João Paulo da Conceição Soares

Mestre em Engenharia Informática

Scaling In-Memory databases on multicores

Dissertação para obtenção do Grau de Doutor em

Engenharia Informática

Orientador: Nuno Manuel Ribeiro Preguiça,
Professor Associado,
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Júri

Presidente:	José Legatheaux Martins
Arguentes:	Fernando Pedone
	Alysson N. Bessani
Vogais:	Rodrigo M. Rodrigues
	João M. Lourenço
	Nuno M. Preguiça

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Outubro, 2015

Scaling In-Memory databases on multicores

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

To my parents and my brother, and to my wife and son.

Acknowledgements

The work presented in this dissertation would not have been possible without the collaboration of a number of people to whom I would like to express my gratitude.

First and foremost I would like to deeply thank my thesis advisor Nuno Preguiça, who, patiently, always helped me to overcome all the challenges that I had to face during my PhD studies. I've learned a lot from him, and I truly hope to be able to advise my students as well as he advised me.

To João Lourenço for always being available to discuss ideas and give excellent tips and suggestions which allowed me to improve the quality of this work.

To all my colleagues with whom I shared the ASC open space, including: David Navalho, Valter Balegas, Bernardo Ferreira, João Leitão, and Tiago Vale.

Finally, my very heartfelt thanks to my parents José and Elisa. To Isabel, for patiently keeping up with me, and for giving me my beautiful son José.

I also would like to acknowledge the following institutions for their hosting and financial support: Departamento de Informática and Faculdade de Ciências e Tecnologia of the Universidade Nova de Lisboa; Centro de Informática e Tecnologias da Informação of the FCT/UNL; Fundação para a Ciência e Tecnologia in the PhD research grant SFRH/BD/62306/2009, and in the research projects RepComp (PTDC/ EIA-EIA/ 108963/ 2008), PEst-OE/ E-EI/ UI0527/ 2011, SwiftComp (PTDC/ EEI-SCR/ 1837/ 2012), PEst-OE/ EEI/ UI0527/ 2014 and EU FP7 project SyncFree (grant no 609551).

Abstract

Current computer systems have evolved from featuring only a single processing unit and limited RAM, in the order of kilobytes or few megabytes, to include several multicore processors, offering in the order of several tens of concurrent execution contexts, and have main memory in the order of several tens to hundreds of gigabytes. This allows to keep all data of many applications in the main memory, leading to the development of in-memory databases. Compared to disk-backed databases, in-memory databases (IMDBs) are expected to provide better performance by incurring in less I/O overhead.

In this dissertation, we present a scalability study of two general purpose IMDBs on multicore systems. The results show that current general purpose IMDBs do not scale on multicores, due to contention among threads running concurrent transactions. In this work, we explore different direction to overcome the scalability issues of IMDBs in multicores, while enforcing strong isolation semantics.

First, we present a solution that requires no modification to either database systems or to the applications, called *MacroDB*. MacroDB replicates the database among several engines, using a master-slave replication scheme, where update transactions execute on the master, while read-only transactions execute on slaves. This reduces contention, allowing MacroDB to offer scalable performance under read-only workloads, while update-intensive workloads suffer from performance loss, when compared to the standalone engine.

Second, we delve into the database engine and identify the concurrency control mechanism used by the storage sub-component as a scalability bottleneck. We then propose a new locking scheme that allows the removal of such mechanisms from the storage sub-component. This modification offers performance improvement under all workloads, when compared to the standalone engine, while scalability is limited to read-only workloads. Next we addressed the scalability limitations for update-intensive workloads, and propose the reduction of locking granularity from the table level to the attribute level. This further improved performance for intensive and moderate update workloads, at a slight cost for read-only workloads. Scalability is limited to intensive-read and read-only workloads.

Finally, we investigate the impact applications have on the performance of database systems, by studying how operation order inside transactions influences the database

performance. We then propose a Read before Write (RbW) interaction pattern, under which transaction perform all read operations before executing write operations. The RbW pattern allowed TPC-C to achieve scalable performance on our modified engine for all workloads. Additionally, the RbW pattern allowed our modified engine to achieve scalable performance on multicores, almost up to the total number of cores, while enforcing strong isolation.

Keywords: In-Memory Databases, Performance Study, Scalability Study, Contention, Serializability, Replication, Latch-Free, Attribute Level Locking, Read-before-Write Pattern

Resumo

Os sistemas de computadores atuais evoluíram a partir de sistemas com uma única unidade de processamento e memória RAM limitada, na ordem de kilobytes ou alguns megabytes, para incluir vários processadores com múltiplos núcleos, oferecendo na ordem de várias dezenas de contextos de execução em simultâneo, e com memória principal na ordem de várias dezenas a centenas de gigabytes. Isto permite a muitas aplicações manter todos os seus dados na memória principal, conduzindo ao desenvolvimento de bases de dados em memória. Comparativamente aos sistemas de gestão de bases de dados suportadas por disco, espera-se que os sistemas de gestão de bases de dados em memória (SGBDM) proporcionem um melhor desempenho por incorrer em menos I/O.

Nesta dissertação, apresentamos um estudo da escalabilidade de dois SGBDM em sistemas de computadores com múltiplos núcleos. Os resultados mostram que as atuais SGBDM não escala em multicores, devido contenção entre fluxos de execução quando executam transações concurrentes. Neste trabalho, exploramos diferentes direções para superar os problemas de escalabilidade de SGBDM em processadores com múltiplos núcleos, preservando uma semântica de isolamento forte.

Primeiro, apresentamos uma solução que não requer quaisquer modificações para ambos os sistemas de bases de dados ou para as aplicações, chamado *MacroDB*. MacroDB replica a base de dados em vários SGBDM, usando um esquema de replicação mestre-escravo, onde transações de escrita executam no mestre, enquanto transações de leitura executam nos escravos. Este modelo de replicação reduz a contenção, permitindo ao MacroDB oferecer um desempenho escalável sob cargas maioritariamente de leitura, enquanto cargas intensas de escritas sofrem de perda de desempenho, quando comparado com o motor independente.

Seguidamente, focando-nos no motor de bases de dados, identificou-se o mecanismo de controle de concorrência usado pelo sub-componente de armazenamento como um gargalo escalabilidade. Propôs-se então um novo esquema de controlo de concorrência que permite a remoção de tais mecanismos no sub-componente de armazenamento. Esta alteração oferece uma melhoria de desempenho em todas as cargas, quando comparado com o motor original. No entanto, a mesma oferece a escalabilidade limitada somente a cargas maioritariamente de leitura. Em seguida, focando-nos na escalabilidade para cargas de escrita intensa, e propor-se uma redução da granularidade do controlo de concorrência

do nível da tabela para o nível do atributo. Esta modificação melhora o desempenho para cargas maioritariamente de escritas, no entanto impõe um pequeno custo para cargas maioritariamente de leitura. A escalabilidade é limitada a cargas maioritariamente de leitura.

Por fim, investigou-se o impacto que as aplicações têm no desempenho dos sistemas de bases de dados, estudando como a ordem das operação dentro de transações influência o desempenho dos mesmos. Então, propôs-se um padrão de interação no qual uma transação realiza todas as operações de leituras antes das operações de escrita, Ler antes de Escrever (LaE). A modificação do TPC-C segundo o padrão LaE permite atingir um desempenho escalável no nosso motor modificado para todas as cargas. Além disso, o padrão LaE permite ao nosso motor modificado atingir um desempenho escalável em processadores com múltiplos núcleos, quase até o número total de núcleos, mesmo quando aplicando uma semântica de isolamento forte.

Palavras-chave: Sistema de gestão de bases de dados em memória, Estudo de desempenho, Estudo de escalabilidade, Contenção, Serialização, Replicação, Controlo de concorrência ao nível dos atributos, Padrão Ler antes de Escrever

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Gordon Moore, back in 1965, observed that the number of transistors in an integrated circuit would double every 12 to 24 months [Moo98]. This has proven to be true, namely in CPUs, and, for a long period of time, as the number of transistors grew so did the processor clock frequency. This increase in clock frequency resulted in a direct improvement in CPU performance, which in turn led to a *free* performance improvement for applications [Pan+08; Sut05].

However, in the late 1990s, the traditional pursuit for improving CPU performance ended, leading to the stabilization of CPU clock frequencies. This results from the fact that further increasing the clock frequency leads to an unmanageable increase in energy consumption and heat emission [Ham+97; Olu+96; Vac+05].

Although the CPU clock frequency has stabilized, it was still possible to continue increasing the number of transistors in a single chip. This has led to a new paradigm for improving CPU performance was adopted, based on new CPU designs. This new paradigm improves CPU performance by increasing the number of processing units instead of increasing the performance of a single unit [Ham+97; Olu+96; Vac+05]. Current modern CPU architectures feature multiple processor cores per chip, and are known as *multicores*. Additionally, each core often provides hardware support for multiple execution contexts, also known as *hyper-threading* [Tul+95].

Other components, namely main memory, have also benefitted from the increase in transistor count. In this case, the increase resulted in a proportional growth of the total amount of main memory supported by computer systems. Like CPUs, memory clock frequencies, has also increased.

In summary, computer systems have evolved from, typically, featuring only a single processing unit and limited RAM, in the order of kilobytes or few megabytes, to include several *multicore* processors, offering in the order of several tens of concurrent execution

contexts, and have main memory in the order of several tens to hundreds of gigabytes.

This evolution, in particular the existence of multiple cores, pose important challenges to the design of application in order to efficiently explore the power of current systems [Bau+09; Har+07; Pan+08; Pap+08; Sal+11]. As a result, considerable research efforts have been exploring new ways for developing applications for multicore systems [AC09; Bau+09; Cle+13; HM93; Lea00; Meh+09; Pan+08; Son+11; Tiw+10; Wam+13; Zha+07]. In this work we focus on some of the challenges that multicore systems impose on *Database Management Systems*.

1.1 Database management systems

Database systems are the underlying building block of information systems. For over 50 years, databases became fundamental to the operations of most organizations, being a central part of our day-to-day life [CB09; Sil+06; Zha+15].

Ever since the development and deployment of the first databases, back in the 1950s, that database research has been an active and challenging field. In 1970, James Codd proposed a new data model and nonprocedural ways of querying data [Cod70]. This gave birth to the relational database, a key component of current information systems architectures [BN97; Niu+13; Xu11].

The foundations of modern general purpose relational database management systems (RDBMS) were laid back in the 1970's, with the pioneer work on early RDBMS, like System R [Ast+76; Cha+81; Gra78] and Ingres [Sto+86]. Traditionally, RDBMSs feature a disk-backed storage system, a log-based transaction manager, and a lock-based concurrency control mechanism [Cha+81; Gra78; GR92].

Current database design is still highly influenced by these initial works, which were developed when online transaction processing (OLTP) databases were many times larger than main memory, and disk I/O was the predominant performance bottleneck [Sto81]. At the time, efficient I/O subsystems were critical for the overall performance of DBMS. Thus, most of the research in DBMS focused, primarily, on buffer pool management, fine-grain concurrency control and sophisticated caching and logging schemes for efficiently multiplexing concurrent transactions, while hiding disk latency [Che+94; GR92].

As discussed before, resources available in current systems are considerably different. When compared to single core processors, multicore architectures offer increased computational power provided by multiple processors running concurrently. Additionally, the increasing amount of main-memory in current computer systems allows the working set of many applications to fit entirely in main memory [Har+08; Joh+09b; Zha+15]. This allows databases to move application data from disk to memory to a large extent or even completely.

This evolution in hardware contributes to the improvement of performance in DBMS. However, it also poses a number of challenges on how to fully explore the resources available in modern computing systems. In fact, for DBMSs to achieve scalable performance,

these must efficiently utilize the available hardware contexts offered by current multicore systems. However, general purpose database design have changed little, still relying on optimizations for computer technologies of the late 1970's [Har+08].

1.2 Problem statement

It has been shown that current general purpose databases can spend more than 30% of time in synchronization-related operations (e.g. locking and latching), even when only a single client thread is running [Har+08; Pan+10]. It has also been shown that running two concurrent database operations in parallel can be slower than running them in sequence, due to workload interference [Pan+10; Sal+11]. These are some of limiting factors for databases to scale on current multicore systems [Joh+09a; Joh+09b; Unt+09; Zho+05].

Several different approaches have been proposed for improving database resource usage of multicore machines. Some of this research aims at using multiple threads to execute query plans in parallel; using new algorithms to parallelize single steps of a plan; or effectively parallelize multiple steps [Che+95; CR08; Cie+10; Ye+11; Zha+13; Zho+05]. Other solutions try to reuse part of the work done during the execution of multiple queries, or using additional threads for prefetching data that may be needed in the future [Gia+12; Pap+08; Zho+05].

Additionally, some works propose the use of techniques from replicated and distributed systems, like data replication or partitioning, to multicore environments[Mao+12; Pan+11; Sal+11]. Others propose a complete redesign of the database engine for efficiently running on multicore machines [Dia+13; Tu+13].

However, most of these approaches require extensive database redesign, which contributes negatively to the acceptance of these proposals by the community. Thus general purpose databases have been slower to adopt them. Nonetheless, some of these solutions start to appear in niche markets.

The goal of this work is to study how general purpose In-Memory Databases (IMDBs) can benefit from the resources available in modern computer systems, namely the multiple cores and abundant memory. Compared to disk-backed databases, IMDBs are expected to provide high performance by incurring in less disk I/O overhead. Additionally, since I/O is the main bottleneck for disk-backed database systems, one could expect IMDBs to scale better with the number of cores.

In this dissertation, we start by presenting an experimental study on the scalability of two general purpose IMDBs, to understand the main bottlenecks that exist (Chapter 3.1). The results from this study show that current general purpose IMDBs do not scale on multicores mainly due to contention among threads running concurrent transactions. This contention occurs both due to concurrency control mechanisms for enforcing transaction serializability, and in the low-level concurrency control mechanisms used by the internal data structures.

Our thesis is that it is possible to scale IMDBs on multicores while enforcing strong isolation semantics. To address this goal, we present the following contributions.

First, we propose a design that requires no modification to either the database systems or to the applications, called *MacroDB* (Chapter 4). MacroDB uses database replication techniques, maintaining several replicas of the same database, in a single multicore system and distributes the load among the replicas. This helps improving the scalability of the system by reducing contention. However, this approach is still limited by the scalability of a single database for update-heavy workloads.

Next, we delve into the database engine and identify the major performance bottlenecks to the scalability problem of IMDBs (Chapter 5). We then propose a series of engine modifications for addressing these bottlenecks, reducing internal contention points for improving the system scalability.

Finally, we investigate the impact applications have on the performance of database systems (Chapter 6). We start by studying how operation order, inside transactions, influence the database performance. We then propose combining modifications to the transactions, defined in the applications, and to database engines for improving concurrency among transactions, which leads to improved database scalability.

1.3 List of Publications

We present the list of publications that resulted from the work presented in this document.

- [Mar+10] Paulo Mariano, João Soares, and Nuno Preguiça. Replicated Software Components for Improved Performance. In proceedings of InForum 2010, September 2010.

- [Soa+13a] João Soares, João Lourenço, and Nuno Preguiça. MacroDB: Scaling Database Engines on Multicores. In proceedings of the 19th International Conference on Parallel Processing (EuroPar'13), August 2013.

  These papers present Macro Components and Macro DB design for improving performance on multicores based on replication (Chapter 4.1))

- [SP12] João Soares, and Nuno Preguiça. Improving Application Fault-Tolerance with Diverse Component Replication. In Euro-TM Workshop on Transactional Memory (WTM 2012), April 2012.

- [Soa+13b] João Soares, João Lourenço, and Nuno Preguiça. Software Component Replication for Improved Fault-Tolerance: Can Multicore Processors Make It Work?. In proceedings of 14th European Workshop on Dependable Computing (EWDC'13), May 2013.

These works discuss how to build on the previous design for improving fault-tolerance.

- [Mar+13] Helder Martins, João Soares, João Lourenço, and Nuno Preguiça. Replicação Multi-nível de Bases de Dados em Memória. In proceedings of InForum 2013, September 2013.

  This work presents a distributed version of MacroDB proposed in [Soa+13a].

- [SP15] João Soares, and Nuno Preguiça. Database Engines on Multicores Scale: A Practical Approach. In proceedings of 30th ACM/SIGAPP Symposium On Applied Computing (SAC 2015), April 2015.

  Report on the performance and scalability evaluation of in-memory databases on multicores (Section 3.1.1). Proposal for a new locking protocol that allows the removal of locks from the underlying data structures, for improving in-memory database scalability (Chapter 5).

1.4 Outline of the Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2 This chapter introduces the fundamental concepts for understanding the remaining discussion.

Chapter 3 This chapter presents a performance study of two general purpose IMDB engines on multicores, and discusses the implications that the different database components have on the performance of these systems. It also presents the state-of-the-art of in-memory database management systems and some related work on the techniques and tools related to the matters addressed by this dissertation.

Chapter 4 This chapter presents a generic design for addressing the scalability problem on IMDBs that requires no modifications to either databases or to the applications. This solution builds on the knowledge from distributed and replicated systems, applying them to multicore environments.

Chapter 5 In this chapter we delve into the database engine to fix the major contributors to the scalability problem, i.e., the engine's major performance bottlenecks. To this end, we propose a series of modifications for addressing these problems.

Chapter 6 This chapter presents a study on the impact applications have on the performance of the database, and propose a set of guidelines for modifying transactions that combined with a modified database engine can offer increased concurrency.

Chapter 7 This chapter summarizes the main results and contributions of the research work described in this dissertation, and lists some directions for future research activities.

Chapter 2

Fundamental Concepts

In this chapter we present the research context of this dissertation. We start by describing the main architectural aspects of database management systems. Next, we discuss the major differences between traditional, disk-backed, databases and their memory-backed counterparts, and discuss the design of two general purpose in-memory databases, HSQLDB and H2.

2.1 Relational Databases

Relational database management systems (RDBMS) are the most mature and widely used database systems in production today [JMHH07]. These systems are a fundamental part of the infrastructure that supports many applications including e-commerce, stock management, billing, health and medical records, human resources and payroll management, to name a few.

The foundations of modern RDBMS were laid back in the 1970's, with the pioneer work like System R [Ast+76; Cha+81; Gra78] and Ingres [Sto+86]. Current RDBMS design is still highly influenced by these initial works.

At its core, a RDBMS has four main components[Cha+81; Gra78; GR92; JMHH07]: the communication manager; the process manager; the query processor and the transaction manager, as well as series of additional subcomponents, as illustrated in Figure 2.1. To better understand the function of each of the main components, as well as their interactions, we briefly describe the life of a transaction during its execution. Transactions are a sequence of query and/or update operations delimited by a commit or rollback operation, performed by database clients. In this document, transactions with, at least, one update operation (be it an insert, delete or update operation) are referred to as *update transactions*, while transactions without update operations are referred to as *read-only transactions*.

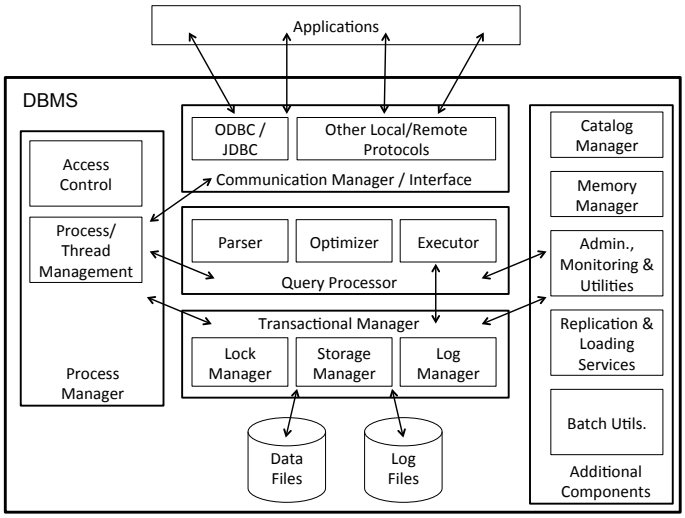


Figure 2.1: General DBMS architecture (Adapted from [JMHH07]).

Consider a simple database interaction where a teacher requests the list of enrolled student in one of its classes. This interaction results in a single-query transaction that follows these, simplified, steps:

1. The client application calls the database API to establish a new connection. A connection can be established directly, via the ODBC or JDBC protocol, or using additional middleware, such as a web or an application server. Each connection is used to send SQL commands from a client to the database and to receive responses from the database (*Communication Manager*).

2. Upon receiving a connection request, the database server assigns a process, or thread, to answer/serve that connection. A connection waits until the *Process Manager* has enough resources available to answer it. When a database process/thread is assigned to a connection, and before executing any SQL commands, it checks and validates user credentials with the *Access Control* component. Connections from non authorized users are dropped.

3. Client can use connections to send SQL operations to the database. These operations are first parsed and checked for syntactic errors by the *SQL Parser*. If error free, operations are compiled and sent to the *Query Optimizer*, which creates a *query execution plan*. *Query Executors* implement a suite of operators, including joins, selection, projection, aggregation, sorting and so on, for executing query plans.

4. Operators that require data from the database use the *Transactional Manager* to access it. This component includes algorithms for managing access to data (*Lock Manager*), and data structures for organizing data on disk (*Storage Manager*). The *Transactional Manager* also ensures that the ACID properties [HR83] of transactions are preserved.

5. After retrieving data from the database, executors produce query results that are sent back to the client. When a client finishes the database interaction it closes the database connection, at which point the database communication manager and transaction manager clean the connection and transaction state, respectively.

Most of the described components impact the overall performance of databases, namely parsers and query optimizers. These components have been extensively studied by the research community and several proposal exist to improve their performance on multicore systems [Che+95; HL09; Kri+09]. Our work is orthogonal to these proposal, focusing on mechanisms for managing concurrent accesses to data.

We now discuss how a database guarantees transactions execute respecting the ACID properties, during (concurrent) execution.

2.1.1 ACID properties

The common database correcteness criteria is that databases must ensure transactions execute according to the four ACID properties: *Atomicity*, *Consistency*, *Isolation* and *Durability*. We now describe these properties, and present how a database guarantees them.

- *Atomicity* defines transaction behavior. According to this property, transactions performed on a database must execute in an "all or nothing" manner, i.e., either all operations of a transaction succeed or non do.

- *Consistency* defines how data evolves from one state to another. Under this property, a database must guarantee its data persist in a consistent state, where all successfully committing transactions must evolve the database from an initial consistent state to a final consistent state.

- *Isolation* defines the allowed interference between concurrent transactions. When taking this property strictly, all operations performed by a transaction must be hidden from all other concurrent transactions, i.e., transactions can only observe changes made by previously committed transactions, prior to its beginning.

- *Durability* defines how the data, and respective changes, must be preserved. According to this property, a database must guarantee that once a transaction completes and successfully commits its results, these results will be visible to subsequent transactions, and will survive the occurrence of any software errors or hardware malfunctions.

Consistency is typically application-specific, with applications being able to capture part of the consistency properties using SQL integrity constraints. Consistency is enforced by runtime checks, that allow transactions to commits only if the constraints are preserved, aborting otherwise.

For the remaining properties, databases guarantee them using a combination of subcomponents, namely: the Lock Manager; the Log Manager; and the Storage Manager. These subcomponents tend to be managed and orchestrated by the Transactional Manager. We now describe each of these subcomponents, and how these are used to ensure the ACID properties.

2.1.2 Storage Manager

The Storage Manager is responsible for managing the database data. Traditional disk-backed databases maintain data in hard disk drives. This includes not only the database data but also the necessary data structures needed for efficiently accessing this data. Typically, a database contains a set of files, with each file containing a set of pages and each page containing a set of records (or tuples). These files, pages and records are used not only to preserve database data but also the necessary meta-data for managing it.

Since the cost of page I/O (*Input* from disk to memory and *Output* from memory to disk) dominates the transaction cost for disk-backed database operations, indices are commonly used for efficiently locating data on disk. Whenever a query requires data from the database, the Storage Manager searches the corresponding index to locate the disk page that holds the data. Traditionally, an index structure is stored in disk, in several blocks, and is loaded into memory as needed, i.e., when the respective blocks are required.

The Storage Manager is also responsible for maintaining the necessary data near the processor, i.e., managing memory buffers that cache the disk data for increased performance. Buffers are extremely important for disk-backed databases due to the difference in bandwidth and latency when accessing data on disk compared to memory. Buffers are arrays of memory stored disk pages called frames, and are used to maintain recently accessed disk pages in memory. For identifying if a given page is presently buffered, a *frame table* is typically used for checking if a frame has already been loaded. When an executor requests for data, if the corresponding frames are present in the buffer, the corresponding data is passed to the executor. If a requested page is not buffered, the Storage Manager issues na I/O request for reading the page from disk into the buffer, before passing the data to the executor.

For preserving database integrity and consistency, buffered frames need to be periodically flushed to disk. Thus, each frame maintains associated metadata for management purposes. This metadata includes, among other information, an access counter (also known as a *pin counter*), used to register the number of executors accessing the frame, and a *dirty bit*, for identifying modified frames. The pin counter allows the Storage Manager to know which frames are being used and which are not, preventing used ones from being evicted from the buffer, while the dirty bit signals frames that need to be written to disk before being removed from the buffer. Periodically, or before replacing a frame in the buffer (for example, due to the buffer being full), modified frames, i.e., frames whose dirty bit are set, are flushed back to disk.

In terms of ACID properties, the Storage Manager is in part responsible for the durability of the database. We say in part since durability does not depend solely of the Storage Manager. While in normal operation, i.e., when the database is properly shutdown, the Storage Manager guarantees that the stored data is consistent, when faults occur this may not be true. For instance, if some buffered frames are dirty and have not been written to disk, database state may become corrupted in the occurrence of a fail stop fault, since some modifications were not propagated to disk. Also, even if dirty buffered frames have been flushed to disk, the corresponding indices stored on disk may not reflect these modifications, since the memory resident versions were not flushed to disk. Like before, a fail stop fault would corrupt the state of the database since, when recovering from the fault (i.e., rebooting), the indices would not be coherent with the stored data. For this reason, additional mechanisms are needed to guarantee durability.

2.1.3 Log Manager

The Log Manager is responsible for recording the modifications performed on a database. Like presented before, durability is not solely guaranteed by the Storage Manager. Instead, databases rely on the Log Manager to achieve this property. To this end, the Log Manager maintains a journal, also know as *log*, of every modification made to the database and writes this log on disk whenever the database is modified. This guarantees the durability of committed transactions, and provides a recovery mechanisms for allowing the database to overcome possible software of hardware faults. Additionally, it also allows aborted transactions to rollback their actions, thus contributing to the atomicity property.

For providing these features, a Log Manager typically maintains a log file on persistent storage (e.g. a file on disk), and a set of related data structures in memory. These data structures maintain the records associated with every update operation performed on the database. For this, databases typically use a Write-Ahead Log (WAL) protocol that operates as follows:

1. Each modification to a database page generates a log record, and the log record must be flushed to the log file before the database page is flushed to disk.

2. Log records must be flushed in order, i.e., a log record cannot be flushed until all log records preceding it have been successfully flushed to disk.

3. Upon a transaction commit request, a commit log record must be flushed to the log file before the commit request returns successfully.

The first rule guarantees that the operations of a transaction can be undone, in the event of a transaction abort. Undoing operations is crucial for achieving atomicity, since without the possibility for recovering actions, a transactions could not undo its modifications in case of an abort. The combination of the other two rules ensure durability, i.e., that the actions of a committed transaction can be recovered after a system crash,

even if these actions are not reflected in the database. This is achieved by redoing every logged operation, including commit requests, and then undoing all operations without a corresponding commit request.

To reduce recovery overhead, by minimizing the work needed to recover from database failures, databases typically rely on *Checkpoints*. A checkpoint reflects all changes made to the database up to a certain point. This allows the corresponding WAL entries to be dropped, since their resulting actions are already reflected in the checkpoint.

2.1.4 Lock Manager

The Lock Manager is responsible for controlling transaction concurrency. Its main function is to guarantee the isolation property. Also, together with the Log Manager provides the atomicity property.

To achieve this, the Lock Manager enforces concurrency control policies on the operations executed by each transaction. Current concurrency control techniques fall into three categories: pessimistic, optimistic or multi-version [Ber+87; JMHH07; KR79]. All concurrency control techniques have the same purpose, to allow non conflicting operations to execute concurrently and prevent conflicting ones from doing so. Conflicts are defined by the type of the operation, and depend on the isolation level being enforced. Generically, read operations do not conflicts with other concurrent read operations, since these do not modify the state of the database, while write operations conflict with every other concurrent operations, since these modify the state of the database. Each concurrency control mechanism prevents the occurrence of conflicts differently, as described next:

- Pessimistic concurrency control enforce concurrency control before operations execute. Under this policy, an operation is only allowed to execute if, at the time the operation will execute, only non conflicting operations are executing. Thus, read operations are only allowed to execute concurrently with other read operations, while a write operation is only allowed to execute in exclusion, i.e., if no other operation is executing concurrently.

- Optimistic concurrency control (OCC) enforces concurrency control after operations execute. Under this policy, an operation in allowed to execute independently of concurrent operations, under the assumption that no conflicts will occur. Additionally, operations do not apply changes directly to the database, tentative changes are kept private to each transaction. Before committing any changes, i.e., before transactions apply the corresponding changes to the database, OCC validates both read and write sets. This validation guarantees that read and written values have not been modified concurrently. A transaction is only allowed to commit if no concurrent modifications are detected, otherwise it must abort.

