID properties for transactions, concurrency control has to take measures to prevent any anomalies that can occur in a multi-user environment where interleaved transaction schedules are allowed. The most prominent technique to deal with these isolation challenges is *locking*, which has proven its feasibility since the rise of relational database systems. Locking database items in compliance with certain rules ensures that each concurrent transaction is under the impression of being the only one accessing the data.

Although many of the ideas originating from the relational world can be transferred to XML databases, new challenges are posed by the increased complexity of XML data and related access operations. In this thesis, we introduce the locking concept used by BrackitDB, a native XML database system, which features fine-grained locks and thereby allows for a high degree of concurrency while still preserving sufficient isolation among transactions. The primary concern in this thesis is the evaluation of this locking approach in the context of modern computer hardware, such as multicore CPUs, high amounts of available main memory, and relatively fast storage devices.

Since processor clockrates have pretty much reached its peak while the number of cores is still increasing, modern software systems should always aim for adequate multithreading scalability. However, unrestricted concurrency in database systems leads to breaches in isolation and consistency. Therefore, apart from ensuring correctness, a good locking implementation seeks for a tradeoff between maximal concurrency offered by fine-grained locks and yet keeping the locking overhead as small as possible. In order to achieve this balance, BrackitDB's lock manager comes with a mechanism known as *lock escalation* where fine-grained locks can be dynamically coarsened to reduce future locking overhead. Optimization techniques like these and their actual impact on performance will also be covered by the benchmarks in this thesis. Ultimately, we hope to demonstrate the merits of fine-grained locking approaches on modern hardware in mostly CPU-bound scenarios.

From a rather practical point of view, conducting locking-related experiments involves extending BrackitDB and the related benchmark software by the necessary means to capture suitable statistical data that provide insight into the system state during workload processing.

The remainder of this thesis is structured as follows: First, a brief overview of BrackitDB's architecture is presented in section 2. Afterwards, section 3 gradually introduces the locking mechanism employed by BrackitDB, starting with the general tasks and challenges of concurrency control and peaking in implementation aspects of lock management. In contrast, section 4 is concerned with theoretical facets of the upcoming benchmarks by outlining how interaction between the benchmark tool and the database server works on an architectural level, and by exhibiting details about the test document and the workload transactions. After the required preparations are made, section 5 specifies the concrete experiments that have been performed and discusses their results subsequently. Finally, section 6 wraps up this thesis and reveals some final thoughts about fine-grained locking techniques.

2 BrackitDB Overview

The object of study in this thesis is BrackitDB [2], a native XML database management system aiming for efficient collaboration on shared documents or collections of documents in a transaction-consistent fashion. It therefore offers fine-grained isolation and full crash recovery based on a sophisticated native storage with advanced indexing capabilities. With the Brackit query engine [1] incorporated into the project, BrackitDB features a powerful and descriptive XQuery [42] interface for applications to jointly interact with data in the native storage or any other plugged-in storage implementation.

While the Brackit query engine was essentially rewritten from scratch as part of the research in [10], a large portion of BrackitDB's codebase originates from the XTC (XML Transaction Coordinator) project. In particular the file and buffer management, the logging and recovery infrastructure, the lock manager, and some of the storage-related data structures (such as the B^+ tree implementation, path synopsis etc.) were adopted without major changes. Consequently, most of the concepts that were attempted in XTC and published in scientific papers still apply to BrackitDB.

This section provides a brief introduction into BrackitDB's anatomy with special focus on the relevant locking- and storage-related components. Although the benchmarks are mainly concerned with evaluating various locking strategies, all the underlying system layers (such as the storage and buffer implementation) implicitly affect the results as well and are therefore also relevant for this thesis to a certain extent.

In general, BrackitDB's architecture (depicted in Figure 1) follows the same ideas and concepts that are well known from relational databases, such as the five layer architecture described in [22, p. 18 ff.] (although it is not always trivial to precisely identify the border between different layers). As a matter of fact, the two bottom layers (file and buffer manager) are not even aware of the employed data model and could easily originate from a relational system. While also featuring a generic B^+-tree implementation (like the one described in [34]), the storage layer introduces many XML-specific data structures and concepts which are not known to relational databases.

2.1 File Layer and Buffer Manager

The bottom layer in BrackitDB is responsible for any kind of disk I/O. At this, it offers different file abstractions from random access files via block-oriented files to segmented log-like files. In order to fulfill its duty, BrackitDB maintains three classes of files on a persistent storage medium.

The *container* file is the primary storage location for the database, retaining all sorts of user data (XML nodes, BLOBs) and related auxiliary data structures. It is divided into blocks (of 8 KiB size by default) and is accessible through the buffer interface. In fact, BrackitDB even allows for several containers, each managed by a dedicated buffer. But if not explicitly configured, only a single system container is used to hold all collections of documents. This system container also features the so-called master document, which provides a catalog of all stored collections and indexes, including necessary hints how to locate them within the container file.

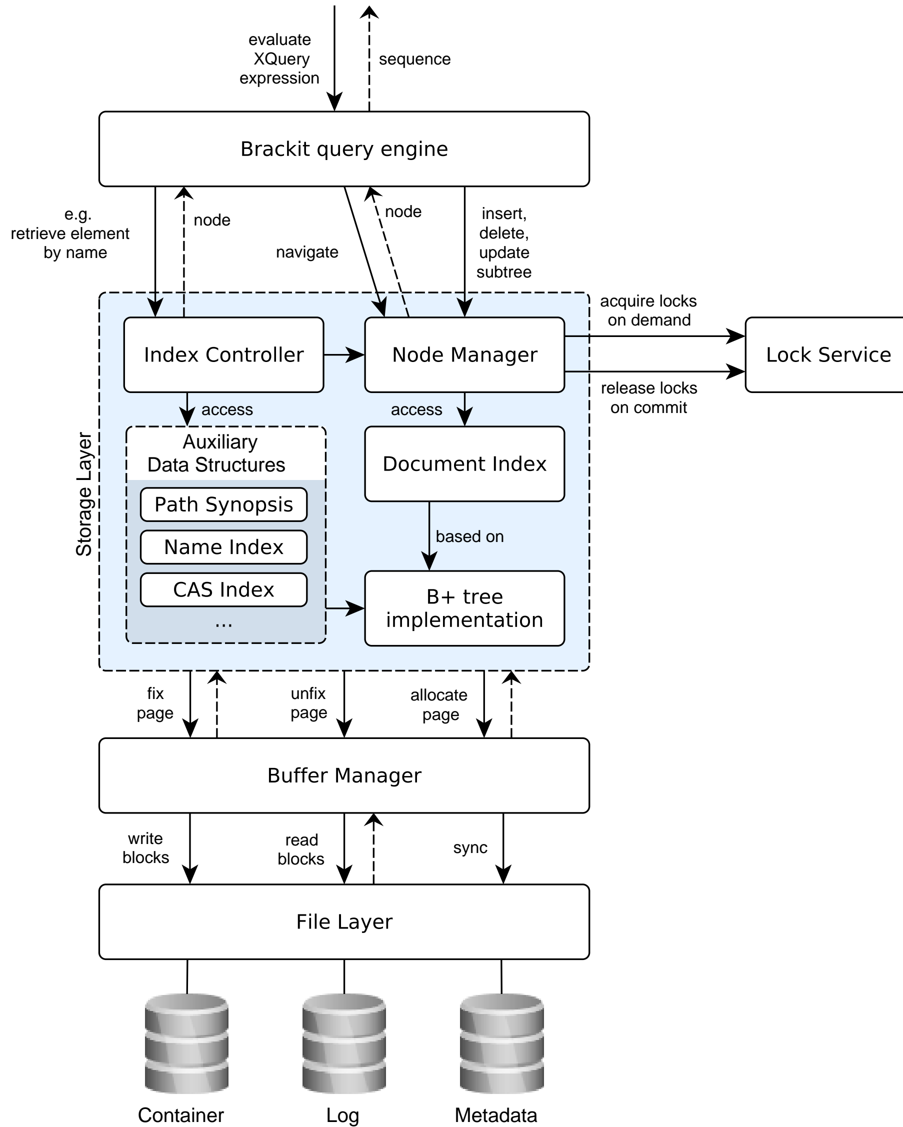


Figure 1: Simplified Architecture of BrackitDB

In addition, each container comes with its own *metadata* file which contains essential management information, such as the chosen block size and a bitmap indicating which blocks are used and which can be allocated.

Finally, the *transaction log* is comprised of a sequence of segment files with a maximal size of 10 MB each. In order to accelerate crash recovery and to prevent these log files from cluttering the whole disk after some time, BrackitDB supports fuzzy checkpoints [23] where the log is truncated as much as possible while still maintaining full support for redo and undo recovery.

Sitting on top of the file layer, the *buffer manager* provides page-based access to the container file and is in charge of transparently loading requested pages from external storage into main memory if they are not already kept there. Although it tries to keep as many pages in memory as possible for faster access, the buffer manager eventually reaches the point where all designated buffer frames are already occupied by other pages. In this case, it has to select one or more

pages for *eviction*, which means either simply discarding the page from its slot, or writing it back to the persistent storage device if it was modified by then. Furthermore, BrackitDB's buffer implementation applies a very basic *prefetching* approach so that page misses usually trigger the loading of several continuous pages from external storage in one go.

2.2 Storage Layer and Locking

The next layer on our bottom-up introduction is the storage subsystem which is concerned with managing XML documents and related index structures, and providing node-oriented access to them. The primary data structure to store documents (or collections of documents) is commonly referred to as the *document index*. In a nutshell, the document index is a B^+ tree where each single XML node is stored as a data record in one of the tree's leaf pages, while the branch pages merely serve as a guide to find nodes by their respective ID [31, p. 159].

In order to identify nodes uniquely in BrackitDB, a prefix-based node labeling schema, known as DeweyIDs [30, 20], is applied. A DeweyID is a sequence of integer divisions, usually containing odd numbers starting from 1. While the root node of an XML document always carries the simple DeweyID `1` with only one division, its children are enumerated by the DeweyIDs `1.3`, `1.5`, `1.7`, and so on. More generally, a node carries its parent's DeweyID appended by an additional (odd) division. Therefore, DeweyIDs reflect the actual position of the corresponding node in the original XML trees, which makes them much more useful than just randomly assigned IDs. Even simple offline inspections of DeweyIDs reveal a good amount of information about their respective nodes: The number of odd division values makes up their level in the document, DeweyIDs of ancestors are easy to determine by extracting prefixes, and many relations among nodes, such as the sibling-sibling or the ancestor-descendent relation, are clearly visible. Moreover, DeweyIDs are stable over the entire lifetime of the node (i.e., they do not need to be reassigned at some point). Inserting a new subtree between the nodes `1.3` and `1.5` results in a new DeweyID of `1.4.3` for the subtree root. This overflow mechanism makes sure that the document order is always preserved, no matter how degenerated the XML tree becomes over time.

It is safe to say that DeweyID labeling is one of the outstanding features of BrackitDB, and possibly the most important ingredient in the storage layer. DeweyIDs are not only used as references in secondary indexes, but they also actively support locking. As we will see in later sections, hierarchical locking protocols require the efficient determination of ancestor IDs for applying *intention locking*, which is trivial in case of DeweyIDs due to their prefix-oriented nature.

Apart from the document index, the storage layer comes with a couple of auxiliary data structures, such as the *path synopsis*. Based on the assumption that an XML document usually contains only a limited set of different paths (compared to the total number of nodes), the path synopsis indexes every existing path in the document and assigns an ID to it, the so-called *path class reference* (or PCR). With the path synopsis in place, it is sufficient for the document index to merely store the leaf nodes of the XML tree, together with their related PCR. Inner nodes (i.e., elements), which only include structural information but no actual data value, are fully reconstructable by the PCR values of their physically stored descendents. Hence, this technique is also referred to as *elementless* [21] or *path-oriented storage* [31, p. 166 ff.].

Query processing can be substantially accelerated by exploiting the best suited secondary index for each kind of query. Therefore, the storage layer features a couple of those, such as name indexes (also referred to as *element indexes* [31, p. 197]), which are used to query elements by their name, or CAS indexes [31, p. 201], which allow for content-based lookups (e.g., given a text or attribute value).

Technically, since BrackitDB applies node-oriented locking rather than logical predicate locking, the lock service and all related components belong to the storage layer as well. Since locking will be covered in later sections, it suffices to say that locking-related aspects are encapsulated and hidden underneath the node interface. Whenever an operation is invoked that returns a node object (such as a navigation step), all necessary locks to ensure proper isolation are transparently acquired on demand so that higher layers do not have to concern themselves with locking-related tasks.

2.3 XQuery engine

The top layer in this simplified architecture is constituted by the Brackit XQuery engine which is responsible for compiling and executing high-level XQuery expressions, thereby providing a bridge between set and node orientation. Among others, its task is to determine the most efficient access plan (i.e., sequence of low-level node operations) for a given expression. Other challenges are posed by join processing, aggregation and sorting.

Since the benchmarks conducted in this thesis essentially circumvent the query engine to have more precise control over node operations and locking aspects, the engine is not further discussed at this point.

3 Locking in BrackitDB

Locking is the prevalent technique for concurrency control in database systems focusing on online transactional processing (OLTP). In terms of the well established ACID properties, locking (if properly implemented) ensures transaction isolation and therefore correctness for every possible interleaved transaction schedule passing through the system. Since locking requires a transaction to acquire a suitable lock prior to any data access, the lock manager and the involved locking protocols are major performance drivers in DBMS with respect to parallelism.

In this section, the reader will be guided through the basic concepts of concurrency control and the various degrees of isolation achieved by different locking techniques. While these concepts have been around in the relational database world for a long time, the increased complexity of the XML data model (compared to flat tables) poses new challenges and considerations regarding the locking protocol. In response to these, the taDOM locking scheme was introduced [25], allowing for concurrent document access at the node level. How taDOM works in detail and how it fits into the BrackitDB architecture will be discussed later in this section.

3.1 Basics of concurrency control

Even though transactions are supposed to be atomic according to the ACID paradigm (i.e., the transactions' effects on the database are either entirely persisted, or not at all), they are clearly not atomic from the processor's point of view. Each transaction consists of a number of smaller database operations like read and write operations on certain data objects, which can again be broken down to plenty of CPU instructions. With this in mind, atomicity is not an instrinsic property of transactions, but need to be explicitly addressed by the DBMS implementation. But what are the implications of this (physical) non-atomicity for transaction isolation?

Whenever two or more transactions execute concurrently, their database operations are typically *interleaved*, potentially causing the programs to behave incorrectly and to leave an inconsistent database state behind. The objective of concurrency control is to prevent this kind of *interference* [16, p. 11] or, in other words, *isolate* the transactions from the effects of others.

For a start, let us investigate the naive approach to cope with this situation. If interleaving can lead to errors and inconsistencies, as stated above, one might come up with the idea to avoid transaction interleaving (and thus concurrency) altogether. Consequently, the DBMS would collect incoming transactions in a queue and run them *serially*, like in batch processing, one transaction after another. By doing this, we eliminate the need for any concurrency control mechanisms entirely, because a serial transaction execution is always correct, assuming that the *Correctness Principle* [36, p. 879] holds. This basic assumption states, that in absence of other transactions (i.e., in complete isolation), a transaction always transforms a consistent database into another consistent state.

Although running transactions in a serial fashion is not practical for real world systems, as it would make poor use of its resources [16, p. 13] (not to mention on multi-core hardware, which was not even available at that time), it constitutes a suitable baseline for the notion of *correctness* in a concurrent environment, as we shall see later.

3.1.1 Serializability

On the one hand, concurrency should be allowed in a DBMS to utilize system resources properly. On the other hand, we claim a certain degree of correctness from the resulting transaction schedule. As pointed out before, correctness inevitably follows from serial transaction execution, which leads us promptly to the concept of *serializability*.

In the database field, serializability is a well known term and, in fact, the definition of correctness for concurrency control [16, p. 14]. In essence, a (potentially interleaved) transaction schedule is called *serializable*, if both of the following requirements are met [16, p. 13]:

1. The transaction schedule has the same effect on the database as *some* serial execution of the same set of transactions...

2. ... and also produces the same output.

At this point, let us elaborate on the importance of the unobtrusive keyword *"some"* in the foregoing definition. When it comes to serializability, it is irrelevant which serial transaction execution produces the same output and the same database state as the interleaved execution, as long as there *is* one in the first place. Incidentally, the serial order of transactions associated with the serializable schedule, is not necessarily determined by the order in which the transactions arrive at the DBMS, start their work, or commit their changes.

In order to refine the terminology a bit more, we denote two transaction schedules *equivalent*, if they produce the same database state and the same output (a formal definition of the *equivalence* of transaction histories can be found in [16, p. 30]). Applying this term facilitates the definition of serializability: A transaction schedule is serializable if there is an equivalent serial execution of the same transactions.

Following this, we need to work out a strategy to detect whether a transaction schedule is serializable and thus acceptable in terms of isolation, or if it must be discarded by the concurrency control mechanism in place. Therefore, we simply model a transaction as a sequence of read and write operations on abstract data objects (labelled as small letters a, b, c, etc.), finalized by the End of Transaction (EoT, which corresponds to a Commit, or a completed Rollback after an Abort has been triggered). A read operation on a data object o performed by transaction T_1 is denoted by $R_1(o)$, a respective write operation by $W_1(o)$. For the sake of simplicity, let the value of a data object be an integer number, although this is by no means a necessity, but useful for the upcoming examples. As far as validity is concerned, the objects can be of any data type (relational tuple, XML node, etc.).

Evidently, an update operation can be translated into a read operation, followed by a write operation on the same object. Even simpler, deletions can be regarded as plain write operations. Note that we deliberately ignore insert operations for the time being and assume the database to be *static*, that is, all data objects are already present. We will examine later why insertions require special care in the scope of serializability.

After having defined our basic transaction model, we can start to analyze various schedules and identify possible interferences caused by interleaving. In order to do so, we still need to introduce

the notion of *conflicting operations*. Two operations are said to *conflict*, if they operate on the same data and if one of them is a write operation [16, p. 28]. Following this definition, we can distinguish three kinds of conflicts: Read/Write, Write/Read and Write/Write. These conflicts substantially determine whether a transaction schedule is serializable or not, because the order of the involved operations affects the transactions' semantics and the resulting database state. Hence, swapping conflicting operations (i.e., running them in reverse order) is not acceptable to find an equivalent serial schedule.

On the other hand, if two operations read the same data (Read/Read situation) or if they operate on different data objects, changing the order in which they are physically executed in the DBMS preserves the semantics and the final database state. That means, non-conflicting operations can be swapped (as long as the internal transaction order is untouched) to generate equivalent schedules. This way, if it is feasible to find an equivalent schedule, the original schedule must be serializable and thus correct.

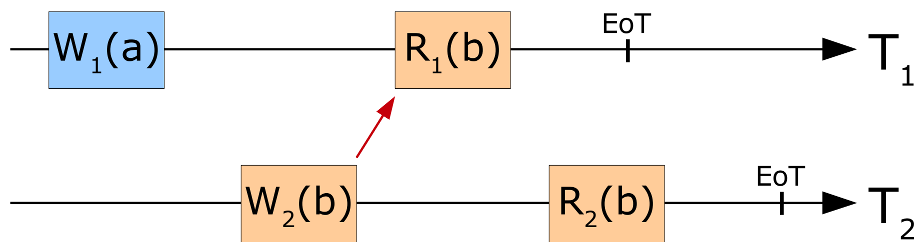


Figure 2: Serializable schedule

Figure 2 depicts a very simple transaction schedule with two transactions T_1 and T_2 running concurrently on the database, and at that, performing read and write operations on the data objects "a" and "b". The temporal placement of these operations is given by the figure's x-axis, i.e., an operation plotted to the right of another operation is said to happen *after* the latter, and vice versa.

Regardless of what exactly the operations read from or write into the objects, this transaction schedule can be shown to be serializable, as it produces the same output and the same database state as the serial execution $T_2|T_1$ (T_2 before T_1). Hence, the equivalent serial schedule performs the operation in the following order: $W_2(b)|R_2(b)|W_1(a)|R_1(b)$.

The only pair of conflicting operations is connected by an arrow, while its direction indicates the temporal constraint that needs to be preserved, i.e., in all equivalent schedules, operation $W_2(b)$ is still performed before operation $R_1(b)$, which is obviously true for the proposed serial order.

Analogous to the definition of serializability, we call transaction schedules *conflict serializable* [16, p. 40][36, p. 925] (commonly abbreviated as CSR) for which a serial order exists that preserves the conflict relation as described above. Bear in mind that a schedule can still be serializable despite not satisfying the criteria for conflict serializability. In fact, the set of conflict serializable schedules is a subset of the serializable schedules (SR), or as formula: $CSR \subset SR$

Conflict serializability imposes stronger conditions on the contained schedules than serializability does, thereby reducing the set size. By simplifying transaction schedules to abstract read and write operations (as done in the CSR model), without further investigating what exactly these

operations do with the data, a CSR-scheduler tends to discard schedules which are, on closer examination, serializable and therefore correct. However, due to the simple model and the easy testing for conflict serializability, most commercial systems implement concurrency control techniques allowing "only" CSR schedules or even a subset thereof [36, p. 925].

For a formulized and complete description of conflict serializability, I refer to [17], [16, p. 25ff.], [36, p. 925ff.], and [37].

3.1.2 Multi-user phenomena

Now that the foundation is laid, we will study some prominent *phenomena* or *anomalies* arising from a lack of adequate concurrency control. These multi-user phenomena are particularly interesting since they form the basis for the various degrees of isolation in a database, as imposed by ANSI SQL.