- Multi-version concurrency control (MVCC) enforces isolation by maintaining multiple versions of each data item, where write operations create new versions of the corresponding item. From a conceptual point of view, transactions execute in a consistent database view (i.e., version), typically defined at the beginning of the transaction. This view remains untouched by any concurrent write operations during the transaction lifespan, i.e., concurrent write operations do not affect the snapshot. Thus, read operations can execute without blocking and never conflict with other operations (by reading in the past). Write operations, made by a transaction, are kept private to each transaction, either in an optimistic or pessimistic way. In an optimistic approach, write operations only modify the corresponding transaction's view, and are maintained by the transaction's write set. This requires an additional validation phase for detecting conflicts. This detection is made before committing changes by validating the transaction's write set. A transaction is only allowed to commit if no concurrent modifications are detected, i.e., if the current version of each item is the same as the transaction's view, otherwise these abort. In a pessimistic approach, write operations execute in place, creating a new version of the corresponding item, and in exclusion. This prevents other concurrent write operations from executing on the same items (e.g., by acquiring locks). This allows transactions to commit without requiring a validation phase.

Concurrency Control		**Concurrent Reads**	**Concurrent Writes**	**Validation**
Pessimistic		Yes	No	-
Optimistic		Yes	Yes	Read and Write Sets
MVCC	**Optimistic**	Yes	Yes	Write Sets
	Pessimistic	Yes	No	-

Table 2.1: Comparison concurrency control mechanisms

Comparing the different concurrency control mechanisms (Table 2.1), OCC provides increased concurrency, since it does not prevent read and write operations from executing concurrently. However, OCC require an additional *validation phase* before transaction commit, which increases overhead. Additionally, OCC can result in higher penalties when conflicting transactions are detected, leading to an higher abort rate [Agr+87; JMHH07].

MVCC mitigates some of the overhead imposed by OCC, since transactions execute in a consistent database view defined at start time. This allows read operations to execute concurrently with every other operations, without conflicts. Thus, under MVCC read-only transactions can execute to completion without aborting. However, MVCC treats write operations differently, either using an optimistic or pessimistic approach.

Under an optimistic approach, MVCC behaves similarly to OCC in respect with writes. Write operations are allowed to execute concurrently. This requires an additional *validation phase* before committing changes, for checking conflicts.

Under a pessimistic approach, MVCC reduces concurrency for write operations, when compared to OCC, since the pessimistic approach prevents other transactions from accessing written items. However, this allows MVCC to commit update transaction without requiring a validation phase.

Compared to these, pessimistic concurrency control imposes additional restrictions to concurrency, since only read operations are allowed to execute concurrently with each other, while write operations execute in exclusion. However, under a pessimistic approach, transactions that try to commit will do so successfully, since this approach prevents any conflicting transactions from executing. Thus, pessimistic concurrency control reduces the waste of computational resources. Additionally, when compared to OCC, it does not require an additional validation phase prior to commit, thus reducing overhead. Still, pessimistic concurrency control requires a deadlock prevention mechanism to guarantee progress.

To mitigate the probability of conflicts, and increase concurrency, databases offer different granularities in which to apply concurrency control. These include:

- Database level, where only non conflicting operations can execute concurrently on the entire database;
- Table level, where only non conflicting operations can execute concurrently on a same database table;
- Row level, where only non conflicting operations can execute concurrently on the same table row;

Besides these logical levels, traditional disk-backed databases also allow concurrency control mechanisms to operate at the storage granularity, by using the Storage Manager internal structures. These, include page level and record level.

The level of concurrency and the conflicts that may occur among concurrent transactions depends on the isolation level enforced. Next, we discuss database isolation levels.

2.1.4.1 Isolation Levels

A common way to increase transaction concurrency is to reduce isolation semantics. The ANSI SQL [ISDLS92] standard defines four isolation levels, which are: *Read Uncommitted*, *Read Committed*, *Repeatable Read* and *Serializable*. From these, the serializable isolation level is the highest isolation level possible, while the read uncommitted is the least intrusive, making no assumption on read data.

The defined isolation levels are the following:

- *Read Uncommitted* - under Read Uncommitted, a transaction can read any data, independently of being committed or not by any concurrent transaction.

- *Read Committed* - under Read Committed, a transaction can only read committed data, i.e., data that has been committed by some transaction. Note that repeated reads of the same item by the same transaction may result in different values, since concurrent transactions may have successfully committed changes between these reads.

- *Repeatable Read* - under Repeatable Read, a transaction can only read committed data, and repeated reads of the same item, by a transaction, always return the same (committed) value. Repeated reads of the same predicate may return additional results.

- *Serializable* - under Serializable, a transaction can only read committed data, and repeated read of the same item and the same predicate always return the same (committed) values.

From the previous description it is possible to see that relaxing a transaction's isolation level may allow otherwise conflicting operations to execute. For instance, relaxing from serializable to the read committed isolation level, allows writing an item that has been read by another concurrent transaction, thus increasing concurrency [Ber+95].

However, this relaxation leads to the occurrence of some concurrency anomalies, called phenomena. These anomalies include: Dirty Read; Non-Repeatable Read; and Phantom Read. The different isolation levels are defined based on these three concurrency anomalies, as presented in Table 2.2.

Isolation level	**Dirty Read**	**Non-Repeatable Read**	**Phantom Read**
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

Table 2.2: Isolation Levels and concurrency anomalies (i.e., phenomena)

The Dirty Read phenomena occurs when transactions are allowed to read non-committed values modified by concurrent transactions. The Non-Repeatable Read phenomena occurs when transactions are allowed to modify and commit an item concurrently read by other transactions, thus repeated reads of the same item may give a different results, where initially read values are non-repeatable. Finally, the Phantom Read phenomena is similar to the Non-Repeatable Read phenomena but applied to the set of rows returned by a given predicate, where consecutive read operations of the same predicate may return different rows.

The presented anomalies are independent of the concurrency control strategy used being optimistic or pessimistic. MVCC supports additional isolations levels, namely *Snapshot Isolation* (SI) [Ber+95]. In SI, a transaction conceptually executes in snapshot of

the database obtained when the transactions starts. At commit time, a transaction must abort if there is a write-write conflict with some other transaction that has committed after the snapshot has been taken. SI is stronger than Repeatable Read, since it may prevent the phantom read phenomena, but is weaker than the Serializable isolation level, since it sometimes allows that phenomena. Also, SI allows an additional phenomena called write skew, which is prevented under the Serializable isolation level [Ber+95].

Summing up, the Lock Manager provides isolation by using concurrency control mechanisms, guaranteeing non conflicting operations execute concurrently and preventing conflicting operations for doing so. These conflicts result in the occurrence of well defined anomalies (phenomena), and different isolation levels provide a tradeoff between performance, due to an increase in concurrency, and the occurrence of these anomalies. For example, the relaxation from Serializability to Read Committed should increase throughput of update transactions since these transactions can update values concurrently read by other transactions. However, the relaxation of isolation levels forces application developers to deal with possible concurrency anomalies.

Finally, it is important to note that, while the relaxation of isolation levels modifies the behavior of the concurrency control protocol, increasing or decreasing concurrency of database operations, it does not in any way guarantee the integrity of the data structures used by the database to store data. Since the database concurrency control may allow operations to access or modify database data concurrently, the internal data structures used for maintaining this data must be able to handle concurrent accesses. To this end, these tend to employ additional concurrency control mechanisms, for guaranteeing their integrity when accessed concurrently, as described next.

2.1.5 Preserving Consistency

Besides transaction concurrency, databases must deal with threads concurrently accessing the database's internal data structures. *Latches* are commonly used for providing mutual exclusion on shared data structures. Contrarily to locks, *latches* are similar to operating system monitors or semaphores [Hoa74].

Latches differ from locks used for concurrency control at the transactional level in a number of ways:

- Locks are kept in the lock table and located via hash functions; latches reside in memory near the resources they protect, and are accessed directly.

- Lock acquisition is driven by data access, where the order and lifespan of lock acquisitions is related to applications and the query optimizer. Latches are acquired by specialized code inside the DBMS, for preventing state corruption of shared database data structures.

- Locks acquisition may produce deadlocks, which must be detected and resolved by the Transactional Manager. Latch deadlock must be avoided, since its occurrence represents a bug in the DBMS code.
- Latches are implemented using an atomic hardware instruction or, via mutual exclusion in the OS kernel.
- Latch calls take at most a few dozen CPU cycles whereas lock requests take hundreds of CPU cycles.
- The Lock Manager tracks all the locks held by living transactions, automatically releasing locks when transactions commit or abort. Latches are not tracked and cannot be automatically released if the task fails. The internal DBMS routines that manipulate latches must track and manage them when an exception occurs.

Latches are used to provide mutual exclusion for concurrently accessed data structures. For instance, a page table, used for managing the buffer of traditional disk-backed databases, maintains a latch associated with each frame, to guarantee that only one database thread is replacing a given frame at any time. Latches are also used in other data structures like the lock table, used by Lock Manager for providing transaction concurrency control. Indices also use latches for preventing the corruption of these data structures when concurrently accessed or modified.

2.2 Classic Isolation Implementation

Classic Lock Managers enforce isolation by using pessimistic concurrency control, obtaining *locks* before accessing items. Typically, read operations obtain shared locks on the accessed items, while write operations obtain exclusive locks.

By convention, database locks are names, used within the system, that represent either physical (e.g., disk pages) or logical items (e.g., tuples, tables). Any name can have an associated lock, and the locking mechanism provides a place to register and check for names. Traditional lock manager typically associate locks with transactions, using a transaction's unique identifier, and have different "modes" [Gra+75].

For this, the Lock Manager maintains a global *lock table* that holds lock names and their respective information. This information includes the mode flag, used to indicate the lock mode, and a wait queue of lock requests consisting of pairs (transactionID, mode). The lock table typically uses a dynamic hash table, where lock names are the keys.

The Lock Manager also maintains a transaction table that associates transactions with their respective locks. Each transaction maintains a list of acquired locks as well as an associated thread state. The thread state allows the DBMS to reschedule threads when transactions have to wait before acquiring locks, while the former is used to facilitate the release of all locks associated with a particular transaction (e.g., upon transaction commit or abort).

For achieving the different isolation levels, lock based concurrency control acquires locks in different ways. Table 2.3 presents the different lock acquisition patterns, including the duration of acquired locks. Long duration means that locks are acquired before accessing an item and are preserved until the transaction commits, while short duration means locks are acquired before accessing an item and released immediately after. Also, *item* means locks are acquired on a single item, while *predicate* means lock are acquired on a set of items that respect a given predicate, or condition.

Isolation Level	**Read Lock Duration**		**Write Lock Duration**	
	Item	Predicate	Item	Predicate
Read Uncommitted	-	-	Long	Long
Read Committed	Short	Short	Long	Long
Repeatable Read	Long	Short	Long	Long
Serializable	Long	Long	Long	Long

Table 2.3: Locks and Isolation Levels (adapted from [Ber+95]).

It is important to note that the different isolation levels differ solely on the duration of acquired locks for read operations, with all write operations acquiring long duration write locks on both item and predicate.

This way, Read Uncommitted is achieved by not acquiring any locks for read operations, which eliminates any conflicts with concurrent write operations, thus allowing read operations to read any value independently of being committed or not. Read Committed acquires short duration read locks on both item and predicate, which conflicts with any concurrent write on the same items. Thus read operations are only allowed to read unlocked items, i.e., committed values. By immediately releasing read locks, it improves concurrency for write operations. Repeatable Read and Serializable locking patterns are identical for single item locking, with both acquiring long duration read locks, which guarantees only committed values are read and prevents any concurrent write operation from executing on the same items during the duration of the transaction. However, this differs for predicate locking. Repeatable Read acquires short duration read locks on predicate, which are immediately released after executing, thus not preventing rows from being inserted or deleted by concurrent transactions during the duration of a transaction. This results in the "phantom read" phenomena, where consecutive reads may return different results. On the other hand, serializable acquires long duration read locks on both item and predicate. This guarantees that write operations do not compromise the results returned by consecutive predicate reads, thus the serializable isolation level prevents the "phantom read" phenomena [Ber+95].

Conflicting concurrent lock requests may result in the occurrence of deadlocks, whenever concurrent transactions require locks held by each other. A deadlock detection algorithm needs to examine the lock table to detect waits-for cycles (a cycle occurs when

two or more executors wait for one another for needed locks). When a deadlock is detected, a transaction is chosen and aborted, thus breaking the cycle. The decision of which transaction to abort is based on heuristics [GR92; Ros+78].

As a particular case of a locking scheme, two-phase locking (2PL) divides locking into 2 separate phases: lock acquiring and lock releasing, with the restriction that no new lock is acquired after the release of a lock. It has been shown that using 2PL allows a database system to provide serializable isolation level [Ber+95; Esw+76].

Lock Managers provide two basic methods: `lock (lockname, transactionID, mode)`, for a transaction *transactionID* locking item *lockname* in mode *mode*, and `unlock (transactionID)`, for unlocking all locks held by *transactionID*. Due to different isolation levels, other than serializable (as discussed in Section 2.1.4.1), additional methods are provided: `unlock (lockname, transactionID)`, for transaction *transactionID* to unlock a given item *lockname*. There is also a `lock_upgrade (lockname, transactionID, newMode)`, for transaction *transactionID* to upgrade lock *lockname* to a higher lock mode *newMode* (e.g., from shared to exclusive), without having to release and acquire a new lock. This is important to avoid breaking 2PL semantics.

2.3 From disk to main memory

Most current general purpose IMDBs have evolved from this traditional design, replacing the disk-backed storage systems with in-memory ones [Sto+07]. In this section we discuss in detail the design differences between disk-backed and in-memory databases, and present the implementation of two general purpose IMDBs, HSQLDB [Gro12] and H2 [H212].

2.3.1 Storage Management

Contrarily to in-memory databases, disk-backed databases store data on disk. To reduce costly I/O, disk-backed databases maintain copies of most recently accessed pages in memory buffers. Thus, main memory is used as a cache mechanism for the accessed data.

Compared to disk-backed databases, in-memory databases store data directly on memory, without need for any buffers. In fact, data is maintained directly on indices, with each table having, at least, one associated index structures. Indices are, typically, tree based implementation (e.g., AVL-Tree, B+-Tree, etc.), and maintain the table rows sorted by the respective key attributes. Since no buffering is used, there is no need for maintaining a complex buffer pool management with executors accessing database data stored directly on the indices.

For SQL SELECT statements, the storage manager locates the corresponding data and returns it to the corresponding executor for returning it to the clients. Updates are typically implemented as a remove operation followed by an insert operations, to

maintain index integrity due to possible index restructuring. Insert and delete operations modify the structure of the index by indexing new tuple(s) or removing existing ones.

2.3.2 Log Management

Database durability is achieved by logging operations. The database guarantees that committed transactions are durable in the sense that the respective database modification will endure (until overwritten by any later transaction) even in the event of database faults. Hence, databases only commit transactions after logging the respective operations (including the commit operation itself) durably.

In-memory database achieve durability in a similar fashion, by recording update operations, and keeping this log in persistent storage. However, since in-memory databases keep both data and index structures in memory, logging operates differently from disk-backed databases. While disk-backed databases need to log every successfully executed update operation, since disk pages may reflect changes made by non committed operations, which need to be rolled back when recovering from faults, their in-memory siblings do not. Since data is kept only in main memory, when a system crash occurs all data is lost. Thus, there is no need to undo previously uncommitted changes since all data is lost when restarting after a crash. Therefore, in-memory databases achieve durability by only logging update operations that successfully commit. Recovery is achieved by redoing all previously committed changes. Like in traditional disk-backed databases, checkpoints may be used to speed-up the recovery process.

For guaranteeing atomicity, both studied databases maintain per session undo logs, for undoing modifications when rolling back operations.

2.3.2.1 Trading durability for performance

Maintaining a REDO log on persistent storage (e.g., disk) can compromise IMDBs performance, since, before committing every update transaction, the database must flush the log to persistent storage to ensure durability.

To deal with this possible performance bottleneck, most in-memory databases trade durability for performance. Instead of flushing the REDO log to disk in every commit operation, this process is done periodically and asynchronously, by batching a series of records before flushing them to disk. This means that, in the event of a failure, some transactions may be lost, since their corresponding log entries have yet to be flushed. This is a tradeoff that allows in-memory databases to offer increases performance, by not incurring in disk I/O operations, while still offering some level of durability. Similar approaches are used in disk-based databases.

An alternative approach to provide durability is to rely on replication techniques, where it is assumed that data replicated in $f + 1$ nodes is durable [Cam+07].

2.3.3 Lock Management

In-memory database offer identical concurrency control mechanisms and isolation levels when compared with disk-backed databases. However disk-backed databases can use locking granularities not available to in-memory databases.

For instance, disk-backed databases can offer *page level locking*, besides table level or row level locking, due to the buffering of database pages from disk. In-memory databases commonly offer table or row level locking, using either pessimistic, optimistic or multi-version concurrency control.

Compared to table level locking, both page level locking and row level locking offer increased concurrency, since conflicts are prevented at a finner grain. However, guaranteeing serializable semantics under these locking granularities requires a complex mix of row level locking and predicate locking, for addressing SQL statements accessing data based on conditions [RS77], and for avoiding phenomena such as Phantom Read. Additionally, the problem of testing predicate locks has been shown to be NP-complete [HR79] (and complex to implement). Table level locking prevents the occurrence of these phenomena, since all modifications are executed in exclusion.

2.3.4 H2 and HSQLDB behavior

We now present the implementation details the two general purpose IMDBs, HSQLDB [Gro12] and H2 [H212], used in our work. Both engines interact with client applications through a JDBC interface. Whenever a client establishes a new connection to the database a new *Session* is created. Sessions are used by the database for guaranteeing atomicity and isolation to the statements performed by different clients, in the context of different transactions. A simplified algorithm of statement execution is presented in Listing 2.1. We omit error and conflict verification for simplicity of presentation.

Listing 2.1: Lock Based Statement Execution.

```
1  var global:
2  Transaction_Manager tx_mngr
3  Storage_Manager storage_mngr
4  Log_Manager log_mngr
5
6  var per client:
7  Session session
8  Result result
9  Command command
10
11 function executeCommon ( statement )
12   if( NOT valid_connection ( session ) )
13     throw DBError
14   if( NOT validate_syntax ( statement ) )
15     throw SyntaxError
16   valid_statement = parse_and_compile ( statement )
17   result = create_result_set ( valid_statement )
18   command = optimize ( valid_statement )
19   tx_manager.aquire_table_locks ( session, command )
```

```
20
21 function executeQuery ( statement )
22   executeCommon ( statement )
23   storage_mngr.read_data ( command, result )
24   return result
25
26 function executeUpdate ( statement )
27   executeCommon ( statement )
28   log_previous_data ( session, command )
29   storage_mngr.delete_row ( session, command )
30   storage_mngr.insert_row ( session, command )
31   storage_mngr.verify_integrity ( session )
32   fire_table_triggers ( )
33   return result;
34
35 function commitCommon ( )
36   tx_manager.unlock_tables ( session )
37   tx_manager.awake_awaiting_txs (  )
38
39 function commitQuery ( )
40   commitCommon ( )
41
42 function commitUpdate ( )
43   log_mngr.log_actions ( session )
44   log_mngr.log_commit ( session )
45   commitCommon ( )
```

Whenever a statement is executed by a client, both databases start by validating the connection state, and checking that the statement is syntactically correct. If no error occurs, then a new result object for that statement is created. This object is used to maintain the statement's result set and its respective metadata (e.g. the information on the tables and columns being read, the columns data types and the number of lines of the result set). After this, an optimization stage selects an execution plan suitable for the statement's execution, as presented in lines 11-18.

Sessions proceed by interacting with the *transaction manager* for executing the statement in isolation (line 19). Both databases support different isolation levels, based on two kinds of concurrency control mechanisms: Multi-version concurrency control and lock based concurrency control (pessimistic) [Ber+87; KR79]. In this discussion we focus on lock based concurrency control.

Both databases implement lock-based concurrency control based on 2PL at the table level, using shared and exclusive *table level locks*, for read and write operations respectively. Sessions are only allowed to execute each statement after acquiring the necessary table locks. If a session fails to acquire a table lock, due to a conflicting concurrent session, then it will wait until the necessary locks are released.

While semantically identical, the locking implementation differs on both engines. In H2, each session maintains a set of table reference for which it has acquired locks (shared or exclusive) during its execution. Also, each table object maintains the set of sessions that have acquired shared locks on it, and a single reference to the session holding it exclusively. On the other hand, HSQLDB follows a more traditional design, where a

single entity, the lock manager, maintains a multi-value map for keeping shared table locks and their associated sessions, and a single-value map for exclusive table locks, and their associated session.

After acquiring table locks, sessions proceed by interacting with the *storage manager*. For query statements, the corresponding data is copied from the database to the session's result set and it is returned to the client (lines 23 and 24). For update statements, the corresponding data items are first read and logged, for recovery purposes (line 28). After this, the *storage manager* updates the necessary table indices (lines 29-32) with the modified values. A similar situation occurs for insert and delete statements. For insert statements, a compensatory action (i.e., a delete of the inserted rows) is defined for undoing the insert, while for delete statements, the deleted items are logged for recovery purposes, in case the transaction aborts or the client issues a rollback. In both databases, sessions maintain *logged* data until a commit or rollback operation ends. Also, both engines implement indices using AVL-trees, and updates are executed as an index delete followed by an insert operation.

All commit operations release acquired locks and notify existing waiting concurrent sessions. For update transactions, sessions first interact with the *log manager*, logging all performed actions (lines 43 and 44). It logs data updated during the transaction execution, old and new values, followed by the commit operation itself. Rollback operations undo the necessary changes before releasing locks.

2.4 Summing up

General purpose in-memory databases have evolved from their traditional disk-backed siblings by trading disk for main memory storage. This allows IMDBs to abandon cache mechanisms, used for reducing latency when accessing data and increase performance, as well as complex buffer management algorithms. Additionally, IMDBs trade durability for performance by eliminating disk I/O during transaction commit phase. Instead these batch several update records before asynchronously writing to disk, for minimizing I/O and efficiently utilizing bandwidth. All these design diferences have the same purpose, to reduce or even eliminate I/O overhead for increasing performance. Thus, one expects these systems to scale on current multicore systems, since available hardware contexts can access data without having to wait for expensive I/O operations. Next we present a scalability study of the studied engines.

CHAPTER 3

RESEARCH PROBLEM

In this chapter we present a study on the scalability of two general purpose in-memory databases, HSQLDB [Gro12] and H2[H212]. In this study, we analyze how different aspects influence the scalability of the databases. This study both demonstrates the research problem being addressed in this dissertation and serves as guidance to the directions explored, which are detailed in Section 3.3. This chapter ends with a brief overview of database research addressing the use of multicore machines, which is complemented in the following sections with related work specific to the techniques being explored.

3.1 H2 & HSQLDB scalability study

To verify the scalability of IMDBs, we studied how two general purpose IMDBs, HSQLDB [Gro12] and H2 [H212], perform on a multicore system. To this end, we measured the throughput of successfully committed transactions, when running a well established benchmark, TPC-C [Cou12]. For these experiments we analyzed the impact of different parameters in the performance of these systems.

First, we have varied the mix of read-only and update transactions in the workload. The goal is to study how the database behaves under different workloads. Second, we varied the isolation level under which transactions execute. Isolation levels have a direct impact on the contention and interference among transactions. Thus, weaker isolation levels are expected to offer increased concurrency, compared to higher isolation levels, by reducing interference between concurrent transactions. For example, the relaxation from serializability to snapshot isolation should increase throughput of read-only transactions since these can execute on a different database snapshot from update transactions, thus are not aborted or delayed by concurrent updates.

This experiment ran on a 16 core Sun Fire x4600, with 32 GBytes of RAM. Workloads

varied from update intensive ones: *8-92* and *50-50* with 92% and 50% update transaction, respectively; to read intensive ones: *80-20* and *100-0* with 80% read transactions and 100% read transactions, respectively. The TPC-C specification defines five different transactions: new order; payment; stock level; order status, and delivery. From these, two are read only: stock level and order status. Also, the 8-92 workload is composed of 45% new order transactions, 43% payment transactions, 4% delivery, and 4% for both stock level and order status. The remaining workloads used for our experiments maintained a similar ratio between the different transaction. For instance, in the 50-50 workload both stock level and order status account each for 25% of all transactions, while new order and payment account for 25% and 23% respectively, with the remaining 2% for delivery.

The studied isolation levels were: *i*) *serializable*, relying on two-phase locking; *ii*) *read-committed*, relying on two-phase locking with early release of read locks; and *iii*) *snapshot isolation (SI)*, relying on a multi-version concurrency control algorithm. To test the scalability of the systems, we have increased the number of clients from 1 to 18. The benchmark ran for 2 minutes, using an approximately 2 gigabyte database (4 warehouses). The presented results are the average of 5 runs, performed on a fresh database, disregarding the best and the worst results.

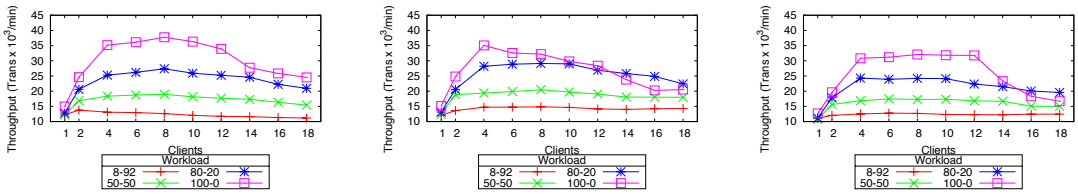
During the remainder of this document, several other experiments will be discussed, including the evaluation of our proposed solutions. All have been performed on this same configuration, unless stated otherwise.

3.1.1 HSQLDB scalability results

We start by presenting the results for HSQLDB. HSQLDB implements *serializable* and *read committed* isolation levels using table level 2PL, acquiring shared table locks for read operations and exclusive table locks for update operations (including SQL UPDATE, INSERT and DELETE statements). Under the serializable isolation level, shared lock are kept until the end of the transaction, i.e., until the transaction commits or rollbacks, while for the read committed isolation level, shared locks are released immediately after the corresponding read operation.

For *snapshot isolation*, HSQLDB allows read operations to execute without locking, executing on a database snapshot defined at the start of each transaction. To this end, table rows have an associated *timestamp*, that defines when these were last modified. Timestamps are assigned during the commit phase of the corresponding update transactions, by incrementing and reading the value of a monotonically increasing counter. Also, before their first operation, transactions read the current value of this counter, thus defining which items are allowed to read. Additionally, HSQLDB uses a pessimistic approach where update operations acquire exclusive table locks before executing, writing tentative values directly in place. These locks are kept until the end of the corresponding transaction, thus preventing other write operations from executing on the same table. This allows transactions to commit without an additional validation phase.

Figure 3.1 shows the TPC-C results for the different isolation levels, serializable (Figure 3.1(a)), read committed (Figure 3.1(b)) and snapshot isolation (Figure 3.1(c)). Each graph shows the measured throughput for the different workloads, varying the number of clients.



(a) Serializable (b) Read Committed (c) SI

Figure 3.1: HSQLDB performance under different TPC-C workloads and isolation levels.

From these results, it is possible to observe that database performance varies with the workload nature, achieving higher performance as the ratio of read-only transactions increases in the workload. This behavior is observable for all isolation levels. This increase in throughput with the ratio of read-only transactions in the workloads is expected. It results from lock based isolation levels acquiring shared locks before executing read operations, which allows them to run concurrently with other read-only transactions on the same tables. Thus, increasing the read-only transaction ratio in the workload reduces the probability of interference between concurrent transactions and increases throughput.

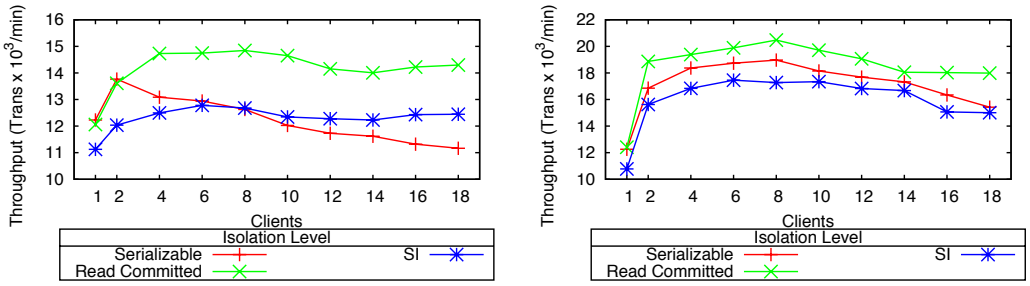
SI achieved a similar throughput to the other isolation levels. We were expecting this isolation level to achieve higher performance than the others, since under SI read operations do not acquire locks, which allows them to execute concurrently with updates. However this does not happen, with the performance of SI being similar to the other isolation levels. For update intensive workloads, the pessimistic approach used by HSQDB, seems to explain why these workloads do not scale. However, the same reason is not valid when the increasing the ratio of read-only transactions in the workload. A possible explanation for this can be that, while read operations do not acquire locks before executing, their results need to be validated before being retuned to the applications. Since this is done in mutual exclusion, this validation increases contention, thus compromising performance.

Next we analyze in greater depth the performance differences achieved by the different isolation levels, for each workload.

3.1.1.1 8-92 and 50-50 workloads

Figures 3.2(a) and 3.2(b) present the TPC-C results for the 8-92 and 50-50 workloads respectively.

When analyzing the results, it is possible to observe that, although the different isolation levels offer different performances, these show an identical behavior. Throughput



(a) 8-92 workload

(b) 50-50 workload

Figure 3.2: HSQLDB TPC-C performance for different isolation levels (8-92 and 50-50 workloads).

tends to increase slightly up to 4 clients, maintains its value as the number of clients increases up to approximately 10, and then decreases as the number of clients approaches the number of available hardware threads.

This is especially true for the 50-50 workload (Figure 3.2(b)), where the differences between the best performing isolation level (read committed) and the worst performing isolation level (SI) does not exceed 15%. While, for the 8-92 workload (Figure 3.2(a)), the difference between these read committed and SI is identical, serializable isolation level decreases performance more quickly as the level of concurrency increases. This is expected, since, under this workload, the conflicting nature of TPC-C has a considerable impact on concurrency. This leads transactions to block or even abort due to conflicting updates, which impact the performance. As expected, increasing the ratio of read-only transaction balances this due to the reduction of conflicting transactions.

As discussed before, an interesting and unexpected result comes from the throughput values obtained for the SI level. In both workloads, SI performed worst than the read committed isolation level, while in the 50-50 workload it was the worst performing.

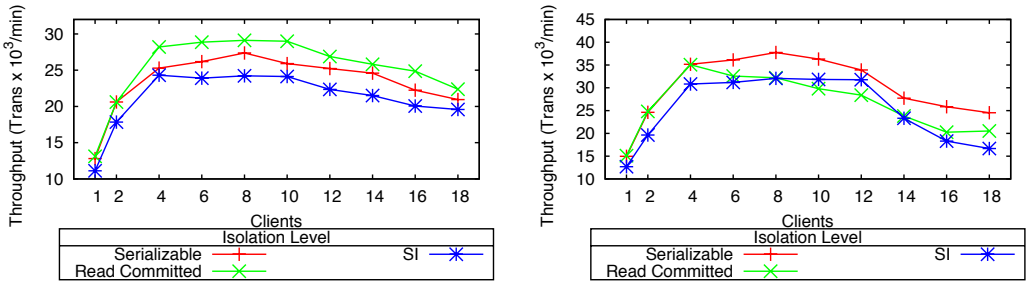
We believe this decrease in performance results from the pessimistic approach used by HSQLDB, and from the increase in overhead due to result validation before returning them to the application. This is not necessary for the other isolation levels since these acquire shared locks before reading an item. This gives them the guarantee that the read values remain the same during the execution of the transaction, since any concurrent updates will block trying to acquire an exclusive lock on the same item.

For these workloads, the read committed isolation level offers the best performance of the three isolation levels. This is a direct result of the immediate release of shared locks after read operations execute, which reduces the waiting time for concurrent updates to acquire exclusive locks on the same tables.

3.1.1.2 80-20 and 100-0 workloads

The results obtained when increasing the ratio of read-only transactions show a similar behavior. These results are presented in Figures 3.3(a) and 3.3(b), for the 80-20 and 100-0

workloads respectively.



(a) 80-20 workload

(b) 100-0 workload

Figure 3.3: HSQLDB TPC-C performance for different isolation levels (80-20 and 100-0 workloads).

When analyzing these results, we can see that although the different isolation levels offer different performances, their performance follows an identical behavior. Throughput increases up to 4 or 6 clients, and then gradually decreases as the number of clients reaches the maximum number of hardware threads. Again, the differences between the best and worst performing do not exceed 18% for both workloads.

Again, like in the previous results, SI is unable to offer the best performance. In fact it offers the worst performance of the three isolation levels in the 80-20 workloads, with the read committed out performing the others or the same workload. For the read-only workload, the serializable isolation level offers the best performance of the three.

Although, the validation overhead of SI can explain its performance penalty for the 80-20 workload, for the read-only workload one would expect that the absence of locking would make up for this overhead. Yet, it performs worst that the lock based serializable isolation level.

An interesting result comes from the fact that the serializable isolation level offers the best performance for the read-only workload. While, for all other workloads, read-committed isolation level provided higher concurrency, this does not occur for this workload. This results from the fact that releasing shared locks after read operations execute does not increase concurrency for read-only transactions. In fact, this contributes to an increase in overhead, since every operation forces lock management algorithms to execute in two different moments, when operations begin (for acquiring locks) and when operations end (for releasing locks). Since lock management algorithms execute in mutual exclusion, it prevents concurrent transactions from acquiring or releasing any lock, increasing contention and reducing concurrency. By releasing all acquired locks during its commit or rollback phase, the serializable isolation level reduces contention due to lock management, which results in better performance.

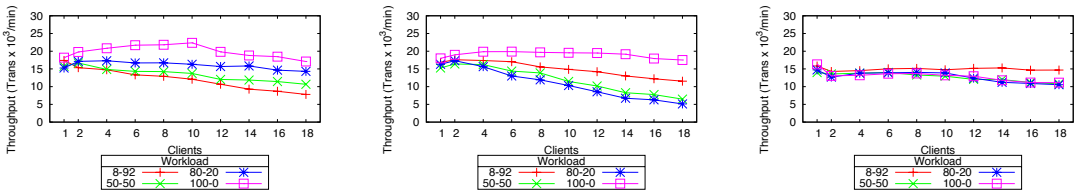
Above all, these graphs show that, although different workloads offer different performances, HSQL does not scale with the number of clients, independently of isolation level and workload.

3.1.2 H2 scalability results

We now present the results for H2. Like HSQLDB, H2 also implements *serializable* and *read committed* isolation levels using table level 2PL, acquiring shared table lock for read operations and exclusive table locks for update operations (including SQL UPDATE, INSERT and DELETE statements). Under the serializable isolation level, shared lock are kept until the end of the transaction, i.e., until the transaction commits or rollback, while for the read committed isolation level, shared locks are released immediately after the corresponding read operation.

For *snapshot isolation*, H2 uses MVCC allowing concurrent read and write operations to execute on the same table. Under SI, each table index is composed by two distinct indices, a *base index* and a *delta index*. The base index only contains committed modifications, while uncommitted modifications are reflected in the delta index. Update operations only modify the delta index, while read operations use both indices before returning values for reading values, since queries may return previously modified values made by the same transaction. H2 detects conflicts during transaction execution at the row level. Transactions abort whenever update or a read operations encounter uncommitted changes made by concurrent transactions. Otherwise, operations execute successfully. Although this is a pessimistic approach, it allows H2 to commit transaction without further validation overhead. Thus, transactions that execute without conflicts commit successfully.

Figure 3.4 shows the TPC-C results for the different isolation levels, serializable (Figure 3.4(a)), read committed (Figure 3.4(b)) and snapshot isolation (Figure 3.1(c)).



(a) Serializable (b) Read Committed (c) SI

Figure 3.4: H2 performance under different TPC-C workloads and isolation levels.

Like in the previous results, H2 performance varies with the workload nature, achieving higher performance as the read ratio increases in the workload. This is specially true for the serializable and read committed isolation levels (Figures 3.4(a) and 3.4(b)), where the 100-0 workload achieves the best performance. Again, this behavior is expected, since, for lock based isolation levels, read operations are allowed to execute concurrently with each other by acquiring shared locks on the accessed tables.

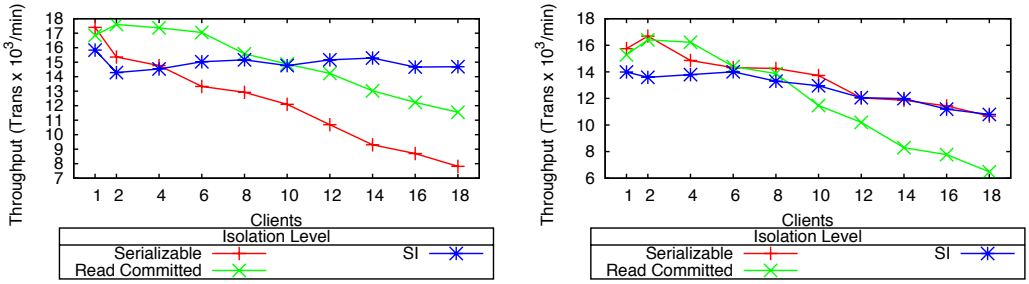
Again, the throughput achieved by SI in H2 does not correspond to what was expected. The throughput is fairly constant with the increasing number of clients, independently of the workloads. This is unexpected since read and write operations are allowed to

execute concurrently. However, we believe that the pessimistic approach used by H2 is responsible for this behavior.

Next we analyze in greater depth the performance differences offered by the different isolation levels, for each workload.

3.1.2.1 8-92 and 50-50 workloads

Figures 3.5(a) and 3.5(b) present the TPC-C throughput under the 8-92 and 50-50 workloads.



(a) H2 Scalability 8-92 Workload (b) H2 Scalability 50-50 Workload

Figure 3.5: H2 TPC-C performance under different Isolation levels.

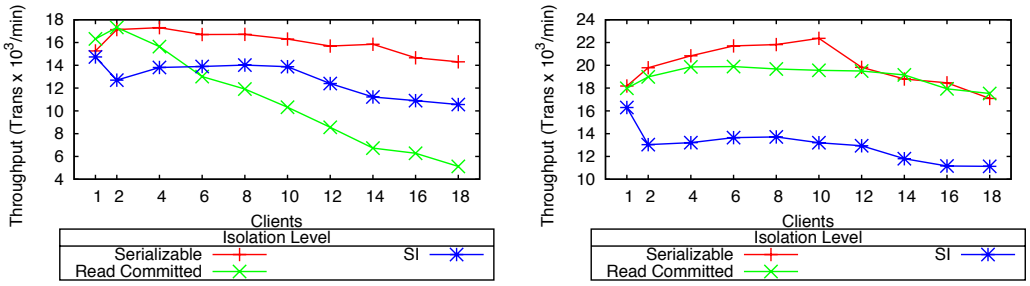
When analyzing the results for update intensive workloads, it is possible to observe that the serializable isolation levels offers the worst performance of the three for the 8-92 workload (Figure 3.5(a)). While the read committed offers the best performance up to 8 clients, SI outperforms both when increasing load beyond 10 clients. A similar behavior is observed for the 50-50 workload (Figure 3.5(b)), although, for this workloads, the read committed isolation level is the worst performing of the three.

These results are somewhat expected for the lock based isolation level, since the conflicting nature of TPC-C greatly compromises concurrency. However, we were expecting a better performance from SI. Although SI is able to offer better performance than the other isolation levels, the throughput achieved by SI remains constant with the increase in the number of clients, independently of the workloads. This is unexpected since read and write operations are allowed to execute concurrently. However, we believe the pessimistic approach used by H2, combined with the conflicting nature of TPC-C, prevents SI to achieve better performance under these workloads. Also, since result validation requires coordination between transactions it increases contentions, which reduces concurrency and performance.

Another interesting result comes from the fact that serializable isolation level is able to outperform the read committed when increasing the ratio of read only transactions in the workload. Although this behavior is different from the one observed for HSQLDB, it can be explained by the different implementations these databases use for managing transactions and locks.

3.1.2.2 80-20 and 100-0 workloads

Figures 3.6(a) and 3.6(b) present the TPC-C throughput for the different isolation levels, under the 80-20 and 100-0 workloads.



(a) H2 Scalability 80-20 Workload

(b) HSQLDB Scalability 100-0 Workload

Figure 3.6: H2 TPC-C performance under different Isolation levels.

When analyzing the results for the 80-20 and 100-0 workloads, we can see that, as before, the serializable isolation level achieves better performance as the ratio of read-only transactions increases. In fact, for these workloads, serializable achieves the best performance of the three, with read committed achieving the worst for the 80-20 workload (Figure 3.6(a)) while SI is the worst performing for the read-only workloads (Figure 3.6(b)).

Again, this is somewhat expected since increasing the ratio of read-only transactions reduces the probability of conflicts, thus allowing higher levels of concurrency. However, the behavior of read committed isolation level is difficult to explain, since it evolves from the worst performing of the three, for the 80-20 workload, to achieving a performance similar to the serializable isolation level, for the 100-0 workload. This is unexpected since the immediate release of shared locks should reduce the wait time for update transactions to acquire exclusive locks on the same tables, which should increase throughput. Also, the contention imposed by the immediate release of shared locks, compared to serializable, should compromise its performance for read-only workloads (a behavior presented by HSQLDB). Again, this could be explained by the difference in the implementations of transaction and lock management between the two databases.

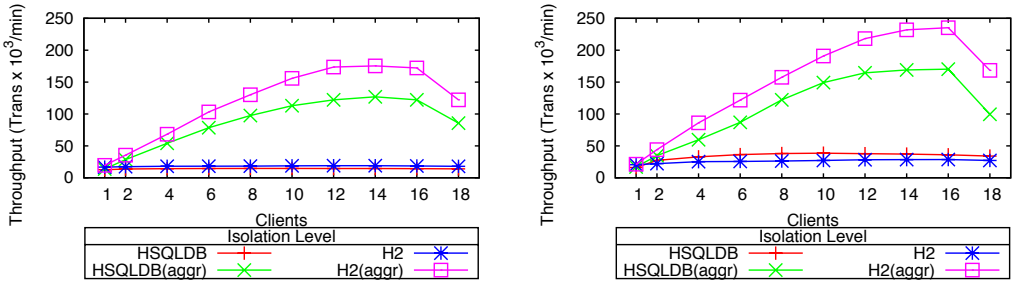
Again, like in the previous results, SI offers a fairly constant throughput as the level of concurrency increases. This seems to confirm that the increased overhead for validation purposes considerably compromises performance. This compromises any performance improvement that SI should offer, since read operations are allowed to execute without using locks.

Also, like HSQLDB, although H2 offers performance differences between the different isolation levels, no single isolation level is able to offer scalable performance for any workload.

3.1.3 Understanding Scalability Results

To better understand if the lack of scalability of the two databases are due to the lack of computational resources, we conducted an experiment that ran an increasing number of pairs client/database concurrently on the same machine, i.e., one database per client. We used two different workloads for this experiment, 8-92 and 100-0, and the database used the serializable isolation level (although not relevant in this experiment).

Figures 3.7(a) and 3.7(b), show the aggregate throughput for each of these experiments, represented as *HSQLDB (aggr)* and *H2 (aggr)*, for the 8-92 and 100-0 workloads respectively. These results show that the aggregate throughput increases with the number of clients, under both workloads.



(a) Scalability under 8-92 workloads

(b) Scalability under read-only workloads

Figure 3.7: H2 Isolation level impact on TPC-C performance.

In fact, the aggregate throughput achieves near linear scalability up to 12 clients for both engines and both workloads. Above 12 clients the increase in performance in not as accentuated, due to operating system overhead for managing the concurrent execution of all processes. Above 16 client there is a considerable drop in performance. This results from lack of computational resources, namely main memory, since the aggregate space needed for 18 databases exceeded the total amount of main memory of the machine, and hardware threads, since the number of concurrent processes exceeds the number of available hardware threads requiring expensive (cache wise) context switching between them, something that is disguised when running fewer processes.

Above all, these results put into evidence that the general lack of scalability for general purpose IMDBs is not due to lack of system resources, but it is a direct result of their design.

Additionally, this experiment allows us to define the *highest expected achievable throughput* for update intensive and read-only workloads, thus establishing the upper bound for the achievable throughput for these databases. Although for read-only transactions it is realistic to expect a database to offer a throughput identical to the one obtained here, this is less realistic for update transaction due to interference. Nonetheless, if the conflict rate of updates is low enough we believe this lower bound can be representative of the highest possible achievable throughput for update intensive workloads.

3.2 Identifying performance bottlenecks

The previous study exposed the general lack of scalability of both database engines. In this section, we will try to identify possible bottlenecks responsible for this lack of scalability.

3.2.1 Transaction Management

We begin by discussing the possible implications transaction management may have in the performance of the databases. As discussed in Section 2.1.4, databases employ concurrency control mechanisms to preserve ACID properties, namely isolation.

Unlike traditional disk-backed databases, both databases implement transaction concurrency control using table-level locking. This is a pragmatic approach that avoids the complexity and overhead of semantic locks. While locking has an impact on database performance, specially for update workloads, such as 8-92, 50-50 and 80-20, locking is not the solely responsible for the lack of scalability, specially for read-only workloads.

3.2.1.1 Update workloads

Focusing on update workloads (8-92, 50-50 and 80-20) and the serializable isolation level, the previous results show that increasing the ratio of read-only transactions influences performance positively, especially for HSQLDB (Figures 3.1(a) and 3.4(a)). This puts into evidence that table level locking has a considerable impact on database performance, since decreasing the ratio of update transactions in the workload increases concurrency. This results from the fact that all update operations have to acquire an exclusive lock before executing, which prevents every other operation from concurrently accessing the same tables.

However, if table level locking were the solely responsible for the lack of scalability, SI should scale much better, especially for moderate update transaction (50-50 and 80-20). In both engines, SI allows read operations to execute without acquiring any locks. This allows read-only transaction to execute concurrently with every other transaction. Thus, under SI, one would expects database throughput to increase with the ratio of read-only transactions in the workload, since the total number of read-only transactions increases. However, this does not occur, with the performance of SI being the worst of the three isolation levels for most of the workloads (except to the 8-92 workload).

Although one may argue that, under SI, both engines still use table level locks for update operations, this does not influence read operations, since these execute without locking. Thus, increasing the ratio of read-only transactions should compensate, by far, the possible conflicts of update transactions. This should be especially true for the 80-20 workload since the total amount of read-only transactions is more than double the total amount of update transactions.

Yet, this is not the case, as confirmed by the previous results. In fact, independently of the workload, SI tends to offer worst performance than the serializable isolation level,

with the only exception being the 8-92 workload in both databases. Although one may argue that conflicting nature of TPC-C may increase the abort rate for transactions executed under SI, this does not apply to read-only transactions since these do not need validation before committing. Even so, if this was true read only workloads would scale much better under this isolation level yet they do not, as discussed next.

3.2.1.2 Read-only workloads

When focusing on read-only workloads, one expects these workloads to scale with the number of clients. This is based on the fact that read-only transactions do not interfere with each other, thus can successfully execute concurrently. However, read-only workloads does not scale in either database, independently of the isolation level. This is put into evidence by the results presented in Figures 3.3(b) and 3.6(b).

It is true that, for lock based isolation levels, read-only transactions still need to acquire shared locks before executing. Also, while shared locks do not prevent concurrent read-transactions from executing, their acquisition, i.e., transaction management, increases overhead that could be responsible for restricting performance. However, if transaction management where the solely responsible for this behavior, performance for read-only workloads would have to scale much better under SI, since, under SI, read-only transaction are allowed to execute without acquiring any locks. However, like in the remaining workloads, SI provides worst performance than lock based serializable isolation level, for read-only workloads.

SI further puts into evidence that the additional factors, other than transaction management, are responsible for the lack of scalable performance of both databases. This is supported by the fact that SI performance does not scale in either database, for read-only workloads. While, under update workloads, one may argue that a significative number of transactions abort due to conflicts detected during their validation phase, under read-only workloads no such conflicts exist. Thus, if transactions are allowed to execute without acquiring locks, and no conflicting transactions exist in the workloads, then there is no reason why read-only workloads should not scale. This put into evidence that additional factors, other than transaction management, restraint the performance of both databases.

Summing up, while we do believe that table level locking has an impact in database performance, we do believe that the lack of scalability of these databases is not solely due to lock based concurrency control. Additionally, while we do believe that transaction management, be it lock based or not, has a negative impact in the database performance, we also believe additional factors are responsible for this as well.

This impact can be observed by the decrease in performance when increasing the level of concurrency, specially for H2.

3.2.2 Logging

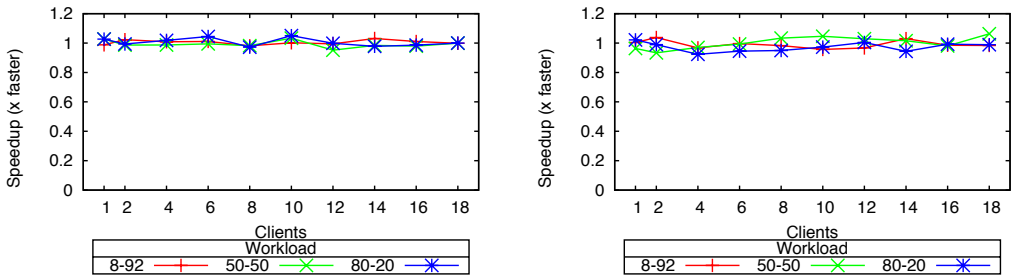
Now we discuss the possible implications logging may have in the performance of the databases. As discussed in Section 2.1.3, databases rely on logging for providing durability. Update transactions are added to the log, and the log is written to disk before transactions commit and return to clients. This allows databases to recover from possible failures, by replaying missed logged operations or rolling back incomplete transactions Thus, logging allows databases to guarantee that transactions, when successfully committed, are durable, while also guaranteeing database consistency.

Both studied databases provide durability by writing their log to disk. However, for minimizing overhead, and contrarily to traditional disk-backed databases, this process is done asynchronously and periodically, instead of being performed in every commit operation. While this means that, is case of failure, some transactions may be loss during the process, it also minimizes overhead for successfully committing transactions. As discussed in Section 2.3.1, this is the tradeoff IMDBs make between durability and performance.

However, logging does not seem to be the main performance bottleneck for these databases. If this was the case, read-only workloads would scale much better since these transactions do not log any information during execution. As presented in the previous results, read-only workloads also do not scale.

Nonetheless, we studied the impact logging has on the performance by configuring both databases to bypass logging and compared the TPC-C results withand without logging. For this experiment, read-only workloads were not used since these do not modify the database, thus do not interact with the durability log.

Figures 3.8 and 3.9 present the performance differences resulting from disabling the durability log for serializable and snapshot isolation levels respectively. Each Figure shows the speedup for the different workloads, for both HSQL (Figures 3.8(a) and 3.9(a)) and H2 (Figures 3.8(b) and 3.9(b)).

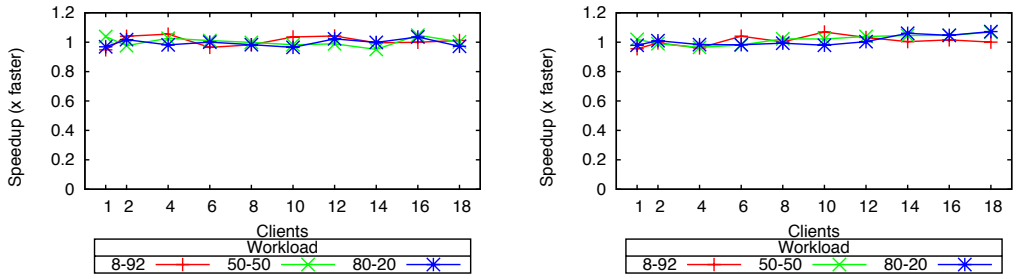


(a) HSQLDB speedup

(b) H2 speedup

Figure 3.8: Durability log overhead under serializable isolation level.

When analyzing the results, we can see that the performance differences between



(a) HSQLDB speedup

(b) H2 speedup

Figure 3.9: Durability log overhead under snapshot isolation level.

both configurations are negligible, with no single configuration offering consistently better performance than the other. This puts into evidence that logging has no significative performance impact on the performance of both databases, independently of isolation level and workload. Like discussed before, this is expected since durability of both databases is guaranteed by asynchronously and periodically writing the log to disk, instead of doing so at every commit operation.

3.2.3 Storage subsystem

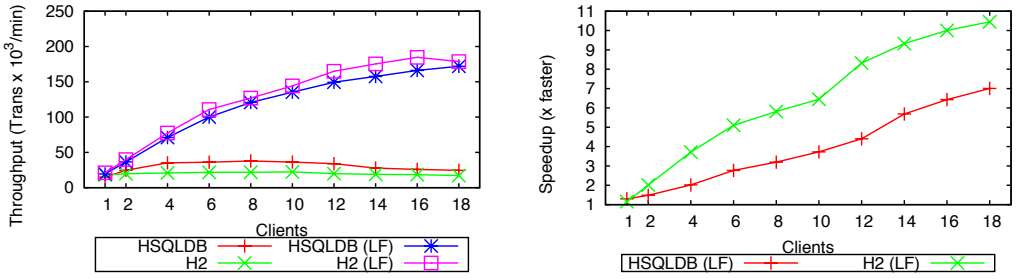
From the results presented so far, we can see that the relaxation of isolation levels does not help scalability. In fact, and contrarily to what one may expect, in both engines snapshot isolation level offers worst performance than serializable isolation level. Additionally, the performance for read-only workloads does not scale independently of the isolation level. This is true even for SI, under which read operations execute without acquiring any locks. In fact, the obtained results show that the performance for read-only workloads exhibits a behavior identical to all other isolation levels, with a decreasing throughput as the concurrency degree increases (Figure 3.1(a) and 3.4(a)).

Since, for read-only workloads, we can rule out the transactional management and the durability logging from being responsible for this lack of performance, we investigated if this lack of performance was a direct result of the underlying storage component. As discussed in Section 2.1.5, databases rely on latching for guaranteeing the underlying data structures, used to manage and store data, remain consistent when accessed concurrently. The increase in concurrency attained from relaxing isolation levels increases contention on the storage subsystem, due to the use of latching mechanisms, thus producing no benefit on performance.

To test this theory we modified both databases by removing all latches used in the storage sub-component, i.e., we removed all concurrency control mechanisms (e.g., locks) from the index data structures, and ran the TPC-C benchmark under a read-only workload. For this experiment, both databases used the serializable isolation level.

Figure 3.10(a) compares the TPC-C results from the original HSQLDB and H2 engines

with the ones obtained by the latch free modified engines, presented as *HSQLDB (LF)* and *H2 (LF)*, respectively. These results show the significative performance restraint imposed by latches used in the storage sub-component of these systems. Their removal allowed an almost 11 and 7 fold performance increase for the modified H2 and HSQLDB compared to the unmodified engines, as presented in Figure 3.10(b).



(a) Throughput (b) Speedup

Figure 3.10: TPC-C read-only workload on the modified engines.

Moreover, both modified engines scale up almost to the number of cores in the systems. Thus, one can conclude that the scalability of DBMS is greatly restricted by the concurrency control mechanisms of the storage subsystem.

3.3 Research Questions

As presented in this chapter, general purpose IMDBs are unable to offer scalable performance on multicore machines. Our study has shown that this lack of scalability is independent of the type of the workloads and also isolation level. Contrarily to what one could expect, relaxing the isolation level, although theoretically reduces contention among transactions, seems to be inefficient in improving scalability (and in some cases even performance). The presented results put into evidence that this lack of scalability is true even on a system with a reduced number of processor cores (16 cores in total).

Although this is an active research topic among the database research community, many works improve database scalability on multicores by proposing a complete redesign of the database engine. Thus, general purpose databases have been shy on adapting them since these require considerable architectural modifications.

In our work we explored the approach of scaling IMDBs on multicore without a complete redesign of the database engine. Our study suggests that not all components have the same influence in the lack of scalability of databases. For instance, our results show that the traditional implementation of underlying data structures, used by the storage subcomponent, imposes a considerable bottleneck in the scalability of these systems. Our work focused on designing techniques for improving the scalability of IMDBs while enforcing strong isolation semantics (i.e., Serializability).

To this end, we followed different paths for addressing the scalability problem on general purpose IMDBs. We started by trying to answer the question of whether it is possible to improve scalability without modifying the database and the applications (Chapter 4). To address this question, we designed and implemented a system, MacroDB, that explores database replication in a single multicore machine. This work treats multicore machines as extremely low latency clusters extended with some shared memory, and builds on the knowledge of distributed and replicated databases, addressing the scalability problem by distributing and balancing the load among different replicas.

Next, we tried to answer the question of how to scale IMDBs by modifying the database engine, without fully rethinking its architecture. In this approach, we delved into the database engine and tried to fix the major performance bottlenecks (Chapter 5). We propose a series of modifications to the database for addressing these bottlenecks. We start by addressing the scalability problem under read-intensive workloads, and then proceed by addressing the problem for update-intensive workloads.

Finally, we tried to address the question of what is the impact of the application code in the performance of the database, and whether it is possible to change the application code to improve scalability while retaining the same semantics (Chapter 6). We start by analyzing how operation order inside transactions influences the database performance. We then study how transactions can be modified in a database friendly way, i.e., for reducing interference, and improving database performance. Finally, we study how the database can take advantage of these modifications to further improve database performance, which again required modifying the database engine.

3.4 Related Work

Research in database system is an active research topic, with researchers focusing on all aspects related to the design, deployment, and use of these system [SZ97; Sil+91]. With the introduction of the multicore architecture, databases have been presented with the challenge of efficiently utilizing the increased concurrency offered by these systems.

In this section, we discuss previous studies on the influence of multicore processors on database management systems, namely on scalability, and compare them with our own. Additionally, we overview some of the proposals to address these problems, namely those that focus on the contention imposed by multicore processors. We present the state of the art for in-memory databases on multicore systems. However, we defer comparing these works with our proposed solutions to the corresponding chapters. Finally, we present some additional works that have influenced our research.

3.4.1 Database performance and scalability studies

Previous works have study the implications of multicore processors on databases performance, by focusing on disk-based database systems. Hardavellas et al. [Har+07],

presented a study on the impact processor cache stalls (due to cache misses) have on database performance. The study shows that stalls, due to cache misses, have a considerable impact on database performance, with databases spending up to 60% of computational time waiting on these stalls.