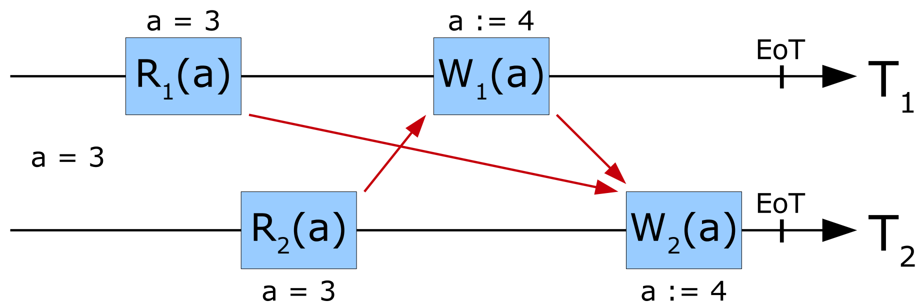


Figure 3: Lost update

The first anomaly we examine is known as *lost update*, exemplified by Figure 3. In this scenario, data object "a" initially holds a value of 3, before two transactions concurrently perform an *increment* operation on this variable. As noted previously, an update is generally not atomic, but implies a sequence of three steps: a read access, a phase in which the retrieved value is processed in some way (in this example, the internally cached value is incremented), and finally a write operation to manifest the new value. Due to unfortunate scheduling, both transactions read the initial value of 3 and thus obtain an incremented value of 4 in their caches. Subsequently, the same value is written back by the two transactions. In the end, "a" was incremented only once even though two increments were intended by the program. The update operation of T_1 is said to be *lost* as it leaves no trace in the final database state due to transaction T_2 overwriting T_1's changeset. Analyzing the schedule in terms of conflicting operations confirms that there is no serial execution order equivalent to the presented schedule.

The multi-user anomaly shown in Figure 4 is referred to as *dirty read* [36, p. 405] and involves a transaction T_2 reading modified data that has not yet been committed by the writing transaction T_1. Before being finalized by a respective commit, this so-called *dirty data* is prone to transaction rollbacks or further modifications by that same transaction.

In the provided example, the value of "a" starts out to be 3 but is soon updated to 5 by transaction T_1, and also perceived as 5 by T_2. However, instead of committing this new value, T_1 is aborted for some reason (e.g., requested by the user himself) and consequently, "a" is reverted

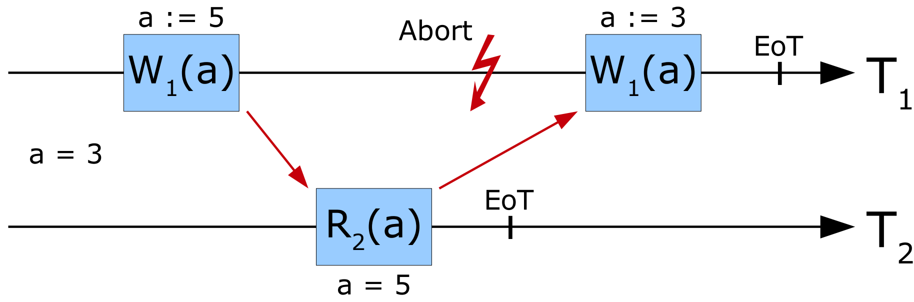


Figure 4: Dirty Read

back to its initial value 3 by corresponding recovery mechanisms. Although T_1 ultimately had no effect on the database, T_2 was able to catch a glimpse on this (dirty) intermediate value and might have returned it to the user, thereby exposing an inconsistent database state.

Another prominent anomaly for concurrent transaction workloads is the *non-repeatable read* [22, p. 410] where a transaction is able to see two different states of the same object during its execution, which is caused by a concurrent update transaction. Or in other words, reading an object repeatedly within one transaction could reveal different values. As a matter of fact, even if the transaction does not access the same object again after its read operation, it could perceive an inconsistent picture of the database, such as illustrated by Figure 5.

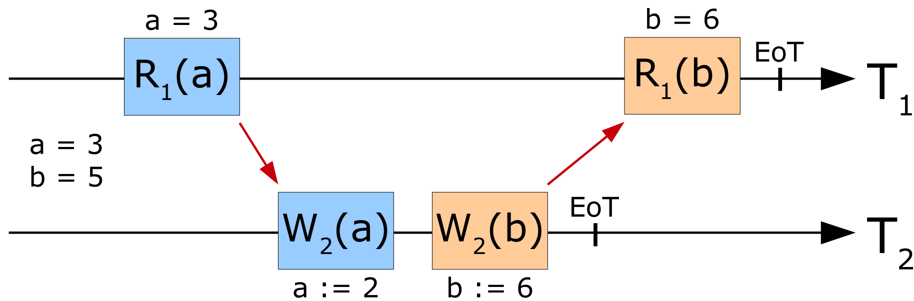


Figure 5: Inconsistent Analysis

Initially, "a" is set to 3 and "b" to 5. Update transaction T_2 modifies the database by decrementing "a" and incrementing "b", accordingly. Hence, in every transaction-consistent database state, the sum of both objects is 8. However, T_1 reads data item "a" before it is modified and "b" after it has already been updated. As a result, transaction T_1 would draw the wrong conclusion that both values add up to 9. The schedule described here is also referred to as *inconsistent analysis*. Take note that T_1 does not read dirty data at any time.

The last phenomena we will discuss in this context are called *phantoms* and are arguably the hardest interferences to deal with. Up to this point we assumed the database to be static, i.e., no new data objects were inserted. In the real world, however, databases are dynamic (i.e., they shrink and extend from time to time), which is where phantoms start to play a role [16, p. 65].

To demonstrate the effects of phantoms in a database with only three data objects ($a = 1$, $b = 2$, $c = 3$), consider a schedule in which transaction T_1 calculates the average value of these items. Now let us assume that T_1's algorithm splits this calculation in two separate database scans. In

its first scan, T_1 sums up the object values, resulting in 6, while in the second iteration, T_1 simply counts the number of objects in the database, which equals to 3. Dividing the first number by the second yields the average value, which is obiously 2. But think about an unfortunate schedule where a second transaction T_2 inserts a new object $d = 4$ between the two scans performed by T_1. In this case, T_1 still computes a sum of 6 (sum over a, b and c). However, the second pass suddenly reveals an object count of 4, thereby misleading T_1 to the false conclusion that the average value amounts to 1.5, which is not true for any transaction-consistent database state.

As a matter of fact, the phantom problem exceeds the expressiveness of the simple transaction model we have used so far. In terms of conflict serializability, phantoms are undetectable, because they do not introduce any conflicting operations. Our CSR model would infer that the example schedule from above is equivalent to the serial execution of $T_2|T_1$. However, due to limited knowledge about T_1's algorithm, it ignores the fact that for the suggested serial order, T_1 would already have read the inserted object d during its first iteration, leading to a consistent picture once again. To overcome this issue, the transaction model needs to be extended to explicitly allow for *scan* operations, as performed by T_1 in the example, which will however not be discussed within the scope of this thesis.

3.1.3 Isolation levels

The widely-adopted SQL standard for realtional database systems defines four isolation levels or *degrees of consistency* which are essentially specified in terms of the multi-user phenomena from above. The main idea is that every transaction may decide for itself which degree of consistency is required and which kinds of anomalies are acceptable for the application. By choosing lower isolation levels, application designers can trade throughput for correctness [15, p. 1]. After all, reduced consistency implies that more transaction schedules are considered acceptable by the concurrency controller, which therefore leads to increased parallelism.

From lowest to highest isolation level, `READ UNCOMMITTED` allows any kind of anomaly discussed. As the name indicates, the next level `READ COMMITTED` ensures that data read by the corresponding transaction is always committed, thus preventing *dirty reads*. Increasing consistency a step further, the isolation level `REPEATABLE READ` additionally avoids *lost updates* and *non-repeatable reads* from happening. Only the hightest isolation level, appropriately named `SERIALIZABLE`, rules out phantoms.

Despite not an SQL database, BrackitDB offers the same isolation levels, which can be selected on transaction begin.

As a side note, some database systems provide isolation levels with slightly deviating consistency guarantees. For instance, the level `CURSOR STABILITY` known from DB2 [7], is situated between `READ COMMITTED` and `REPEATABLE READ`, and was specifically designed to prevent *lost updates* [15, p. 6].

3.2 Introduction to Locking

After discussing the purpose and challenges of concurrency control, we will pursue the question how locking helps to achieve serializability in a concurrent environment. First of all, locking is not the only approach for implementing concurrency control. In fact, a variety of different mechanisms can be found which aim for a proper balance between correctness on the one hand and high parallelism on the other. Generally, concurrency control techniques can be classified as either *pessimistic* or *optimistic*. While pessimistic (or *conservative)* schedulers tackle serializability by delaying transaction operations in order to avoid inconsistent schedules in the first place, optimistic (or *aggressive*) approaches try to schedule operations immediately. By doing so, optimistic schedulers take the risk of forcing transactions to rollback if the resulting schedule turns out to be non-serializable [16, p. 47/48].

3.2.1 Basic RX Locking

According to the previous classification, locking clearly belongs to the pessimistic techniques. The general requirement in a locking-based system is that a transaction always has to obtain a lock on the respective data item before it is allowed to work on it (i.e., perform read or write operations). Since only one lock exists for each data item, it is guaranteed that a particular object is accessed by at most one transaction at any time. If we further assume that every transaction can be split up into two phases, while locks are only acquired in the first phase and only released in the second phase (known as two-phase locking or 2PL), conflict serializability of the resulting schedule can be proven [16, p. 53 ff.]. Bear in mind that 2PL only works under the premise that transactions always commit successfully. Otherwise, a stricter protocol (*strict 2PL* [22, p. 418]) is required where locks are only released after transaction commit is ensured.

Although locking data objects exclusively in a 2PL fashion achieves (conflict) serializability, it is by far stricter than it needs to be. For instance, allowing two transactions to read the same data at the same time would not sacrifice serializability because read operations are not conflicting in terms of CSR. However, not protecting read accesses at all would introduce read/write and write/read conflicts and thus permit *dirty reads* and *non-repeatable reads*.

Therefore, to enable increased parallelism without renouncing serializability, more sophisticated locking protocols offer several *lock modes*. The two modes encountered in almost any locking scheme are the *exclusive* mode X and the *read* mode R (or alternatively S for *shared* mode). It lies in the responsibility of the corresponding locking protocol to define the *compatibility* among the introduced lock modes, which is usually done in the form of a compatibility matrix. This matrix has to answer the question whether a lock mode requested by transaction T_1 is compatible with the lock mode already held by T_2. If the answer is positive, the lock request can be granted. Hence, in a multi-mode lock protocol, a data item can be locked multiple times by different transactions. Depending on the concrete implementation, a lock manager might even allow a transaction to hold several lock modes on the same object (e.g., both X and R at the same time). In BrackitDB, however, the underlying assumption applies that any transaction can only hold at most one lock on each data item.

3.2.2 Lock Conversion

The previous discussion leads us to the question how the locking manager deals with the situation where a transaction first reads data and later decides that it needs to modify the same data. Complying to our simple RX locking protocol, the transaction would first acquire a read lock prior to the actual read operation, and would later try to obtain an exclusive lock. In this situation, the lock manager has to perform *lock conversion*. As a result, the read lock already held by the transaction is *upgraded* to an exclusive lock, given that the X lock is compatible with all the other granted locks on this object. If not, the conversion is delayed until this condition is met, just like for new requests.

In the previous example, it was quite evident that an R lock followed by an X request would lead to an X mode after conversion. In general, however, the locking protocol has to explicitly define how the conversion is performed, i.e., which mode results from which combination of held and requested lock mode. Again, this relation can be defined in terms of a matrix, namely the *conversion matrix*. In a manner of speaking, conversion has to yield a lock mode that covers both the current and the requested mode or, in other words, conversion generally makes the acquired lock stronger. Obviously, an obtained X lock is not downgraded to R when an object is read after being written to, because this would expose uncommitted data to other readers.

3.2.3 Update Locks

Another commonly used lock mode next to the basic RX modes, is the *update* lock U [36, p. 945 f.]. It gives a transaction the privilege to read an item but not to modify it. At that, the U mode is pretty similar to the read mode. However, an update lock can both be downgraded to R or upgraded to X, depending on the value the transaction reads from the object. The advantage of upgrading an U instead of an R lock, stems from the fact that a U→X upgrade does not involve the risk of a conversion deadlock. Upgrading an R lock to X, on the other side, leads to a deadlock whenever this upgrade is performed by two transactions at the same time, which is a perfectly valid scenario due to the R mode being compatible with itself. In contrast, an item can only be update-locked once, while a second update transaction would have to wait until either the first transaction downgrades to R or commits altogether (if it decided to upgrade and modify the data). In order to avoid indefinite delays for U→X upgrades, i.e., the conversion is never granted due to continuously incoming readers, the update lock mode is often defined asymmetrically. As a result, U is compatible to existing R locks but new readers have to wait for the update transaction to finish or downgrade.

3.2.4 Locking Challenges

Although it was stated earlier that locking techniques avoid incorrect (i.e., non-serializable) transaction schedules before they even occur, locking does not guarantee liveness. If read or write operations from the participating transactions arrive at the database system in an unfavorable order, some (if not all) transactions might end up in a deadlock, waiting for other blocked transactions to release their locks. As long as transaction are allowed to access data items in an arbitrary order (i.e., without any protocol), deadlocks are bound to occur in classical 2PL systems [16, p. 52].

Another big challenge for locking techniques is phantom prevention. As a matter of fact, the two-phase RX locking described previously does not solve the phantom problem and hence only provides `REPEATABLE READ` isolation at most. Protection against phantoms would technically involve locking data items that do not exist but might be inserted later [36, p. 961]. As a consequence, phantom protection requires additional mechanisms on top of a classical lock manager, e.g., locking based on logical predicates.

3.3 Hierarchical Locking

Up to this point, we modeled a database as a set of generic data items holding some values, and discussed how appropriately locking these objects achieves serializability. However, we did not say any words about the granularity of these items. In terms of relational databases, a lockable object might be a particular column value of a tuple, a whole record, a table, or even the entire database. The good news is, the serializability considerations and locking techniques from above apply in any case. Thus, the granularity of data items does not affect correctness in any way [16, p. 69]. But what it does have an impact on is the system performance. Finer granularities are more precise and improve parallelism, in that the transactions only lock what they actually access. On the other side, we have an increased locking overhead due to the higher amount of involved locks that need to be acquired and managed. Hence, choosing a suitable lock granularity requires finding a balance between locking overhead and degree of concurrency [16, p. 70].

But instead of statically deciding on a global lock granularity, we can even allow different granularities at the same time to achieve an optimal locking behavior for each transaction or operation. For instance, a transaction scanning through a whole table would ideally request a shared lock on the table, while a transaction selecting a handful of tuples satisfying a certain condition, might better only acquire locks on these specific records and therefore enable other transactions to concurrently modify other tuples of that same table.

The technique that deals with locks of different granularities is called *multi-granularity locking* (MGL) and even dates back to the seventies [19]. For a start, let us assume the database is hierarchically structured in terms of granularities, i.e., each granule is contained in a coarser granule on the next higher level, so that the database can be regarded as a tree of lockable items (MGL also works on directed acyclic graphs, but for our purposes, tree structures are sufficient). In relational systems, this tree is intuitively induced by the database/table/tuple hierarchy. However, lock granularities can also be defined based on rather physical characteristics, such as database/container/page/record. In order to decide on lock compatibility on different hierarchy levels, MGL introduces a new class of lock modes, called *intention* locks.

Intention locks are acquired for inner (non-leaf) nodes to indicate that the owning transaction intents to read or modify an item further down the tree, i.e., on a finer granularity level. For the basic RX locking scheme, its hierarchical generalization therefore includes two new lock modes: IR (for *intention read*) and IX (for *intention exclusive*). An R (or X) request on a node is processed according to the following algorithm: For each ancestor of said node, starting from the root, obtain an IR (or IX) lock. Once every ancestor is appropriately intention-locked, acquire an R (or X) lock on the actual data item. The resulting locking situation might look like depicted in Figure 6. Note how R/X locks also protect the underlying subtree.

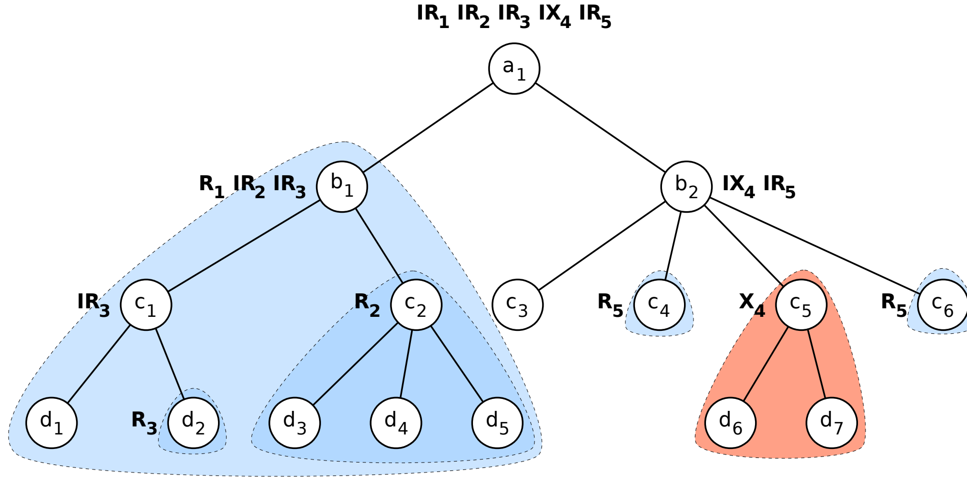


Figure 6: Possible locking situation for hierarchical RX protocol

Essentially, locking ancestors with intention locks ensures that lock compatibility can always be verified locally based on a single node. For instance, imagine there were no intention locks and a transaction requested a read lock on a table, while another transaction held exclusive locks on some table records below. The lock manager could not decide whether to grant the R lock on the table without checking compatibility with each record contained in that table, whereas, with an IX lock present on the table, the lock manager would immediately deny the R request as long as the other transaction modifies data on record granularity.

Another mode often supplementing the above described hierarchical lock protocol is RIX [22, p. 429], combining the privileges of R and IX locks in one mode. RIX improves the common case where a transaction reads through a set of data items (e.g., a table) but only modifies a fraction of these finer-grained objects (e.g., tuples). Without RIX, the transaction would either have to lock plenty of items on the fine granularity level with R or X modes, or rather lock the whole subtree exclusively. As a matter of fact, unifying lock modes for frequently encountered locking scenarios is a cheap way for enhancing concurrency in a system that restricts transactions to hold at most one lock per item, like in BrackitDB. The compatibility matrix for MGL including the RIX lock mode is given in Table 1.

	R	**X**	**IR**	**IX**	**RIX**
R	+	-	+	-	-
X	-	-	-	-	-
IR	+	-	+	+	+
IX	-	-	+	+	-
RIX	-	-	+	-	-

Table 1: Compatibility matrix of RIX protocol

Up to here, we took for granted that transactions magically knew which lock on which granularity would achieve the best result in terms of both locking overhead and concurrency. In reality, though, identifying the most appropriate locks for a sequence of data operations is by far not trivial. Evidently, the more knowledge about the actual transaction we have, the better we can optimize the lock request pattern. On the one hand, the goal is to acquire enough locks

to cover all data accesses adequately, but on the other hand, the total amount of locks should be minimized. However, locks are usually not explicitly requested by the database application itself, since ensuring correctness in a concurrent environment is the duty of the database system. Instead, lock acquisition is automatically triggered when the transaction performs read or write operations on certain data items. Unfortunately, the lock manager does not see the whole picture and is thus not able to perfectly predict upcoming data accesses. The choice of the appropriate lock granularity is therefore based on the transaction's recent access behavior.

A heuristic approach to tackle this challenge is called *lock escalation* [16, p. 75]. In essence, the transaction starts out obtaining locks on the finest granularity (e.g., records in a relational system) until the number of held locks in a particular subtree (e.g., records belonging to the same table) exceeds a certain threshold. Once this is the case, lock requests are escalated to the corresponding parent node in anticipation that plenty more data items in that subtree will be accessed in the future.

It should be noted that hierarchical locking schemes provide phantom protection to some extent, if the lock granularity is cleverly chosen. For instance, when a transaction scans a table multiple times, but requires for its correctness that all iterations retrieve the same set of records, read-locking the entire table instead of every single record avoids unwanted insertions during its processing.

3.4 taDOM Isolation

So far, the discussed correctness principles and locking approaches are not bound to any particular data model. Although they were originally invented for relational database systems, the above-described techniques can be conveyed and adjusted specifically to XML data. In terms of its structure, access operations, and querying capabilities, the XML data model is undeniably more flexible and complex than a set of tuples organized in flat tables. Therefore, XML data requires more sophisticated transaction isolation mechanisms than presented thus far, especially when it comes to phantom protection.

In order to cope with these challenges, taDOM (transactional DOM) was designed [27, 25, 24, 28]. It extends the well-known DOM API [41] for XML documents by incorporating the core concepts from hierarchical locking and adding tailor-made, fine-grained lock modes to allow for efficient transaction isolation on the node interface. DOM itself is an API that represents an XML document as a tree structure of node objects and defines various operations on these nodes, such as navigation to related nodes (e.g., next sibling or first child), document manipulation (inserting, modifying or deleting subtrees) and even basic querying (e.g., retrieve an element by its ID attribute). However, DOM is designed for single user environments [27]. If multiple users work on the same XML file, they will merely operate on their local DOM tree constructed in their respective main memory, rather than collaborating with others. In contrast, taDOM lets multiple users access the same document tree concurrently while maintaining serializability.