In a complementary study, Harizopoulos et al. [Har+08], dissected the kernel of Shore [Car+94] database management system, and identify performance bottlenecks. The study shows that traditional database engines can spend more than 30% of the time in synchronization related operations (locking and latching) during transaction execution. Additionally, the authors study the impact of modifying the database engine by consecutively removing different features of the database. These features include: buffer management, lock management and latching and logging. The end result was a single-threaded, lock-free, in-memory database kernel without recovery that achieved considerable performance improvement over the original engine. Although, the authors do not propose specific changes to any of the different components, they give some insight on possible DBMS modifications for reducing their overhead.

Johnson et al. [Joh+09b] focus on the scalability of several disk-backed database engines. Their study shows the general lack of scalable performance of the studied engines on multicore processors. Additionally, the authors focus on the performance of Shore's storage engine, and propose a series of modifications to address its performance limitations. While focusing primarily on the storage engine, the authors considerably modify other parts of the engine for improving its scalability, namely, *log management* and *transaction management*. Although the authors do not propose fundamental design modifications, with their primary focus being the reengineering of the original design, the attained performance improvements are considerable, showing that databases can scale on multicores without a complete system redesign.

Jung et al. [Jun+13], present a study on the scalability of disk-backed databases on multicores. In this study the authors show that contention on lock management algorithms greatly reduce concurrency on multicores, thus restricting database performance even under non-conflicting workloads (such as read-only). Additionally, the authors propose a series of modifications to the transactional manager, by reengineering it using a read-after-write (RAW) coding style [Att+11; HS08]. Two common synchronization patterns are frequently used in the design of concurrent algorithms: read after write (RAW) and atomic write after read (AWAR). RAW patterns consist on a thread writing to some shared variable A, followed by the same thread reading a different shared variable B, without writing to B in between. The AWAR pattern consists of a thread reading some shared variable followed by the same thread writing to a shared variable (the write could be to the same shared variable as the read), where the entire read-write sequence is atomic. Examples of the AWAR pattern include read-modify-write operations such as a Compare-and-Swap (CAS).

Our study is complementary to these works in several aspects. While these works study the bottleneck of disk-based database, we focus on general purpose in-memory

databases. Additionally, we compare two different database implementations for a better understanding of how different development techniques affect the performance of the database. Furthermore, we try to understand how database engines behave when subjected, not only to different workloads, but also under different isolation levels. Finally, we will later show how applications can influence the performance of the database, by modifying the TPC-C bechmark in a database friendly way. Like some of these works [Joh+09b; Jun+13], for addressing the scalability problems identified, we follow a similar research path by reengineering some components for improving their efficiency in multicore systems, rather than radically changing the database design.

3.4.2 Improving scalability by reducing contention

Several works have focused on contention related problems that reduce database efficiency on multicores. This problem influences database performance by reducing throughput, since it greatly decreases concurrency. Proposed solutions for this problem tend to mitigate synchronization overhead, by focusing on transactional management (locks), and/or on low level concurrency control mechanisms used by the engine's internal data structures and algorithms (latches). As further discussed in Chapter 5, in our work, we also follow a similar direction but we take a more radical approach by completely avoiding any latches/locks in the data structures used to maintain data. Next we describe some of these works.

Cha et al. [Cha+01] address the scalability problem by focusing on the concurrency control mechanisms and algorithms used by main-memory index data structures. In this context, the authors proposes *OLFIT*, a latch-free index transversal algorithm for index trees (B+-Tree [Com79] and CSB+Tree [RR00]). OLFIT allows tree transversals without acquiring latches, while preventing updates from interfering. This is achieved by each node in the tree maintaining a latch and an associated version. This way, update operations acquire the corresponding node's latch (or latches if the update modifies a set of nodes) before executing, incrementing the node's version after updating it and before releasing the latch. Read operations execute without latching, starting by reading a node's version, followed by reading the corresponding values. It then tests if the node is latched and again reads its version. If the versions differ or if the node is latched, the operation aborts and repeats, completing successfully otherwise. While this is not a pure latch-free approach, it is one of the first proposals that address the problem of latching in multi-processor environments.

Sewall et al. [Sew+11] propose an adaptation of the Bulk Synchronous Parallel (BSP) concurrent execution model [Val90] for multicore environments, for allowing multiple read/write queries to execute atomically on B+-Trees without the use of latches. Instead of the traditional approach, where queries execute independently of each other, PALM groups multiple queries (including write operations) into batches, executing batches

sequentially. In each batch, the corresponding index operations are divided among concurrent threads. PALM executes read operations before write operations. This allows all tree transversals to execute concurrently without synchronization overhead, since no modification results from tree transversals. After executing read operations, PALM coordinates the remaining threads (i.e., threads executing write operations) for performing the corresponding tree modifications. These are allowed to execute concurrently on leaf nodes, when modifying distinct nodes. Otherwise, the corresponding threads coordinate, where one of them is elected for sequentially executing all the corresponding (batched) modifications. Likewise, modifications to the internal nodes of the tree are also batched and applied by a single thread. All necessary modifications are propagated up the tree in a similar fashion, where the corresponding tree modifications are batched and executed by a single thread. Thus, when reaching the root node, only a single thread executes all the necessary modifications. This allows the removal of latches since all tree modifications are executed sequentially by a single thread at the node level. This also allows PALM to eliminate possible deadlock occurrences that can occur when using conventional latching mechanisms.

Pandis et al. [Pan+11] follow a different path for reducing contention at the database storage level. The authors propose *physiological partitioning* (PLP), that combines techniques taken from shared-nothing and shared-everything designs. Contrarily to the traditional approach, where indices are implemented using a single data-structure, typically a B+-Tree, in this work the authors propose partitioning of data among several data structures, similar to the work of Graefe et al. [Gra03]. Thus, PLP uses a multi-rooted B+-Tree (called MRBTree), that partitions data among several B+-Trees. For identifying which partition to access, PLP uses an additional structure, called a partitioning table, which acts as the root of the index. Each tree maintains a subset of the entire key-space. This partitioning scheme allows PLP to reduce contention, since different partition may be transversed concurrently without coordination. Additionally, PLP further reduces synchronization overhead by restricting access to each partition to a single thread. This way, no concurrency control mechanisms (i.e., latches) are used at the index level. While additional latches may still be necessary, like page level latches, the authors argue these impose less contention then index latches.

Mao et al. [Mao+12] address database storage contention building on some of the features of the previously described systems [Cha+01; Gra03; Pan+11]. In Masstree [Mao+12] the authors present a key value store that partitions data among several B+-Trees, concatenated into a trie-like structure [Fre60], where each partition coveres a subset of the key-space. Like OLFIT, tree transversals do not acquire latches, while update operations acquire fine grain latches (only on the involved tree nodes). Additionally, for preventing inconsistent states from being exposed, Masstree employs an optimistic concurrency control, where: update operations, before updating a node, mark updated nodes as dirty and modify their version afterwards; and read operations, before reading a node, check the version of the node. If the version of all read nodes remain consistent,

and no node has been marked dirty, read operations execute successfully, otherwise these retry on a fresh snapshot.

Bw-Tree [Lev+13b; Lom+13] is an adaptation of a classic B+-Tree index for secondary storage (i.e., disk), designed for scaling on multicores. Bw-Tree addresses index contention on multicores by using a latch free approach. Thus, Bw-Tree eliminates traditional latches in favor of compare-and-swap (CAS) operations, for atomically modifying its state. Additionally, Bw-Tree improves cache consistency by using out-of-place modifications, i.e., deltas. Contrarily to the traditional approach, update operations (such as inserts, deletes or updates) do not directly modify tree nodes, instead these atomically append modification deltas to the corresponding nodes. Deltas represent the corresponding state changes and are transversed prior to the actual node, behaving as a stack of node modifications. Bw-Tree follows a Blink-tree approach [LY81], where each node maintains a pointer to the next node of the same level, allowing transversals on Bw-Tree to execute without latching (since link pointers prevent transaversals from observing inconsistent states due to concurrent state changes, i.e., node splits or merges [LY81]). Additionally, Bw-Tree also allows state modifications, due to update operations, to execute without latching (although assuming conflicting updates are prevented by external mechanisms, such as lock management). For preserving state consistency, Bw-Tree treats state modification operations (node splits or merges) as a kind of "transaction", where a *termination delta* is used to identify if the corresponding modifications have ended. Additionally, any state modifying operations is only allowed to execute after all previously initiated modifications have ended. To this end, whenever an operation identifies an incomplete modification, for example when a state modifying operation detects an unfinished concurrent node split (by reaching a new node through a link) that has yet to be propagated to the father node, it will complete the previous operation before executing its own. This guarantees that concurrent state modification operations execute in the same serialized order in all nodes.

Jung et al. [Jun+13] address the contention problems of lock management algorithms. In their study, the authors show that MySQL database performance in greatly compromised due to contention created by concurrency control mechanisms (latches) used by the lock management and deadlock detection algorithms. In this study, the authors discuss that some data races are benign, and that *memory barriers* combined with read-after-write (RAW) coding style [Att+11; HS08] have advantages over atomic write-after-read operations (CAS operations), due to increased cache efficiency. Thus, the authors propose a solution for reducing latches used by the lock manager, that combines the RAW coding style with a new lock acquisition and release pattern, that separates allocation and de-allocation of lock data structures from lock acquisition and release. Additionally, the authors pre-allocate and de-allocate locks asynchronously. Contrarily to the original approach where locks are allocated when needed and de-allocated when released, under the proposed modification locks are allocated before being needed (being maintained

in a pool), and are de-allocated after transactions have release them. This further reduces contention by reducing the number of operations executed in exclusion (since lock de-allocation is done asynchronously of locks releases).

3.4.3 State of the art in-memory databases

The mechanisms described in the previous section reduce contention in specific database components. In this section we describe some state-of-the-art database systems designed for multicore systems.

VoltDB [SW13], the commercial follow up of H-Store [Kal+08], follows the same design principles as H-Store. H-Store [Kal+08; Sto+07] is an OLTP system designed for distributed clusters of shared-nothing machines. H-Store orchestrates the nodes of a cluster, by partitioning the database among several single-threaded engines, called sites, each running on a single processor core of a node. H-Store further replicates each partitions to improve both performance and availability. Applications interact with H-Store by predefined stored procedures, each identified by a unique name, consisting of structured control code mixed with parameterized SQL commands. Data partitioning is done based on the predefined stored procedures, in order to maximize efficiency. Transactions may execute on a single site, or on multiple sites. When data is accessible on a single site, transactions execute to completion on the corresponding site, without additional coordination or logging overhead, since each site is single threaded. Serializability is achieved by running each transaction in sequence. When transactions require data from different sites, additional coordination is required to provide isolation. In this case a global controller is used for deciding a serial order for operations to execute in the corresponding sites. Durability is maintained using asynchronous transaction-consistent checkpoints of the state on each site's main memory. Additionally, a transaction log records each transaction with the corresponding identifiers.

In Oracle TimesTen [Lah+13] in-memory database, concurrency control mechanisms are designed to scale on multicores by trading locks for latches, whenever possible, and using fine-grain locking. Additionally, contrarily to the traditional approach, where accessing items requires translating logical addresses (kept by indices) for physical addresses (associated to memory cache buffers), TimesTen maintains physical addresses directly in the indices, i.e., indices maintain pointers to tuples. Durability in achieved combining checkpointing and write-ahead logging. For minimizing overhead, logging is divided into multiple partitions, that are written in parallel. Sequential order is restored when reading the log from disk. Like most in-memory databases, TimesTen trades durability for performance, by allowing transactions to commit without waiting for log records to be flushed to disk. SolidDB uses a pessimistic concurrency control allowing row or table level locking, offering the highest isolation level, serializable.

SolidDB [Lin+13] is a relational database that combines in-memory and disk-backed tables. Pure memory configurations maintains data directly on indices, using custom

built trie like structures [Fre60]. These data structures use a variation of path- and level-compression and are built on top on leaf nodes similar to B+-trees. For addressing contention, indice transversals do not use locks or latches, while write operations use two-level locking for protecting agains conflicting concurrent writes. Indices uses optimistic concurrency control, associating data with versions. These are used for detecting conflicts during read operations, similar to OLFIT [Cha+01]. SolidDB uses a pessimistic concurrency control using row level locking (optimistic concurrency control is only available for disk-backed tables), offering repeatable read as the highest isolation level (serializable isolation level is available only for disk-backed tables).

Tu et al. [Tu+13] have proposed Silo, an in-memory transactional database for multicore systems. Silo addresses scalability limitations of traditional databases by building on a Masstree-inspired [Mao+12] storage engine, and relying on an optimistic concurrency control (OCC) that offers serializable isolation semantics. Silo assumes a one-shot request model, where all parameters for each request are available at the start. Thus, requests complete without further client interaction. For providing serializable isolation semantics, the authors define time periods called *epochs*. Committing transactions start by acquiring locks on all modified records. After this point, the epoch number is registered (a memory fence/barrier is used to prevent code reordering due to processor optimizations). This defines the serialization order for the transactions. On a second phase, read records are examined to guarantee these have not been modified during the duration of the transaction (using the records *transaction identifier* (TID)). If some record has been modified, or is locked by a concurrent transaction, the committing transaction aborts releasing all locks. Otherwise the transaction is allowed to commit. Finally, all modified records and respective TIDs are updated accordingly.

Hekaton [Dia+13] is an in-memory extension for SQLServer, built on top of Bw-Tree [Lev+13b; Lom+13] and LLAMA [Lev+13a]. Transaction isolation is supported by a multi-version concurrency control that requires no locks or lock tables. Read only transactions are serialized in the past, thus do not require additional validation. For update transaction, serialization is achieved by each transaction maintaining both its read and write set, for validation during commit. This is done by revisiting all previously read locations and verify their validity (if their versions remain unchanged) [Lar+11]. Hekaton builds on the Bw-Tree, used as index data-structure, and LLAMA [Lev+13a] a cache/storage subsystem designed for modern hardware. LLAMA's design is based on traditional cache/storage subsystems, where pages are read from secondary storage to main memory on demand. However, LLAMA supports latch-free page modification operations, using compare-and-swap atomic operations, replacing traditional latches used to guard pages from concurrent accesses. Additionally, page modifications are accomplished using deltas, appended to pages using compare-and-swap operations, thus avoiding in-place modifications. Deltas are applied to physical pages based on heuristics (when the number of deltas per page exceeds 10). This allows LLAMA to deal with the overhead on write operations in modern SSDs, since modifying a disk page results in writing a new page,

i.e. requires reading the original page and writing it with the respective modification to a different locations. Deltas prevent original pages from being modified, thus only delta pages are written to disk. Additionally, LLAMA organizes data on secondary storage in a log structured manner [RO92], where all updates are written sequentially in a log-like structure. Like before, this approach helps in reducing the overhead when writing pages to disk. Durability relies on both logging and checkpointing to external storage, with no logging being done during transaction execution. Only transactions that pass their validation phase write to log their corresponding modifications. Similar to TimesTen [Lah+13], Hekaton allows log to be partitioned over multiple devices, since commit ordering is determined by each transaction's end timestamp.

System	Architecture	Concurrency Control	Isolation Level (highest)	Index Conc. Control	Client Interactions
VoltDB/ H-Store	Distributed (Partition + Replication)	Single threaded (1 per CPU core)	Serializable	Non (Single threaded storage)	Single shot
Oracle TimesTen	Centralized + Replicated	Pessimistic	Serializable	Latches + Fine-grained locking	ODBC/ JDBC
SolidDB	Centralized + Replicated	Pessimistic	Repeatable Read	Non + Locks (w/ writing)	ODBC/ JDBC
Silo	Centralized	OCC	Serializable	Non + latches (w/ writing)	Single shot
Hekaton	Centralized + Replicated	MVCC	Serializable	Non + CAS ops. (w/ writing)	ODBC/ JDBC

Table 3.1: State of the art comparison.

Table 3.1 compares some of the features of the previously described systems. From this comparison it is possible to see that no consensus exists in terms of concurrency control mechanisms, with each system using a different approach. It is also possible to see that most systems try to offer the highest isolation level, serializable. This is of considerable importance since it relinquishes application developers from reasoning about isolation phenomena. Finally, all systems take into consideration the implication that latching has on index, and consequently, database performance, with most systems using latch free index transversal solutions. In our work, we also follow a similar direction, to reduce the overhead of latching in data structures. However, we take a more radical approach by completely eliminating latches/locks in the data structures used to maintain data.

3.4.4 Alternative database designs

Column based Databases Besides previously described works, which rely on traditional row-based storage, recent works have proposed the use of column-based storage. These approaches are often focused on business intelligence and analytical processing (OLAP) workloads, we briefly describe some of these proposals here and discuss the major differences of column-based storage.

MonetDB [Bon+08; Man+09], C-Store [Sto+05], and SAP HANA [F+12; Sik+12] are examples of database management systems that trade traditional row-oriented storage systems for column-oriented ones. Row-stores typically store the database tables as arrays

of records (i.e., tuples), where each entry of the array holds a record. Also, each record holds the corresponding values of its respective attributes. On the other hand, column-store vertically partition each database table into a collection of individual columns, each stored separately. Thus, in column-stores, database tables are decomposed in a collection of arrays (one for each attribute), where the corresponding attribute values of each record are stored.

Row-stores have been the norm for relational databases, since these offer improved data access efficiency for disk based storage by allowing records to be written to, and read from, disk in a single operation[Aba+12]. This is especially important to reduce I/O overhead due to disk search, since reading or writing to the same disk position reduces latency. On the other hand, column-stores tend to require additional disk accesses whenever reading or writing more than one attribute [CK85]. This is particularly prominent for insert and remove operations, since all table attributes need to be accessed. The same may not hold true for main memory, since memory access time is identical independently of being sequential or random.

Row-stores offer some advantages, over column-based stores, on memory resident databases. Especially when reading only a subset of attributes on a set of records. This is a result of CPUs caching entire memory rows. Since row-stores store records contiguously in main memory, cache space is wasted by values of attributes that are not required. On the other hand, column-stores use the CPU cache more efficiently by eliminating waist, since the set of values of each attribute are contiguously stored in main memory, thus occupy cache lines without waste [Aba+12; Man+09]. For this reason column-stores have been used for some specific read-intensive workloads, like OLAP. Update operations, such as inserts and deletes, still require additional memory accesses when using column-stores. Under these conditions cache utilization is not as efficient as in read operations, further compromising performance.

Thus, the major weakness of column-stores, compared to traditional row-stores, is their performance under update workloads. For dealing with this, some systems resort to a slipt architecture by using a "read-store" and a "write-store" [Man+09; Sto+05]. The read-store maintains stale information using a column-store, while the write-store maintains the more recent updates. MonetDB uses two additional columns for each column base in the scheme, for marking pending inserts and deletes (updates are mapped as a delete followed by an insert). C-Store follows a different approach, maintaining stale data in a column-store and updates on a row-store. Both systems merge read- and write-store information during query execution. For minimizing the size of the write-store, updates are propagated periodically to the read-store. SAP HANA uses a hybrid approach for dealing with these scenarios, replicating data among row and column storage, and dividing operations accordingly. OLTP transactions execute primarily on a row-store being propagated asynchronously to the column-store, while OLAP transactions execute on a column-store [F+12].

These approaches result from the traditional practice for data warehouses, where

large data set are queried ad-hoc and updated in bulk. This is compatible with OLAP workloads, since business intelligence and analytical processing may use stale data without compromising results [Pla09; Sto+05]. Column-stores are also favorable to OLAP workloads, since these workloads tend to analyse data from specific subsets of attributes.

Focusing on concurrency control mechanisms, these systems tend to use multiversion concurrency control, providing snapshot isolation [F̈12; Man+09; Sik+12].

Chapter 4

Modification-free Solution

As put in evidence in the previous chapter, current general purpose IMDBs do not scale on multicore systems, independently of the isolation semantics and workload nature. Additionally, we have identified contention on the storage sub-component as a major performance bottleneck of these systems under read-only workloads.

In this chapter we present a generic solution, that requires no modifications to either databases or to the applications, for reducing the contention among transactions. By treating multicore machines as extremely low latency clusters, extended with some shared memory, we propose a middleware system, called *MacroDB*, that coordinates a set of database replicas. MacroDB, an example of a Macro-Component [Mar+10], builds from the knowledge of distributed and replicated databases and improves scalability by distributing and balancing load among different replicas.

4.1 Modification-free approach

In our quest to improve the scalability of in-memory databases, our initial approach is to use databases as black-boxes. This approach uses database replication techniques and builds on the knowledge from distributed and replicated database systems. By treating a multicore machine as an extremely low latency cluster, extended with shared memory, we design a middleware system, called *MacroDB*, as a collection of coordinated database replicas.

MacroDB is designed to offer improved database scalability and performance by distributing load among the several internal replicas, reducing contention. To this end, it uses a master/slave replication approach, where update transactions execute on the master replica, which holds the primary copy of the database. The slaves maintain independent secondary copies of the database, receiving read-only transactions from clients,

while updates are asynchronously propagated to them, upon committing on the primary replica. This approach reduces contention since: *i*) Update transactions do not block, nor are they blocked, by concurrent read-only transactions, since different types of transactions execute in different replicas; *ii*) Read-only transactions are fully executed on slave replicas, reducing the number of transactions each replica processes, thus distributing load among the available replicas, and *iii*) Update transactions executed in slave replicas as sequential batches of updates, thus leading to no contention among them.

MacroDB provides a scalable data management solution that does not require any modifications to neither the database engines or to the applications. Our experiments show that MacroDB is able to provide performance benefits up to 4× over the standalone database engines under read-dominate workloads, while for write-dominant workloads MacroDB suffers from a 25% overhead over the non-replicated database.

4.2 Macro-Components

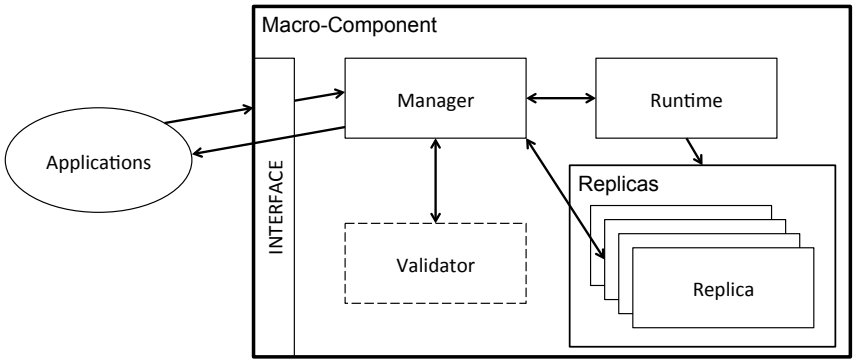


Figure 4.1: Macro-Component design and behavior

MacroDB follows the design principles of *Macro-Components*. Macro-Components are software components designed to offer improved performance and/or reliability compared to their standard siblings. Taking advantage of the available computational resources offered by multicore machines, Macro-Components maintain a set of, possibly diverse, component implementations of the same interface (called *replicas*), as presented in Figure 4.1. Applications can benefit from the use of Macro-Components, simply by replacing standard components by their Macro-Components siblings.

By treating multicore machines as extremely low latency clusters, extended with shared memory, Macro-Components adopt identical principles as traditional replicated systems, applying them on a single multicore machine. Thus, instead of distributing replicas on to distinct machines, Macro-Component maintain replicas on the same machine.

This allows Macro-Components to offer improved performance, over standard components, in a similar manner as traditional replicated systems, i.e., by distributing and

balancing concurrent method calls among the available replicas. Additionally, Macro-Components offer an asynchronous execution model that allows methods to execute concurrently with the calling application thread. This, prevents application threads from being held by methods that return no results, or by methods returning values that are non-relevant for the application.

Following the same principles as N-version programming [Avi85; AK84], when using replicas with diverse implementations, Macro-Components offer improved reliability, compared to standard components. Implementation diversity allows Macro-Components to detect buggy behavior of replicas, preventing these bugs from being exposed and affecting the reliability of applications. Since different implementations offer the same functionality and maintain the same abstract state, buggy behavior is detected by identifying state or result divergences amongst replicas.

Combining diversity and method call distribution allows Macro-Components to exploit the performance differences of the diverse replicas. Since different implementations tend to offer different performances, and no *single* implementation is the fastest for all operations, Macro-Components can be implemented to exploit these differences by executing each operation in the corresponding fastest replica, allowing the fastest result to be returned to the calling application. This way, Macro-Components can offer the best performing result to the application. Next we detail the generic design and behavior of the Macro-Component abstraction.

4.2.1 Macro-Component design and behavior

As previously presented, Macro-Components are software components that encapsulate several, possibly diverse, implementations of the same specification, called *Replicas*. The generic design of a Macro-Component, as depicted in Figure 4.1, is composed by three main sub-components: the *Manager*, responsible for coordinating method execution on the replicas, the *Runtime*, responsible for executing operations on the replicas, and the *Replicas*, the components responsible for maintaining the state. An additional optional component, the *Validator*, is responsible for validating the results returned by the replicas.

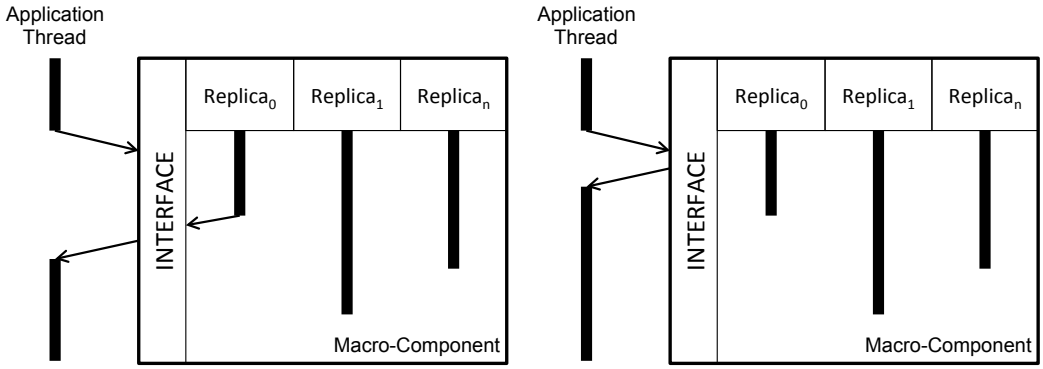
Whenever a method is called on a Macro-Component, the Manager decides if the corresponding method should be called on all or a sub-set of the Replicas. Normally, write methods execute in all Replicas, since these modify their internal state, while read methods can execute on a sub-set of Replicas, since no state modification results from their execution. The Runtime is then responsible for executing the respective method on the set of Replicas, with the gathered results being passed back to the Manager.

After results are gathered, the Manager may pass them to the Validator for detecting possible inconsistencies, or return one result to the application without validation. When using the Validator, only valid results, i.e., results in accordance to the majority, are returned to applications. Whenever inconsistent results are detected, the corresponding faulty replicas are marked for recovery, and temporarily removed from the set of active

replicas. Also, if some replica is unable to produce a result within a certain time limit, the replica is considered faulty and threads executing in the replica are aborted. The time limit is defined by the time taken by the majority of the replicas to reply plus an additional tolerance.

This allows Macro-Components to offer improved fault-tolerance, by executing methods on the majority of replicas and comparing results, or to offer improved performance, by executing operations on the fastest replica, or by distributing method invocation among different replicas, for minimizing contention.

Additionally, the Runtime decouples the execution of the callee from the method, i.e., the thread calling the method can be different from the thread that executes it. This allows Macro-Components to offer two different execution models: *synchronous* or *asynchronous*.



(a) Synchronous Execution (b) Asynchronous Execution

Figure 4.2: Macro-Component Concurrent Execution Model

The synchronous execution model forces application threads, when performing method invocations on a Macro-Component, to wait for it to return, i.e., waiting for methods to execute before continuing, as presented in Figure 4.2(a). While, the asynchronous execution model allows application threads, when performing method invocations on a Macro-Component, to proceed execution concurrently with the method, i.e., without waiting for the methods to return, as presented in Figure 4.2(b).

The latter can be used whenever a method returns non relevant results, or no result at all, to the application, while the former can be used in all other situations. The asynchronous execution model also further contributes for Macro-Components to offer improved performance over standard components, since applications and method execution can execute concurrently. Next we detail the generic implementation of the Macro-Component abstraction.

4.2.2 Implementation and Runtime

The following discussion focuses on a Macro-Component implementation using the Java programming language. For simplicity, we use the example of a simple counter with

three operations:

- *incrementCounter*, that increments the current value of the counter and returns no result;
- *getCounter*, that returns the current value of the counter; and
- *incrementCounterAndGet*, that increments the current value of the counter and returns the new value.

Listing 4.1: Macro-Counter Implementation.

```
1  class MacroCounter {
2      // each Replica is a Counter
3      Replica<Counter> [] replicas;
4      VersionVector version;
5
6      void incrementCounter() { ... }
7
8      int getCounter() { ... }
9
10     int incrementCounterAndGet() { ... }
11 }
```

Listing 4.2: Replica Implementation.

```
1  class Replica<T> {
2      T replica;
3      ReplicaID id;
4      VersionVector version;
5      ...
6  }
```

In conformity with the presented design, a Macro-Component has a set of Replicas that implement a common interface, and an associated version-tracking mechanism, where each Replica is a wrapper of a standard component implementation, with an associated unique identifier and a version-tracking mechanism, as presented in Listings 4.1 and 4.2 respectively.