3.4.1 Node Locks

taDOM establishes a hierarchical locking protocol based on the document tree by mapping XML nodes to granules in MGL's terminology. Before a node can be accessed (by means of traversal or modification), the transaction must obtain an appropriate lock. Apart from the lock modes we already learned about, taDOM introduces a couple of additional lock modes specifically designed to increase parallelism. As mentioned earlier, read and write locks in MGL always protect the entire subtree from concurrent accesses, which is why their taDOM counterparts are denoted SR (*subtree read*) and SX (*subtree exclusive*), respectively.

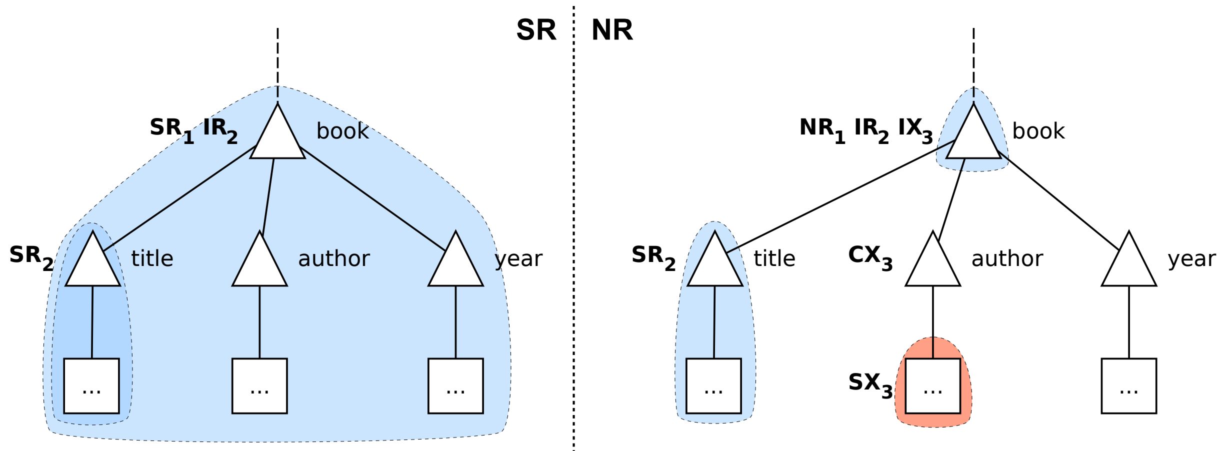


Figure 7: NR lock mode for increased parallelism

In many cases, however, locking the whole subtree when only one node is effectively accessed is an unnecessary restriction that reduces parallelism noticeably. For this reason, taDOM offers a new lock mode NR specifically for reading nodes only. Essentially, the NR mode is implicitly acquired for all nodes returned by any traversal operation (e.g., next sibling) which makes it the most frequently encountered lock mode in many scenarios. NR grants the transaction the privilege to read the node's name (for elements or attributes) and its value (for attributes and text nodes) and furthermore prevents any of these nodes to be deleted concurrently. In contrast to SR, this new lock mode does not affect descendent nodes in any way so that modifications in the underlying subtree are allowed, as demonstrated in Figure 7. For the moment, treat the CX lock mode like a special variant of IX, which will be explained later.

The second novel lock mode employed by taDOM is LR (*level read*) which is obtained for operations like `getChildren()` or `getAttributes()`, and prevents concurrent modifications of the context node itself and its children (but none of its other descendents). With that, LR behaves similarly to a collection of individual NR locks on the context node and each of its children, but only has to be issued once on the respective parent. But aside from the reduced locking overhead, LR features a built-in phantom protection, since child insertions are impossible as long as LR locks are active. One possible use case of a level read lock is depicted in Figure 8. Transaction T_1 iterates over all book elements in the library document, possibly looking for a specific book identified by one of its attributes. At the same time, though, another transaction T_2 is allowed to modify a subtree located on a deeper level of this document. Since read locks are always compatible to each other, another transaction T_3 might hold an SR lock on one of the respective children, thus leading to overlapping lock scopes.

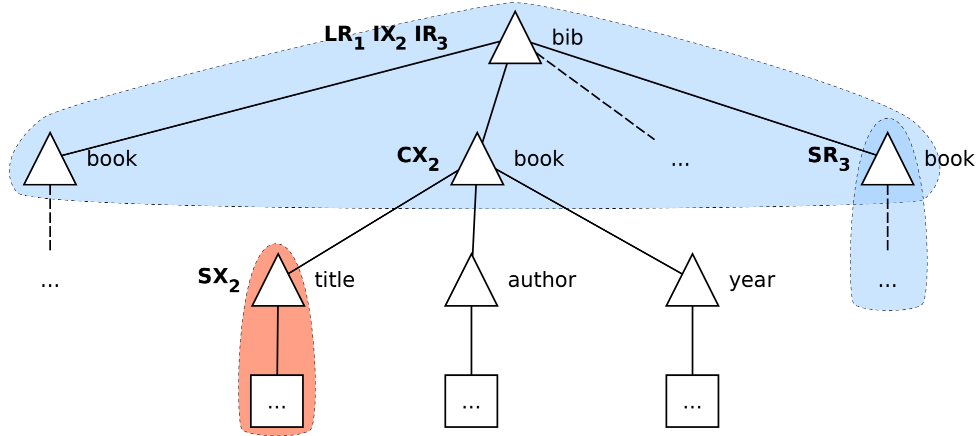


Figure 8: Demonstration of LR lock mode

As pointed out before, it is essential in a hierarchical locking protocol to be able to decide whether an incoming lock request can be granted or not, by simply inspecting the locking situation at the local node. Obviously, the need to examine the underlying subtree leads to unacceptable locking overhead. With regard to the new LR lock mode, the lock manager has to determine whether one of the child nodes is exclusively locked by another transaction before it can grant the LR request. We are already aware of IX locks which are obtained to indicate write activity in one of the descendent subtrees. However, not allowing LR locks on nodes with granted IX locks is again too restrictive in terms of parallelism, since chances are that the X-locked node resides on a deeper level in the corresponding subtree. To this end, taDOM stipulates that the parent of an exclusively locked node must be locked in CX (*child exclusive*) mode, while any other ancestor is still tied to the more generic IX lock mode. By instituting this new intention lock mode, taDOM can allow LR and IX to be compatible, whereas LR and CX are contending.

Another lock mode not explicitly addressed so far is the SU (*subtree update*) lock mode. In its functional principle, SU behaves equivalently to the update lock in the non-hierarchical locking scheme. SU allows for value-dependent updates and therefore supports downgrades to SR or upgrades to SX, respectively. It may be interesting to note that SX requests are always preceded by SU requests in BrackitDB's current implementation, even if the upcoming lock upgrade is already decided and imminent. The reason for this seemingly unnecessary SU lock is related to *instant lock granting*. Whenever a lock request arrives at some node and this request turns out to be compatible with the node's current lock mode, the request is granted without checking the lock request queue for other pending requests. As a result, an SX lock request may never be granted in a hotspot region where new SR requests arrive continuously. To compensate for this "unfair" scheduling behavior, write transactions always start out acquiring an SU lock that is compatible with existing SR locks but prevents newly requested SR locks to be granted instantly.

Finally, the compatibility of taDOM's lock modes discussed up to now is presented in Table 2. It should be mentioned that this compatibility matrix is only part of the truth. As a matter of fact, taDOM constitutes a whole family of locking protocols which become increasingly complex in exchange for improved concurrency characteristics. The lock modes introduced so far make up the taDOM2 variant which is aimed for fine-granular transaction isolation on top of operations defined in the DOM Level 2 standard [39]. However, it does not perform well for some of the

	IR	NR	LR	SR	IX	CX	SU	SX
IR	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	-	-
LR	+	+	+	+	+	-	-	-
SR	+	+	+	+	-	-	-	-
IX	+	+	+	-	+	+	-	-
CX	+	+	-	-	+	+	-	-
SU	+	+	+	+	-	-	-	-
SX	-	-	-	-	-	-	-	-

Table 2: Compatibility matrix of taDOM's core modes

operations specified by DOM Level 3 [40]. For instance, renaming an element node requires the acquisition of an exclusive lock on the entire subtree due to the lack of better suited lock modes, which is why outsourcing element names into *virtual name nodes* is proposed for taDOM2 [25]. In contrast, taDOM3 resolves this matter by establishing new lock modes specifically designed for single node modifications (NU for *node update* and NX for *node exclusive*) which behave just like the NR mode in terms of their scope.

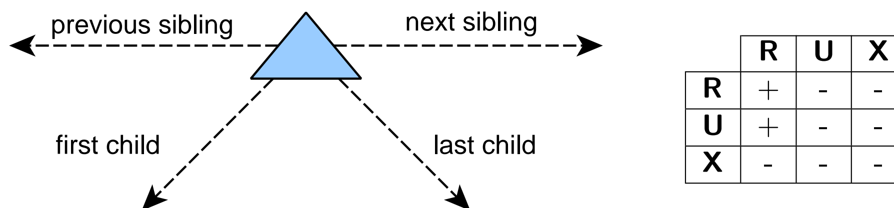
Moreoever, the basic variants taDOM2 and taDOM3 can quickly end up in coarse-granular locking conditions whenever lock conversions take place. As explained earlier, conversions become necessary when a transaction requests different locks on the same node (or subtree) over the course of its lifetime. Since a transaction can at most hold one lock per node, the obtained and the requested lock mode need to be merged into a mode strong enough to imply both original locks. If no appropriate conversion mode is provided by the instantiated protocol, subtrees rapidly become exclusively locked. For this reason, taDOM2+ and taDOM3+ additionally introduce combinations of the lock modes previously discussed, such as LRIX, SRIX, LRCX and SRCX. The semantics of these hybrid modes can be interpreted as the logical conjunction of both, i.e., their row and column entries in the compatibility matrix is simply determined by applying the `AND` operator on the row/column of the original lock modes.

Another scenario where these hybrid modes are beneficial is when separate transactions hold different but compatible lock modes on the same node, such as LR and IX. Since incoming lock requests are not verified against each individual granted mode from the queue but instead against the overall lock mode (which sums up to SX for a granted LR and IX lock), further concurrency is essentially obstructed even if permissible in terms of serializability. Again, hybrid lock modes, like LRIX in this example, help to maximize parallelism.

3.4.2 Edge Locks

The locking mechanism addressed so far, however, is not sufficient to ensure serializability. After all, correct transaction isolation implies that a transaction can read the same data multiple times and will always yield the same result. Reading data in the context of XML mostly involves navigation, i.e., stepping from one node to another along one of the axes, such as `Next-Sibling` or `First-Child`. With this in mind, assume a transaction iterates over a node's children by first retrieving its first child, and then continuing to traverse along the `Next-Sibling` axis until it arrives at its last child. Despite the appropriate acquisition of node locks on every node passed, as explained above, repeating the same sequence of navigation steps at a later time (but within the same transaction) could potentially return phantoms which have not been there the first time. Be aware that in this particular case phantoms can be easily prevented by utilizing the LR lock mode, but in general there is not a perfect phantom-free lock mode for every conceivable sequence of navigations.

For this reason, taDOM employs a second layer of locking which functions on top of the node locking mechanism. These locks are called *navigation locks* [27, 29], or more commonly *edge locks* [25, 12, 11, 13, 14].



	R	U	X
R	+	-	-
U	+	-	-
X	-	-	-

Figure 9: Navigation edges and related lock modes

Figure 9 depicts different edges for which an appropriate lock must be acquired before the corresponding navigation step (e.g., `getNextSibling()`) can be performed. Since edge locks are inherently non-hierarchical, as opposed to node locks, edge locking settles for the basic RUX lock modes. While R is required to traverse along an edge, the U/X combination allows for edge modifications. For instance, when a transaction inserts a new node N_{new} between nodes N_{left} and N_{right}, it needs to hold an X lock on both the `Next-Sibling` edge of N_{left} and the `Previous-Sibling` edge of N_{right} because both edges will point to a different node after the insertion. In consequence, if another (serializable) transaction already traversed over these edges and has therefore perceived N_{left} and N_{right} as adjacent sibling nodes, then this image will be preserved until it eventually commits and releases its R edge locks.

3.4.3 Index Locks

With the combination of node and edge locks established so far, taDOM only achieves serializability if all transactions start executing at the root node and only make use of navigation steps to perform their workload. However, document entry from any secondary index still involves the risk of phantoms. For instance, retrieving a list of element nodes with a specified name from the *name index* [31, p. 197] causes every returned node to be NR locked. While this protects the elements from being renamed or deleted, other transactions might still insert new elements

with matching names. Hence, opening the name index again at a later time with the same query parameters (i.e., given element name), might fetch an extended list of element nodes, including phantom elements that have been inserted concurrently.

In order to prevent this kind of anomalies, BrackitDB implements Key-Value Locking (or KVL [32]) for all B^+ trees employed as secondary indexes. In a nutshell, KVL aims not only for locking single index entries but also protects key ranges traversed by index scans. Therefore, KVL effectively prevents phantoms resulting from secondary index accesses.

Although the above-mentioned approach on top of node and edge locks finally achieves serializability, index locks are considered to be too restrictive in the scope of XQuery and DOM predicates, thereby reducing concurrency more than necessary. For this reason, a different approach to deal with phantoms in taDOM, called *value-based axes locking*, was suggested in [26] but will not be further explored in this thesis.

3.4.4 Lock Escalation

We briefly sketched the purpose of lock escalation in the context of MGL in section 3.3. As a matter of fact, lock escalation is not a technique specifically designed for taDOM, but constitutes a general mechanism in hierarchical locking for finding the proper balance between lock overhead (mainly defined by the number of locks that have to be managed) and level of parallelism (determined by the lock granularities in use). However, lock escalation is deemed an integral part for an efficient taDOM implementation, so that a closer look into its concrete realization in BrackitDB seems appropriate.

The basic problem in MGL and taDOM is indeed the same, namely to figure out which level of granularity provides the best performance. If no lock escalation takes place, taDOM will always assign the finest possible granularity for a specific operation. For instance, reading or navigating over a single node will only acquire a *node read* (NR) lock on said node, while a subtree scan (by means of the `getSubtree()` operation) will instantly lock the whole underlying subtree by obtaining the SR mode. Hence, the granularity is already determined by the kind of operation the transaction invokes.

But imagine the case where a transaction accesses thousands of nodes successively in a low-traffic subtree of the document. Although there is no other transaction present that could interfere, a lock must be acquired for each single node in compliance with the lock protocol. The result is a huge locking overhead for no actual benefit. This is a scenario where lock escalation can prove useful.

In BrackitDB, we can distinguish between two kinds of lock escalations. The first one describes a static *lock depth* [27, 29], which determines the maximal level for which locks are obtained in general. For a lock depth of 0, a transaction will only go for the root node lock, whereas lock depth 1 involves the root node and its children, and so on. If a node is accessed which is located deeper in the document tree than the lock depth allows, its corresponding ancestor will be locked instead. Keep in mind that escalating a lock request to an ancestor potentially widens the lock mode as well. Escalating mode NR upwards the hierarchy results in an SR of the related ancestor subtree. Consequently, if it is known for a certain document on which level the transaction contention usually takes place, the lock depth should be fixed to a value slightly

higher than the hotspot location. That way, fine granularities are used in these high-traffic areas, while everything underneath is locked coarsely.

The other kind of lock escalation in BrackitDB works dynamically at runtime. Apparently, the overwhelming drawback of the lock depth is its inflexibility. Every part in the document is affected by the same static lock depth, even though some parts might accommodate a hotspot region deeper down the tree, while other parts might not even be accessed at depth 1. For this reason, dynamic lock escalation (mostly just called *lock escalation* as such) only performs escalation when it detects a high degree of locality in a certain subtree.

Incidentally, tracking locality is a cheap task, since it takes advantage of the fact that ancestor locks have to be acquired anyway. While performing intention locking, an internal lock request counter is incremented for the direct parent of the requested node, thereby indicating that one of its child nodes is accessed. Once this counter exceeds a certain threshold for a node, future lock requests to one of its children are escalated to this node instead. In this manner, locks are being escalated upwards one level after another if the transaction turns out to be very busy in the corresponding subtree. Needless to say, escalation only takes place if the escalated lock mode is compatible with all the other locks held at the parent node. As a result, only subtrees that are occupied by a single transaction bear the potential of being coarsely locked due to locks being escalated, while high-traffic subtrees are totally unaffected by the escalation mechanism.

The above-mentioned threshold is computed from the node's level in the document tree and two adjustable system parameters specifying the aggressiveness of the escalation policy, which is documented in [14]. More specifically, the concrete formula used by BrackitDB has the following shape:

$$threshold = \frac{maxEscalationCount}{2^{level} * escalationGain}$$

Although this formula is primarily a rule of thumb than exact science, the basic underlying consideration is: The deeper a node is located in the document, the lower is the threshold for performing lock escalation. The parameter from the numerator, `maxEscalationCount`, constitutes an absolute basis for the number of locks that need to be held at most before escalation is applied. This value is not changed throughout different escalation strategies. Whereas the parameter from the denominator, `escalationGain`, affects how the threshold changes from one level to the next, i.e., the higher this value is set, the more aggressive is the escalation policy for deeper nodes.

Throughout the benchmarks in this thesis, the following escalation policies are applied:

	maxEscalationCount	escalationGain
moderate	1920	1.0
eager	1920	1.4
aggressive	1920	2.0

3.5 Implementation Aspects

After we treated the locking mechanisms utilized by BrackitDB in adequate detail, let us have a brief discussion about some of the challenges that arise when implementing an efficient, taDOM-aware lock manager. The main obstacles on the way are synchronization issues right up to sporadic, system-wide deadlocks that are very difficult to track down and fix. Be aware that a *deadlock* in this context does not refer to an actual locking-based transaction deadlock which also occur in flawless lock manager implementations. What is rather meant here are deadlocks due to a lack of carefulness when synchronizing access on shared data structures among multiple threads. An ever so minuscule bug in one of the sensitive code sections can cause the system to crash or halt instantly, or it may work as expected in 99 out of 100 cases, but sooner or later the consequences of incorrect synchronization eventually come to light.

3.5.1 Lock Table

The most important data structure in terms of locking is undeniably the lock table, which stores a so-called *lock header* for every lockable object (in BrackitDB, this is a node). This header contains locking-related information about the corresponding lock object, such as the currently granted lock mode (if several modes are granted to different transaction at the same time, this one describes the "strongest" granted mode). But what is more, the header points to its lock request queue which is managed as a linked list, as depicted by the solid arrows in Figure 10.

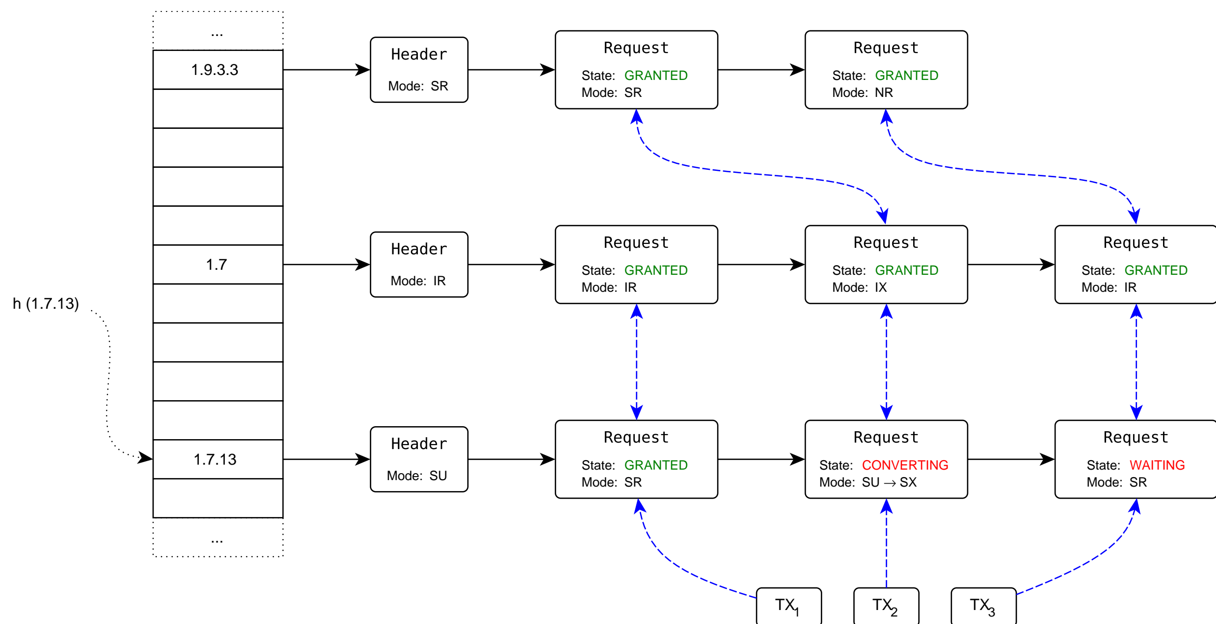


Figure 10: Lock table in BrackitDB

While the header contains a reference to the first request in its queue, every subsequent request again points to the next in the list, accordingly. It should be emphasized that not only pending requests are kept in this list, but also currently granted requests. Therefore, one of the request's attributes reveals its current state (e.g., `GRANTED` or `WAITING`). Depending on this status information, a request includes up to two lock modes: the first lock mode describes the mode

that is currently granted to the transaction (or `null` if the transaction does not hold any lock mode yet). The second one is the lock mode the transaction is trying to obtain. Hence, if both lock mode fields in a request are given, a lock conversion (from the granted to the requested mode) is indicated. Note that in BrackitDB's implementation, a transaction can only issue one request per header. Requesting a second lock mode on the same object at a later time causes its old request to be converted accordingly.