Listing 4.3: Counter Macro-Component Implementation.

```
1  class MacroCounter {
2      // each Replica is a Counter
3      Replica[] replicas;
4
5      static final Action incAction = new WriteAction() {
6          public Object execute(Replica replica, Object ... args) {
7              replica.incrementCounter();
8              return null;
9          }
10     };
11     // Asynchronous write operation.
12     void incrementCounter() {
13         Task t = createTask(incAction, replicas);
```

```
14          Manager.asyncExecuteAll(t);
15      }
16
17      static final Action getAction = new ReadAction() {
18              public Object execute(Replica replica, Object ... args) {
19                  return replica.getCounter();
20              }
21      };
22      // Synchronous read operation.
23      int getCounter() {
24          Task t = createTask(getAction);
25          // 0 -> replica index in which the
26          // task will execute synchronously
27          Manager.syncExecute(t, 0);
28      }
29
30      static final Action incAndGetAction = new WriteAction() {
31          public Object execute(Replica replica, Object ... args) {
32              return replica.incrementCounterAndGet();
33          }
34      };
35      // Synchronous write operation.
36      int incrementCounterAndGet() {
37          Task t = createTask(incAndGetAction);
38          // 0 -> replica index in which the
39          // task will execute synchronously
40          return Manager.syncExecuteAll(t, 0)
41      }
42
43      ...
44  }
```

Listing 4.4: Task Implementation.

```
1   class Task {
2       Action action;
3       Replica[] replicas;
4       VersionVector version;
5       Long callerID;
6       Object[] args;
7       Object[] results;
8
9       public void execute(int replica) {
10          // wait for previous tasks to finish execution;
11          while(!replicas[replica].isUpToDate(version));
12
13          results[replica] = action.execute(replicas[replica], args);
14          if(this instanceof WriteTask)
15              replicas[replica].incrementVersion(callerID);
16      }
17      ...
18  }
```

Whenever a method is called on a Macro-Component, a *Task* object is created, as presented in Listing 4.3 (line 13, 24 and 37). Tasks are abstract representations of the called method, as well as the corresponding arguments and result set, as presented in Listing 4.4. Since Java programing language does not permit to pass methods as arguments,

methods are encapsulated as *Action* objects, as presented in Listing 4.3 (lines 5-10, 17-21 and 30-34). Actions define a single method, called *execute*, that is responsible for calling the corresponding method on a replica (with the corresponding arguments).

Tasks also include additional information, such as the version in which the method should execute, to allow the Runtime to preserve state consistency. Since some operations change the component state while others do not, we represent these differences using ReadTasks and WriteTasks. These differ from one another by incrementing the replica's version.

Listing 4.5: Macro-Component Manager Implementation.

```
1  class MacroManager  {
2      Version macroVersion;
3      Version[] replicaVersions;
4      int replicas;
5
6      Task createTask(Action action, Replicas replicas, Object ... args) {
7          Long callerID = Thread.currentThread().getId();
8          // timestamp used to guarantee ordering of operations
9
10         Version versionstamp;
11         if (action instanceof WriteAction)
12             versionstamp = macroVersion.incrementAndGet(callerID);
13         else
14             versionstamp = macroVersion.get(callerID);
15
16         return new Task(action, replicas, callerID, versionstamp, args);
17     }
18
19     // Asynchronous concurrent execution of Task t
20     void asyncExecuteAll(Task t) {
21         // sending Task to Runtime for execution
22         Runtime.submitTask(t);
23     }
24
25     // Synchronous concurrent execution of Task t
26     Object syncExecuteAll(Task t) {
27         // sending Task to Runtime for execution
28         Runtime.submitTask(t);
29         while(!t.hasResult());
30         return t.getResult();
31     }
32
33     // Synchronous in line execution of Task t in Replica rep
34     Object syncExecute(Task t, int rep) {
35         // rep -> replica index in which the task will execute
36         t.execute(rep);
37         return t.getResult(rep);
38     }
39
40     // Synchronous inline execution of Task t in Replica rep
41     // Asynchronous concurrent execution in other replicas
42     Object syncExecuteAll(Task t, int rep) {
43         // rep -> replica index in which the task will execute
44         t.execute(rep);
45         // sending Task to Runtime for execution on additional Replicas
```

```
46          Runtime.submitTask(t);
47
48          return t.getResult(rep);
49      }
50  }
```

WriteAction are used for methods that modify the state of the replicas, thus advance the Macro-Component version, and the version of the replicas, after executing. On the other hand, ReadActions are used for methods that do not modify the state of the replicas, thus do not advance the Macro-Component version or replica versions, as presented in Listings 4.5 (lines 6-17) and 4.4 (lines 9-16).

It is the Manager responsibility to create and execute Tasks. Although these function could be directly implemented in the Macro-Counter, used in the example, we implement the Manager separately for allowing developing Macro-Components automatically. Thus, the Manager, in conjunction with the Runtime, allows Tasks to be executed: *i*) in-line by the calling thread on a specific Replica. This synchronous in-line execution model minimizes overhead, as the application thread does not need to synchronize or wait for other threads; *ii*) concurrently by the runtime on all Replicas either synchronous, i.e., waiting for it to return, or asynchronously, i.e., without waiting for it to return. Additionally, the Manager offers a combination of both models, executing methods in-line on a specific Replica, i.e., by the application thread, while executing the method concurrently in the remaining Replicas, using the Runtime, as presented in Listing 4.5 (lines 20-23, 26-31, 34-37 and 42-49).

In the Macro-Counter example, presented in Listing 4.3, the *incrementCounter()* method (lines 12-15) illustrates the asynchronous concurrent execution model, where the created Task is executed by the Runtime and the calling thread continues execution without waiting for any result. The *getCounter()* method (lines 23-28) illustrates the synchronous in-line execution model, where the Task is executed by the calling thread on a specific Replica, returning that result to the application. While, the *incrementCounterAndGet()* method (lines 36-41) illustrates the combination of the two models, where the Task is executed in line on a given Replica and sent to the Runtime for executing on the additional Replicas. The first result is then returned to the application.

The synchronous in-line execution approach contributes to a reduction on the number of asynchronous Tasks that need to execute, thus reducing synchronization overhead and resources used by Macro-Components. Also, since only write operations need to be executed in all Replicas, for preserving state consistency, these are normally executed by the Runtime. As presented in the example, this execution can be synchronous, if the result is relevant to the application, for example, using the inline approach presented in the *incrementCounterAndGet()* method; or asynchronously, if the returning is non relevant or no result is returned, for example, using the Runtime as presented in the *incrementCounter()* method. The asynchronous execution approach contributes to an important performance improvement for applications, since these can proceed without "waiting"

for the operation to return.

Listing 4.6: Runtime Implementation.

```
1  class Runtime {
2      OrderedQueue<Task>[] tasks;
3      Executor[] pool;
4      ...
5
6      // adds a task to the queue
7      void submit(Task t) {
8          for(int i = 0; i < tasks.length; i++)
9              tasks[i].add(t);
10     }
11
12     // retrieves the next task from the queue
13     Task getNext(int executorID) {
14         return tasks[executorID].pool();
15     }
16     ...
17 }
```

4.2.2.1 Preserving Replica State Consistency

For managing the execution of Tasks, the Runtime uses a producer/consumer scheme, where submitted Tasks are scheduled for execution according to their version, as presented in Listing 4.6. Ordered queues, one for each replica, are used to maintain Tasks ordered by their respective version. This allows the Runtime to execute Tasks in all Replicas according to their total order [Lam78], thus guaranteeing the semantics of the component and maintaining replica state consistency.

Listing 4.7: Executor Implementation.

```
1  class Executor implements Runnable {
2      int executorID;
3      int replica;
4      ...
5
6      void run() {
7          while(Runtime.isRunning()) {
8              Task t = Runtime.getNext(executorID);
9              if(!f.alreadyExecuted(t.replicas[replica]))
10                 t.execute(t.replicas[replica]);
11         }
12     }
13     ...
14 }
```

A fixed size pool of Executor threads, one for each Replica, retrieves Tasks from the queue and executes them on the corresponding Replica, as presented in Listing 4.7. Tasks guarantee that the corresponding method executes in a Replica accordingly to its version, Listing 4.4 (line 11), thus Macro-Components offer a consistent single copy view of the underlying replicas. Additionally, the Executors guarantees that each Task does not execute more than once in the same Replica, Listing 4.7 (line 9).

By using a fixed number of Executors, we guarantee that, at most, only N+M threads will be executing at any time, where N is the total number of Executors and M is the number of threads of the application.

4.3 MacroDB

We now present MacroDB, an example of a Macro-Component for scaling IMDBs on multicore machines. MacroDB replicates the database on several database engines, all running on the same machine, while offering a *single-copy serializable view* of the database to clients [Ber+87].

MacroDB works independently of the underlying database engine, acting as a transparent layer between applications and the database. SQL statements, received from the application, are passed, without modifications, to the underlying engines, thus making MacroDB easy to deploy, since it does not require any modification to existing applications or database engines. This section details the architecture and algorithms used in MacroDB.

4.3.1 Architecture

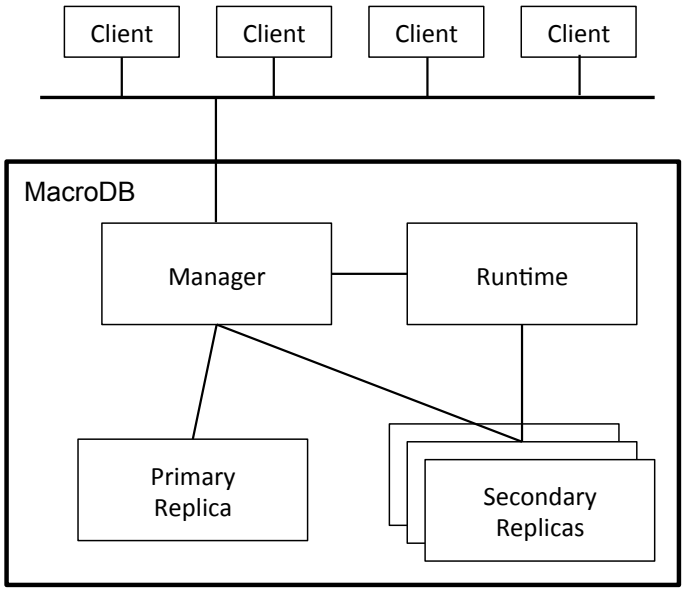


Figure 4.3: MacroDB architecture.

MacroDB uses a master-slave replication scheme [Hel+96; Wie+00], where the master maintains the primary copy of the database, while the slaves maintain secondary replicas. The MacroDB architecture, depicted in Figure 4.3, is composed by the three main components of a Macro-Component:

- the *manager*, responsible for coordinating transaction execution in the replicas;

- the *runtime*, responsible for propagating operations from the primary to the secondary replicas, i.e., executing operations in the secondary replicas; and
- the *replicas*, i.e., the database engines responsible for maintaining copies of the database.

Clients remain oblivious of the replicated nature of MacroDB since it offers them a standard JDBC interface, and provides them with a *single-copy serializable view* of the database [Ber+87]. To this end, clients do not communicate directly with the database engines, instead they communicate with the MacroDB JDBC compliant front-end.

This front-end, in conjunction with the manager, coordinate client queries and their respective execution in the underlying replicas. MacroDB receives statements from clients and executes them, without modification, in the appropriate replica, replying to clients the respective results. All update transactions are executed on the primary replica, while read-only transactions execute on a single secondary replica.

The manager also propagates update transactions, that successfully commit in the primary, to the runtime for maintaining consistency of the secondary replicas. Thus, its main functions are: *i*) to route client requests and execute them on the appropriate replicas, replying the results back to the clients; *ii*) to manage operation execution to guarantee that the system provides a single consistent serializable view of the replicated database to the applications, and *iii*) to detect and recover possible replica failures.

Update transactions, received from clients, execute concurrently on the primary copy, while read-only transaction execute concurrently on secondary replicas. For preserving consistency each replica maintains an associated *version* that registers the number of update transactions committed in the respective replica. This value is incremented every time a commit is successfully executed. Contrarily to the general approach used by MacroComponents, the version of the primary replica also defines the MacroDB version. This is possible since all update transactions are executed synchronously by the primary.

Upon successfully committing on the primary, and advancing the replica's version, update transactions are sent to the runtime for execution on the secondary replicas. These are propagated in statement batches, stamped with the version value of the primary at commit time. The runtime guarantees these batches execute on the secondary replicas in the same order as committed on the primary, i.e., according to their respective version values, similarly to CBS [Van+07]. By sequentially executing each commit in the master replica, and advancing the version counter, we define a correct serialization order for update transactions, without forcing an *a priori* commit order.

This means that secondary replicas might not be completely up to date at a given moment. To guarantee linearizability[HW90], read-only transactions only execute on a secondary replica which has, at least, the same version as the master. To achieve this, before the first operation of each new transaction, the MacroDB manager registers the current version value of the master replica. This defines the expected replica version, i.e., the state of the database, in which the transaction must execute. Since all update

transactions execute directly on the master replica, this is only used by read-only transactions, since read-only transaction can only execute on up-to-date replicas. This means read-only transactions may be held up waiting for a secondary replicas to be updated, before executing.

For maintaining the secondary replicas updated, the runtime maintains a queue of pending update batches and a set of *executor* threads, one for each secondary replica. These threads execute the update batches on their respective secondary replica in the same order these commit in the primary replica. Respecting this order guarantees that all replicas evolve to the same consistent state.

To achieve this, the runtime maintains the queue of update batches ordered by their respective version. Additionally, each executor only applies an update if its replica is up to date. Since the runtime maintains only one executor per replica, and the updates are kept ordered, the runtime guarantees state consistency. We will now describe, in greater detail, the implementation of MacroDB.

4.3.2 Implementation details

MacroDB offers applications a JDBC compliant front-end. During this discussion we present some implementation details of this front-end and try to show how this is integrated with the remaining MacroDB components. Above all, we try to present a clear view of how MacroDB abstracts applications from its replicated nature, and offers clients a single copy serializable view of the database.

4.3.2.1 Requirements

Since MacroDB has been designed to use database engines as blackboxes, i.e., without modifications, we require the underlying engine to use lock bases concurrency control, thus guaranteeing commit operations do not block. Additionally, for providing single copy serializable view of the database, we require the underling engines to use serializable isolation level. These requirements are supported by, possibly, every major database implementation, including the ones used for the MacroDB evaluation.

4.3.2.2 Connecting to MacroDB

As presented before, MacroDB offers a JDBC compliant front-end, thus applications establish connections to MacroDB using the JDBC getConnection operation.

Whenever a client tries to establish a new connection to MacroDB, the manager tries to connect to all MacroDB replicas, as presented in Listing 4.8. If this is the first connection to MacroDB, the manager will instantiate each replicas and create the appropriate metadata, as presented in Listing 4.9 (lines 7-11 and 14-17).

As an example of a Macro-Component, each replica of MacroDB has an associated unique identifier and a version tracking mechanism. These are unique in the systems,

i.e., each replica maintains the same identifier and version counter independently of the number of client connections.

Subsequent client connections to MacroDB, result in the creation of a new connection to each replica, sharing the previously created metadata. Thus, each MacroDB connection has an exclusive associated context in each replica, i.e., has an exclusive connection to each database replica.

Listing 4.8: MacroDB Driver Implementation.

```
class MacroDBDriver implements java.sql.Driver {
    ...
    // return a JDBC Connection object that maintains
    // a connection to each database replica
    Connection connect(String url, Properties p) {
        Replica[] connectionsToReplicas = MacroDBManager.connectTo(url, p);

        return new MacroDBConnection(connectionsToReplicas);
    }
    ...
}
```

Listing 4.9: MacroDB Driver Implementation.

```
class MacroDBManager{
    int replicaCount;
    Version[] replicaVersions;
    ...
    // returns the version of a given replica,
    // creating a new one if it does not exist
    Version getReplicaVersion(int replica) {
        if replicaVersions[i] == null)
            replicaVersions[i] = new Version();
        return replicaVersions[i];
    }

    // establishes a connection to a database replica
    Connection createConnection(String url, Properties p, int replica) {
        String replicaURL = url + i;
        return DriverManager.getConnection(replicaURL, args));
    }

    // establishes a connection to the database replicas
    Replicas[] connectTo(String url, Properties p) {
        Replica[] connections = new Replica[replicaCount];
        for(int i = 0;i < replicaCount; i++) {
            Connection con = createConnection(url, p, i);
            Version v = getReplicaVersion(i);
            connections[i] = new Replica<Connection>(con, v);
        }
        return connections;
    }

    Task createTask(Action action, Replicas reps, Object ... args){
        if (action instanceof ReadAction)
            return new Task(action, reps, replicaVersions[0].get(), args);
        else
```

```
34              return new Task(action, reps, replicaVersions[0].incrementAndGet(), args);
35      }
36
37      Task createTask(Action action, Replicas reps, Version version, Object ... args){
38          return new Task(action, reps, version, args);
39      }
40      ...
41  }
```

After establishing a connection, clients interact with MacroDB using the standard JDBC Connection interface. When the clients calls methods that directly access the database, such *setAutoCommit()* or *setTransactionIsolation()*, the respective method are called on the master replica, forwarding the call to the runtime after successful execution, for executing on the remaining replicas to guarantee consistency, as presented in Listing 4.10 (lines 14-18).

Since these methods are context related, i.e., since these methods only modify the state of the corresponding connection (session) and not of the entire database, these do not modify the version of the replica. On the other hand methods that modify the state of the database, such as the *commit()* method, advance the version counter when successfully executed, as described later.

When a client wants to initiate a new transaction, by calling, for instances, the *createStatement()* or *preparedStatement()* methods, the manager calls the corresponding methods on all replicas, and creates a new MacroDBStatement/MacroDBPreparedStatement object. These maintain the corresponding Statement/PreparedStatement that resulted from executing these calls on the replicas, as presented in Listing 4.10 (lines 22-33).

Again, since these methods do not change the state of the database, these do not modify the version value of the replicas. Next we detail how MacroDB manages the execution of transactions, i.e., how MacroDB manages Statement method calls. Although this discussion focused on the Statement JDBC interface, a similar approach is used for the PreparedStatement JDBC interface.

Listing 4.10: MacroDB Connection Implementation.

```
1   class MacroDBConnection implements java.sql.Connection {
2       // database engine connections
3       Replica<Connection>[] connections;
4       boolean autoCommit;
5       boolean readOnly;
6       int currReplica;
7       Version txInitVersion;
8       Queue batch;
9       ...
10
11      static final Action setAutoCommit = new ReadAction {
12          Object execute(Replica replica, Object ... args) throws SQLException{
13              ((Connection)replica.getReplica()).setAutoCommit((Boolean)args[0])
14          }
15      }
16      // JDBC setAutoCommit method;
17      // Defines if the database auto-commits every statement
```

```
18      void setAutoCommit(boolean auto) throws SQLException {
19          Task t = Manager.createTask(setAutoCommit, auto);
20          Manager.syncExecuteAll(t, 0);
21          this.autoCommit = val;
22      }
23
24      // JDBC createStatement method;
25      // Creates a new Statement for this connection
26      Statement createStatement() throws SQLException {
27          int count = Manager.replicaCount;
28          Replica<Statement>[] statReplicas = new Replica[count];
29          for(int i = 0;i < count; i++) {
30              Statement stat = ((Connection)connections).createStatement();
31              statReplicas[i] = new Replica(stat, Manager.getReplicaVersion(i));
32          }
33          return new MacroDBStatement(stats);
34      }
35
36      static final Action setReadOnly = new ReadAction {
37          Object execute(Replica replica, Object ... args) throws SQLException{
38                  ((Connection)replica.getReplica()).setReadOnly(args[0]);
39              }
40          }
41      }
42      void setReadOnly(boolean read) throws SQLException {
43          readOnly = read;
44          if(read) {
45              currReplica = nextSlave();
46          }
47          else {
48              currReplica = 0;
49              batch = new Queue();
50          }
51          Task t = Manager.createTask(setReadOnly, read);
52          Manager.syncExecute(t, currReplica);
53          if(!read)
54              batch.add(t);
55      }
56
57      static final Action readCommit = new ReadAction {
58          Object execute(Replica replica, Object ... args) throws SQLException{
59                  ((Connection)replica.getReplica()).commit();
60              }
61      }
62      static final Action writeCommit = new WriteAction {
63          Object execute(Replica replica, Object ... args) throws SQLException{
64              synchronized (replica) {
65                  ((Connection)replica.getReplica()).commit();
66                  replica..getAndIncrementVersion();
67              }
68          }
69      }
70      // JDBC commit method;
71      void commit() throws SQLException {
72          if(readOnly) {
73              Task t = Manager.createTask(readCommit, txInitVersion);
74              Manager.syncExecute(t, currReplica);
75              txInitVersion = null;
```

```
76          }
77          else {
78              Task t = Manager.createTask(writeCommit);
79              Manager.syncExecute(t, currReplica);
80
81              batch.add(t);
82              Runtime.submitBatch(batch);
83              batch = null;
84          }
85      }
86
87      static final Action rollback = new ReadAction {
88          Object execute(Replica replica, Object ... args) throws SQLException{
89                  ((Connection)replica.getReplica()).rollback();
90          }
91      }
92      // JDBC rollback method;
93      void rollback() throws SQLException {
94          if(readOnly) {
95              Task t = Manager.createTask(rollback, txInitVersion);
96              Manager.syncExecute(t, currReplica);
97              txInitVersion = null;
98          }
99          else {
100             Task t = Manager.createTask(rollback);
101             Manager.syncExecute(t, currReplica);
102             batch = null;
103         }
104     }
105
106     ...
107 }
```

4.3.2.3 Transaction Execution

In this discussion we assume that, prior to the start of a transaction, a client calls the *setReadOnly* method from the JDBC interface. All subsequent queries and/or updates executed after this method and prior to a commit or rollback operation define a transaction. Additionally, we define as read-only transactions that do not execute update operations. All transactions that execute, at least one update operations, be it an UPDATE, INSERT or REMOVE operation, are defined as update transactions. We also assume that the argument value passed to the setReadOnly method is coherent with the transaction type.

Read-only transactions When a read-only transaction starts, MacroDB selects a slave replica in which the transaction will execute, as presented in Listing 4.10 (line 45). After starting the transaction, clients submit queries using the executeQuery of the JDBC interface. All queries are executes on the selected replica, as presented in Listing 4.11 (lines 14-23), using the context of the calling thread, returning the result to the client.

The first query of each new transaction additionally records the current version of the primary replica. This version defines the state of the database for this transaction. This is

necessary to preserve single copy serializable semantics, since, due to runtime overhead, a client may start a read-only transactions on a secondary replicas that has yet to commit a previous update transaction from that same client, thus violating serializable semantics. To this end, the selected secondary replica needs to have, at least, the same version as the one recorded. This is guaranteed by our runtime, since, when executing a Task, it will wait until the selected replica has the same version as the one defined, i.e., is in a correct state.

Note that, at the time of the calling, the replica may have a superior version to the one expected, i. e., the recorded version of the primary was inferior to the one the replica is in (this can occur due to OS scheduling policies). This is not problematic, still guaranteeing single copy serializability, since any modifications that occurred were due to concurrently executed transactions. Note also that subsequent queries do not need to wait for the replica to be on the same state as the primary, since these are concurrent to any update transactions executing on the primary. Only the first query needs to wait for the replica to be up-to-date.

When a client commits a read-only, by calling the *commit* JDBC method, the method is called on the secondary replica using the context of the calling thread. Since no modification occurred to the database, a ReadAction is used to commit the transaction on the secondary replica. This commits the transaction without advancing the replica's version.

Similarly, if the client decides to rollback the transaction, by calling the *rollback* JDBC method, the corresponding method is called on the secondary replica using the context of the calling thread. Again, since rolling back a transaction does not change the state of the database, a ReadAction is used to rollback the transaction without advancing the replica's version.

Listing 4.11: MacroDB Statement Implementation.

```
1  class MacroDBStatement implements java.sql.Statement {
2      // father MacroDB Connection
3      MacroDBConnection father;
4      // database statements
5      Replica<Statement>[] stats;
6      ...
7
8      static final Action executeQuery = new ReadAction {
9          Object execute(Replica replica, Object ... args) throws SQLException{
10             return ((Statement)replica.getReplica()).executeQuery((String)args[0])
11         }
12     }
13     // JDBC executeQuery method;
14     ResultSet executeQuery(String sql) throws SQLException {
15         if(father.txInitVersion == null)
16             father.txInitVersion = Manager.getReplicaVersion(0);
17         Task t;
18         if(father.readOnly)
19             t = Manager.createTask(executeQuery, father.txInitVersion, sql);
20         else
21             t = Manager.createTask(executeQuery, sql);
22         return Manager.syncExecute(t, father.currReplica);
```

```
23      }
24
25      static final Action executeUpdate = new ReadAction {
26          Object execute(Replica replica, Object ... args) throws SQLException{
27              return ((Statement)replica.getReplica()).executeUpdate((String)args[0])
28          }
29      }
30      // JDBC executeUpdate method;
31      int executeUpdate(String sql) throws SQLException {
32          Task t = Manager.createTask(executeUpdate, sql);
33          int res = Manager.syncExecute(t, father.currReplica);
34          father.addToBatch(t);
35          return res;
36      }
37
38      ...
39  }
```

Update transactions Update transaction are treated in a similar manner, only these execute directly on the primary replica. When an update transaction starts, by calling the *setReadOnly* JDBC method, the primary replica is selected for executing the transaction, and a new *batch* object is created. This batch is used for maintaining any update operation, i.e., any SQL update, insert or remove operations, performed in the context of the transaction, for maintaining consistency of the secondary replicas. Additionally, the setReadOnly method is called on the primary replicas, using the context of the calling thread and its associated task is added to the current transaction batch, before returning to the client.

All queries, submitted by a client using the *executeQuery* JDBC method, are executed on the primary replica, using the context of the calling thread, as presented in Listing 4.11 (lines 14-23), and the respective results are returned to the client. Note that, contrarily to read-only transactions, these queries execute directly on the primary replica without waiting for the replica to be in a specific state, since update transactions all execute directly on the primary replica.

Similarly, update operations, submitted by a client using the *executeUpdate* JDBC method, execute on the primary replica, using the context of the calling thread, as presented in Listing 4.11 (lines 31-36). However, and contrarily to queries, before their respective results are returned to the client, its associated task is added to the transaction's batch.

When a client decides to commit an update-transaction, by calling the *commit* JDBC method, the corresponding method is called on the primary replica using the context of the calling thread. Since committing an update transaction modifies the state of the database, a WriteAction is used to commit the transaction on the primary. This commits the transaction and advances the replica's version atomically. Additionally, the corresponding task is added to the transaction batch, and the batch is sent for the runtime to execute in the remaining replicas, i.e., in the secondary replicas.

If, prior to committing it, a client decides to rollback an update-transaction, by calling the *rollback* JDBC method, the corresponding method is called on the primary replicas using the context of the calling thread. Since any modifications, performed in the context of this transaction, have only affected the primary replica, there is no need to rollback the transaction in the secondary replicas, since only successfully committed transactions propagate their actions to the secondaries. Thus, and before returning to the client, the rollback simply discards the batch. Again, since rollback operations do not modify the state of the database, a ReadAction is used to execute the rollback without advancing the replica's version.

4.3.2.4 Concurrent Transactions

MacroDB does not explicitly restrict concurrency in any way, i.e., several clients can interact concurrently with MacroDB. MacroDB delegates the concurrency control mechanism to the underlying database. This is possible since, each client connection maintains associated connections to all replicas, thus all statements execute in their respective contexts.

Also, these connections are shared with the runtime, thus all operations execute under their respective contexts, thus leaving concurrency control to the underlying databases. Additionally, since the manager and runtime force the first operations of a new read-only transaction to wait for any prior transactions to commit on the selected replica, it guarantees that all statements of the new client transaction execute after all previous transactions from that same client have committed on the replicas. Thus, never two operations from different transaction will ever execute concurrently on the same context, i.e., as if they were part of the same transaction.

4.3.2.5 Closing connections

When a client closes a Connection or a Statement, by calling the *close* JDBC method, the corresponding method is called synchronously on the primary replica, using the caller thread, and asynchronously on the secondary replicas, using the runtime.

4.3.2.6 Replica State Consistency

For propagating updates to the secondary replicas, MacroDB uses the, previously described, Macro-Component runtime. The runtime maintains a queue of pending update batches and a set of executor threads, one for each secondary replica. Each thread waits for the next update batch to be inserted into its queue, and executes it on the respective replica.

To preserve consistency, the runtime maintains the queue of update batches ordered by their respective version. Additionally, each executor only applies an update if the replica is up to date, i.e., if the version of the update batch is the immediate successor to the current value of the replica. In the case where the update batch is not the immediate successor, the executor returns the update to the runtime and retrieves a new one, possibly

the immediate successor. This situation can happen since only the commit and increment of the version counter are atomic, not the insertion of the batch in the queue. However, since the runtime maintains only one executor per replica, and the updates are executed accordingly to their version, the runtime guarantees eventual state consistency.

After successfully executing an update batch, the executor atomically commits the transaction and advances the version associated with the replica. Since these updates are performed sequentially, we guarantee that no deadlock will occur on the secondaries when performing updates. This guarantees that all update batches will commit successfully.

MacroDB guarantees single-copy serializable view in different ways:

1. By running all replicas run under the serializable isolation level, we guarantee that update transactions execute in isolation from each other on the master replicas, and that read-only transactions execute in isolation from concurrent updates on the slave replicas;

2. By retrying updates on the slave replicas in case of these abort due to conflicting read-only transactions.

Since the propagation of update on the secondaries can conflict with concurrently executing read-only transactions, leading to possible aborts, the runtime guarantees consistency and isolation by retrying updates whenever these abort. This guarantees both consistency and isolations, since relaunching updates guarantees these will eventually succeed.