On top of the request queue starting from the header object, there is a second chain of requests connecting all lock requests of a particular transaction in a doubly linked fashion (illustrated as dashed arrows in Figure 10). That means, a request does not only link to the next request in the header queue, but it also holds references to the next and previous request of the same transaction. Evidently, this transaction-oriented chain is useful at commit time for releasing all acquired locks in one go.

For now, we only talked about the static data structures that make up the lock table. Incidentally, the object structure explained above is not only accessed by one thread at a time. Instead, for scalability reasons, the lock table is allowed to be traversed and modified by a multitude of transaction threads at the same time. Therefore, preserving the physical consistency of this data structure becomes one of the biggest challenges from an implementation viewpoint. As a matter of fact, a variety of latches and critical sections are involved when performing operations on the lock table. The implementation of the hash table itself (in Figure 10 portrayed as a set of *buckets* on the left) is based on the design of the `ConcurrentHashMap` class supplied by OpenJDK, to provide a high degree of parallelism within the lock table.

The second means for synchronization is provided by the header objects, which can be read- or write-latched. In the database context, a *latch* denotes a short-lived lock that is used to protect a data structure from concurrent access. They should not be confused with database locks. While latches preserve the physical consistency of some data structure, locks ensure transactional consistency [11]. Latching the header in read mode is needed to get a consistent view of the header state and the request queue. Therefore, changing header attributes or modifying the request queue (e.g., when a new request arrives or an old lock is released) requires the thread to X-latch the header before. Yet another latch that plays a role for the physical consistency of the lock table is provided per transaction, and protects the (dashed) doubly linked request chain from concurrent modifications. As a result, in order to be allowed to enqueue a new lock request, the related thread must first obtain both an X latch on the header and an X latch for the corresponding transaction request list. Failure to adhere to the concurrency protocol imposed by the lock table implementation results in deadlocks and inconsistent lock entries.

3.5.2 Edge Lock Internals

One might have noticed that the lock table format previously presented relies on DeweyID hash values for finding corresponding lock headers. Thus, the question arises how edge locks fit into this architecture. As a matter of fact, BrackitDB implements a *unified* lock service that keeps node and edge locks in the same physical lock table. This is done by introducing an extended DeweyID format which includes an additional "division", called a *tail*. If this tail is set to zero, the related DeweyID describes a node. If it is non-zero, however, the extended DeweyID denotes an edge, while every edge specifies its own specific tail value. For instance, the lock header for the node `1.5.3` is stored under the name `1.5.3#0` where `#` separates the actual DeweyID from its

tail. On the other hand, the `Next-Sibling` edge of the same node is identified by `1.5.3#-4` while `-4` is simply the internal value that was arbitrarily chosen for `Next-Sibling`.

The practical implication of this unified node/edge lock approach is that edge locks must also adhere to the hierarchical locking protocol, i.e., locking some node's `Next-Sibling` edge requires the transaction to obtain the corresponding intention locks on the whole ancestor chain (including the context node). Or from a different perspective, locking a subtree in the document implicitly protects all underlying edges as well. It should be stressed that this hierarchical dependency between node and edge locks does not restrain parallelism on the edges, since navigation over edges imply that the transaction at least holds a shared lock on the context node anyway.

3.5.3 Deadlock Detection

Another major component of BrackitDB's lock management is the *deadlock detector*. As already mentioned earlier, partial or even system-wide deadlocks are bound to occur in locking-based systems. They are caused by unfortunate transaction schedules where two or more transactions wait for each other in a cyclic manner. For this reason, a deadlock resolution mechanism is necessary. In BrackitDB, the deadlock detector (DD) runs periodically in a separate thread so that deadlock resolution is still possible, even when all transaction workers are already blocked. Every time the DD thread wakes up, its task is to crawl through the lock table and to construct a *wait-for* graph on its way. This graph adopts transactions as nodes, and waiting conditions as edges among them. If the wait-for graph turns out to contain a cycle, one of the participating transactions is selected to be aborted.

The decision for a suitable abort candidate is determined by the number of obtained locks. In general, a low number of locks is an indication that the corresponding transaction has not performed too much work yet, so that aborting this transaction does not inflict the same amount of damage than aborting a long-running transaction that is about to commit. After the transaction rollback is triggered, the DD remains active and reevaluates the wait-for graph (possibly aborting further transactions) until the deadlock is finally resolved.

One of the biggest issues with the DD is however caused by its inconsistent view on the actual locking condition in the system. While it builds the wait-for graph, it has to comply with the same latching protocol as every other thread accessing the lock table. As a consequence, the resulting wait-for graph does not reflect a perfectly consistent snapshot of the actual wait-relationships in the system. Sometimes false positives are detected so that perfectly alive transactions are aborted to no avail. Moreoever, under rare circumstances, the DD is not able to find a suitable transaction for rollback in a deadlock cycle because all participating transactions are already committed or rolled back. In this case, the DD ignores the supposed "deadlock" and continues its work.

After the theoretical and practical aspects of locking and lock management have been sufficiently discussed, the remainder of this thesis deals with the preparation and execution of benchmarks that we need to conduct in order to evaluate the performance and effectiveness of locking-related techniques introduced in this section.

4 Benchmark Architecture and Setup

The following section is meant to provide an introduction to the benchmark suite, an extensible framework for the definition and execution of software benchmarks, and how this tool interacts with BrackitDB on an architectural level. Subsequently, the benchmark semantics is discussed by describing the structure and meaning of the actual benchmark document (generated by the XMark tool), and which kinds of transactions produce the workload on the database during the test runs. Ultimately, a quick overview over the hardware and software environment is presented.

4.1 The Benchmark Suite

All experiments conducted as part of this thesis were performed and evaluated by a utility application written in Java, known as the *Benchmark Suite* or *BenchSuite* for short. While mainly developed as an extensible benchmarking tool for XTC/Brackit, the BenchSuite provides ambitious benchmark-related features for any kind of project. For a better understanding of the measurements presented later, this section sheds some light on the BenchSuite's characteristics, its architecture, and how it interacts with BrackitDB in the first place.

In essence, the BenchSuite's features can be summarized as follows:

- XML documents to specify benchmark configurations
- extensibility through custom (e.g., project-specific) actions and work units
- interactive or non-interactive shell for running and managing benchmarks
- basic data browsing capabilities to crawl through the measurements of a stored benchmark
- data aggregation and HTML export
- generation of plots using *gnuplot* [3]

4.1.1 Terminology and Benchmark Layout

In order to sketch how a benchmark is specified, we need some basic insight into the BechSuite's terminology. A *benchmark* describes the most coarse-grained unit of execution and typically takes a few hours up to several days, while running workloads repeatedly under varying system conditions and gathering various measures along the way. In the scope of this tool, a benchmark consists of one or more *series*, which are again split up into different *scenarios*, while each scenario of each series is *run* multiple times to smoothen the results and thus gain more accuracy and stability.

The benchmark layout introduced by the BenchSuite (as illustrated in Figure 11) essentially allows to compare two *variables* and their effect on the system. One of these variables can

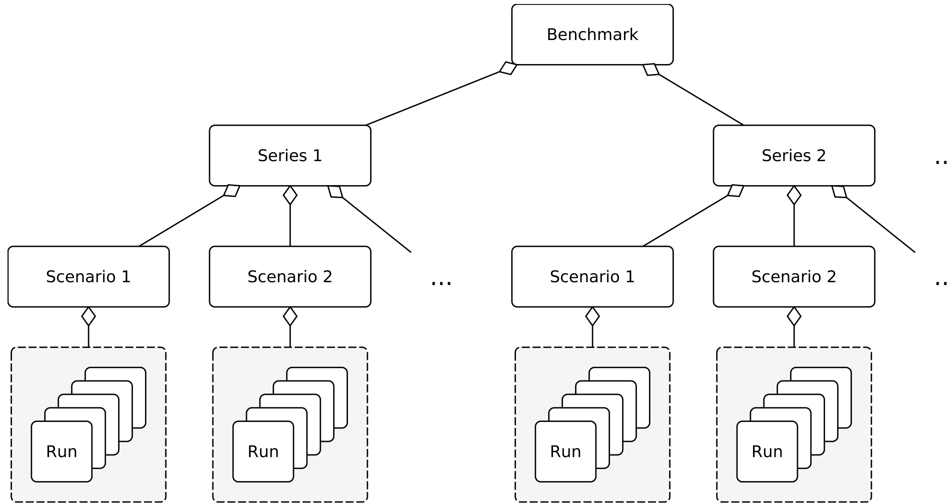


Figure 11: Basic benchmark model in the BenchSuite

change from series to series, while the other is modified from scenario to scenario. Although one might think that the notion of a *Run* establishes a third dimension in our benchmark model, it is however not possible nor desirable to vary the system configuration from one run to the next, as their sole meaning is to yield a larger data sample.

A *variable* in this terminology is an abstract notion, which might stand for a simple configuration parameter of the software being benchmarked (e.g., size of the page buffer in the database system), or it might even describe distinct software systems altogether. For instance, to compare different relational database systems (e.g., DB2, MySQL, Oracle, etc.) under similar workloads, one would possibly dedicate one series to each of these competing systems, while the scenarios might vary the query type. Hence, the effect of two variables ($var1 \in \{DB2, MySQL, Oracle, ...\}$ and $var2 \in \{query1, query2, ...\}$) is investigated within one benchmark. Although technically not necessary, the examined scenarios are similar throughout all series. In terms of comparability, it would hardly make sense, for instance, to investigate an entirely different set of queries in DB2 and MySQL and eventually put the results next to each other and analyze them. In fact, the distinction between series and scenarios is non-essential for the benchmarks conducted in this thesis, as the two confronted variables are always independent from each other, and therefore the hierarchy imposed by the series/scenario distinction can easily be inverted by assigning the *inner* variable to the series and the *outer* variable to the scenario. In the end, we always obtain a two-dimensional matrix of various measurements, while one axis captures the changes in one variable (series), and the other axis depicts the changes in the second variable (scenario), accordingly.

4.1.2 XML Configuration

Not surprisingly, the described benchmark structure is reflected by the XML format of the configuration files, as briefly sketched in Listing 1.

```
<benchmark name="MyBenchmark">
 <globalParameters>
  <!-- set global benchmark parameters -->
  ...
 </globalParameters>

 <!-- define benchmark setup -->
 <setup>
  <series name="DB2">
   <scenario name="query1">
    <beforeRun>
     <!-- define actions that need to be executed before each scenario run -->
    </beforeRun>
    <afterRun>
     <!-- define actions that are executed after each run -->
    </afterRun>
    <agentSetup>
     <!-- specify which agents run which workload -->
    </agentSetup>
   </scenario>
   <scenario name="query2">
    ...
   </scenario>
  </series>
  <series name="MySQL">
   ...
  </series>
  ...
 </setup>
</benchmark>
```

Listing 1: Basic (incomplete) structure of a benchmark configuration file

Up to this point, we introduced the relationship of the terms *benchmark*, *series* and *scenario*. But in fact, the hierarchy goes much deeper. A scenario is again defined by one or more *agents* running potentially different *workloads*, while a workload is composed of smaller *work units* which are either run in sequence, or which are randomly selected according to a predefined weight distribution. Furthermore, at any level of this hierarchy, *actions* can be performed (e.g., as preparation steps) before or after a series, a scenario, or between single scenario runs.

Considering these numerous layers of composition, one can imagine how deeply nested and confusing the configuration file gets for a fully-fledged benchmark definition. Moreover, redundancy becomes an issue for this kind of hierarchical specification. Remember the benchmark example with different queries executed on different database systems. Conforming to the presented XML format requires the repeated definition of the exact same scenario workloads (queries) for each series (database systems).

Fortunately, the BenchSuite features an inheritance mechanism that makes a benchmark configuration much more manageable and non-redundant. This mechanism allows us to specify global

scenario templates, global workloads etc., which are later assembled (or *extended*) to shape the complete benchmark definition. A small demonstration of this templating feature is shown in Listing 2.

```
<benchmark name="MyBenchmark">
  <globalParameters>
    ...
  </globalParameters>

  <!-- define global scenario templates -->
  <scenarios>
    <scenario name="query1">
      ...
    </scenario>
    <scenario name="query1">
      ...
    </scenario>
    ...
  </scenarios>

  <!-- extend the templates to inherit their properties -->
  <setup>
    <series name="DB2">
      <scenario extends="query1"/>
      <scenario extends="query2"/>
      ...
    </series>
    <series name="MySQL">
      <scenario extends="query1"/>
      <scenario extends="query2"/>
      ...
    </series>
    ...
  </setup>
</benchmark>
```

Listing 2: Templating of benchmark components

The not yet mentioned `globalParameters` section incorporates the declaration of global benchmark parameters which can be globally accessed and overwritten at any level. For instance, one might declare common default values in this section, and later vary these parameters from scenario to scenario. The scope of a parameter value is determined by the location of its declaration or alteration. Declaring a parameter on the series level makes its value accessable from the entire underlying series, including its scenarios and work units, while a modification of such parameter in one of the scenarios only applies for the duration of the corresponding scenario. From a technical perspective, parameters are resolved in a bottom-up fashion by traversing the ancestor chain, until the respective parameter declaration is found. Incidentally, a parameter is basically untyped and therefore simply defined by an empty XML element with the two attributes `name` and `value`.

4.1.3 BrackitDB-specific Extensions

Aside from the powerful templating and parameter mechanism, the BenchSuite's strength originates from its extensibility in conjunction with Java. In its main development branch, the BenchSuite only provides very few generic *actions* and *work units*, which can be chosen from to put a benchmark together. Most notably, there is a default work unit that runs an arbitrary command as an external process, thus allowing for benchmarks of any kind of executables. However, the downside of this default work unit is exactly its generality: due to its lack of internal knowledge about the invoked process, it only measures the work unit duration (timespan between start and end of triggered process). For more sophisticated insights into BrackitDB, we therefore rely on custom written actions and work units encouraged by the BenchSuite's extendable architecture.

In terms of extensibility, there are two abstract Java classes that can be extended from in order to tailor the BenchSuite to a specific project. One of them is the `Action` class, which simply requires a `run()` method to be implemented. While this method itself does not pass any useful configuration parameters, every class extending from `Action` can access an inherited variable that allows for lookups of parameters defined in the XML configuration. Although these parameters are essentially untyped in the configuration file, type checking is eventually enforced in the lookup mechanism. Hence, if the user assigns incompatible values in the XML description, he will face an exception during runtime.

Actions are usually employed to perform preparation tasks before the benchmark begins, and to properly clean up resources afterwards. Within the scope of this thesis, the following BrackitDB-specific actions were created to support the benchmarks:

StartDB / StopDB: As the name suggests, `StartDB` and `StopDB` control the BrackitDB server lifetime. `StartDB` causes a server instance to start in a separate process (and therefore in a different Java VM), whereas `StopDB` triggers the server to stop. In order to initiate the server shutdown, a second process is started to establish a TCP connection to BrackitDB's management port and to send the proper shutdown message.

RunQuery creates a client connection to BrackitDB and transmits a preconfigured query, which is then executed on the server. In the context of later benchmarks, this action is mainly used to warm up the buffer by traversing the data before the actual workload puts stress on the system.

LoadXML loads an external XML document (from local filesystem or the network) into the database. In fact, the `LoadXML` action could be substituted by the more general `RunQuery`, as it only adds some convenience for locating documents and URL handling.

BackupDB / RestoreDB: The combination of the `BackupDB` and `RestoreDB` actions provides a simple backup mechanism for BrackitDB's container files. For the sake of equal starting conditions, we want all the scenario runs to start their work on a freshly loaded document instead of taking over the database state from previous runs. Since the benchmark workload includes write transactions, thereby causing indexes to split uncontrollably and thus degenerating the data structures, later scenarios might produce inferior results. To counter this effect, a freshly loaded XML document is backed up in advance and is repeatedly restored before a new run begins.

The second crucial class in terms of customization is called `WorkUnit`. For more sophisticated benchmarks that go beyond simple start/end measurements of external processes (offered by the default `RunProcess` work unit), it is mandatory to supply the BenchSuite with more project-specific work units, which is done by extending the `WorkUnit` class. Once again, the abstract base class provides access to benchmark parameters so that work units can be configured in the corresponding XML file. Similarly to action handling, the `WorkUnit` class specifies an abstract `run(...)` method that makes up the actual logic in the benchmark. Unlike the `Action` class, however, the `run(...)` method from `WorkUnit` passes a `Context` object as argument, which allows the work unit to store an arbitrary amount of measurements during its execution.

With regard to the benchmarks presented in this thesis, a `RunQuery` work unit had to be implemented in order to put load on the BrackitDB server. Much like the same-titled action class, `RunQuery` acts as a database client that acquires a connection, transmits the predefined query and eventually commits. On top of that, the work unit implementation takes measurements during query execution and finally stores various transaction-related measures in the designated `Context` object. While the query runtime is simply measured on the client side (that is, inside the `RunQuery.run(...)` method), most of the measures in the benchmark, like the lock count or the block time, need to be collected by the server and transmitted to the client. The `RunQuery` work unit solves this by adding the function call `bdb:tx-stats()` to the actual payload query, so that all transaction-related statistics are conveniently returned as query result.

In a nutshell, the `RunQuery` work unit adds the following BrackitDB-related measures to the BenchSuite, which are all captured per transaction.

Query runtime: The query runtime measures the time needed to execute the specified query and subsequently commit potential changes. The overhead for establishing a database connection and for transaction preparation (such as configuring transaction-related settings like the lock depth), is not included in this measure. What is however covered in the query runtime, is the cost for local TCP communication. More precisely, two roundtrips are needed: First, the query is sent to the server, which then returns the gathered transaction statistics. The second interaction eventually issues the commit. In section 5, where the experiments are evaluated, this measure is also called **total transaction runtime**.

Commit value: The commit value is simply a boolean value that equates to 1, if the transaction committed successfully, while a value of 0 implies a rollback due to a locking induced deadlock. This measure is even useful in two respects, as we can derive two interesting numbers from it by applying different aggregation functions. If we later calculate the average over these values, we obtain the *commit rate* (percentage of successful transactions) or more commonly used, the **abort rate** (i.e., $1 - commitrate$). Additionally, by taking the sum over the collected commit values, we gain easy access to the **transaction throughput**, that is, the number of committed transactions per time interval.

Lock request count: The lock request count denotes the total number of lock requests submitted by the corresponding transaction. It should be noted that lock requests in a hierarchical locking protocol entail further lock requests along the ancestor chain, which are also classified as lock request in the context of this measure. Hence, locking a node on the third document level increases the lock request count by three, even though only one node lock is explicitly requested.

Lock request time: The lock request time describes the total time a transaction spends for any locking-related activities, i.e., lookups in the lock table, requesting locks along the ancestor chain and in particular waiting for other transactions if the locking situation makes it necessary. Therefore, this measure provides a good overview of the general costs of locking in relation to the overall transaction runtime.

Escalation count: This measure sums up the number of performed lock escalations due to high subtree locality. In general, we would expect more escalations in case of a more aggressive escalation policy.

Block count: The block count refers to the number of occasions where a transaction needs to be blocked due to the current locking situation, i.e., when a lock request can not be granted immediately. Although there are also other parts of the system in which a transaction thread might be stalled, e.g., waiting for a latch or entering other critical sections in the system, this measure only counts locking-based waiting conditions.

Block time: The block time tracks the duration a transaction spends in a state waiting for other transactions to release their locks. If a transaction is required to wait multiple times, the block time expresses the sum of all individual wait durations within that transaction.

I/O read count: The I/O read count denotes the number of physical hard drive read operations the buffer manager has to perform in order to make a requested page available in main memory. Obviously, the larger the buffer or the denser the access pattern on the document is, the less I/O reads are to be expected. Bear in mind, that the just described I/O read count is not the same as the number of pages loaded from disk, since the buffer in BrackitDB incorporates a prefetching mechanism that always leads to multiple page loads per disk access.

I/O write count: Analogous to the read count, the I/O write count captures the number of physical disk writes. Whenever the full buffer capacity is reached and another not included page is requested, the buffer manager determines one or more victim pages for replacement. If the victim pages have been modified while residing in the buffer, they must be flushed (i.e., physically written) to disk in order to reflect their changes on a persistent storage. Only under these circumstances will a page request cause a physical I/O write and thus increment this measure.

Interaction between the BenchSuite and the database server is accomplished by the BrackitDB *driver*, a small Java library implementing the client-side network communication to the server, based on a simple proprietary TCP protocol. Its main class, the `BrackitConnection`, features an API for evaluating XQuery expressions and managing transaction demarcation. While per default each query is executed in its own transaction (similar to the auto-commit mode in JDBC), transactions can as well be controlled manually by operations like `begin()`, `commit()` or `rollback()`.

Speaking of TCP interaction, one of the initial benchmark attempts revealed a weakness in the implementation of the `RunQuery` work unit. In scenarios with increased thread load, the benchmark machine's operating system was apparently not able to allocate enough TCP client ports to allow for the repeated connection acquisition and termination, as done by the `RunQuery` work unit. Although at any point in time, there are at most as many active TCP connections as

there are threads producing the workload, the operating system (or the Java runtime) refused to establish new connections after some time during the benchmark run. It turned out that this issue might have been caused by the 60 seconds interval imposed by the `tcp_fin_timeout` kernel parameter [5], which keeps the TCP sockets (and ports) reserved after connection tear-down. Hence, after some time of execution, the agent threads managed to deplete all available client ports on the machine, thereby causing exceptions on further attempts to create database connections.