Also, possible conflicting read-only transactions that abort in the same way, i.e., due to a conflict with an update, will rollback their actions, and have a different version when they relaunch. Thus, these will read a new version value from the master, and will have to wait for previous updates to be executed before continuing.

In extremis an update batch can execute in exclusion, on a given secondary replica, due to all concurrent read transactions, scheduled for that replica, having started when the version of the master already reflected that update.

4.3.3 Correctness

For the correctness of the system, it is necessary to guarantee that all replicas evolve to the same state after executing the same set of transactions. Also, for guaranteeing that MacroDB provides a single consistent view of the replicated database, it is necessary to guarantee that a transaction is always executed after all update transactions that may precede its commit. This is achieved because the system enforce the following properties.

Proposition 1. *All replicas commit all update transactions in the same, serializable, order.*

Proof. At the primary, as commits execute atomically in isolation, the serializable order is defined by the order of each commit. Since secondary replicas execute update transactions

in a single thread, i.e., sequentially, by the same order, all replicas commit all update transactions in the same order. □

Proposition 2. *A transaction is serialized after all update transactions that precede its commit.*

Proof. For update transactions, this is guaranteed by the database engine at the primary. For read-only transactions, MacroDB enforces this property by delaying the beginning of a transaction until the secondary replica has executed all transactions committed at the moment the begin transaction was called. □

4.3.4 Minimizing Contention for Efficient Execution

As presented earlier, the master-slave replication approach used in MacroDB executes update and read-only transactions in different replicas. Thus, read-only transactions never block update transactions and vice-versa, since these execute in distinct replicas.

Also, the execution of update transactions in secondary replicas may interfere with read-only transactions, depending on the concurrency control scheme used in the underlying database. Both H2 and HSQLDB support multi-version concurrency control that allows read-only transaction to not interfere with update transactions. Since only a single update transaction executes at a time, in secondary replicas, this approach guarantees serializable semantics.

Read-only transactions still need to wait until the secondary replica is up-to-date before starting. Our approach, of executing update transactions as a single batch of updates, minimizes the execution time for these transactions, thus also minimizing waiting time.

Finally, we minimize contention on the database engine by reducing the number of transactions that execute in the same replica at the same time - by executing only a fraction of the read-only transactions in each secondary replica and by executing update transactions quickly in a single database operation.

4.3.5 Evaluation

In this section we evaluate MacroDB performance, by measuring the throughput of the system. Additionally we compare the results from the standalone uncoordinated versions of HSQLDB and H2 with the ones from MacroDB using HSQLDB and H2, referred to as MacroHSQL and MacroH2 respectively. For this comparison we evaluated the performance impact of varying the number of secondary replicas of MacroDB.

4.3.5.1 Prototype Considerations

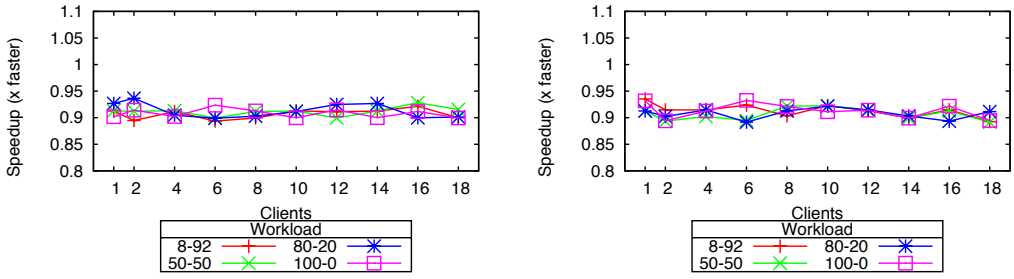
Our MacroDB prototype is built in Java, and includes the necessary runtime system, as well as a custom JDBC driver. By using this simple approach, developers are able to integrate MacroDB into their applications by simply adding its library and modifying the URL used to connect to the database engine, without additional changes to the application

code. The number of replicas and underlying database engines used are defined in the connecting URL. When the first client connects to the database, replicas are instantiated and the runtime system is started.

Our prototype takes advantage of the embedded functionalities offered by both databases. Thus, contrarily to traditional middleware solutions that execute the database engine on a separate process from the middleware, in our approach the middleware shares the same process space as the database engines. This simplifies communication, reducing its overhead, and allows for MacroDB to be practical, even when increasing the number of replicas, as presented in Section 4.3.5.2.

4.3.5.2 TPC-C

Overhead First we wanted to measure the overhead imposed by MacroDB. To this end we compared the throughput of the standalone uncoordinated database engines with MacroDB configured with a single replica, i.e., only the primary replica. Under this configuration MacroDB behaves similarly as the standalone versions, i.e., without distributing load among several replicas, thus allowing us to measure the overhead imposed by our system.



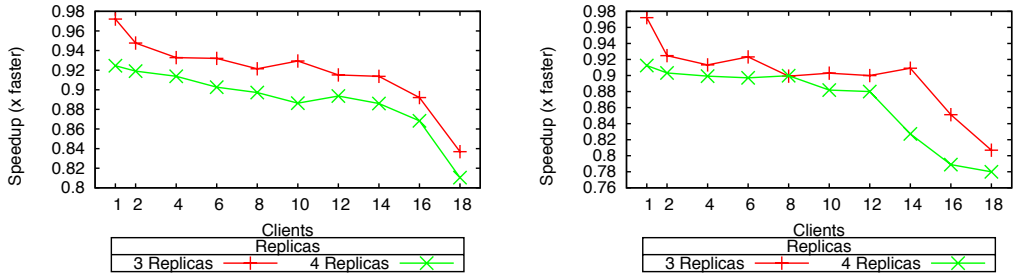
(a) MacroHSQLDB (1 replica) (b) MacroH2 (1 replica)

Figure 4.4: MacroDB overhead results (all TPC-C workload).

The results, presented in Figures 4.4(a) and 4.4(b) for MacroHSQLDB and MacroH2 respectively, show the MacroDB speedup compared to their standalone siblings, for all TPC-C workloads. These results show that MacroDB imposes a, fairly constant, overhead of approximately 10%. Moreover, this overhead remains constant independently of the number of clients and type of workload.

MacroDB speedup Next we measured the speedup of MacroDB, compared to the standalone database engines. To this test we varied the number of replicas, using 3 and 4 replica, i.e., a primary plus 2 and 3 secondary replicas. Again, we used the TPC-C benchmark to measure the speedup of MacroDB, varying the number of clients and type of workload.

8-92 workload Figures 4.5(a) and 4.5(b) present the results obtained running TPC-C with a 8% read and 92% write workload. As expected, under update intensive workloads, our system is unable to offer any performance improvements. This occurs since the majority of transactions (92%) execute on a single replica, the primary, thus MacroDB is unable to benefit from the additional replicas for load balancing.

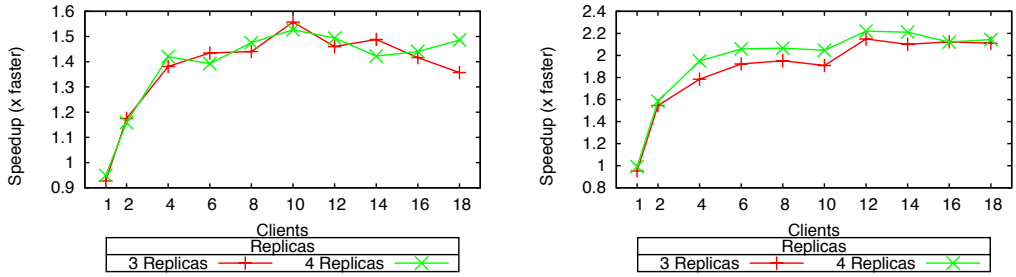


(a) MacroHSQLDB (b) MacroH2

Figure 4.5: TPC-C 8-92 workload results (3 & 4 replicas).

Nonetheless, the results allow us to measure the overhead imposed by the runtime system, since, and contrarily to the previous experiment where all transactions execute on a single replica, some transactions (8%) execute on a secondary, thus need to wait for previous updates to be executed on the selected replica. This overhead increases with the number of clients and the number of replicas, ranging up to approximately 25%. This is expected since the increasing number of clients increases contention due to coordination among concurrent threads accessing MacroDB. Also, increasing number of replicas increases increases the number of queues used by the runtime system for propagating updates, which increases overhead.

50-50 workload Figures 4.6(a) and 4.6(b) present the results obtained running TPC-C with a 50% read and 50% write workload. Under these workloads, MacroDB is able to achieve higher throughput than the standalone versions, offering up to 1.6× and 2.2× the performance of HSQLDB and H2 respectively.

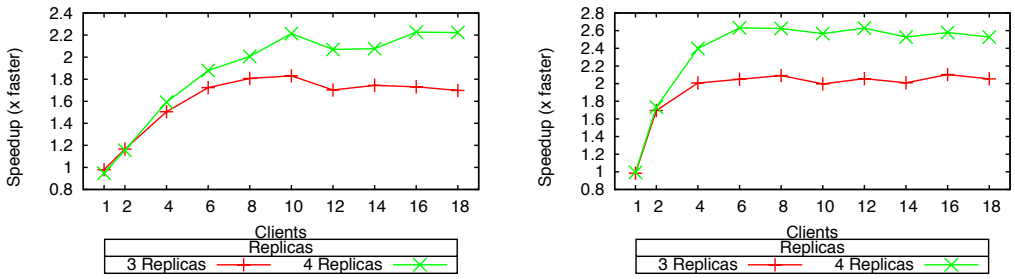


(a) MacroHSQLDB (b) MacroH2

Figure 4.6: TPC-C 50-50 workload results (3 & 4 replicas).

Although MacroDB is able to offer better performance than the standalone engines, its scalability is still limited by the nature of the workload. While secondary replicas are able to balance read-only transactions, the moderate update nature of this workload still imposes great stress at the primary replica, thus limiting scalability. This is put in evidence by the, almost negligible, performance difference when MacroDB is configured with 3 or 4 replicas, i.e., 2 or 3 secondary replicas. Nonetheless, these results show that, even with considerable update rates, MacroDB is able to offer improved performance compared to the standalone engines.

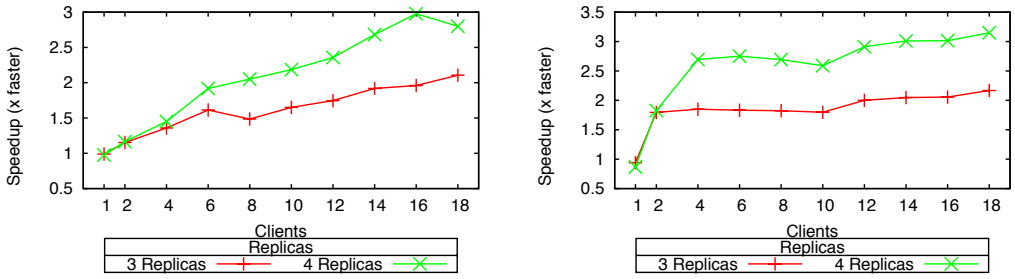
80-20 and 100-0 workloads The nature of read intensive workloads allows MacroDB to take full advantage of its replicated architecture. Figures 4.7(a) and 4.7(b) present the results obtained running TPC-C with a 80% read and 20% write workload. Under these workloads, MacroDB is able to improve throughput of the standalone versions, offering up to 2.2× and 2.6× the performance of the original HSQLDB and H2 engine, respectively, when MacroDB is configured with 4 replicas.



(a) MacroHSQLDB (b) MacroH2

Figure 4.7: TPC-C 80-20 workload results (3 & 4 replicas).

Figures 4.8(a) and 4.8(b) present the results for the read-only workload. Again, MacroDB is able to offer considerable performance increases, outperforming the original engines in up to 2.9× and 3.1× for HSQLDB and H2, respectively.



(a) MacroHSQLDB (b) MacroH2

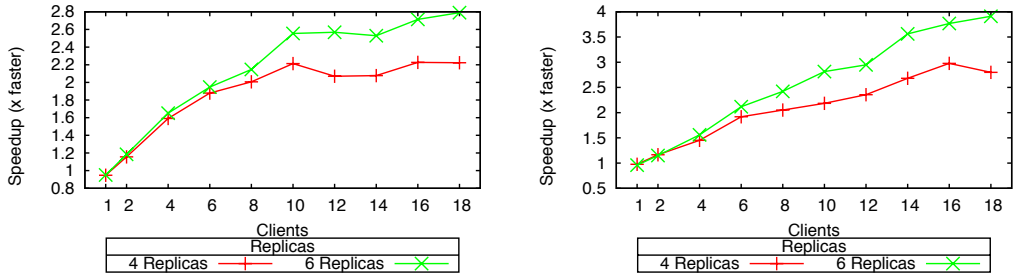
Figure 4.8: TPC-C 100-0 workload results (3 & 4 replicas).

Additionally, under these workloads, the increase in the number of replicas directly

increases performance. This is expected, since the read intensive nature of these workloads allows MacroDB to efficiently distribute load among the additional replicas. Also, and contrarily to the update intensive workloads, these workloads reduce contention on the primary replicas, thus contributing to the scalability of MacroDB.

Additional Replicas To study the performance impact of the number of replicas, we repeated the previous experiment increasing the number of replicas of MacroDB. This experiment was conducted using the HSQLDB engine.

Figures 4.9(a) and 4.9(b) present the results obtained running TPC-C under the 80% read workload and the read-only workload, respectively, using a MacroDB configuration of 6 replicas (1 primary and 5 secondary replicas).



(a) TPC-C 80-20 Workload (b) TPC-C 100-0 Workload

Figure 4.9: MacroHSQLDB with 6 replicas

The results show that MacroDB is able to further improve its performance with the increase in the number of secondary replicas. It improves performance from 2.2× to 2.8×, for the 80% read workloads, while, for read-only workloads, the additional replicas allow MacroDB to achieve a 4× performance improvement, almost 1× more than with 3 secondary replicas.

Again, these results put into evidence how current multicore CPUs are underutilized by current IMDB engines, since MacroDB was able to successfully improve performance, over standalone engines, even when running 6 engines on a single machine.

	Replicas		
MacroDB	2	3	4
H2	1.37×	1.74×	2.13×
HSQL	1.39×	1.79×	2.18×

Table 4.1: Memory overhead

Memory Usage To measure the practicality of our proposal, we measured the memory overhead imposed by MacroDB, over the standalone database engines (Figure 4.1), varying the number of replicas. This was measured using the Java Virtual Machine methods for consulting memory usage.

Contrarily to what may be expected, the memory used by MacroDB is not directly proportional to the number of replicas. This results from MacroDB exploiting the embedded configuration option offered by both HSQLDB and H2. Under such configuration, the different replicas run in the same Java Virtual Machine, thus sharing the same address space. This way, different replicas share immutable objects Java, such as Strings, instead of fully replicating data. Thus, increasing the number of replicas does increase memory usage proportionally.

The obtained results show that MacroDB, when configured with either H2 or HSQLDB replicas, uses at most 1.4× more memory than the standalone engine, when configured with 2 replica, and 2.2× when using 4 replicas. This makes deploying MacroDB practical on a single multicore machine, even with large numbers of replicas.

4.3.5.3 TPC-W

Workload	Throughput(WIPS)	
	H2	MacroDB (3 H2 replicas)
Browsing Mix	261.6	458.4
Shopping Mix	202	428.6

Table 4.2: TPC-W results

As an additional experiment, we compared the results obtained running TPC-W benchmark on a single, uncoordinated, H2 engine and a MacroDB using three H2 replicas (Rep3). The results obtained, presented in Figure 4.2, show the throughput, in web interactions per second (WIPS), obtained running TPC-W browsing mix (95% read transactions) and shopping mix (80% read transactions), on the machine previously described with a database of 2 gigabytes, for 20 minutes and using 128 emulated browsers, with no thinking time. The performance improvements of MacroDB over the standalone version of H2 ranges from 1.75× to 2.12× the performance of the original engine, thus showing the benefits of our system.

4.3.6 Additional discussion

From the presented results, we can conclude that, while MacroDB is able to reduce contention and improve performance over standalone systems, scalability is restricted to read-intensive workloads. This is expected, since update intensive workloads impose high contention on master-slave replication schemes, with all update transactions being directed to the same primary replica.

Additionally, even under read-intensive workloads, MacroDB performance improvements, over standalone versions, is limited by the number of replicas in the system. Again this is expected, since the underlying replicas maintain the original performance limitations, although the replicated nature of the system only exposes these limitations under a higher level of concurrency.

Above all, MacroDB performance is limited by the performance of its replicas, since throughput is a function of the load on a bottleneck entity [Por+15]. In our design the bottleneck is the master in update-heavy workloads or the slaves in read-intensive workloads. Thus, the major performance restriction for MacroDB is the performance of each replica. So, for efficiently addressing the performance and scalability problem of IMDBs on multicores, we should modify the database engine. In the next chapters, we focus on this problem.

4.3.7 Fault Handling

While the previous discussion focused on improved performance, the Macro-Component abstraction can also be used to offer improved fault-tolerance over standard components. When focusing on this goal, we can follow principles similar to diverse component replication [Avi85; AK84; CA78; Gar+11; Gas+07] and byzantine fault-tolerant systems [CL02a; CL02b; Gar+11; Pre+08; Rod+01; Van+07], and apply them at the component level.

Fault-tolerant macro-components can provide fault tolerance by replacing original components by their Macro-Component siblings, where each Macro-Component coordinates a number of diverse replicas with the same interface, running on a single machine multicore system. Operations can perform arbitrary deterministic computations using the component state and operation arguments. Also, applications interact with the original components by invoking operations and blocking for a reply.

Following a state machine replication scheme [Lam78; Sch90], we assume a Byzantine failure model, where a component may fail arbitrarily by crashing or failing to respond, as well as returning erroneous results due to bugs. Also, we assume different implementations of the same specification are available (i.e., component diversity), and that faults can be detected by comparing the results from the different implementations. Finally, we assume component faults do not compromise the computer system integrity and functionality, i.e., do not cause the computer system to fail.

By applying a state machine replication scheme on a single machine multicore systems, we assume replica coordination is not compromised by the applications, since these remain oblivious to the replicated nature of a Macro-Component. Thus, contrarily to the traditional approach that requires additional replicas to guarantee liveness of the replicated system [CL02a], Macro-components offer Byzantine fault-tolerance assuming no more than $(n-1)/2$ replicas fail. Therefore, Macro-Components provide improved fault-tolerance over non-replicated components, by requiring $2f+1$ replicas, where f is the number of faults the system is able to tolerate.

4.3.7.1 Behavior

Contrarily to non-fault-tolerant Macro-components, where all operations execute on a single replica, either on the primary or on a secondary replica (with writes being propagated

asynchronously from the primary replica to the secondary replicas), fault-tolerant Macro-Components execute operations in more replicas. A fault-tolerant Macro-Component detects faults by: *i*) invoking a method in a set of replicas; *ii*) each replica executes the corresponding method concurrently; *iii*) the coordinator waits for, at least, $f + 1$ equal results; *iv*) the result from the majority of replicas is returned to the application.

For reducing overhead, read operations initially execute in only $f + 1$ replicas, executing in additional replicas only if the results diverge. To promote fault manifestation and distribute load, read operations execute in a randomly selected set of replicas. This guarantees that every replica is used for executing read operations.

Whenever inconsistent results are detected, the corresponding replicas are marked as faulty and selected for recovery, being temporarily removed from the set of active replicas. Also, if some replica is unable to produce a result within a certain time limit, the replica is considered faulty. The time limit is defined by the time taken by the majority of replicas to reply, plus an additional tolerance.

4.3.7.2 Fault-Tolerant MacroDB

We have built a fault-tolerant Macro-Component for in-memory databases, called *FT-MacroDB*. FT-MacroDB shares most of the design of MacroDB, adding an additional validation subcomponent, and requiring the number of replicas to be $2f + 1$, where f is the maximun number of replicas that can be faulty.

Like its sibling, it is based on a master-slave replication scheme, with the master maintaining the primary copy of the database and slaves maintaining secondary copies. However, and contrarily to its sibling, in FT-MacroDB each operation executes on a set of replicas with the same state. Thus, all operations are initially executed by the master replica for defining the order in which these must execute in the remaining slaves. The first result, from each statement is returned to the applications, while the remaining results are gathered and compared asynchronously. Query statements are initially executed on $f + 1$ replicas, while update statements execute in all replicas. Queries only execute in additional replicas if results differ.

When a client wants to commit a transaction, if all statements have produced coherent results, the transaction commits successfully, failing otherwise. This approach combines a speculative and asynchronous verification model in a way that is transparent to the application. If the asynchronous verification detects that incorrect results were returned to the application, the transaction aborts.

4.3.7.3 Additional remarks

Our preliminary results have shown that FT-MacroDB is even more dependent on the scalability of each replica.

Contrarily to the performance approach, where only update operations need to execute in all replicas, in a fault-tolerant approach all operations need to execute in, at least,

$f + 1$ replicas. This increases the concurrency in each replica, stressing the scalability problem of a single database. Additionally, the obtained results from these operations need to be gathered, for comparison, before committing each transaction. Both these prerequisites require considerable coordination between replicas, since operations need to execute in the same order in all replicas. This coordination, in turn, results in considerable overhead and runtime contention, which greatly compromises performance and practicality.

Thus, although this was an interesting research path, we ended up not exploring it too much on this work. We have decided to follow the path of addressing the performance limitations of In-Memory Databases on multicore machines.

4.4 Related Work

Database replication techniques have been used to improve database performance and/or availability in many systems [Cec+08; CB09; KA10; Kem+10; Sil+06; WS05; Wie+00]. Availability is achieved by minimizing the system's downtime, whether due to scheduled system maintenance or due to unpredictable faults. Replication improves availability by maintaining additional database replicas that guarantee that the service remains available in the presence of a fault.

Focusing on performance, replication offers improved performance, over non-replicated databases, by distributing load among the available replicas. This increases throughput by minimizing contention between concurrent transactions, as well as resource usage. The former is accomplished by routing transactions to distinct replicas, which allows them to execute concurrently without interference. The latter is achieved since workloads are split among the available replicas, which consequently reduces the total number of transactions executed in each replica. This has the potential to increase throughput, since fewer transaction share the same resources, allowing additional transactions to exploit the remaining resources.

Traditional database replication falls into two categories: *eager* and *lazy* [Gra+96]. Eager replication schemes keep replicas up to date, i.e., synchronized, within the boundaries of each transaction. These typically provide 1-copy serializability [Ber+87], where resulting schedules are equivalent to a serial execution order on a single database. However, this approach imposes considerable overhead, restricting the scalability of these solutions [Gra+96]. On the other hand, lazy replication schemes trade consistency for performance by synchronizing replicas outside the scope of the transaction. Thus, this scheme offers better scalability at the expense of some replicas presenting stale data, since these have yet to apply the latest updates [Gra+96].

Additionally, different replication schemes have been adopted, including single or multi-master replication schemes, as well as different replication architectures, relying solely on internal database feactures or on additional middleware [Kem+10]. We now detail some middleware-based replication solutions that inspired MacroDB.

Sprint In Sprint [Cam+07], the authors propose a middleware solution that orchestrates commodity IMDBs running on a cluster of shared nothing servers. For managing the database, Sprint ensemble is composed of three different components: *edge servers* responsible for managing data partitions and living transactions; *data servers* responsible for maintaining data; and *durability servers* responsible for providing durability (Sprint separates durability property from the database to a specialized component for reducing I/O).

Sprint does not replicate the entire database, instead it partitions the database, replicating the different partitions among several data servers. Each data server may hold replicas of several partitions of the database. Clients interact with edge servers, which modify and forward submitted statements to the corresponding data servers. Sprint needs to process statements since a single statement may need to access different partitions to be executed. All statements from the same transaction are managed by the same edge server.

Sprint offers a single copy serializable view of the database by serializing transactions in the past. Each new transaction is assigned a sequence number, that defines the order in which the transaction should execute. When all statements of the same transaction can be satisfied on a single data server, these execute locally in that data server, and conflict detection is left to the database executing the transaction. However, when statements need to access data from different data servers, it is necessary coordination among the different data servers. Thus, requests are multicast to all data servers, thus defining the transaction's respective execution order. Conflicts are detected by the edge server.

For committing read-only transactions, edge servers send commit messages to the respective data servers and wait for an acknowledgement before committing to the client. For update transactions, Sprint relies on total order multicast relying on the Paxos protocol [Lam98]. The edge server multicasts a prepare commit request to all data servers involved in the transactions. Each data server multicasts its vote to the edge server, all participating data servers and the durability servers. Each data servers wiling to commit the transaction multicasts, together with its vote, the update statements executed by the transaction. If no data server opposes the transaction commits, aborting otherwise. Total order multicast allows durability servers to guarantee both isolation and durability of committed transaction.

Like Sprint, MacroDB also uses a master-slave replication scheme that requires no modifications to the underlying database engines. However, contrarily to Sprint, MacroDB does not partition data. Instead, it replicates the entire database on all replicas. This allows MacroDB to avoid some of the complexity of Sprint, allowing transactions to commit at the primary without coordinating with additional replicas. Also, by batching update statements before executing them on the secondary replicas, MacroDB allows secondary replicas to acquire exclusive locks for shorter periods of time. This is essential for reducing contention, which greatly reduces concurrency in multicore systems.

Pangea Pangea [MN09] is an eager database replication middleware that provides snapshot isolation semantics without requiring modifications to the database servers. It uses a similar approach to the master slave replication scheme, which the authors entitle leaders and followers. Update operations execute initially on the leader (primary replica), being sent to the followers (secondaries replicas) if executed successfully. Read operations execute on a follower.

Pangea defines its isolation semantics as *global snapshot isolation*. Under this isolation semantics, before the first statement of each new transaction, Pangea elects a secondary replica (follower) and defines the snapshot for the transaction. Snapshot definition is done in mutual exclusion, for guaranteeing that no transaction commits during this process. Likewise, commit operations are also executed in mutual exclusion, guaranteeing no snapshot is being defined during its execution.

Although complementary to our work, MacroDB builds on some of the techniques from this system, applying them to multicore systems. Like Pangea, MacroDB also follows a master-slave replication scheme, although using a stronger isolation level, serialziability. Also, MacroDB follows an identical consistency protocol used by Pangea, adapting it to a stronger isolation level. Thus, while Pangea requires coordination at both the start and commit of transactions, MacroDB only requires coordination for the commit for guaranteeing transactions execute in the same order in all replicas. This is only possible due to locking protocol used by its replicas, which guarantees that a transaction that successfully reaches its commit phase it will commit successfully, since it holds all the necessary locks to complete the operation. Also, by executing updates sequentially in the secondary replicas, MacroDB guarantees replica state is consistent without additional coordination, since an update is only performed after all previous updates have successfully committed. Reducing coordination among replicas is crucial for scalable performance on multicores, since contention imposed by such procedures reduces concurrency, greatly penalizing performance.

Ganymed and Multimed Ganymed [PA04], the ancestor of Multimed [Sal+11], is a database replication middleware for scaling databases on clusters of shared nothing machines. It replicates the database using a master-slave replication schemes, where the master replica holds the primary copy of the database, while slave hold secondary replicas. Update transaction execute on the primary replicas, while all read-only transaction execute on the secondaries. Ganymed passes SQL statements to the replicas without modification. It uses a lazy replication scheme, where updates are asynchronously propagated to the secondary replicas. To this end, Ganymed extracts *write sets* from successfully committed transactions in the primary replicas, for propagation to the secondary replicas. For guaranteeing replica consistency, commits are serialized in the master replica, and write sets are propagated using a FIFO queue. Write sets are extracted using row-level insert, delete and update triggers in the master database. Additionally, Ganymed provides SI semantics by using a scheduling algorithm that delays the execution of read-only transactions when

needed, to prevent them from executing on a stale secondary replica.

Similar to its predecessor, Multimed [Sal+11] is a database replication middleware, intended for scaling databases on single machine multicore environments. It uses an identical master-slave model, lazy replication scheme, and update propagation mechanism as Ganymed. Also, it provides a single consistent view of the database offering SI semantics.

MacroDB follows a similar path as Multimed, proposing multicores should be seen as extremely low latency distributed systems (clusters) extended with shared memory [Bau+09; Cec+08; Son+11]. However, MacroDB aims at in-memory databases, which presents different challenges for providing scalability, by incurring in less I/O overhead. Contrarily to these systems, MacroDB offers clients single-copy serializable view of the database, while both offer weaker snapshot isolation semantics. Additionally, while Multimed requires special server side functions and triggers, implemented by the database replicas, for extracting and applying write sets, our approach is completely transparent to both applications and databases. MacroDB guarantees consistency without requiring any modifications to either applications or databases, by simply propagating SQL update statements, from successfully committed transactions, from the master to slave replicas. This allows MacroDB to coordinate any database engine, independently of the manufacturer, as long as these offer a standard JDBC interface. Finally, by considering a multicore system as a distributed system *extended with shared memory*, we explore the shared memory for efficient coordination and data sharing among replicas. As it has been shown in our evaluation, by reusing the same constant data objects in different replicas, data does not grow linearly with the number of replicas in MacroDB.

Chapter 5

Database Modifications

The approach presented in the previous chapter allowed to address some of the scalability problems of database. However, for update-heavy workloads, our solution still does not scale, as all update transactions need to execute in every database replica. In this chapter, we delve into the database engine for improving its scalability by addressing some of the major performance bottlenecks.

As put in evidence by the results presented in Section 3.1, all the isolation levels supported by the studied systems are unable to offer scalable performance. Although some performance variations are observable among the different isolation levels, throughput does not scale with the number of clients.