In order to cope with this issue, a resource pool for instances of `BrackitConnection` was introduced, which allows for shared usage among agent threads during the benchmark runs. This connection pool is built on Apache Commons Pool [8], an open source Java library for object pooling, and is configured to grow on demand but to reuse old connections whenever feasible. Hence, when an agent thread executes the `RunQuery` work unit, it first *borrows* a database connection from the shared pool and *returns* the same connection object after being finished. If no connection instance is available on request, the pool grows by establishing and returning a brand-new connection to BrackitDB. Therefore, the connection pool does not block any thread longer than the time needed to acquire a physical database connection, while it will soon grow to its maximal size, which clearly coincides with the number of concurrent transactions in the benchmark. However, in order to completely pull out the connection setup overhead from the benchmark measurements, the pool is configured to sustain a sufficient number of connections at all times.

4.1.4 BenchSuite Architecture and Workflow

The revised architecture of the BenchSuite is depicted in Figure 12. When a benchmark is started, the user-supplied benchmark configuration file is parsed and transformed into an internal benchmark representation (made of Java objects) which then gets passed to the `Coordinator`, the core class to manage the overall benchmark workflow. This component is responsible for setting up the worker threads, initializing the workload scheduler (which later assigns workunits to the workers), performing before- and after-actions according to the benchmark description, starting and stopping the actual scenario runs, and persisting the collected measurements in a database.

Regarding measurement persistence, the BenchSuite employs Hibernate ORM [35] on top of Apache Derby [9], an open source relational database. Combining these technologies allows for easy persisting and retrieval of Java object networks in a flat, relational database. Some sort of object-relational mapping is in fact required, as the collected benchmark measurements are again hierarchical in nature, since measurements can be taken on every benchmark level (like series, scenario or run level). Therefore, the raw measurement data is represented in a tree of Java objects, which is then mapped to relational tables by the persistence framework in use.

Synchronization of the volatile raw data and the persistent database does not take place while the benchmark runs are in progress, to prevent the inflicted I/O overhead from distorting the measurements. Instead, the raw data is cumulated in main memory by simply adding measurement nodes to the afore-mentioned object tree. After the run has been finished, the results eventually get written to the database. Depending on the achieved transaction throughput, which mainly determines the amount of collected data, persisting the measurement tree takes longer than the scenario run itself. For a one minute run duration, storing the results could

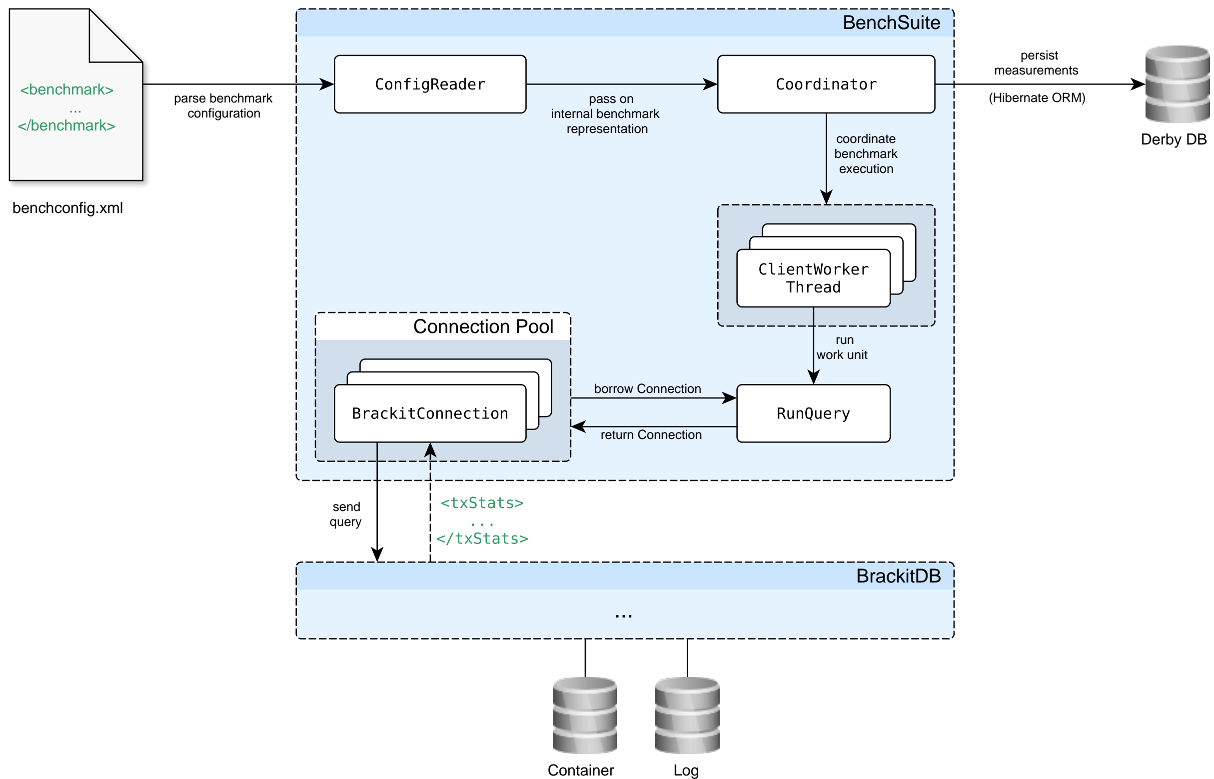


Figure 12: Simplified architecture of the Benchmark Suite

require up to four minutes at worst, thereby exposing the persistence mechanism as the main cause for prolonged benchmark durations.

What also turned out to be a major downside of the BenchSuite's persistence strategy, is its excessive main memory demand. As the benchmark progresses, the heap space continuously gets cluttered with measurement data, until the benchmark is either finished or the JVM reports an error due to memory shortage. For that reason, keeping an eye on memory consumption and regulating the size and amount of measurements posed a constant challenge throughout the benchmarks. Reducing the number of runs per scenario in exchange for lower accuracy is one way to shrink the size of gathered data. Another option is to selectively disable tracking of some of the transaction-related measures. For instance, in a pure main memory scenario, keeping track of the number of hard drive accesses caused by buffer reads or writes hardly leads to new insights, as we expect these measures to be zero anyway.

Once a benchmark terminates successfully, the collected data can be aggregated and exported as an HTML report exhibiting summary and detailed statistics about the benchmark. Again depending on the number of measures we deem useful for a certain benchmark and the number of aggregation functions to apply on them (such as *average* or *sum*), this post-processing phase requires a similar amount of main memory and time as the actual benchmark before.

4.2 Benchmark Document

The XML document utilized for the benchmarks was created by the XMark Benchmark Data Generator [6]. The corresponding command line tool `xmlgen` produces documents that model auction data in a typical e-commerce scenario. Needless to say, the generated documents are always well-formed and valid according to a certain DTD that can be inspected by running the command with the `-e` option appended. The words for text paragraphs are randomly picked from Shakespear's plays so that character distribution is fairly representative for the english language. Moreover, the XML output includes referential constraints across the document through respective ID/IDREF pairs, thus providing a basis for secondary index scenarios.

But the most outstanding feature is arguably the *scaling factor* which is specified on XML generation and determines the document's output size. Varying this factor allows for generating documents from the Kilobyte range up to several Gigabytes. According to claims made by the authors, the generated documents still preserve their characteristics under scaling so that bottlenecks found on smaller documents would apply for larger documents proportionally. It is worth noting that scaling works only horizontally (envisioning the XML as tree). That means, the document's depth and complexity remains untouched, while scaling appends new elements of different types with respect to an underlying probability distribution, and thereby widening the overall document tree.

The benchmarks in this thesis are based on an XMark document with a scaling factor of 10, resulting in a file of approximately 1.1 GiB in size. Independent of its scaling factor, the document exhibits the following layout.

```
<site>
 <!-- items clustered by region/continent -->
 <regions>
  <africa>
   <item id="item0">...</item>
   <item id="item1">...</item>
   ...
  </africa>
  <asia>
   ...
  </asia>
  ...
 </regions>

 <!-- item categories -->
 <categories>
  <category id="category0">...</category>
  ...
 </categories>

 <!-- category graph -->
 <catgraph>
  <edge from="..." to="..."/>
  ...
 </catgraph>

 <!-- registered persons (both bidders and sellers) -->
 <people>
  <person id="person0">...</person>
```

```
  ...
 </people>

 <!-- auctions open for bids -->
 <open_auctions>
  <open_auction id="open_auction0">...</open_auction>
  ...
 </open_auctions>

 <!-- closed auctions -->
 <closed_auctions>
  <closed_auction id="closed_auction0">...</closed_auction>
  ...
 </closed_auctions>
</site>
```

Listing 3: XMark document layout

As mentioned earlier, the XMark document models an auction application where *items* coming from a certain *region* are put up for auction. Each item is thereby associated with one or more *categories*, which is realized by a sequence of child elements referring to category elements by their respective ID. Furthermore, every item contains a *mailbox* element surrounding a list of *mails*. Although it might seem odd, but in our fictive auction application mails are primarily associated with an item instead of the mails' recipients.

Following on the item declarations, the *categories* element accumulates the available item categories in the system which are simply characterized by a name and a description text. Even though irrelevant for our benchmarks, the *catgraph* element defines a graph structure on top of the categories, whereas each *edge* ties two categories together. However, the meaning of this category graph is not clearly evident, but it certainly implies some sort of relationship between connected categories (e.g., similarity). At the end of the day, the exact interpretation is left to the benchmark designer.

The *person* elements gathered under the *people* element act as users of the virtual auction platform. A few personal information like the name or the email address are exposed at this place, but their main usage is to be referenced by *auction* elements as bidders or sellers, respectively.

In terms of auctions, XMark distinguishes between open and closed auctions and stores them separately as either *open_auction* or *closed_auction* element in the corresponding subtree. But in fact, their XML structure is quite similar. Both kinds of auctions contain a description, a reference to the traded item and its seller. Additionally, an open auction offers information about the current bidding situation. In this application, each submitted bid on an item is represented by a *bidder* element under the related auction. However, a bid does not specify an absolute price the bidder is willing to pay. Instead, every bid specifies an increase in price relative to the previous bid, starting out with a fixed initial price (presumably determined by the seller). Additionally, the auction's current price is stored subsequent to the list of bidders. In contrast, a closed auction does not contain the history of bids anymore, but simply links to the eventual buyer and reveals the final price.

Ultimately, to provide a better grasp on the document's fanout and how the above-described elements compare to each other in terms of quantity, we should have a look at the exact cardinalities for a document with scaling factor 10:

Items (all regions):	217,500
Categories:	10,000
Catgraph Edges:	10,000
Persons:	250,000
Open auctions:	120,000
Closed auctions:	97,500

4.3 Benchmark Workload

After having introduced the document structure and the meaning of its elements, we can proceed with describing the benchmark workload itself. In essence, the workload that is putting stress on BrackitDB is a mix of eight different transactions. Some of these are read-only, others perform both read and write accesses. Moreover, transactions are picked randomly from this pool according to predefined weights, which are supposed to reflect a distribution we might find in a realistic auction-based application.

4.3.1 Architectural Placement

As already mentioned before, the Benchmark Suite interacts with BrackitDB through its Java driver, which allows for sending queries to be evaluated on the database. However, the workload transactions described later are not defined by complex XQuery expressions. Instead, the BenchSuite agents merely trigger low-level routines to run directly on top of the node interface, as depicted in Figure 13. Accordingly, in order to call one of the XMark operations, a simple XQuery function is sent to the server. This `workload` function takes the operation's name as argument, as well as the document name to apply it on. Based on the first argument, the code for the specified operation is invoked, and with it, the engine's responsibility ends.

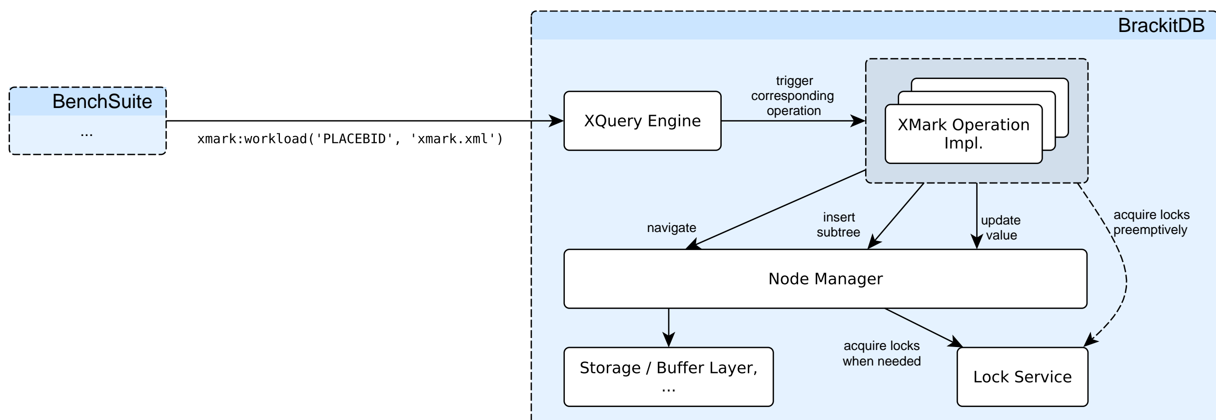


Figure 13: Workload operations as "low-level programs" based on the node interface

There are a few reasons why the workload is not constructed as a set of XQuery (e.g., FLWOR) expressions. As a matter of fact, XQuery is powerful and expressive enough to describe equivalent operations on the query level. But for a start, we would rely on the XQuery engine to determine the best suited access plan, which means we have no direct influence over the node operations called by the engine to perform the transactions. While on the other hand, a low-level procedure on top of BrackitDB's node interface provides the necessary means to control all locking-related aspects of the workload. For instance, if we are looking for a child node satisfying a certain condition, we have the power to decide between performing a `First-Child`, followed by a sequence of `Next-Sibling` navigation steps, or rather utilize the stream-based version `getChildren()` for this task. Needless to say, the choice of employed node operations can have a huge impact on the resulting locking situation and ultimately the system performance. To come back to our previous example, manually iterating over child nodes in a node-by-node fashion leads to many fine-grained node and edge locks, whereas requesting a children stream only implies a single lock, namely a level read on the respective parent node (given taDOM is enabled). On the other hand, if the desired child node is one of the first among its siblings, locking every single child node up to the matching node might prove to be superior over locking the whole level through one lock.

At some point, we might even decide to circumvent the node interface as a whole, by preemptively requesting a lock on a certain node. In general, programs working with the node interface do not have the obligation to acquire locks at all. In fact, all locking-related tasks are performed by the node implementation on demand, while higher layers usually do not even have access to the lock service. But if we know, for instance, that a node we retrieve via the node interface (which only comes shared-locked), will subsequently be updated anyway, acquiring an update lock right in the beginning might be beneficial to avoid delicate lock conversions. In other words, context knowledge about the operation we implement allows us to manage locks more reasonably in certain situations, compared to the default locking policy pursued by the node implementation.

The bottom line is that implementing the workload operations as tailor-made XQuery functions, based on the low-level node interface, offers enhanced flexibility and influence compared to the equivalent XQuery expression. As we will see later, the workload implementation comes in different variants. One of these makes use of the `First-Child/Next-Sibling` navigation pattern whenever child access is necessary, while another employs the children stream in conjunction with the LR mode provided by taDOM. Yet another variant aims for avoiding conversion deadlocks by preemptively obtaining the strongest necessary lock mode before any node is accessed for the first time.

4.3.2 Secondary Indexes

When an XML document is loaded from the file system into BrackitDB, the so-called document index is created, a B^+-tree with DeweyIDs as keys, and the corresponding XML nodes stored as values in the leaf pages. This document index acts as the *primary* index, since it provides a mapping between the unique node labels and the node data itself. Therefore, it can serve as entry point for document access, whenever the DeweyID of the starting node is known in advance. Obviously, access to the document's root node with its statically determined DeweyID can simply be accomplished by consulting the document index.

However, for any other kind of document access (e.g., retrieving a node by its name, by its path, or by its content), a secondary index is required. In general, accessing nodes via secondary index adds an additional layer of indirection, as this procedure usually implies a two-fold access pattern:

1. Access secondary index: Restore qualifying node (most importantly its DeweyID).
2. Access document index: Based on the node returned from step one, perform any relevant navigation steps (e.g., scanning through its subtree).

It should be mentioned that step two can be omitted in certain situations. If, for some reason, the node retrieved by step one is not used for further navigation steps, the document index can in fact be bypassed, since the secondary indexes in BrackitDB contain sufficient information to fully restore the indexed node, but none of its attributes, text values and so on. Hence, further navigation is very likely.

Considering our benchmark workload, the transactions need a way to randomly jump into the document to begin their processing. For instance, placing a bid requires the transaction to pick a random auction element, while another operation might start on a random person element as context node. Another feature needed by some operations is the possibility to follow ID references within the document, e.g., jump from the auction to the corresponding item element. Although both requirements (random entry and navigation via references) could be met without utilizing any secondary indexes by accessing the root node and scanning for the desired node, system performance could drastically suffer from this unfavorable access plan. For that reason, we provide the two following secondary indexes on the XMark document:

Name Index: This index maps from element names to a set of DeweyIDs, from which the appropriate element node can be reconstructed. It therefore satisfy the requirement of random entry to an element with a given name.

CAS Index: A CAS index [31, p. 201] allows for content-based node lookups, i.e., attribute and text nodes can be retrieved based on their string value. Hence, it constitutes the perfect candidate to deal with IDREF cross jumps.

4.3.3 Skewed Access

When fetching random elements from the name index (which is always the first step for every benchmark transaction), the elements are selected with respect to some predefined *skew*. Since the XMark document is over 1 GB in size, locking-related contention has to be artificially enforced by making certain parts of the document accessable with a higher likelihood than other parts, which is referred to as skew in the context of this thesis. A higher skew implies denser access patterns and increased contention.

From an implementation's viewpoint, descending a name index is based on normal distributions [4, 38] laid over the entries of each index page, as depicted in Figure 14. Consequently, pointers stored in the center of the corresponding page have an increased probability of being chosen for the next descent. This pattern is continued until arriving on the element level. Although a normal distribution is assumed for each individual page, the overall element access is not

necessarily normally distributed. However, this approach succeeds in generating hotspots in the document, which is sufficient in our context.

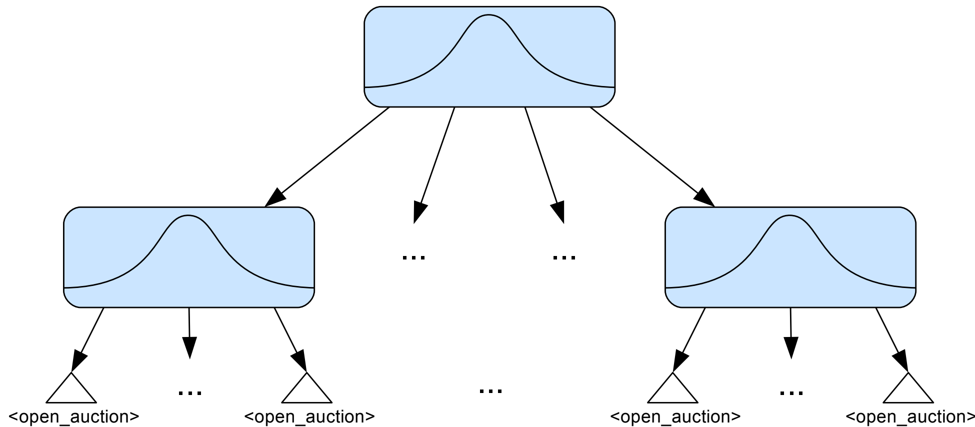


Figure 14: Name index for auction elements. The waveforms reflect the likelihood of a particular pointer to be selected for the descent.

The skew is now defined as $1-\sigma$, while σ denotes the relative standard deviation of the underlying normal distributions. Hence, a skew of 99% (or a relative σ of 1%) implies that only 1% of the records in the page are picked with a probability of approximately 68%, effectively producing a hotspot in the page's center.

4.3.4 Workload Transactions

The following part is dedicated to the description of each individual XMark operation that is performed over the course of the benchmarks.

Place Bid: The first and arguably most important activity on an auction platform is the bidding process. Putting a bid on an open auction in an XMark document basically implies two steps which need to be performed atomically, or in other words, within a single database transaction. First and foremost, a new *bidder* element has to be inserted, specifying an increase in price relative to the previous bid. The order in which the bids appear in the auction element is significant, however. Following the initial price, the timeline of placed bids is defined by the sequence of *bidder* elements, while the last of them is the highest bidder at this moment (in the little from Listing 4, this would be the person with ID "person4711").

```
<open_auction>
  <initial>1.07</initial>
  <bidder>
    <date>...</date>
    <time>...</time>
    <personref person="person1234"/>
    <increase>7.50</increase>
  </bidder>
  <bidder>
    <date>...</date>
    <time>...</time>
    <personref person="person4711"/>
    <increase>16.50</increase>
  </bidder>
  <current>25.07</current>
  <seller person="..."/>
  ...
</open_auction>
```

Listing 4: Excerpt from an auction element

The second crucial step in order to regain consistency, is the update of the item's current price, which is represented as an element following the latest bidder. From an implementation viewpoint, there are two options to calculate the current price. One is to read the old (current) price, add the new bid's increase to that value, and write the new price back. Unless an update lock is explicitly requested, this procedure would first acquire a shared lock on the *current* element (or rather on its text child node), which is later converted to an exclusive lock to perform the update. Although this would indeed work if we assumed the old price was consistent to the previously submitted bids, the bidding transaction pursues a different approach. While iterating over the auction's child elements to determine the correct insertion position for the new bid, it also keeps track of the current price by starting with the initial value and adding the increase from each traversed bid to its internal price variable. After having inserted its new bid between the last *bidder* and the *current* element, it can instantly set the new price tag along with it.

Read Seller Information: The second operation in our transaction mix first selects a random auction element and determines the seller ID by iterating over the auction's children until it arrives at the *seller* element. Next, with the appropriate seller ID in mind, the transaction jumps to the desired person element by accessing the CAS index. From here, it traverses the seller's children in order to collect some personal information about him, such as his name, his email address and his phone number.

Register: In the registration transaction a new person subtree is added to the list of existing persons in the system. Recalling the XMark document structure, the users of the auction system are accumulated under the global *people* element. Therefore, this operation fetches the (only) people element from the name index and first navigates to its last child. From here, it determines the latest assigned person ID by reading the last child's ID attribute, and finally inserts a new person with an incremented ID after the currently last person, thereby becoming the new last child of the *people* element.

As it turns out, this operation provides a good opportunity for preemptive locking. Since we first navigate over the `Last-Child` edge of the *people* element, thereby locking this edge in shared mode, the insert operation and the resulting X-upgrade of this edge lock is a recipe for deadlocks. Hence, it may prove beneficial to exclusively lock the `Last-Child` edge in the first place.

Change User Info: The next operation in our mix is concerned with changing user data for a random person element. In a manner of speaking, this transaction forms the antipole to the transaction retrieving the seller information. Because otherwise, reading from a person subtree could never even produce any locking-related clashes.

What this operation does in detail after fetching a random person is to iterate over its children until it arrives at his email and phone element where it updates both text values. For that matter, updating an element value involves navigating to its only text child node and updating the value of this text node accordingly.

Check Mails: As already discussed, the mailing system in the XMark auction application seems quite unusual, since mailboxes do not belong to persons but are associated with items and are therefore stored as a subtree under the respective item element. Checking mails thus implies fetching a random item element to start with. With this item as context node, the transaction navigates to its last child, which is always the item's mailbox element according to the schema. From there, the transaction runs through the single mail elements (i.e., the mailbox' children nodes) and reads every mail by scanning through the entire mail's subtree by means of the `getSubtree()` method on the node interface.

Read Item: This operation emulates the behavior of a customer who is interested in a specific, but randomly picked auction, and who wants to find out details about the item that is put up for sale. To do so, the transaction traverses the auction element's children until crossing the *itemref* element which points to the respective item by its unique ID. Like for determining the auction's seller, the CAS index is consulted to retrieve the referenced item. Once there, the complete item subtree is scanned (again by calling `getSubtree()`).

Add Mail: Adding mails to the auction system involves fetching a random *mailbox* element from the name index and prepending a new *mail* element to it. Hence, the resulting mail sequence is ordered from newest to oldest mail, which is also the common order emails are displayed in various email clients.

Unfortunately, due to intrinsic characteristics of DeweyIDs, prepending elements is an unfavorable operation which eventually requires a reorganization of stored nodes. It is known from section 2.2 that DeweyIDs are considered stable, i.e., they allow for arbitrary insertions without ever needing to be changed due to their overflow division approach. While that is undoubtedly true, certain insertion patterns lead to bulky and ever growing DeweyIDs. If we keep prepending elements to a non-empty root node, we end up with DeweyIDs of the following structure: `1.2.2.2.2.2.....3`

In conclusion, DeweyIDs support both prepending and appending operations, but only appending ensures a constant number of divisions for the resulting IDs. With the DeweyID compression algorithm utilized in BrackitDB, an increased amount of divisions also raises their physical storage size, while higher division values are less of a concern in terms of storage consumption. As a result, continuously prepending elements leads to a situation at some point, where the physical page layout of the document index is no longer able to store DeweyIDs in their compressed binary format.

Since BrackitDB does not support DeweyID reorganizations at this moment, a workaround to make the constant prepending operations of mail elements possible was therefore introduced. Before the new mail is inserted, the transaction checks the DeweyID of the currently first mail in the sequence, and verifies based on the number of divisions whether it is still safe to prepend further elements to that mailbox. If not, the new mail is appended to the end of the mail sequence instead.

Add Item: The last operation of our transaction mix adds items to the system. As previously mentioned, items are clustered by their respective region, as a result of which a random region needs to be selected as parent element for the new item. To this end, a random item is fetched from the name index and its parent is chosen as insertion target. The remaining procedure is similar to the registration transaction. In a first step, the last child of the chosen *region* element is read in order to determine the last used item ID, which is then incremented to make up the new item's ID. Ultimately, the new item is inserted after the current last child.

Once again, we can reduce the potential for lock conversion deadlocks by preemptively X-locking the region's last child edge in preparation of the upcoming append operation.

4.3.5 Workload Variants

Previously, the workload operations were described on a rather semantical level. However, there is still much room for different implementations which have the same effect on the persistent data, but deviate in their choice of node operations and acquired locks. Since it is not easy to predict which strategy leads to the overall best system performance, the workload is configurable by two parameters with two possible values each, thereby resulting in four different workload variants. The two mentioned parameters are specified below.

Child Access (navigation-based / stream-based): The first workload parameter determines whether child traversals in the XMark transactions are performed in a navigation- or stream-based fashion. While the navigation-based approach makes use of a `First-Child` followed by a sequence of `Next-Sibling` operations, the stream-based strategy requests a child stream instead by calling `getChildren()` on the parent. Although this decision might sound like a minor detail, it produces a completely different locking situation in the respective subtree and might thus affect system performance noticably.

Navigation-based child access entails a series of fine-granular lock acquisitions (more specifically NR locks) on the parent as well as on every visited child node. Furthermore, serializability calls

for shared edge locks on the parent's `First-Child` edge and on each passed `Next-Sibling` edge. Therefore, it potentially allows for more concurrency in that subtree at the expense of increased locking overhead. On the other hand, the stream-based child scan brings taDOM's LR (*level read*) mode into operation, which protects the parent, its children and all the gaps in between (with regard to phantoms) from parallel modifications. Reduced locking costs in return for coarser granularities is the logical consequence.

Depending on the objective of the corresponding child access, the navigation-based pattern might benefit from its flexibility to cancel the child scan at any time, e.g., after the desired child node was retrieved. In this case, locks are only obtained for the nodes and edges that were effectively visited, while the remaining children remain unaffected. Whereas the LR mode imposed by the stream-based scan keeps its wide protection range until commit, even if the stream is closed soon after traversing a handful of children. Another advantage for the navigation-based strategy is its potential for lock escalation due to the high subtree locality resulting from high the number of lock requests on the children level. However, what militates again for the stream-based approach is its more efficient storage-level implementation compared to the equivalent sequence of single navigation steps.

In the end, it comes down to fine-granular locks and increased concurrency (navigation-based) versus reduced locking overhead and more efficient storage evaluation (stream-based). The experiments will reveal which of these aspects prevail in practice.

Preemptive Locking (disabled, enabled): In the context of this thesis, *preemptive locking* denotes a strategy where the strongest necessary lock mode is obtained before any particular node is accessed for the first time. In general, the main problem when designing transactions on top of the node interface is the lack of control over the acquired lock modes, since the node implementation encapsulates any locking-related aspects underneath its public API. Consequently, the choice over the received lock mode is merely determined by the type of node operation invoked by the transaction. In the current version, no further knowledge regarding the intention for the requested node can be passed to the node manager. As a result, traversed nodes are always returned read-locked unless the concrete need for a stronger lock mode becomes apparent, such as for insert, update or delete operations. When a node has to be inspected prior to an update operation, a lock conversion (or more specifically a lock upgrade) is carried out. Unfortunately, lock upgrades always incur the risk of deadlocks.

Taking context knowledge about the transaction and its purpose into account allows us to drastically reduce the potential for deadlocks. This is achieved by always obtaining the strictest lock mode we require in the context of a particular transaction before a node is accessed (and thereby read-locked) for the first time. In order to do so, however, the node interface needs to be manually bypassed to get direct control over the lock service in the transaction code.

For instance, if a transaction requests a child stream with the intention to insert a new subtree between one of the children, the preemptive locking approach makes sure the corresponding parent is locked in LRCX mode before `getChildren()` is called. Otherwise, a later upgrade from LR to LRCX would be inevitable. If two transactions happened to perform this upgrade at the same time, a deadlock would occur.

It should be noted that the preemptively acquired locks in the workload implementation are not necessarily sufficient to guarantee the complete absence of conversion deadlocks, but they aim for avoiding the most frequently encountered deadlocks that have been observed over time.

4.4 Hardware/Software Environment

If not stated otherwise, all benchmarks performed in the scope of this thesis were executed on the following machine. Note that the benchmark suite as well as the BrackitDB server were both running concurrently on the same node during the measurements.

CPU: 2x Intel Xeon E5420 @ 2.5 GHz, totalling in 8 cores

Memory: 16 GiB DDR2 @ 667 MHz

Operating System: Ubuntu 13.04 (GNU/Linux 3.8.0-35 x86-64)

JVM:

java version "1.7.0_51"
OpenJDK Runtime Environment (IcedTea 2.4.4) (7u51-2.4.4-0ubuntu0.13.04.2)
OpenJDK 64-Bit Server VM (build 24.45-b08, mixed mode)

5 Benchmark Execution and Evaluation

This section is concerned with the actual benchmark executions and the conclusions we can draw from the collected measurements. As described in the previous section, the benchmark architecture allows us to vary two system parameters per benchmark, while the others are constant throughout the run. Needless to say, in a complex database system like BrackitDB there are tens or hundreds of configuration parameters that affect general system performance. Therefore, finding pairs of particularly interesting parameters for comparison, while the others are set to reasonable default values, is a research topic for itself.

In order to provide a clear picture about how BrackitDB's locking mechanism performs and scales, we run a sequence of benchmarks in which different locking aspects and their effects on various performance measures are investigated. After presenting the relevant BrackitDB system parameters, which either serve as variable or as constant in the following benchmarks, we first establish a baseline where some coarser-grained locking protocols compete with each other. Subsequently, taDOM and the effectiveness of related optimization techniques like *lock depth* or *lock escalation* will be studied under the same conditions as the baseline experiments (i.e., tight access pattern and reduced I/O overhead).

Furthermore, we prepared different but semantically equivalent workload variants which are evaluated afterwards. Like before, this benchmark relies on taDOM as underlying locking approach. The last measurements in the scope of this thesis explore the implications of different I/O setups, such as slow and frequent I/O interactions versus disabled external storage I/O.

5.1 BrackitDB System Parameters

The following set of system parameters defines the framework for each conducted experiment. Depending on our goal, they are either varied to determine their impact on different measures, or they are fixed to some constant value throughout the benchmark. The parentheses next to the parameters' names contain the available values.

Buffer-ratio (small buffer, large buffer): The buffer-ratio denotes the ratio of available frames in the page buffer relative to the size of the whole container file. Hence, a value of 0.01 or 1% implies that 1% of the XMark document (incl. secundary indexes) fits into main memory. When the buffer is full and yet another page is requested, one of the pages in the buffer is evicted (possibly written back to disk) to make room for the new page. If the ratio value is greater than 1, no eviction takes place at all, since all pages from the container fit into the buffer at the same time. Hence, after every page was requested at least once, no further page-related disk I/O during the scenario run is involved. Incidentally, every measurement taken below starts on a warm buffer. In the scope of these benchmarks, we are mainly interested in the two corner cases of a *small buffer* (with a buffer-ratio of 0.01) and a *large buffer* (buffer-ratio > 1).

Log (HDD, Flash, disabled): BrackitDB pursues an ARIES-style logging and recovery approach [33] which requires it to flush its log buffers to a persistent storage before dirty pages are evicted from the buffer (for Undo recovery) and when a transaction commits (for Redo processing). Since BrackitDB does not offer any group commit [18] behavior, the frequent

log flushes triggered for every commit quickly turned out to be a major system bottleneck in terms of transaction throughput. For this reason, instead of keeping the persistent log file on an HDD with disastrous random access performance, the log can also be configured to reside on flash memory. Due to its low latency characteristics, the I/O bottleneck is considerably defused. However, for scenarios in which no external storage I/O is desired at all, log flushes may also be entirely *disabled*. While this option makes Redo recovery impossible (which is irrelevant for our benchmarks anyway), transactions can still be rolled back if aborted by the deadlock detector.

Skew (low, medium, high): The purpose of skew is to artificially create hotspot elements in the benchmark document which are consequently accessed more frequently than others. How this skew is realized was already covered in 4.3.3. Suffice it to say that we will only set this parameter to 50% (low skew), 90% (medium skew) or 99% (high skew). For obvious reasons, we expect higher skews to produce more locking conflicts and thus a reduced transaction throughput.

Number of threads: This parameter specifies the number of worker threads supplied by Brackit-DB to process the workload. Hence, this number determines the maximal amount of concurrency in the system. As there is currently no intra-transactional parallelization, the number of worker threads is equivalent to the number of active transactions. We should keep in mind, however, that the same number of threads are additionally spawned within the BenchSuite process to generate the workload.

Workload variant: As introduced in section 4.3.5, this parameter specifies which concrete workload implementation is utilized for the experiment, i.e., whether child access is performed via navigation steps or via children stream, and whether preemptive locking is exploited or not.

Lock protocol (X, RIX, URIX, taDOM): Thanks to BrackitDB's extensible locking framework (also referred to as meta-locking [12]), plugging different locking protocols into the system is an easy task. Every locking protocol merely has to define its lock modes, the compatibility and conversion relationship among them, and which lock mode must be acquired for which kind of node operation. Once this is specified, a system parameter (defaulting to taDOM) determines the lock protocol used by BrackitDB on startup. In the scope of this thesis, we will have a look at some alternative protocols in addition to taDOM. The first one is simply named X, and refers to the usage of a single exclusive document lock. Although many transactions can be active at the same time, only one at a time can access the XMark document. The next protocol on this list, RIX, is a hierarchical protocol like taDOM but is restricted in terms of lock modes. Only R, I, and X are available in our RIX implementation. URIX extends RIX in a two-fold fahsion: it introduces a subtree update lock mode and additionally distinguishes between IX and IR instead of the single intention lock offered by RIX. Compared to taDOM, URIX still lacks some more advanced lock modes, such as node read/update/exclusive or level read locks.

Lock depth: This parameter corresponds to the lock depth described in 3.4.4.

Lock escalation (disabled, moderate, eager, aggressive): As introduced in 3.4.4, this parameter specifies the underlying lock escalation policy. From left to right, escalation becomes increasingly aggressive.

5.2 Baseline Benchmark

In our first series of experiments, we establish a baseline for taDOM to compete against. In order to do so, we run the same XMark workload with different lock protocols plugged into BrackitDB, more specifically X, RIX, and URIX. The number of threads as well as the skew are chosen to cause as many locking conflicts as possible. As a result, these measurements should be regarded as the worst case in terms of contention. Furthermore, storage I/O is reduced to a minimum where crash recovery is still feasible. That way, the investigated lock protocols can prove their efficiency in a CPU-bound environment.

Variables:

- Lock protocol: X, RIX, URIX

Constants:

- Number of threads: 8
- Skew: high
- Buffer: large
- Log: Flash
- Lock depth: infinite
- Lock escalation: disabled

The results for above-described benchmark setup are presented and interpreted next. The most important measure in terms of the overall system performance, given by the transaction throughput (per minute), is shown on the left side of Figure 15. At first glance, the outcome appears counter-intuitive because the most simplistic and concurrency-pruning lock protocol X strikingly prevails against the finer-grained tree protocols RIX and URIX. Although the X setup employs the same number of worker threads, only one of them is effectively accessing the document at one point in time, while the others have to wait for the global exclusive lock to be released. In contrast, RIX and URIX allow the transactions to process their work concurrently, as long as they are accessing different parts of the document. Read-only transactions are even more favored by RIX/URIX in that they do not impede each other with respect to locking. Despite all these aspects, however, squeezing the transactions sequentially through the system achieves a transaction throughput that is roughly three times as high as the throughput provided by the other contenders.

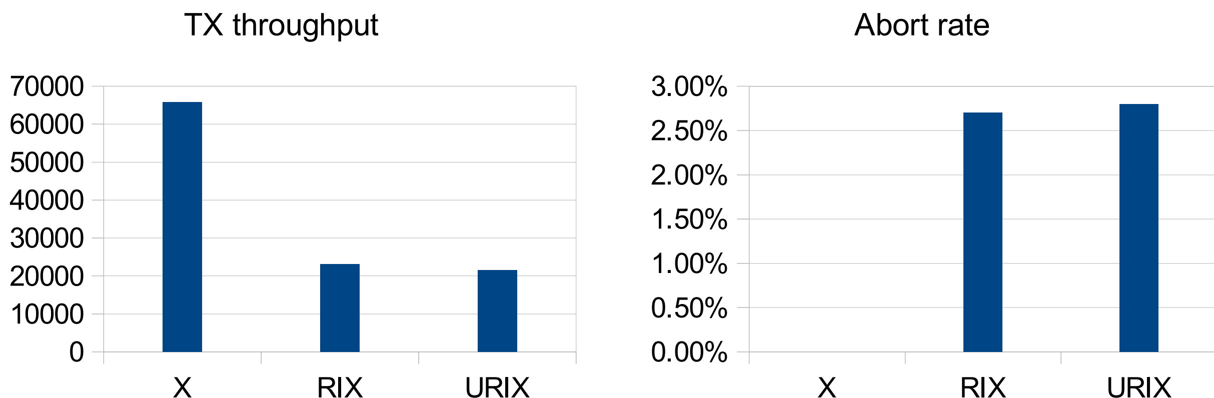


Figure 15: Baseline results (Transaction throughput and abort rate)

Looking at the abort rate (Figure 15, right) delivers at least a slight indication about the merit of exclusively locking. No deadlock conditions and thus transaction abortions ever occur in our X scenario, since only one lock exists in the first place. Moreover, by only providing a single lock mode, lock conversions (essentially the main reason for deadlocks) are avoided entirely. Although the abort rates for RIX and URIX are less than 3%, which is almost negligible, the time it takes for the deadlock detector to eventually resolve these deadlocks might be too high for a CPU-bound benchmark setup. The subtly increased abort rate of URIX compared to RIX may well be within the margin of error, as there are in fact plausible reasons to support exactly the opposite. After all, URIX allows for deadlock-reduced upgrades from U to X (which only results in a deadlock if other locked objects are involved as well). But the original problem remains in case of R→U upgrades, which are equally as dangerous as classic R→X upgrades.

Further insights about the transaction timings, with focus on locking overhead, is provided by Figure 16. In terms of the individual query runtimes, the two hierarchical protocols are the definite winners, whereas RIX and URIX generate almost the same timing values. While a transaction - from begin to commit - takes nearly seven milliseconds for an exclusively locked document, this runtime is significantly reduced by the gain in parallelism when utilizing RIX or URIX, where a transaction only requires slightly over one millisecond in average. The reason why the X lock protocol takes longer for an individual transaction is consistently reflected by the block time (rightmost bar). Since all parallel transactions have to wait in the lock request queue of the document node, they are blocked and remain in this state for slightly less than six milliseconds until it is finally their turn to access the document. Idle times caused by being stuck in queue are essentially non-existent with RIX and URIX (50 to 80 microseconds in average).

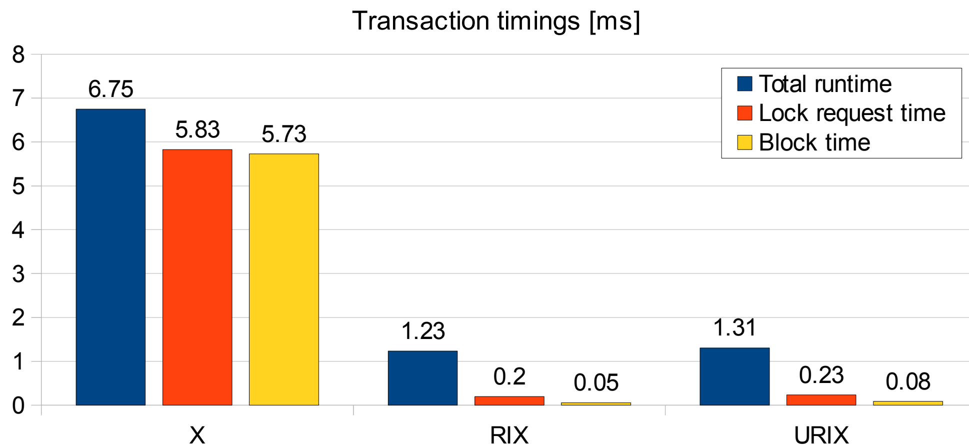


Figure 16: Baseline results (Transaction timings)

Another interesting observation from Figure 16 is the constant and miniscule difference between the block time (rightmost bar) and the total lock request time (bar in the middle). What this effectively tells us is the neglectable overhead for general lock management (including access to the lock table and other locking-related data structures). If blocking is required for an issued lock request (like in the X lock scenario), the block duration substantially dominates the time a transaction spends for acquiring this lock.

Furthermore, analyzing the deviation between the total transaction runtime (left bar) and the cumulated lock request time (middle bar) makes clear that the non-locking-related overhead is almost constant and makes up around one millisecond per transaction, independent from the

applied locking scheme. This overhead includes TCP interaction between the BenchSuite and BrackitDB (although no real network traffic is involved), accessing the storage and evaluating navigation steps, logging every modification, fixing pages at buffer level and much more. For locking protocols like RIX and URIX, these fixed costs are unavoidable and make up a major portion of the total query runtime. In fact, less than 20% of the transaction time is spent for locking-related tasks. Although locking does not seem to be the bottleneck in transaction execution, the choice of employed locking protocol turns out to have a huge impact on the overall transaction throughput (with regard to Figure 15).