As discussed earlier (Section 3.1.3), the increase in concurrency, attained from relaxing isolation levels, seems to increase contention on the data structures used by the Storage Manager, which leads to the general lack of scalability showed by both systems. This has been confirmed by the removal of latches used by the index data structures, which allowed the throughput of both modified engines to scale with the number of clients, under TPC-C read-only workloads.

We now discuss how to eliminate latches in database without compromising database consistency, even when accessed concurrently. We then propose a series of modifications to the database for addressing other performance bottlenecks. We start by addressing the scalability problem under read-intensive workloads, and then proceed by addressing the problem for update-intensive workloads.

5.1 Concurrency implications

As presented before (Section 2.1.4.1), databases offer a tradeoff between strict ACID properties and performance. One way in which this is achieved is through the relaxation

of transaction isolation levels, which allows otherwise conflicting transactions to execute concurrently. However, this increase in concurrency may expose applications to, otherwise prevented, concurrency anomalies due to transaction interference.

For lock based concurrency control, all isolation levels acquire exclusive locks, on both items and predicates, before updating them. This prevents write operations from concurrently executing with other operations. However, some isolation levels (independent of being supported by lock based concurrency control) allow read operations to execute concurrently with write operations [Ber+95]. For example, the READ UNCOMMITTED isolation level, which allows transactions to read uncommitted data, when supported by lock based concurrency control, allows read operations to execute without acquiring any read locks (as discussed in Section 2.2), thus allowing read operations to execute concurrently with write operations.

Therefore, read and write operations may concurrently access the Storage Manager. To prevent these operations from compromising the system's health, databases employ additional concurrency control mechanisms for protecting vital data structures against conflicting concurrent accesses.

These additional concurrency control mechanisms are normally referred to as *latches*, as presented in Section 2.1.5. Like most concurrency control mechanisms, latches allow non conflicting operation to execute concurrently, while preventing conflicting ones from doing so. To this end, storage data structures, namely index data structures, use latches for coordinating concurrent accesses, preventing both state corruption and the exposure of inconsistent states, thus guaranteeing data structure integrity under concurrent access. Removing latches may result in state corruption due to concurrent write/write operations, or may expose data inconsistencies under concurrent read/write operations. While the latter can occur when transactions execute under the READ UNCOMMITTED isolation level, write/write conflicts may occur under some granularities of transactional concurrency control, for example, when applied to row level.

Most isolation levels, when supported by table level locking, prevent conflicting operations from concurrently accessing the Storage Manager. Under these conditions, transactions acquire shared or exclusive table locks before reading or updating a table respectively. This prevents read operations from concurrently executing with write operations when accessing the same table, only allowing read operations to execute concurrently with other read operations.

Also, by preventing conflicting operations from executing concurrently, these isolation levels prevent concurrent execution of both write/write and read/write operations at the storage level. This prevents both scenarios that may cause data structure corruption (assuming that data structures are not modified on reads). This way, all isolation levels that acquire both shared or exclusive table locks before each operation, guarantee that only non conflicting operations concurrently access storage data structure, thus guarantee their integrity even if these employ no internal concurrency control mechanism.

As a particular case of a locking scheme, two-phase locking (2PL) divides locking into

2 separate phases: lock acquiring and lock releasing, with the restriction that no new lock is acquired after the release of a lock. It has been shown that using 2PL allows a database system to provide serializability [Ber+95]. Thus, 2PL at the table level prevents concurrent read/write and write/write conflicts from executing at the storage level, while offering serializable transactions.

Summing up, assuming that the databases implement table level locking, which most implementations do, and read locks are acquired before accessing a data item (which is the case for most isolation levels), then there is no need for concurrency control at the storage level. Thus, if all isolation levels are redefined to require acquiring table level locks before any read or update operation, this would guarantee that *only* read operations execute concurrently with each other in the same table, while update operations execute in mutual exclusion.

Using table level locks instead of lower-granularity ones has the advantage of simplifying the management of locks, with no need to manage predicate locks.

5.1.1 Removing storage latches

As a consequence of the previous observations, we propose a modification to both the Storage Manager and Lock Manager of general purpose IMDBs. Specifically, we propose Lock Managers to provide concurrency control using table level locking, and provide the ANSI SQL [ISDLS92] isolation levels by acquiring locks as described in Table 5.1.

Isolation Level	**Read Lock Duration**		**Write Lock Duration**	
	Item	Predicate	Item	Predicate
Read Uncommitted	Short	Short	Long	Long
Read Committed	Short	Short	Long	Long
Repeatable Read	Long	Short	Long	Long
Serializable	Long	Long	Long	Long

Table 5.1: Modified Locking behavior and Isolation Levels

These differs from the traditional implementation (presented in Section 2.1.4.1) by requiring the READ UNCOMMITTED isolation level to acquired short duration shared locks and short duration exclusive locks.

Additionally, we propose modifying the Storage Manager by removing all latches used by the storage data structures. Since the Lock Manager guarantees only non conflicting operations access the Storage Manager concurrently, data structures are never concurrently accessed by conflicting operations, thus their integrity and state consistency is preserved even when removing latches.

Although this are simple modifications, the achieved performance improvements are considerable, allowing IMDBs to scale on multicores, as put into evidence by the obtained evaluation results (Section 5.1.3).

5.1.2 Additional Remarks

It is important to note that the traditional implementation of the ANSI SQL isolation levels, presented in Section 2.1.4.1, guarantees isolation semantics even if different isolation levels coexist concurrently, i.e., even if concurrent transactions run under different isolation levels.

This is possible since all isolation levels acquire long duration exclusive locks before write operations, even READ UNCOMMITTED.

This guarantees that transactions running under other isolation levels can only read values written by a READ UNCOMMITTED transaction after it commits. If all transactions were running under READ UNCOMMITTED mode, it would not be necessary to have long duration locks for writes, as a written value could immediately be read by a concurrent READ UNCOMMITTED transaction.

To allow the use of short duration exclusive locks, we propose the following approach. The database keeps track of the maximum isolation level for running transactions. If this maximum isolation level is READ UNCOMMITTED, a transaction acquires only short duration write locks. Otherwise, it acquires long duration write locks. Additionally, when a transaction with isolation level stronger that READ UNCOMMITTED starts, the database automatically acquires long duration locks for the tables modified by on-going transactions running under READ UNCOMMITTED.

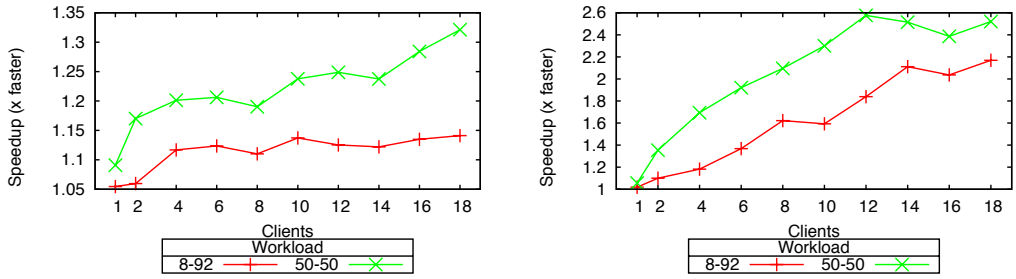
5.1.3 Evaluation

In this section we evaluate the proposed modifications on both HSQLDB and H2, named *HSQLDB (LF)* and *H2 (LF)* respectively, and compare them to their unmodified siblings, by measuring throughput gains (speedup) and scalability.

5.1.3.1 8-92 Workload

Under the standard workload, our proposed modified engines offers limited performance benefits over their unmodified siblings, as presented in Figures 5.1(a) and 5.1(b) for HSQLDB and H2 respectively. This speedup is about 1.14× and 2× when compared to the unmodified HSQLDB and H2 engines respectively. Also, its performance is unable to scale.

This is an expected situation, since all TPC-C transactions interfere, thus restricting concurrency. Under this workload, two out of the five transaction defined by TPC-C account for 80% of the workload. Also, since these transactions access mostly the same tables this creates a considerable bottleneck for the system, which is reflected by the presented results. Also, as discussed previously (Section 3.2.1), table level locking greatly restricts concurrency under highly conflicting workloads.



(a) Latch Free HSQLDB speedup over HSQLDB (b) Latch Free H2 speedup over H2

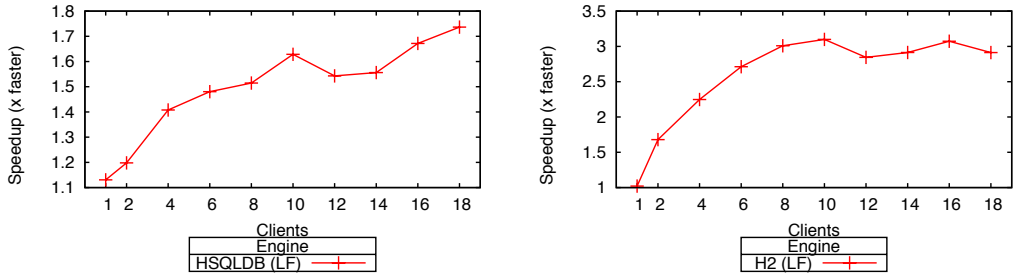
Figure 5.1: TPC-C performance under 8-92 and 50-50 workloads.

5.1.3.2 50-50 Workload

In the 50-50 workload, where the number of read-only and update transactions is the same, our modified versions of both HSQLDB and H2 are able to achieve better throughput, with 1.3× and 2.5× speedup over the unmodified HSQLDB and H2 database engines respectively. Like before, the conflicting nature of TPC-C and the concurrency restrictions imposed by table level locking, have a considerable impact on scalability on both modified engines.

5.1.3.3 80-20 and 100-0 Workloads

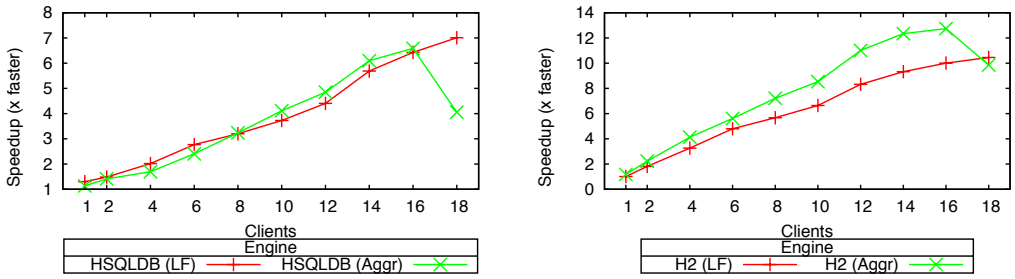
The nature of read intensive workloads is favorable to the proposed modifications, with both modified versions achieving higher throughput than their unmodified siblings. The modifications offer an increase in performance of approximately 1.8× and 3×, over the unmodified HSQLDB and H2 engines respectively, for the 80-20 workload (Figures 5.2(a) and 5.2(b)).



(a) Latch Free HSQLDB speedup over HSQLDB (b) Latch Free H2 speedup over H2

Figure 5.2: TPC-C performance under 80-20 workloads.

Under read-only workloads, our modified engines achieve an approximately 7× and 10× throughput increase over the unmodified HSQLDB and H2 engines respectively, as presented in Figures 5.3(a) and 5.3(b).



(a) Latch Free HSQLDB speedup over HSQLDB

(b) Latch Free H2 speedup over H2

Figure 5.3: TPC-C performance under read-only workloads.

The results show that under read intensive workloads both modified engines scale on multicore machines. Above all, for read-only workloads, we can see that the obtained results, for both modified engines, are fairly close to the expected maximum achievable under these workloads (as defined in Section 3.2.3), presented as *HSQLDB (Aggr)* and *H2 (Aggr)* in Figures 5.3(a) and 5.3(b), respectively.

In fact, the performance of the modified versions surpasses the corresponding aggregated value when the number of clients exceeds the number of cores. This results from the lack of resources, mainly main memory, in the aggregate experiment, which requires flushing caches when a new process is scheduled, and from the use of virtual memory.

Additionally, these workloads allow the modified engines to scale near linearly with the number of cores, thus taking advantage of the computational resources offered by current multicore systems.

5.1.4 Additional discussion

The presented results allow us to conclude that the impact from storage contention, on database performance, is considerable for read-dominant workloads, since under these conditions our proposed modification achieved its largest speedup compared to the original engines.

Under update intensive workloads, the proposed modification, while offering some performance improvements over the original engines, still presents scalability problems. This is specially true under write predominant workloads, such as STD and 50-50. This lack of scalability results from reduced concurrency due to high transaction interference. Thus, one can conclude that, under write dominant workloads, the transaction manager becomes a considerable performance bottleneck, by locking entire tables.

Next we discuss how to address the limitation imposed by the transactional manager, for improving throughput under update predominant workloads, while still avoiding index structures from using concurrency control mechanisms.

5.2 Scaling update-intensive workloads

The previous results show that removing storage latches allows our modified engines to scale on read-intensive workloads, such as 80-20 and 100-0. On the other hand, update-intensive workloads do not benefit in the same way, since contention created by the Lock Manager has a considerable impact on performance.

Thus, the limiting factor for scalability under update-intensive workloads is the enforcement of transactional semantics. More specifically, the contention created due to table level locking, which, for update operations, prevents concurrent transactions from operating on the same tables, even when no *real* conflicts exist.

5.2.1 Increasing write concurrency

Serializable isolation semantics states that: concurrent execution of two or more transactions is serializable if the outcome result is equivalent to the one obtained from executing the same transactions in some sequentially order [Ady+00; Ber+95; Pap79].

While 2PL at the table level achieves this, it is considerable restrictive in terms of concurrency, since whenever a transaction updates a table, it does so in exclusion, preventing every other operation from concurrently accessing the same table.

Although one may argue that table level locking restricts concurrency too much, the alternatives requires a complex mix of row level locking and predicate locking for addressing SQL statements accessing data based on conditions [Ber+95]. Not only the problem of testing predicate locks has been shown to be NP-complete [HR79] (and complex to implement), but also, enforcing concurrency control at lower granularities, such as row-level, allows read and write operations to execute concurrently at the storage level. Thus, this requires index data structures to use latches for preventing state corruption under these scenarios, since rows can be inserted (or deleted) into a table while concurrent transactions read from the same table. As already presented, storage latches have a considerable impact on database performance [SP15; Sto+07].

In this section, we show how to increase transaction concurrency, while enforcing strong isolation semantics, i.e., serializability, and without having to use latches at the storage level.

5.2.1.1 Concurrency without latches

As discussed before, storage latches may be removed as long as the Lock Manager prevents write operations from executing concurrently with read operations at the storage level. Table level locking guarantees this since write operations (including SQL UPDATE, INSERT and DELETE) execute in exclusion. Thus, table level locks prevent the occurrence of concurrent structural changes to the index structure, which may result from write operations.

On the other hand, reducing locking granularity from table to row level, may allow write operations to execute concurrently with every other operation (i.e., both reads and writes). Thus, index structures are required to use latches for preventing the execution of concurrent operations with any structural changes that may result due to a write operation.

However, one may allow some write operations to execute concurrently with every other operation (i.e., both reads and writes), without requiring index structures to use latches, as long as their execution does not compromise the integrity of the index structure, i.e., as long as structural changes execute in isolation.

For preserving serializability, read and write operations may execute concurrently as long as these do not modify values previously read or written by any concurrent transaction. Although, table and row level locking guarantee this, one may further reduce locking granularity, and allow concurrent read and write operations to execute on the same table row, as long as they access distinct attributes.

To exemplify, consider table *A*, with attributes *(a,b,c)*. Consider transaction *T1* that modifies attribute *b*, referred as *A.b*, using attribute *a* for selecting the row (e.g. SQL statement *UPDATE A SET b = b + 1 WHERE a = 10;*) and transaction *T2* that modifies attribute *A.c*, also using attribute *a* for selecting the row (e.g. SQL statement *UPDATE A SET c = c - 1 WHERE a = 10;*). These transactions do not interfere if *T1* does not access *A.c* and *T2* does not access *A.b*. Additionally, both operations may execute concurrently at the storage level, as long as their execution does not perform any structural changes to the data structures used to store table A and associated indices. Thus, both transactions can execute concurrently and be serialized in any order.

5.2.2 Proposed algorithm

To increase concurrency among update transactions, we propose a concurrency control protocol based on *attribute level locking (A2L)*. Under A2L, operations lock table attributes, i.e., (table, column) pairs, instead of locking the complete table.

A2L differs from table level locking due to the granularity of the locks, i.e., by locking attributes instead of the entire table. Operations lock different attributes in different ways, with the following locks being obtained for each SQL operation:

- *SELECT* operations acquire shared locks on the read attributes. These include any attributes used by WHERE clauses;
- *UPDATE* operations acquire exclusive locks on the modified attributes and shared locks on the read attributes (including attributes used in WHERE clauses); and
- *INSERT* and *DELETE* operations acquire exclusive locks for all attributes of the table, as these operations lead to structural changes in the underlying data structures, and cannot proceed concurrently with any other operation.

Whenever update operations update an index key, these are treated as delete followed by an insert operations. Lock acquisition, under A2L, is performed using the usual two-phase locking approach, with a first phase where new locks are acquired followed by a second phase where locks are released. Additionally, A2L is also compatible with standard ANSI SQL isolation levels. Like in traditional implementations, the different isolation levels are enforced by acquiring read and write locks using the rules defined previously, with the difference that locks are obtained at the attribute level instead of the full table level.

To illustrate how A2L works, in the context of the previous example, transaction *T1* obtains a exclusive locks on pair *(A,b)* and a shared lock on pair *(A,a)*. Transaction *T2* obtains a exclusive locks on pair *(A,c)* and a shared lock on pair *(A,a)*. Since no write dependency exists between the two transactions, as they write different table attributes, both transactions can acquire the needed locks and proceed concurrently. This can occur even if both transactions access the same row.

5.2.3 Improving A2L

While the base algorithm achieves its purpose, it imposes a considerable overhead for INSERT and DELETE operations, since these operations have to acquire locks on all table attributes. Thus, for reducing this overhead, we have improved the base algorithm, so that these operations require only a single lock.

Under the modified A2L algorithm, operations lock a combination of tables and attributes in different ways, with the following locks being obtained for each SQL operation:

- *SELECT* operations acquire shared locks on the respective tables as well as on the corresponding read attributes. These include any attributes used by WHERE clauses;

- *UPDATE* operations acquire shared locks on the respective table and on the corresponding read attributes (including attributes used in WHERE clauses), and exclusive locks on the modified attributes; and

- *INSERT* and *DELETE* operations acquire exclusive locks on the target tables.

These modifications reduce the number of locks acquired by INSERT and DELETE operations, while still guaranteeing their execution in exclusion, since these operations lead to structural changes in the underlying data structures, and cannot proceed concurrently with any other operation. Although, this reduction comes at the cost of an additional lock for other operations (SELECT and UPDATE), it reduces the complexity for computing dependencies among operations used for deadlock detection algorithm. Contrarily to the base approach, INSERT and DELETE operations only depend on a table locks, and not on every attribute.

5.2.4 Correctness

For the correctness of A2L it is necessary to guarantee that transactions executed under A2L respect serializability semantics, i.e., that the result of concurrently executing two or more transactions under A2L is identical to some sequential order of those same transactions.

Proposition 3. *A2L algorithm enforces serializability of concurrent transactions.*

Proof. It is known that two-phase locking (including predicate locks) guarantees serializability [Ady+00; Ber+95]. Thus, as our approach uses two-phase locking, we only need to show that our locking scheme does not allow read/write or write/write conflicts on any data item, including read/write conflicts between rows defined by a condition. In A2L, read and write operations need to obtain shared locks on attributes used to select rows. As write operations must obtain an exclusive lock before modifying the value of an attribute in any row, it follows that two transactions cannot have a read/write conflict for any attribute. Also, as transactions obtain exclusive locks in the attributes modified, which for insert and delete include all attributes, there is also no write/write conflict. Thus, A2L algorithm enforces serializability.

□

For proving the correctness of our implementation, we further need to prove that A2L can run with a modified engine that includes no concurrency control on the data structures used for maintaining data.

Proposition 4. *A2L preserves consistency with data structures that include no concurrency control.*

Proof. A2L locking scheme acquires shared or exclusive lock on an item before accessing it, thus preventing any concurrent operation from acquiring an exclusive lock on the same attribute. This prevents concurrent read/write and write/write operations from execution on the same data attribute. Additionally, A2L exclusive locks *all* attributes of a table before executing *inserting* or *deleting* an item. This prevents a concurrent read operation from acquiring shared locks on any attribute, thus preventing transversing the data structures while structural changes occur.

□

In the next section we discuss how we address the cases where *update* operations involve structural changes to the storage data structures.

5.2.5 Implementing A2L

In this section we discuss the requirements for supporting A2L, and describe a prototype based on a modified version of HSQLDB [Gro12].

5.2.5.1 Transactional Component

Most general purpose database transaction managers guarantee transaction isolation by acquiring shared table locks before executing read operations, and exclusive locks before performing write operations. The latter include SQL INSERT, DELETE and UPDATE operations.

HSQLDB implements shared locks using a multi-value map, that associates tables with the sessions accessing it, while exclusive locks are implemented as a single-value map, associating tables with a single session. Read operations are only allowed to execute if the corresponding table is not exclusively locked, i.e., contained in the exclusive lock map, while write operations can only execute if the corresponding tables are not locked exclusively or shared, i.e., are not contained in either the shared or exclusive lock maps.

A2L differs from table level locking by the granularity of the locks. Thus, we have extended the HSQLDB transactional manager to map both table and (table, attribute) pairs with sessions, instead of the traditional approach that maps only tables with sessions. All transaction coordination and deadlock detection mechanisms have been extended to consider this modification.

Additionally, operations lock attributes in different ways, as explained before (Section 5.2.2). To this end, we extract attribute information from all data manipulation language (DML) and data query language (DQL) statements, where attributes used in 'WHERE' clauses are mapped as read attributes. INSERT and DELETE operations lock the corresponding tables, for preserving isolation semantics.

Beside the modifications to the transaction component, A2L also requires modifications to both storage and index components, as described next.

5.2.5.2 Storage Component

Both studied general purpose memory database engines implement, at the storage level, row update operations as a remove followed by an insert row operation, where the newly inserted row embodies the update modifications. This is due to the fact that tables have, at least one, index data structure to improve efficiency for search operations. Since these indices use the values of an attribute, or set of attributes, as search keys, any modification to an index key, without the corresponding structural modification, may corrupt the index state.

For supporting A2L, we require database storage engines to only implement update operations as remove/insert pairs when modifying index key attributes. This way, updates on attributes not associated with an index key, should update the row directly, while updates that modify an index key, should be treated in the traditional way, i.e., as a remove followed by an insert operation.

To this end, we have modified HSQLDB storage component by adding support for directly updating attribute values on an index node, i.e., a row object, when executing

an update. This method is used whenever the attribute being modified is not a key of an index.

Additionally, for providing atomicity, all HSQLDB DML statements perform direct in-place updates, while registering *action* objects that revert the performed operations for recovering state in case of aborts or rollbacks. In case of a commit, all registered actions are simply discarded and acquired locks are released, with no further action needed. In case of a rollback, all registered actions are executed before releasing locks, thus reverting previously made changes.

Insert and delete actions were left unchanged, while a new update action was added for reverting update operations. Session objects were further modified for supporting this modification. At commit, these values are discarded, while aborts restore previous values to the corresponding row. Since only the modified attribute values are restored, any concurrently operations are not affected by these changes.

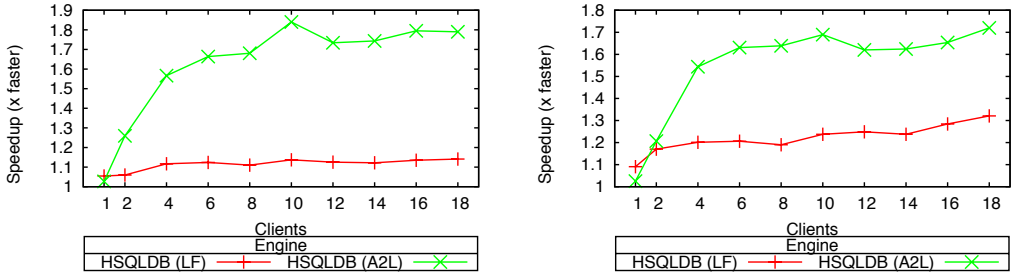
5.2.6 Evaluation

In this section we present the evaluation of A2L when combined with our modified storage engine, by comparing its throughput and scalability against the original HSQLDB, as well as the modified version of HSQLDB with no latches in the data structures, as described in Section 5.1 (HSQLDB-LF) and a version including A2L, as described in Section 5.2 (HSQLDB-A2L)

Although we previously described a way to reduce the number of locks acquired by A2L (Section 5.2.3), the evaluation results did not reflect significative performance differences between the two.

5.2.6.1 Update-intensive Workloads (8-92 and 50-50)

Under update-intensive workloads, A2L is able to offer performance improvements over the original HSQLDB. Figures 5.4(a) and 5.4(b) show the speedup achieved by the A2L modification, over the original HSQLDB engine, for update intensive workloads (8-92 and 50-50 respectively).



(a) 8-92 workload

(b) 50-50 workload

Figure 5.4: HSQLDB A2L modification under update-intensive workloads.

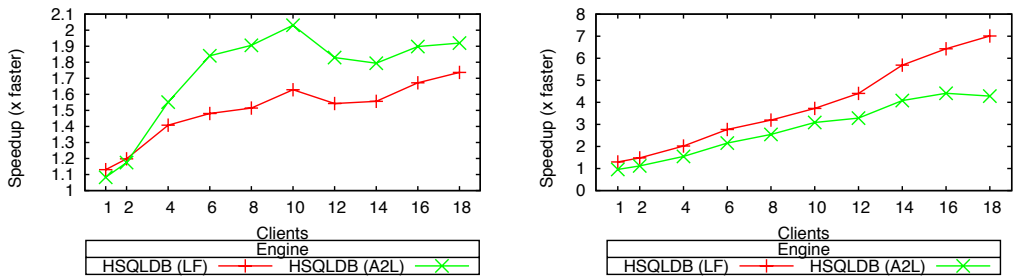
Under these workloads, A2L is able to offers a 1.8× performance improvement over the unmodified database engine. This shows that A2L is able to efficiently reduce transaction conflicts, thus increasing performance.

However, while it improves performance, A2L is still unable to offer scalable performance under these workloads. This comes from the fact that, while A2L reduces conflicts between different transactions, since these transactions access different attributes in different ways (even when accessing the same tables), update transactions end up conflicting with themselfs. Since TPC-C defines only five different transactions, of which two are read-only, due to the update intensive nature of these workloads the probability of two different clients executing the same update transaction increases quickly with the number of clients. Thus, performance scalability under these workloads is considerably penalized, as put into evidence by these results.

Furthermore, SQL INSERT and DELETE operations considerably restrict concurrency since, under A2L, these operations require locking the entire table. Thus, when a transaction includes such operations, A2L suffers from the same problems as table level locking. As future work, we intend to study a way to allow these operations to execute concurrently at the storage level, while offering serializable semantics and without requiring the use of latches.

5.2.6.2 Read-Intensive Workloads (80-20 and 100-0)

Like most concurrency control mechanisms, read-intensive workloads allow A2L to offer increased performance improvements over the original HSQLDB engine. Figures 5.5(a) and 5.5(b) show the speedup achieved by the A2L modification, over the original HSQLDB engine, for the 80-20 and 100-0 workloads, respectively.



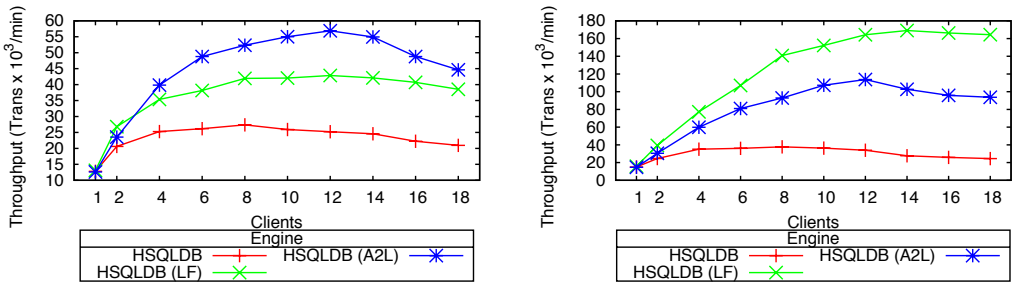
(a) 80-20 workload

(b) 100-0 workload

Figure 5.5: HSQLDB A2L modification under read-intensive workloads.

Under these workloads, A2L is able to offer, at most, 2× and 4× performance improvements over the unmodified database engine, for the 80-20 and 100-0 workloads respectively. Again, this shows that A2L is able to efficiently reduce transaction conflicts, which results in a performance improvement over traditional locking schemes.

However, A2L increases locking overhead compared to table level locking, since some operations acquire a higher number of locks (equal to the number of accessed attributes). This increase in locking overhead limits the scalability of A2L under read-intensive and read-only workloads, as put into evidence by the results from the 80-20 and 100-0 workloads presented in Figures 5.6(a) and 5.6(b), respectively. These results show a considerable drop in performance when concurrency surpasses 10 clients.



(a) 80-20 workload

(b) 100-0 workload

Figure 5.6: A2L throughput compared to table level locking (80-20 and 100-0 workloads).