5.3 Lock Depth and Lock Escalation

After we got over the initial surprises presented by the baseline benchmark, it is time to put taDOM and its optimization techniques to the test. Since taDOM acts as an extension to URIX, providing the same set of lock modes plus some new taDOM-specific locks (such as NR, NU, and NX), we would expect this protocol to behave more similar to URIX than to the deadlock-free X mechanism.

Variables:

- Lock depth: 0, 1, ..., 8
- Lock escalation: disabled, moderate, eager, aggressive

Constants:

- Lock protocol: taDOM
- Number of threads: 8
- Skew: high
- Buffer: large
- Log: Flash
- Workload variant: navigation-based

To make the following benchmark comparable to the baseline, the starting conditions are pretty much the same. Though the difference is that we vary the lock depth and lock escalation strategies to attest their impact on the results. At this, the next benchmark resembles the experiments that have already been conducted in XTC as part of the research in [14]. However, the numbers should not be compared one-to-one since the benchmark conditions differ.

The first observation we can make from examining Figure 17 is the huge impact on the transaction throughput when varying the lock depth in the range between 0 and 3. While a lock depth of 0 basically escalates every lock request to the root node and thereby achieves a dreadful result in terms of system performance, taDOM seems to reach its climax when restricting the lock depth to 3. Allowing even finer-grained locks by further increasing the lock depth does not result in higher throughputs anymore. One could even argue that performance first deteriorates for lock depths higher than 3 but quickly starts to stagnate at lock depth 4 or 5. In fact, this pattern could be well explained by the structure of the XMark document and the workload. Increasing throughput values for lock depths up to 3 indicate that most of the transactions clash at the fourth level of the XML tree, which is where most of the document modifications take place. Consequently, incrementing the lock depth to 4 does not improve concurrency any further but entails more locking overhead.

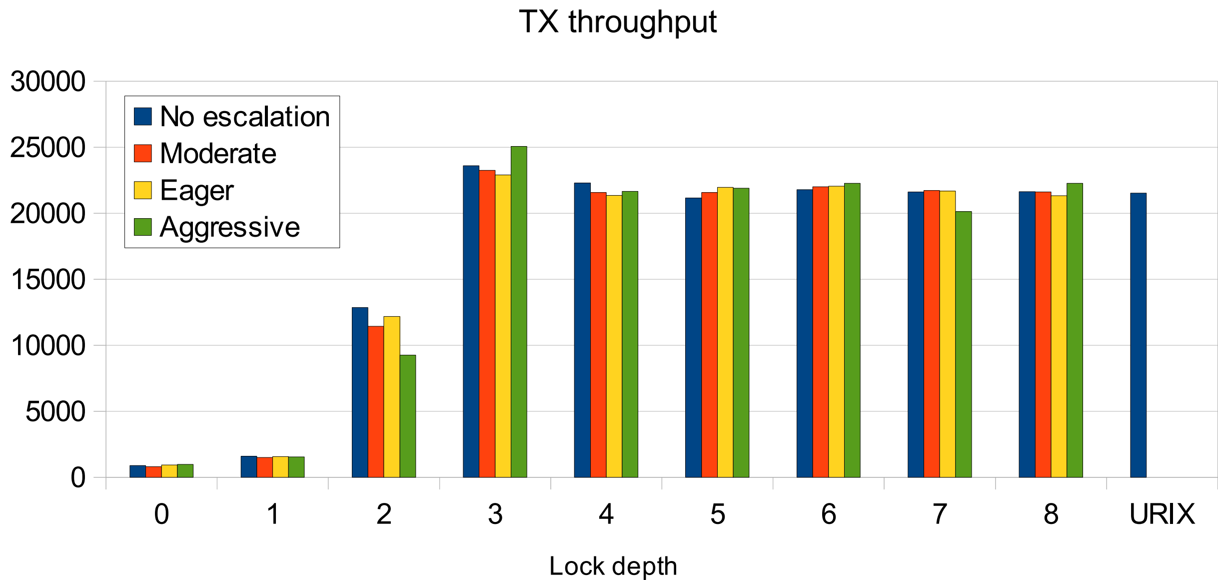


Figure 17: taDOM with lock depth and lock escalation (Transaction throughput)

Figure 18 confirms our observation regarding increased contention for lock depths less than 3. In fact, the abort rate is somewhat complementary to the throughput. High locking contention poses a higher risk of deadlocks which again leads to high abort rates and eventually a drastically reduced transaction throughput.

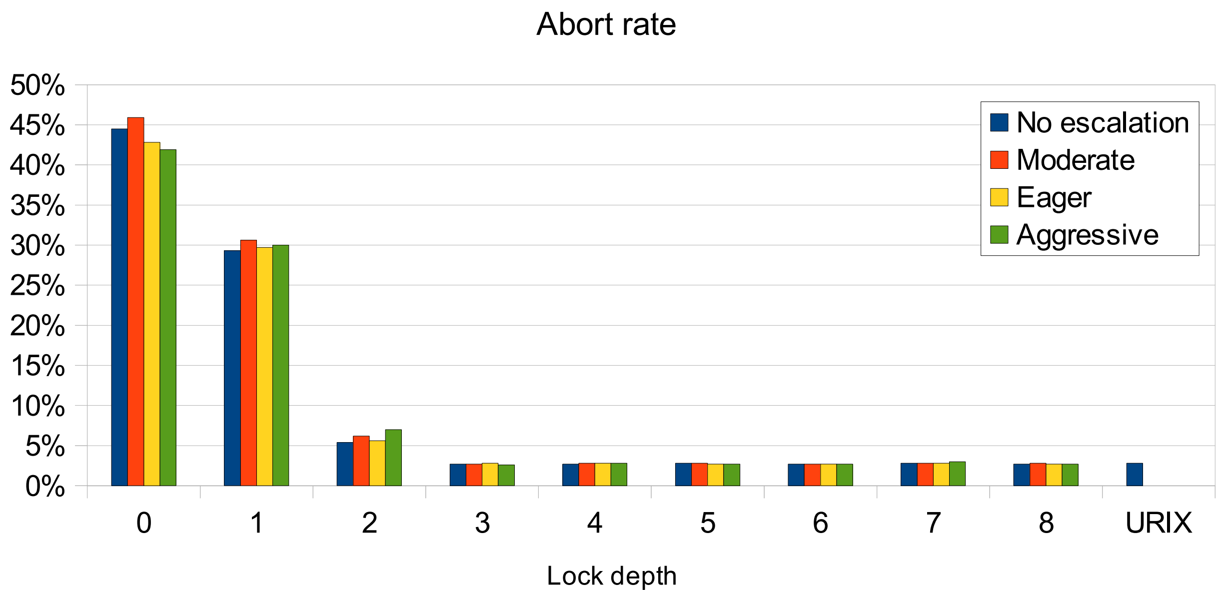


Figure 18: taDOM with lock depth and lock escalation (Abort rate)

The second benchmark variable, the applied lock escalation policy, seems to have no remarkable effect on the throughput or abort rate whatsoever. All visible fluctuations between the different strategies could be caused by the random aspects of our workload. Otherwise, there would at least be a stable pattern where one strategy is always slightly better or worse than another one. A possible reason for this observation is quickly found in Figure 19. It essentially tells

us that only aggressive lock escalation leads to actually performed lock escalations, while lock requests for any other escalation policy never satisfy the conditions it takes for escalating. Hence, the strategies *no escalation*, *moderate* and *eager* are not only coincidentally producing similar results, but they literally constitute the same scenario.

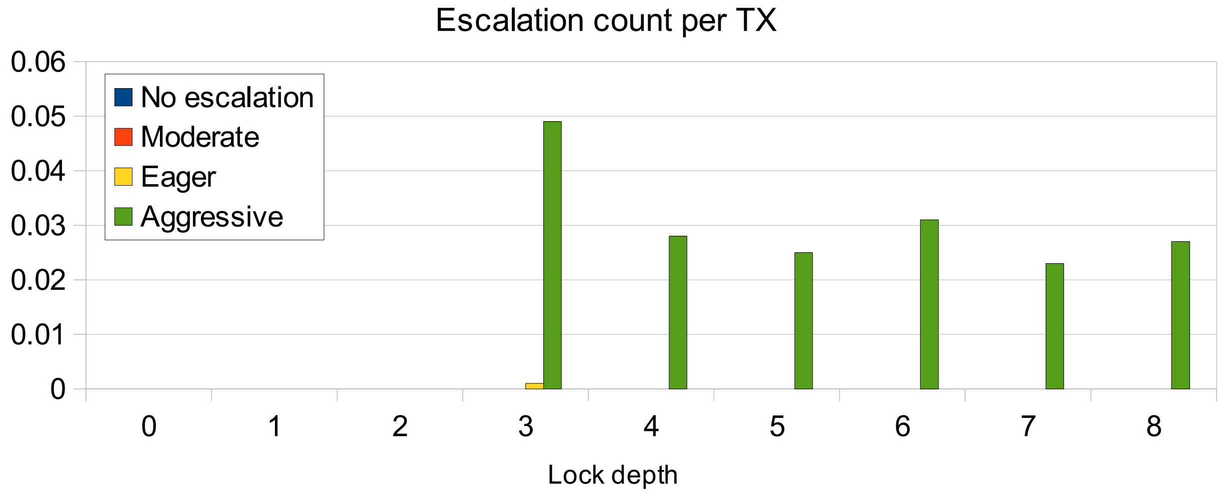


Figure 19: Only aggressive escalation leads to some actual escalations

By showing no notable effect for more aggressive escalation approaches, our findings dissent from the results in [14], where aggressive escalation constantly proved to be slightly favorable. We identified the following potential reasons why escalation occurs very infrequently in our benchmark:

1. The absolute values for the two escalation parameters (explained in 3.4.4) are not chosen properly. In order to increase the tendency for escalation, the value for `maxEscalationCount` must either be lowered, or `escalationGain` must be increased for every escalation policy. Only if these parameters are appropriately set up, we create the basis for the "optimal" amount of performed escalations.

2. If a transaction meets the conditions for escalation in a certain subtree, there is a second reason why escalation might fail. Independent from the escalation policy, a lock request will only be escalated if the parent's granted lock modes are in fact compatible with the escalated lock mode. Or in simpler terms, a transaction needs to be alone in its current subtree in order to trigger any escalation. Due to the intentionally high skew in our benchmark, this condition might not be satisfied often enough.

3. The benchmark transactions are generally too short and thus do not hold enough locks in a particular subtree for initiating escalation, although the lock escalation parameters (from point one in this list) are reasonably chosen for "real-world" workloads.

Now that the effects of lock depth and lock escalation have been discussed, we still need to compare taDOM to alternative locking approaches, such as URIX from the baseline. By examining the throughput and abort rate, we quickly realize that taDOM plays in the same league as URIX, even though taDOM offers finer-grained lock modes than URIX does. Maybe the benefits we anticipate from finer-grained lock modes (such as NR versus SR) do not properly come into

effect for the workload we have set up. Moreover, without presenting a new diagram, the query timings (total runtime, lock request time, block time) for taDOM look almost indistinguishable from the URIX values in Figure 16.

As a short preliminary conclusion, all locking mechanisms which impose deadlocks and therefore trigger transaction rollbacks (may it be RIX, URIX, or taDOM) can currently not compete with the deadlock-free exclusive document lock in terms of throughput, but they are more responsive with respect to individual transaction runtimes.

5.4 Workload Variants

But before we acknowledge defeat, there are still optimizations for taDOM left that we have not considered yet. As introduced in 4.3.5, we prepared different implementations of the same XMark transactions. Semantically, these workload variants are equivalent, but they utilize slightly different node operations to fulfill their task, or explicitly acquire locks here and there to reduce the risk of deadlocks caused by lock conversions.

For the next benchmark we therefore distinguish between four different workload variants:

Navigation-based:
Navigation-based child access and no preemptive locking

Stream-based:
Stream-based child access and no preemptive locking

Navigation-based + Preemptive:
Navigation-based child access with preemptive locking

Stream-based + Preemptive:
Stream-based child access with preemptive locking

The previous experiment to evaluate the effect of lock depth and lock escalation strategy made use of the first variant, i.e., the transactions always perform navigation steps like `First-Child` and `Next-Sibling` whenever child traversal is necessary. The other alternative is to call `getChildren()` in order to retrieve a child stream and a single LR lock on the context node. Although stream-based access might sound like the clear winner, there are situations where single navigation steps make more sense, e.g., when only a few of the first children are actually accessed.

If indeed the deadlock detection and resolution was the limiting factor for transaction throughput in the previous benchmark, we should also notice a difference when applying preemptive locking, since this strategy is meant to drastically reduce deadlocks or even prevent them altogether.

The remaining benchmark conditions are again similar to those in previous measurements to ensure comparability. However, since the lock escalation strategy did not seem to have a huge impact on the performance, it is disabled for the following benchmark. As it turned out, applying lock escalation involves a small risk of conversion deadlocks. In general, locks are only escalated if the granted mode of the parent node is compatible with the intended escalation mode. But there is a short time frame between checking the parent's lock mode and the actual process of lock escalation, in which another transaction could in fact obtain a lock on the parent and thereby block the escalation request, ultimately leading to conversion deadlocks on occasion.

In total, the next benchmark is characterized by:

Variables:

- Workload variant:
  - Navigation-based
  - Stream-based
  - Navigation-based + Preemptive
  - Stream-based + Preemptive

Constants:

- Lock protocol: taDOM
- Number of threads: 8
- Skew: high
- Buffer: large
- Log: Flash
- Lock depth: infinite
- Lock escalation: disabled

As ususal, let us first discuss the achieved transaction throughput for each of these taDOM variants and confront it with the primitive exclusive protocol, as illustrated by Figure 20. Keep in mind that the first workload variant (*navigation-based*) corresponds to the configuration of the previous benchmark but without lock depth and lock escalation tweaks. Accordingly, this scenario achieves a throughput of slightly more than 20,000 transactions per minute. Modifying the workload implementation by substituting single navigation steps by child streams (*stream-based*) does not have a striking impact on performance either.

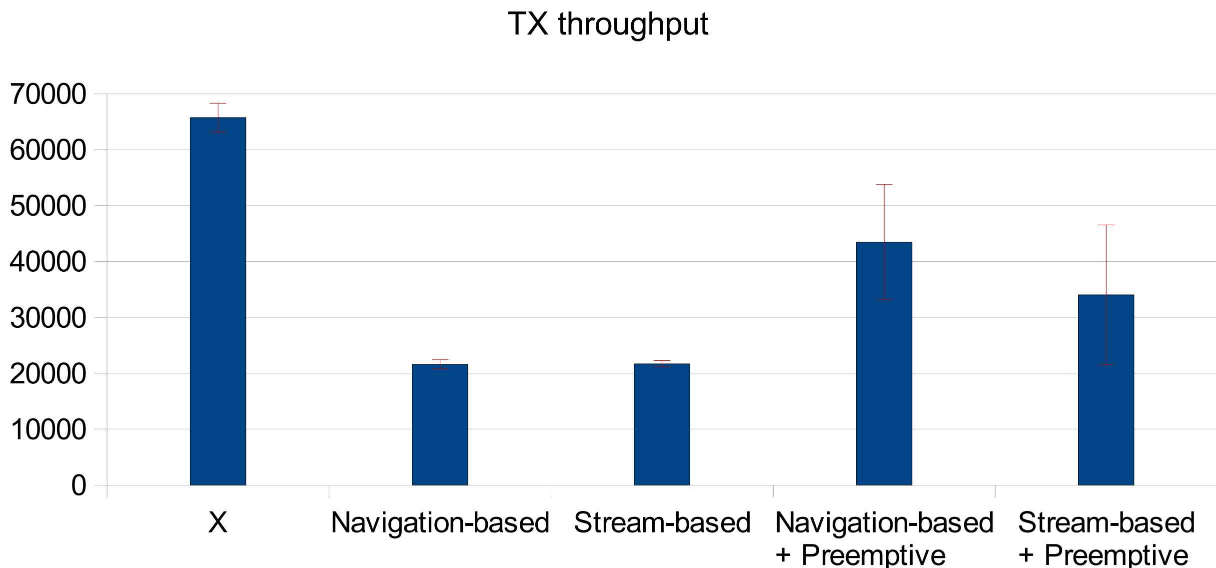


Figure 20: Different workload variants (Transaction throughput)

What improves the throughput significantly, however, is reducing the risk of deadlocks, which is done in the third and fourth workload variant by exploiting preemptive locking. As a result, the abort rate could be brought down from 2.8% to literally 0%. In case of the navigation-based approach, the preemptive locking strategy roughly doubles the throughput, while it also improves the stream-based implementation considerably, yet not to the same extent.

The reason why this diagram displays an additional error bar is to emphasize the high standard deviation for the two measurements involving preemptive locking. No matter how often the experiment is repeated, these high fluctuations in the transaction throughput remain. Which means in essence that some benchmark runs with preemptive locking achieve extremely low throughputs (such as 20,000), and other runs succeed in processing a multitude of transactions per minute (such as 70,000) under the exact same starting conditions. But what can be observed regardless of this high jitter is the overall increased throughput by eliminating deadlocks.

Yet again, neither of the provided workload variations comes close to the exclusive document lock for some reason. As a matter of fact, the two preemptive strategies have the potential to surpass the X results, but in average, considering all measurements, the preemptive variants stand only between the classical taDOM variants and the simplistic X lock approach.

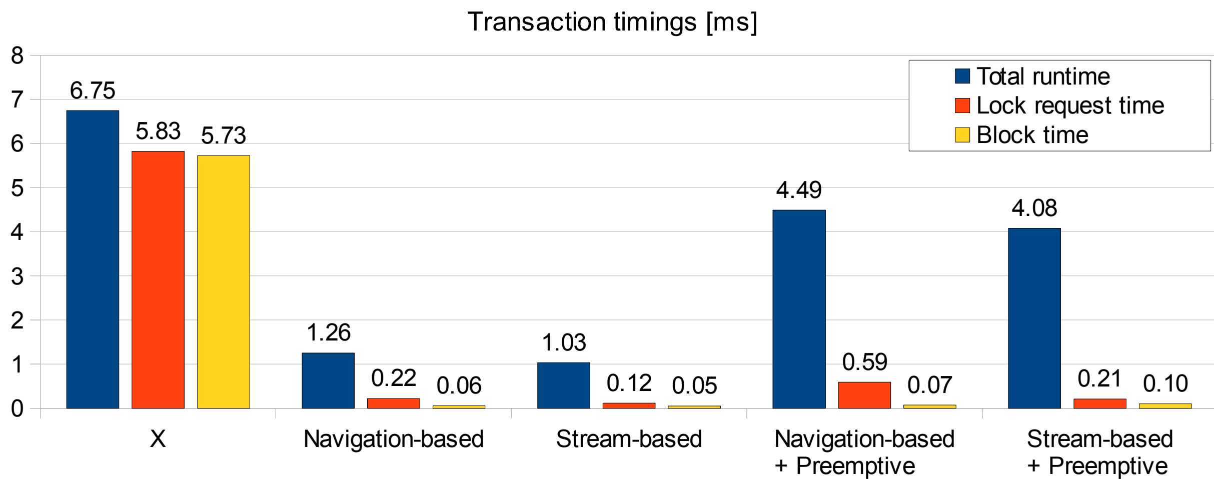


Figure 21: Different workload variants (Transaction timings)

Comparing *navigation-based* and *stream-based* child access in terms of their average transaction timings, as shown in Figure 21, supports some of our previously mentioned ideas. For instance, the lock manager overhead (i.e., gap between second and third bar) is reduced when obtaining a single LR lock instead of a multitude of node and edge locks. Although this might appear as an effect of random fluctuations once again, the results for the non-preemptive workload variants are pretty stable. In fact, the stream-based strategy consistently exposes lower lock request times throughout all benchmark repetitions. Also the total query runtime turns out to be lower if child streams are utilized instead of navigation steps, which might indicate a slight performance benefit resulting from storage-related aspects (e.g., more efficient sibling traversal for stream-based access).

More puzzling are again the results for the two preemptive scenarios. Although preemptive locking merely adds a handful of manual lock acquisitions to the workload implementation, it has a devastating impact on the transaction timings, in particular the total transaction duration. We know from the X-lock measurements that higher runtimes do not necessarily imply degraded system performance in terms of throughput. But up to this point, we considered the gap between the total runtime and the lock request time (that is, the overhead due to non-locking-related components) as a fixed quantity of roughly one millisecond per transaction. After several reruns of the same benchmark configuration, we have to assume that this non-locking overhead is indeed significantly increased for both workload variants with preemptive locking. It would not

be surprising if this phenomenon is somehow related to the unstable throughput measurements from Figure 20.

Next up, we should discuss the question how preemptive locking can drastically affect the performance of non-locking-related parts of the system, as indicated by the timings in Figure 21. Although the lock request time is also alarmingly increased for the preemptive navigation-based scenario, the gain in non-locking overhead clearly dominates the picture. One theory for this observation might suggest that by eliminating deadlocks through preemptive locking, the throughput is increased in such a way that the transaction threads run into a new kind of synchronisation bottleneck, which was not measurable for the non-preemptive scenarios. That might explain why the transaction throughput for preemptive locking is indeed improved, but not to the extent we would hope for. Furthermore, the exclusive document lock strategy might not suffer from the same effect because only one single thread effectively accesses the document at each point in time.

There are many components in BrackitDB apart from the locking system where thread synchronisation must be performed, for instance at the buffer manager or the log manager. Depending on the skew, the transaction threads might also be blocked due to the page latching mechanism in the document index. One last benchmark attempt might give us some more clues about potential system bottlenecks.

5.5 I/O Edge Cases

In our last series we are going to examine the influence of different I/O setups in combination with different locking strategies. Intuitively, we would assume that increased I/O delays due to buffer page loads or log flushes favors a configuration where more parallelism can take place. While threads waiting for I/O calls could easily be evicted from their CPU core and thereby make room for active threads, a single-threaded I/O bound system basically leaves the CPU unutilized during disk interaction.

In order to investigate these effects, the next benchmark again compares the single-threaded X locking approach and the four different workload variants employing taDOM under different I/O setups. Up to this point, we only saw results for a medium amount of I/O, accomplished by frequent log flushes to an SSD device but a sufficiently large page buffer to keep the entire document in main memory. For our next test, we therefore look at the two edge cases *full I/O* (i.e., as much I/O as our benchmark can deliver) and *no I/O* at all.

On that note, full I/O is achieved by a tiny buffer ratio of 0.01 and rather slow log flushes to an HDD. No I/O, in contrast, makes use of a large page buffer and disables log flushes completely. While log records are still kept in memory to allow for transaction rollbacks, they are never persisted so that potential crash recovery and thus ACID durability is abandoned in this scenario. Moreover, the skew is lowered to make sure that a larger portion of the document is accessed, thereby provoking page misses for the small buffer in the full I/O scenario.

Variables:

- Lock protocol: X, taDOM (+ workload variants)
- I/O setup:
  - Full I/O (Buffer: small, Log: HDD)
  - No I/O (Buffer: large, Log: disabled)

Constants:

- Number of threads: 8
- Skew: low
- Lock depth: infinite
- Lock escalation: disabled

Following our usual procedure, we begin with an evaluation of the transaction throughput in reference to Figure 22. Not surprisingly, adding I/O delay cuts back the throughput considerably. The absolute impact of I/O on performance is particularly evident in the X locking scenario where disk interaction results in a throughput of roughly 20,000 while removing all I/O costs almost achieves 120,000 transaction per minute.

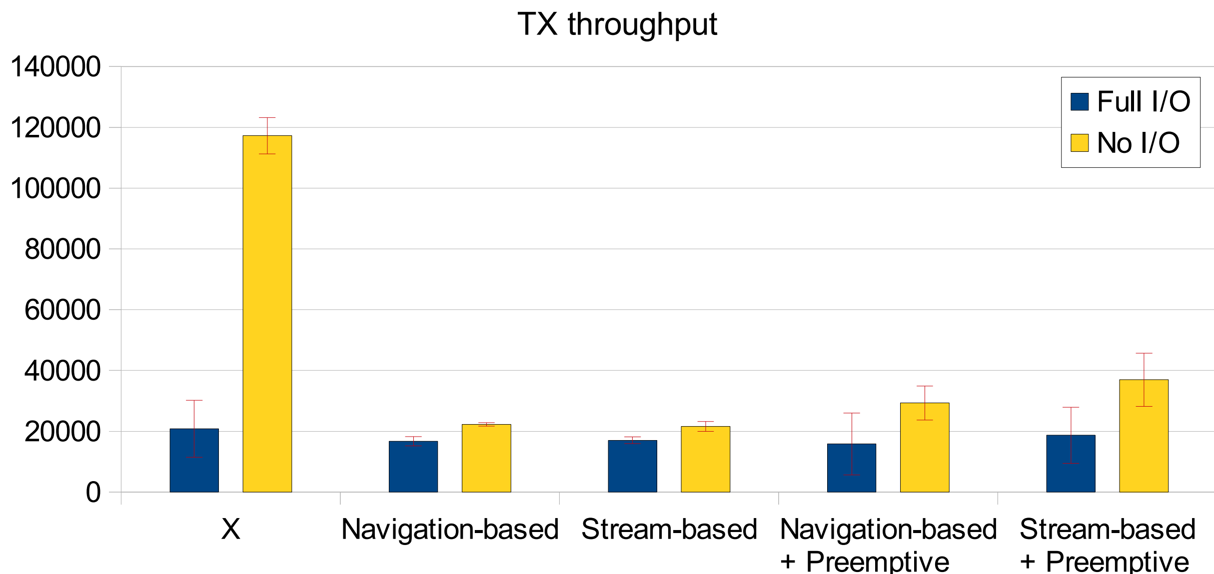


Figure 22: I/O edge cases (Transaction throughput)

Unfortunately, the results do not confirm our proposition that the fine-grained, multithreaded scenarios gain a clear advantage against the sequentially executed baseline in the I/O scenario. Indeed, the gap between the X lock throughput and the corresponding taDOM throughputs has shrinked noticeably when I/O is involved, but the baseline still holds the upper hand on average. It should again be noted that the gathered throughput values were quite jumpy from run to run despite several repetitions. This is particularly apparent for the full I/O setups. Furthermore, although the skew is already lowered in this benchmark, there is still potential for increasing buffer I/O even more, by relying on uniform, random access to the name indexes. Maybe with just a little more I/O involved, the multithreaded execution outperforms the exclusive strategy at last.

While the standard taDOM workloads (navigation-based and stream-based) yield rather consistent results with regard to previous measurements, the numbers for their deadlock-free variants

are once again more puzzling. Instead of improving the throughput values from Figure 20, disabling log-caused SSD interaction even seemed to have degraded performance for preemptive locking. And this is although we see from the X locking results that removing log I/O generally has a positive effect on throughput.

Finally, for the sake of completeness, the corresponding transaction timings for both I/O corner cases are presented in Figure 23. The full I/O timings reveal significantly increased transaction runtimes due to buffer delays (and possibly other effects as well). Looking at the default navigation-based and stream-based taDOM variants, we observe that adding I/O adds more than six milliseconds to the total runtime of any individual transaction, while the locking overhead only slightly increases (by less than 0.2 ms in absolute numbers). This is exactly one of the benefits we anticipate from fine-grained locking: Waiting times are less dependent from the duration of other transactions, as opposed to exclusive locking where each transaction has to wait for over 23 milliseconds in the lock request queue until all preceding transactions finally commit.

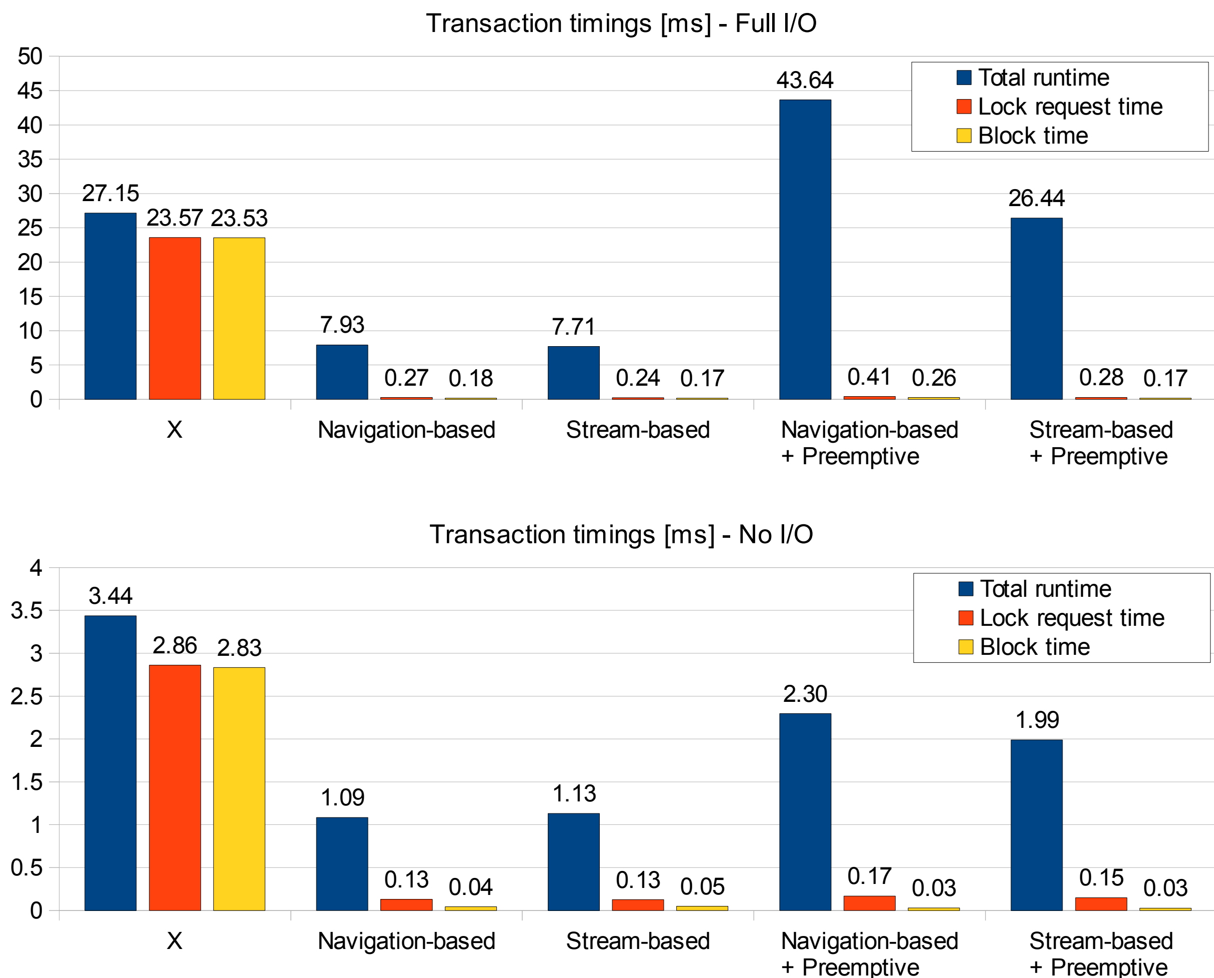


Figure 23: I/O edge cases (Transaction timings)

The preemptively locked variants again turn out to behave suspiciously in terms of their timing values. Since we have to assume that every locking variant suffers from roughly the same I/O overhead, the considerably higher total transaction runtimes of the two preemptive locking

approaches is hard to comprehend. If they were caused by thread synchronisation bottlenecks (such as latching issues), as speculated in the previous benchmark discussion, we would rather expect that additional I/O smoothens these effects out, instead of scaling them up even more.

5.6 Analyzing Throughput Bottlenecks

Without further in-depth experiments, it is difficult to tell why the fine-grained locking approaches do not achieve comparable transaction throughputs compared to the "one at a time" strategy of exclusive document locking. We found some indications that the lock manager and the applied protocol is not the limiting factor, such as implied by the constantly low locking-related overhead (illustrated as the central bars in the timing diagrams). We could identify the deadlock detection and resolution mechanism as one of the bottlenecks, since bringing down the abort rate to zero improved the overall throughput considerably, in exchange for newly arising questions.

At this point, one might even go so far as to theorize that disabling locking altogether and running the same amount of worker threads concurrently on the document without any isolation control would expose the same kind of throughput bottleneck as with taDOM enabled. Unfortunately, running a highly concurrent workload without any locking mechanism in place might quickly render the benchmark document inconsistent.

What might prove insightful as to finding multithreading issues is a step-by-step concurrency analysis of every lower layer component that involves any thread synchronisation. For instance, first check the buffer manager in an isolated testbed by running a multithreaded workload that excessively fixes, unfixes and allocate pages, then proceed with evaluating the log buffer implementation appropriately, followed by a benchmark of the page latching mechanism imposed by the B^+ tree implementation, and so on. If all these components turn out to have an adequate multithreading performance, we should again focus on locking mechanisms.

But until then, it would be premature to blame the applied locking protocol for meager transaction throughputs. The prevailing locking mechanism according to the previous benchmarks (which was simply referred to as X), might only yield the best throughput because it actually prevents any concurrency issues from occurring in the first place.

6 Conclusion

In this thesis, we introduced the general architecture of an XML database management system, exemplified by BrackitDB. Many long-serving, well established techniques and design principles from relational systems are applied, combined with more XML-specific innovations the higher we climb up the architectural ladder.

We discussed the purpose of concurrency control which is required whenever transactions are allowed to access shared data in an interleaved fashion. In this context, the notion of serializability as the general measure for correctness was brought into play. Locking data items prior to accessing them, while holding these locks until the transaction commits, is one way of achieving serializability.

Various locking techniques have been invented for relational database systems, one of them being multi-granularity locking where the database is arranged as a tree of different granularities, from the complete database at the top, down to single records at the bottom. It was recognized that this locking principle can be transferred to XML data as for its hierarchical nature. In fact, the locking protocol employed by BrackitDB, known as taDOM, extends these ideas considerably by establishing new lock modes of varying granularity. These lock modes are specifically tailored to provide minimal and yet sufficient transaction isolation on top of the DOM interface. Depending on the invoked operation, single nodes up to entire subtrees are transparently locked on demand to ensure adequate protection. However, node and subtree locks are not quite sufficient to cope with phantoms that can be encountered by performing navigation steps. For this reason, edge locks play a vital role for achieving true serializability. If secondary indexes are utilized to retrieve data, they also need to implement measures against phantoms, which is covered by index locks.

Evaluating the above described locking concept was the primary concern in this thesis. In order to set up a suitable benchmark environment, BrackitDB had to be extended by appropriate statistic-tracking features that needed to be accessible to the benchmark software. Furthermore, many practical issues had to be overcome for BrackitDB to cope with highly concurrent workloads without ending up in unexpected deadlock situations.

Eventually, taDOM was put to the test against alternative locking protocols, such as RIX, URIX, and a simple exclusive document lock. Since taDOM offers finer-grained locks compared to its competitors, more concurrency within the document should result from it. Ultimately, the question is whether this increased potential for concurrency compensates for the higher locking overhead that is entailed by finer-grained locks.

We observed that techniques for dynamically coarsening the lock granularity (known as lock escalation) to reduce the overhead of managing plenty of fine-grained locks do not impact transaction throughput noticeably in our benchmark setup. However, according to the measurements, lock escalation was rarely applied at all during the runs. This issue might be fixed by making the escalation strategy more aggressive or adding some long-running transaction to the workload mix. Another taDOM-related parameter, the statically fixed lock depth, has a major influence on system performance, especially when set to very low values. It might have a slight positive impact compared to infinite lock depth, but the optimal value heavily depends on the document structure and workload (for our benchmark, the optimum was roughly on lock depth 3).

Unfortunately, with reference to the baseline experiments, neither of the taDOM setups was able to surpass the incredibly simple X lock approach that essentially reduced the database to a sequential transaction processor. We tried to overcome this issue by providing different workload variants, two of which even offered deadlock-free execution. By eliminating deadlock resolution overhead we could indeed observe an increased transaction throughput that is, however, still exceeded by the global exclusive lock strategy. Besides, some of the measurements taken from the deadlock-free workload variants raised more questions than they answered.

Considering all facts from the benchmark, it appears the general problem with respect to transaction throughput is not related to taDOM in particular, as the effect was evident for every locking protocol that allowed for concurrency in the document to a certain extent (e.g., RIX and URIX). Furthermore, the individual transaction timings (between one and two milliseconds per transaction) looked quite reasonable. Especially the rather short duration that a transaction usually spends in locking-related code is an indication to not blame the locking technique for low throughputs.

In its current state, multithreading in general seems to be the cause for the limited throughput, even though the benchmark CPU features several cores for truely parallel thread execution. While linear scalability in parallel programming is of course unrealistic, we should at least see slight benefits when allowing an increased amount of threads accessing the data at the same time. Therefore, the next step would be to further investigate the observed synchronization bottleneck.

In the light of the multicore era, there is no doubt that today's software should aim for adequate multithreading scalability in order to be competitive. Another trend emanates from advancements in storage media which become progressively larger, cheaper, and faster. Consequently, database systems will be increasingly restrained by the CPU and the efficiency of their algorithms. In the end, it thus comes down to how much concurrency the database allows while minimzing the CPU overhead inflicted by appropriate concurrency control mechanisms. Fine-grained lock protection with the potential to dynamically adjust lock granularity, such as offered by taDOM, therefore poses a promising candidate for this challenge. However, presumably caused by a synchronization bottleneck that has yet to be identified, the experiments were unable to provide the practical evidence for this assumption.

7 References

[1] Brackit - Google Project Hosting. http://code.google.com/p/brackit/. [Last Access: 2014-02-26].

[2] BrackitDB - Google Project Hosting. http://code.google.com/p/brackit/wiki/BrackitDB. [Last Access: 2014-02-22].

[3] gnuplot - Project Homepage. http://www.gnuplot.info/. [Last Access: 2014-01-24].

[4] Normal distribution - Encyclopedia of Mathematics. http://www.encyclopediaofmath.org/index.php/Normal_distribution. [Last Access: 2014-01-10].

[5] TCP parameters, Linux kernel - The STAR experiment. https://drupal.star.bnl.gov/STAR/blog-entry/jeromel/2009/feb/18/tcp-parameters-linux-kernel. [Last Access: 2014-02-07].

[6] XMark - An XML Benchmark Project. http://www.xml-benchmark.org/index.html. [Last Access: 2014-01-09].

[7] IBM Systems Information Center - Isolation Levels. http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.common.doc/doc/banner.htm, May 2007. [Last Access: 2014-01-20].

[8] Apache Software Foundation. Apache Commons Pool - Project Homepage. http://commons.apache.org/proper/commons-pool/. [Last Access: 2014-02-03].

[9] Apache Software Foundation. Apache Derby - Project Homepage. http://db.apache.org/derby/. [Last Access: 2014-01-31].

[10] S. Bächle. *Separating key concerns in query processing: set orientation, physical data independence, and parallelism*. PhD thesis, Verl. Dr. Hut, Müünchen, 2013.

[11] S. Bächle and T. Härder. Realizing Fine-Granular and Scalable Transaction Isolation in Native XML Databases. In *SYRCoDIS*, 2008.

[12] S. Bächle and T. Härder. Tailor-made lock protocols and their DBMS integration. In *Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management*, page 18–23, 2008.

[13] S. Bächle and T. Härder. The real performance drivers behind XML lock protocols. In *Database and Expert Systems Applications*, page 38–52, 2009.

[14] S. Bächle, T. Härder, and M. P. Haustein. Implementing and optimizing fine-granular lock management for XML document trees. In *Database Systems for Advanced Applications*, page 631–645, 2009.

[15] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[17] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *Software Engineering, IEEE Transactions on*, (3):203–216, 1979.

[18] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS fast path. *IEEE Database Eng. Bull.*, 8(2):3–10, 1985.

[19] J. Gray and R. A. Lorie. Granularity of locks and degrees of consistency in a shared data base. *... on Modelling in Data Base ...*, pages 365–394, 1976.

[20] T. Härder, M. P. Haustein, C. Mathis, and M. Wagner. Node labeling schemes for dynamic XML documents reconsidered. *Data & Knowledge Engineering*, 60(1):126–149, Jan. 2007.

[21] T. Härder, C. Mathis, and K. Schmidt. Comparison of complete and elementless native storage of XML documents. In *Database Engineering and Applications Symposium, 2007. IDEAS 2007. 11th International*, page 102–113, 2007.

[22] T. Härder and E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung.* Springer, Berlin; Heidelberg; New York; Barcelona; Hong Kong; London; Mailand; Paris; Tokio, 2001.

[23] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[24] M. P. Haustein. Eine XML-Programmierschnittstelle zur transaktionsgeschützten Kombination von DOM, SAX und XQuery. In *BTW*, page 265–284, 2005.

[25] M. P. Haustein. *Feingranulare Transaktionsisolation in nativen XML-Datenbanksystemen.* Verl. Dr. Hut, München, 2005.

[26] M. P. Haustein. Verhinderung von Phantomen in XML-Datenbanksystemen mit wertbasierten Achsensperren. In R. Eckstein and R. Tolksdorf, editors, *Berliner XML Tage*, pages 79–92, 2006.

[27] M. P. Haustein and T. Härder. taDOM: A tailored synchronization concept with tunable lock granularity for the DOM API. In *Advances in Databases and Information Systems*, page 88–102, 2003.

[28] M. P. Haustein and T. Härder. An efficient infrastructure for native transactional XML processing. *Data & Knowledge Engineering*, 61(3):500–523, 2007.

[29] M. P. Haustein and T. Härder. Optimizing lock protocols for native XML processing. *Data & Knowledge Engineering*, 65(1):147–173, Apr. 2008.

[30] M. P. Haustein, T. Härder, C. Mathis, and M. Wagner. DeweyIDs-The Key to Fine-Grained Management of XML Documents. In *SBBD*, volume 5, page 85–99, 2005.

[31] C. Mathis. *Storing, indexing and querying XML documents in native XML database management systems*. PhD thesis, Verl. Dr. Hut, München, 2009.

[32] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *VLDB*, page 392–405, 1990.

[33] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[34] C. Mohan and F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-ahead Logging. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, page 371–380, New York, NY, USA, 1992. ACM.

[35] Red Hat, Inc. Hibernate ORM - Project Homepage. `http://hibernate.org/orm/`. [Last Access: 2014-01-31].

[36] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Upper Saddle River, 2001.

[37] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[38] E. W. Weisstein. Normal Distribution – from Wolfram MathWorld. `http://mathworld.wolfram.com/NormalDistribution.html`. [Last Access: 2014-01-10].

[39] World Wide Web Consortium (W3C). Document Object Model (DOM) Level 2 Core Specification. `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/`. [Last Access: 2014-02-13].

[40] World Wide Web Consortium (W3C). Document Object Model (DOM) Level 3 Core Specification. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`. [Last Access: 2014-02-13].

[41] World Wide Web Consortium (W3C). W3C DOM4. `http://www.w3.org/TR/dom/`. [Last Access: 2014-01-10].

[42] World Wide Web Consortium (W3C). XQuery 1.0: An XML Query Language (Second Edition). `http://www.w3.org/TR/xquery/`. [Last Access: 2014-02-26].