Also, while A2L is able to offers a 4× performance improvement over HSQLDB (Figure 5.5(b)), it suffers a performance loss when compared to the traditional 2PL scheme under read-only workloads, as put into evidence by the result in Figures 5.5(b) and 5.6(b). The reason for this loss of performance is due to the transactional locking and deadlock detection mechanisms of A2L being more complex, since A2L typically acquires a higher number of locks. Since both these tasks need to be coordinated among concurrent threads, i.e., execute in mutual exclusion, the increase in the number of operations increases contention, thus reducing concurrency. However, we believe these limitations could be addressed by further modifying the database engine, using some of the ideas presented in recent works for increasing concurrency for locking algorithms [Jun+13; Ren+12; Tu+13].

Nevertheless, A2L offers significative performance and scalability improvements over the original HSQLDB under different workloads natures, showing database engines can offer scalable performance under strong isolation semantics.

5.2.7 Additional discussion

From the presented results we can conclude that A2L contributes positively for improving database performance and scalability, especially under update intensive workloads. However, this improvements comes at a cost for read intensive workloads, with the results showing some reduction in throughput, when compared to the traditional 2PL at the table level.

The limiting factor for improvements, under A2L, results from interference between transactions of the same type. TPC-C is composed of five different transactions, with only 3 different update transactions. This results in high contention between transaction of the

same types in update heavy workloads. Nonetheless, A2L is still able to offer performance improvements under these conditions.

Also, while A2L is able to efficiently increase concurrency for SQL UPDATE operations, INSERT and DELETE behave similarly to table level locking, preventing every other operation from executing. This is an important point that we intend to address in the future.

Finally, these results put into evidence the performance tradeoffs one must take into consideration when modifying a database engine. In the next chapter, we present a study on the impact the order of operation issued by applications have on database performance.

5.3 Related Work

With the introduction of multicore processors and the evolution of in-memory databases, several works have focused on the storage component of these systems, namely index data structures. Early works focused on the reduction of memory footprint of traditional data structures, making them cache-conscious. Among these works is the proposal of T-Trees [LC86], that reduces the number of pointers used by traditional AVL-Trees [VDG74]. CSB+-Tree [RR99; RR00] applies the same principals to the B+-Tree, reducing the number of used pointers. These works are complementary to ours, and the techniques proposed could be used in our work for further improving performance.

Additional works, like Masstree [Mao+12], Foster B-Trees [Gra+12], CTrie [Pro+13] or Kiss-Tree [Kis+12] propose multicore-conscious data structures, that reduce synchronization costs. While Masstree and Foster B-Trees reduce synchronization costs using fine grain latches, allowing concurrent updates in distinct parts of the data structures, CTrie and Kiss-Tree follow a latch-free approach, where latches are removed in favor of atomic compare-and-swap instructions.

These works focus on designing and implementing efficient data structures for multicores. While we do not propose new data structures for in-memory databases, we follow a similar path, by proposing the removal of latches from the data structures. In contrast to these works, our work proposes a transactional-conscious storage management strategy, where the storage system and transaction concurrency control are designed jointly. For instance, as put into evidence in our approach, the use of pessimistic concurrency control, allows the removal of latches in the index structures, without requiring them to be replaced by compare-and-swap operations.

Our solution uses a new locking scheme that increases concurrency without requiring index structures to use concurrency control mechanisms. Although *attribute level locking* (A2L) model was proposed in the past [MAH08], to our knowledge there was no practical application of it. Also, and to the best of our knowledge, this is the first application of A2L in the context of in-memory databases. We show that A2L increases concurrency compared to table level locking, and that it can be combined with latch-free index structures.

We now detail additional systems that inspired our modifications.

Hekaton Hekaton [Dia+13], a main-memory storage engine for SQL Server [Mic15], also uses latch-free implementations of B-Trees as indexes, called Bw-Tree [Lom+13]. Bw-Trees trade latches for compare-and-swap instructions, allowing the data structure to be transversed while being concurrently modified. Like traditional B-Trees, these are used to index disk pages, instead of maintaining the data directly on the index [Lev+14]. Thus, in Hekaton, an additional cache layer is used, similar to the traditional buffer management, that caches database data. To avoid waiting for disk pages, Hekaton caches the entire database when possible, i.e., caches all pages. Mapping between logical and physical pages is managed by the cache layer using a *mapping table*. To prevent the use of latches for managing cache, write operations do not modify pages directly. Instead, write operations append modification *deltas* to the corresponding page. These deltas represent required page modifications due to updates, inserts, removes, page split or join operations.

Unlike our work, Hekaton trades latches for compare-and-swap operations, thus dealing with concurrent operations. Although compare-and-swap operations are more efficient than latches, studies shows that, like their counterparts, their still performance degrades with increasing levels of concurrency [Dic+13]. We follow a more extreme approach of completely removing any synchronization from the data structures and keeping all control of concurrency in the Lock Manager. This approach allows simpler implementations while providing scalable performance.

Doppel Doppel [Nar+14] is a shared-everything in-memory database based on Silo [Tu+13]. Applications interact with the database in the form of stored procedure, where single-shot transactions execute to completion without client interaction. The authors propose a concurrency control mechanisms called *phased reconciliation*, that reduces contention by "creating" on-demand replicas, that allow specific write operations to execute concurrently. Dopple achieves this by cycling the database between tree different states: joined; split, and reconciliation. Under the joined phase, the database uses traditional optimistic concurrency control. When conflicts are detected, i.e., when concurrently trying to modify the same value, the database changes to a split phase, where each core aplies modifications to a "private copy" of the database. All modifications performed under this phase result in logical changes, i.e., the database guarantees the corresponding operation will execute without conflict. When a result from these changes needs to be read, the database changes to the reconciliation phase in which the final (physical) values are calculated. Dopple provides this functionality for a small number of operations, including calculating the maximun and minimum between two values, additions, ordered insert, etc.

Although Doppel can scale very well for some workloads, where transactions can update the database without having to read its value or using commutative operations, it

is not a solution for general transactions. We could leverage a similar idea to allow updates of the same attributes to proceed concurrently. This could be achieved by postponing update operations until a transaction's commit phase. If updated values are not read by the respective transaction, these can be delayed until the transaction tries to commit. This reduces the duration for exclusive locks, thus reducing contention and increasing concurrency.

Chapter 6

Application impact on database performance

As presented so far, the type of workload affects the performance of databases. This is a direct result from the inherent degree of interference, since update-intensive workloads interfere more than read-intensive ones, since read operations are allowed to execute concurrently with other read operations, while update operations need to execute in exclusion. Additionally, our experiments have shown that addressing performance for a specific workload may not benefit different workloads the same way.

Besides this, most applications tend to access databases using well defined database access patterns, where transactions are pre-defined in the application code, and operations inside each transaction are known *a priori*. Thus, there is some freedom to rewrite operations inside a transaction in different orders, without compromising the application behavior, as long as transaction semantics is preserved.

Although irrelevant to the application, the order of operations inside a transaction can have impact on performance, as the order of operations dictates how long locks will be held by a transaction. If a transaction starts with an update operation, it will hold an exclusive lock, which will prevent any concurrent operations from executing, during the duration of this transaction. On the other hand, acquiring a shared lock in the beginning of a transaction still allows other read operations to execute concurrently during the duration of a transaction. Thus, if update operations (including SQL UPDATE, DELETE and INSERT) are postponed until the end of the transaction, exclusive lock will be acquired for smaller periods of time, which reduces contention and increases concurrency, thus potentially improving database performance.

In this chapter we discuss how to rewrite transactions using a *Read before Write* (RbW) pattern, where *all read operations* of a transaction are executed prior to *all write operations*, whenever possible. Also, we present a study on the impact applications have on database performance. This study analyzes the transactions defined in the TPC-C benchmark,

identifying their access pattern, and studies how the RbW pattern influences the database performance. We study how to take advantage of the knowledge of database transactions, and their respective operations, for reducing database contention and further improving system performance.

6.1 Reordering Operations Inside Transactions

The process of reordering operations inside a transaction needs to guarantee that the original transaction semantics is preserved. In this discussion, we will describe a process for reordering read and write operations inside a transaction that guarantees the transaction semantics. In this discussion, we assume that two transactions are semantically equivalent if they modify the same set of attributes with the same values.

Although this discussion tries to be as exhaustive as possible, covering as many read-/write combinations as possible, some situations may not be covered due to their application related semantics. Also, while the following examples could be applied in most applications, some queries may not be reordered without compromising the transaction semantics, due to specific application related semantics.

As an additional note, in this discussion we focus exclusively in the reordering between read and write operations pairs, as the goal is to move read operation to the beginning of transactions. We also assume that the relative order between write operations is preserved during the transformation process, since this may influence the semantics of the transaction. Finally, we make no assumptions on the automation of this process, since operation reordering is largely related to the application semantics.

To this end, we will focus on the reordering of following operation pairs: UPDATE/SELECT; INSERT/SELECT, and DELETE/SELECT. For each pair, we will discuss how to reorder them if they operate over a *single row* or a *range of rows*, contemplating if dependencies exist or not between the operations, i.e., if write operations modify (or not) values later read. We will consider UPDATE, INSERT and DELETE operations as WRITE operations whenever the same solution can be applied to all, and address each separately otherwise.

6.1.1 Non-Dependent operations

We start with the case of reordering operations without dependencies, i.e., modified attributes (due to UPDATE, INSERT or REMOVE operations) are not read by any subsequent SELECT operation inside the same transaction.

When no dependencies exist between a WRITE and a SELECT operations, then the read set of the SELECT operations will be the same independently of executing before or after any WRITE operation. Thus, these operations are commutative and their order can be changed without compromising transaction semantics.

To illustrate this scenario, consider the example presented in Listing 6.1, where an UPDATE increments the value of attribute *x* for a given table row (row whose *id* is equal to *zero*) and a SELECT operation read the value of attribute *x* of a different row of the same table. Since the UPDATE does not modify the row returned by the SELECT operation, their relative order is irrelevant to the semantics of the transaction.

Listing 6.1: Single row UPDATE/SELECT, without dependencies.

```
1 UPDATE table_a SET x = x + 1 WHERE id = 0;
2 SELECT x FROM table_a WHERE id = 2;
3 ...
```

Thus, the two operation can be reordered without consequences, as presented in Listing 6.2 This can be applied to other WRITE operations (INSERT and DELETE) in the same way, as long as no dependencies exist between them and subsequent SELECT operation.

Listing 6.2: Reordered single row UPDATE/SELECT, without dependencies.

```
1 SELECT x FROM table_a WHERE id = 2;
2 UPDATE table_a SET x = x + 1 WHERE id = 0;
3 ...
```

6.1.2 Dependent operations

When operations have dependencies between them, i.e., when a WRITE operation modifies values read by a SELECT operation, the reordering process can be more complex, and no single generic solution can be applied to them. This results from the fact that WRITE operation can affect the results returned by the SELECT operation, thus influencing its results when executed prior or after the WRITE operation.

Nonetheless, we describe how to deal with these scenarios. Additionally, and contrarily to the previous scenario, when considering dependent operations, the reordering process has to be adapted accordingly to the number of rows affected by the operations, i.e., if these operate on a single row or multiple rows (referred to as *range* operations).

6.1.2.1 WRITE single/SELECT single

We start by addressing the reordering of dependent operations, executing over the same row. Specifically, we consider a single row UPDATE that modifies the data item accessed by a single row SELECT, exemplified in Listing 6.3.

Listing 6.3: Single row UPDATE/SELECT, with dependencies.

```
1 UPDATE table_a SET x = x + 1 WHERE id = 0;
2 SELECT x FROM table_a WHERE id = 0;
3 ...
```

In the example, attribute *x*, of row 0, is modified by the UPDATE operation and is latter read by the SELECT operation (this example assumes attribute *id* is unique). These

operations are not commutative since the value returned by the SELECT will be different if executed before or after the UPDATE. However, if the updated value can be computed outside the scope of the database, i.e., by the application, this can be reordered by first reading *x*'s initial value and incrementing it by the corresponding amount, in this case 1. Thus, the reordered transaction could be rewritten in a semantically identical way, as presented in Listing 6.4.

Listing 6.4: Reordered single row UPDATE/SELECT, with dependencies.

```
1 SELECT x + 1 FROM table_a WHERE id = 0;
2 UPDATE table_a SET x = x + 1 WHERE id = 0;
3 ...
```

The same can be applied even if the update value is a direct result of a given sub-query. In this case the sub-query, i.e., the SELECT operation responsible for defining the new value, can be executed prior to the execution of the UPDATE, and its result used by the subsequent query, or stored in the application for later use. Since databases tend to be accessed by application code, and transactions are known *a priori*, this reordering should be fairly straightforward to achieve.

When considering single row INSERT operations, the newly inserted values tend to be the result of previous computations made by the application. Thus, any subsequent SELECT operation on the inserted row should return results that are already present in the application.

However, some applications may resort to the database for automatically defining some attribute value when inserting a new table row. For instances, by using auto-increment columns or sequences for defining values of row identifiers, as presented in Listing 6.5.

Listing 6.5: Single row INSERT/SELECT, with dependencies.

```
1 // id_sequence is a database sequence
2 INSERT INTO table_a(row_id,user_id) values (id_sequence, 10);
3 SELECT row_id FROM table_a WHERE user_id = 10;
4 ...
```

Most databases allow the SQL SELECT operation to be used to consult and return these values to the applications, thus allowing to circumvent the initial problem, as presented in Listing 6.6.

Listing 6.6: Reordered single row INSERT/SELECT, with dependencies.

```
1 nextID = SELECT NEXT VALUE FOR id_sequence FROM table_a;
2 INSERT INTO table_a(id,user_id,x) values (nextID, 10, 100);
3 ...
```

We will not consider the reordering of DELETE/SELECT operations on a single row, since executing a SELECT operation on a row that has been deleted makes no apparent sense.

6.1.2.2 WRITE single/SELECT range

We now consider a single row WRITE operation followed by a range SELECT. Consider the example, presented in Listing 6.7, for a dependent single row UPDATE and a range SELECT. In this example, the transaction updates the value of x, to $x+1$, for row 0 and selects all rows (id) that have a value of x larger than 10.

Listing 6.7: Single row UPDATE and range SELECT, with dependencies.

```
1 UPDATE table_a SET x = x + 1 WHERE id = 0;
2 SELECT id FROM table_a WHERE x > 10;
3 ...
```

In this case, row 0 can be included in the result returned by the SELECT operation, if its initial value for attribute x is 10. Thus, executing the SELECT after the UPDATE could return additional rows that would not be returned if executed prior to the UPDATE.

However, selecting the value of x for row 0 before the UPDATE allows the application to know if the increment will include that row in the result for the SELECT operations. Thus, the reordered example presented in Listing 6.8 is equivalent to the original one. In the reordered transaction, the SELECT operation includes the id of row 0 if its value of attribute x is equal to 10 by using the UNION SQL operator.

Listing 6.8: Reordered single row UPDATE and range SELECT, with dependencies.

```
1 SELECT id FROM table_a WHERE id = 0 and x = 10
2     UNION
3         SELECT id FROM table_a WHERE x > 10;
4 UPDATE table_a SET x = x + 1 WHERE id = 0;
5 ...
```

When dealing with INSERT operations, an approach similar to the one described for single INSERT-single SELECT (Section 6.1.2.1) can be used. In the example shown in Listing 6.9, an INSERT operation adds a new row to a table before executing a range SELECT operation on the same table. A simple reorder of operations could compromise the semantics of the transaction, since the SELECT operation can include the newly inserted row if the query condition is satisfied.

Listing 6.9: Single row INSERT range SELECT, with dependencies.

```
1 INSERT INTO table_a(id,x) VALUES(10,100);
2 SELECT id FROM table_a WHERE x > 10;
3 ...
```

However, before inserting a row the application tends to know, *a priori*, the values it will insert. Thus this information could be added to the result set of the SELECT operations, as presented in Listing 6.10. When database mechanisms are used for automatically defining the value of attributes, when inserting rows in a table, the solution presented for single INSERT-single SELECT (Section 6.1.2.1) can be applied.

Listing 6.10: Reordered single row INSERT range SELECT, with dependencies.

```
if(x > 10)
  SELECT 10 as id UNION SELECT id FROM table_a WHERE x > 10;
else
  SELECT id FROM table_a WHERE x > 10;
INSERT INTO table_a(id,x) VALUES(10,100);
...
```

When dealing with dependent DELETE and SELECT operations, again, the simple reorder of operations could compromise the transaction semantics. Consider the following single row DELETE followed by a range SELECT example presented in Listing 6.11. In this case, if the deleted row satisfies the query condition, reordering the two operations would return a different result.

Listing 6.11: Single row DELETE range SELECT, with dependencies.

```
DELETE FROM table_a WHERE id = 0;
SELECT id FROM table_a WHERE x > 10;
...
```

Again, the *a priori* knowledge of the removed row can be used to reorder operations, as presented in Listing 6.12. In this case, by knowing the condition for the DELETE operation, it is possible to add it to the query condition for excluding the deleted rows.

Listing 6.12: Reordered single row DELETE range SELECT, with dependencies.

```
SELECT id FROM table_a WHERE x > 10 AND NOT id = 0;
DELETE FROM table_a WHERE id = 0;
...
```

6.1.2.3 WRITE range/SELECT single

We now consider a range WRITE operation followed by a single row SELECT.

Consider the example presented in Listing 6.13, for dependent range UPDATE followed by a single row SELECT operation. In this example, the transaction updates the value of x for all rows in which the initial value of x is grater than 10 and then selects the value of x for row 0.

Listing 6.13: Range UPDATE single SELECT, with dependencies.

```
UPDATE table_a SET x = x + 1 WHERE x > 10;
SELECT x FROM table_a WHERE id = 0;
...
```

By having an *a priori* knowledge of the increment value and the range condition, we can rewrite the transaction accordingly. In the example presented in Listing 6.14, the initial SELECT operation returns the current value of attribute x of row 0 and the application then increments the value by one if the value satisfies the update condition. In this case, part of the adaptation is done in the application code.

Listing 6.14: Ordered range UPDATE single SELECT, with dependencies.

```
app_x = SELECT x FROM table_a WHERE id = 0;
if (app_x == 10)
  app_x = app_x + 1;
UPDATE table_a SET x = x + 1 WHERE x > 10;
...
```

For a range DELETE operation followed by a single row SELECT operation, consider the example presented in Listing 6.15.

Listing 6.15: Range DELETE single SELECT, with dependencies.

```
DELETE FROM table_a WHERE x > 10;
SELECT x FROM table_a WHERE id = 0;
...
```

With the knowledge on the DELETE condition allows the reordering of operations. As presented in Listing 6.16, if row 0 satisfies the remove condition, then it is excluded from the result of the SELECT operation, thus preserving the transaction semantics.

Listing 6.16: Reordered range DELETE single SELECT, with dependencies.

```
SELECT x FROM table_a WHERE id = 0 AND NOT x > 10;
DELETE FROM table_a WHERE x > 10;
...
```

6.1.2.4 WRITE range/SELECT range

we now consider a range WRITE operation followed by a range SELECT.

Consider the example presented in Listing 6.17, for dependent range UPDATE followed by a range SELECT operation. In this example, the transaction updates the value of x for all rows in which the initial value of x is greater than 10, and then selects the values of attribute x for rows that have a value of attribute id greater than 10. In this example, the UPDATE operation may increment the value of attribute x in rows that have a value of attribute id grater than 10, if the UPDATE condition is satisfied. Thus the SELECT operation may return a different value if executed before of after the UPDATE.

Listing 6.17: Range UPDATE range SELECT, with dependencies.

```
UPDATE table_a SET x = x + 1 WHERE x > 10;
SELECT x FROM table_a WHERE id > 10;
...
```

Again, by having an *a priori* knowledge of the increment value as well as the UPDATE condition, it is possible to rewrite and reorder the operations as presented in Listing 6.18. In the reordered version, the SELECT operation is transformed into the UNION of two SELECT operations. One selects the values of attribute x for all rows that satisfy the original condition and do not have an x value greater than 10, while the other the values of attribute x with the respective increment for all rows that satisfy the original condition and have an x value greater than 10 (the update condition).

Listing 6.18: Reordered range UPDATE range SELECT, with dependencies.

```
SELECT x FROM table_a WHERE id > 10 AND x <= 10
    UNION
        SELECT x+1 FROM table_a WHERE id > 10 AND X > 10;
UPDATE table_a SET x = x + 1 WHERE x > 10;
...
```

We now consider the case of a range DELETE followed by a range SELECT, as exemplified in Listing 6.19.

Listing 6.19: Range DELETE range SELECT, with dependencies.

```
DELETE FROM table_a WHERE x > 10;
SELECT x FROM table_a WHERE id > 0;
...
```

The *a priori* knowledge on the DELETE condition allows the reordering of operations. As presented in Listing 6.20, if a row satisfies the remove condition (in the example the value of attribute *x* is greater than 10), then it is excluded from the result of the SELECT. In the reordered version, this is achieved with an additional condition for the SELECT operation.

Listing 6.20: Reordered range DELETE range SELECT, with dependencies.

```
SELECT x FROM table_a WHERE id > 0 AND NOT x > 10;
DELETE FROM table_a WHERE x > 10;
...
```

Note that, the majority of the presented examples are application related, thus there is no generic solution that can be applied to every application. However, we believe that most transactions can be rewritten to satisfy the read-before-write behavior, be it by using additional SQL queries or in conjunction with additional application code.

Next we employ the previously described techniques for reordering the transaction of the TPC-C benchmark, and study how these modification influence the performance of the database.

6.2 TPC-C

TPC-C has been adopted as an industry standard benchmark for online transaction processing (OLTP). Thus it presents itself as a good example for studying how the order of operations, inside a transaction, influences database performance. TPC-C defines five different transactions: *i*) New Order; *ii*) Payment; *iii*) Stock Level; *iv*) Order Status, and *v*) Delivery. These transactions are composed by the operations presented in Table 6.1, ordered according to the TPC-C specification. From this set of transactions, *Stock Level* and *Order Status* only perform read operations, while all others perform both read and update operations.

New Order	Payment	Stock Level	Order Status	Delivery
S(*customer* ⋈ *warehouse*)	U(*warehouse*)	S(*district*)	S(*costumer*)	S(*new_order*)
S(*district*)	S (*warehouse*)	S(*order_line* ⋈ *stock*)	S(*orderr*)	D(*new_order*)
U(*new_order*)	U(*district*)		S(*order_line*)	S(*orderr*)
U(*district*)	S(*district*)			U(*orderr*)
I(*orderr*)	S(*customer*)			U(*order_line*)
S(*item*)	U(*customer*)			S(*order_line*)
S(*stock*)	I(*history*)			U(*customer*)
U(*stock*)				
I(*order_line*)				

Table 6.1: TPC-C transactions. 'S', 'U', 'I' and 'D' refer to *select*, *update*, *insert* and *delete* operations, respectively, and the accessed tables, with $t1 \bowtie t2$ representing a join between tables *t1* and *t2*.

6.2.1 Transaction Analysis

From a first analysis one can see that read and write operations, inside each transaction, occur without any specific relative order. For instance, the *Payment* transaction starts by updating table *warehouse* before doing any other operations, while a *New Order* transaction starts by reading the join between tables *warehouse* and *customer* before anything else.

As previously described, from the application perspective, the order of operations inside each transaction is not relevant as long as the transaction semantics is preserved. Under this condition, operations inside a transaction can be reorder without consequences to the application, since both the original transaction and its reordered version evolve the database from the same initial state to the same final state, thus producing the same result.

On the other hand, operation order can have a considerable performance impact in the performance of the database. For instance, starting a transaction with a table update operation will prevent any concurrent operations on that same table from executing, since the database locks the table exclusively for the duration of the transaction, thus restricting concurrency for all other concurrent operation on the same table. However, when transactions start by performing read operations, since these acquire shared locks on the respective tables, other concurrent read operations, on the same tables, are still allowed to execute.

Therefore, if write operations (including INSERT, DELETE and UPDATE operations) are delayed until the end of each transaction, i.e., after all read operations have been executed, there is a chance of improving concurrency at the database level. This modification is expected to improve performance in two ways: i*)* due to increased concurrency between update transactions, since these transactions can execute their read phase concurrently, allowing transactions to make progress concurrently; and ii*)* by allowing read-only transactions, such as *Stock Level* and *Order Status*, to execute to conclusion concurrently with update transactions, since read-only transactions can successfully execute concurrently with other update transactions that are in their read phase.

6.2.2 TPC-C reordering

To assess the performance influence the order of operations has inside transactions, we applied some of the previously described mechanisms and modified the TPC-C benchmark. This modification focuses on the reordering of operations inside each transaction, making them compliant with the *Read before Write* pattern, as presented in Figure 6.2.

New Order	Payment	Stock Level	Order Status	Delivery
S($customer \bowtie warehouse$)	S ($warehouse$)	S($district$)	S($costumer$)	S(new_order)
S($district$)	S($district$)	S($order\_line \bowtie stock$)	S($orderr$)	S($orderr$)
S($item$)	S($customer$)		S($order\_line$)	S($order\_line$)
S($stock$)	U($warehouse$)			D(new_order)
I($orderr$)	U($district$)			U($orderr$)
U(new_order)	U($customer$)			U($order\_line$)
U($district$)	I($history$)			U($customer$)
U($stock$)				
I($order\_line$)				

Table 6.2: Reordered TPC-C transactions.

We will now detail the modifications made to each TPC-C transaction, focusing only on the database interactions and dependencies in the operations. This discussion will present the original database interaction of TPC-C and, for each one, describes the necessary changes for reordering them to be RbW compliant.

6.2.2.1 New Order

Listing 6.21 presents the database interactions made by the *new order* transaction. The reordering process for this transaction is fairly straightforward since no dependencies exist between WRITE and SELECT operations. As expected, WRITE operations modify the database based on values previously read by SELECT operations.

Listing 6.21: TPC-C new order transaction.

```
1  SELECT c_discount, c_last, c_credit, w_tax
2      FROM customer x warehouse
3          WHERE w_id = ? AND c_w_id = ? AND c_d_id = ? AND c_id = ?;
4  SELECT d_next_o_id, d_tax
5      FROM district
6          WHERE d_id = ? AND d_w_id = ?;
7  INSERT
8      INTO new_order (n_o_id, n_d_id, no_w_id)
9          VALUES (?, ?, ?)
10 UPDATE d_next_o_id
11     FROM DISTRICT district
12         WHERE d_id = ? AND d_w_id = ?;
13 INSERT
14     INTO orderr (o_id, o_d_id, o_w_id, o_c_id, o_entry_d, o_ol_cnt, o_all_local)
15         VALUES (?,?,?,?)
16
17 for 1 to order_lines
18     SELECT i_price, i_name, i_data
19         FROM item
```

```
20              WHERE i_id = ?;
21      SELECT s_quantity, s_data, s_dist01...10
22          FROM stock
23              WHERE s_i_id = ? AND s_w_id = ?;
24      UPDATE s_quantity, ol_quantity, s_remote_cnt_increment
25          FROM stock
26              WHERE s_i_id = ? AND s_w_id = ?;
27      INSERT
28          INTO order_line(ol_o_id, ol_d_id, ol_w_id, ol_number,
29          ol_i_id, ol_supply_w_id, ol_quantity, ol_amount, ol_dist_info)
30              VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?);
31  end for
```

The original Transaction does not respect the RbW pattern only due to a *for loop* used to define the order lines for the new order. Thus, for the reordering process, only the two SELECT operations inside the loop (lines 18 and 21) needed to be reordered. Again, since these SELECT operations only read values used by the two WRITE operations (lines 24 and 27) that follow them, the reordering process is straightforward.

Thus, the application was modified to first read the needed values, for each of the lines of the new order, before executing the write phase, as presented in Listing 6.22. Note that the relative order of WRITE operations is preserved.

Listing 6.22: Reordered TPC-C new order transaction.

```
1   # READ PHASE
2   SELECT c_discount, c_last, c_credit, w_tax
3       FROM customer x warehouse
4           WHERE w_id = ? AND c_w_id = ? AND c_d_id = ? AND c_id = ?;
5   SELECT d_next_o_id, d_tax
6       FROM district
7           WHERE d_id = ? AND d_w_id = ?;
8   for 1 to order_lines
9       SELECT i_price, i_name, i_data
10          FROM item
11              WHERE i_id = ?;
12      SELECT s_quantity, s_data, s_dist01...10
13          FROM stock
14              WHERE s_i_id = ? AND s_w_id = ?;
15  end for
16
17  # WRITE PHASE
18  INSERT
19      INTO new_order (n_o_id, n_d_id, no_w_id)
20          VALUES (?, ?, ?)
21  UPDATE d_next_o_id
22      FROM DISTRICT district
23          WHERE d_id = ? AND d_w_id = ?;
24  INSERT
25      INTO orderr (o_id, o_d_id, o_w_id, o_c_id, o_entry_d, o_ol_cnt, o_all_local)
26          VALUES (?,?,?,?)
27  for 1 to order_lines
28      UPDATE s_quantity, ol_quantity, s_remote_cnt_increment
29          FROM stock
30              WHERE s_i_id = ? AND s_w_id = ?;
31      INSERT
```

```
32          INTO order_line(ol_o_id, ol_d_id, ol_w_id, ol_number,
33          ol_i_id, ol_supply_w_id, ol_quantity, ol_amount, ol_dist_info)
34              VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?);
35  end for
```

6.2.2.2 Payment

Listing 6.23 presents the database interactions made by the *payment* transaction. The reordering process for this transaction is also fairly straightforward since no dependencies exist between WRITE and SELECT operations. Like in the previous transaction, WRITE operations modify the database based on values previously read by SELECT operations or previously defined by the application.

Listing 6.23: TPC-C payment transaction.

```
1   UPDATE w_ytd
2       FROM warehouse
3           WHERE w_id = ?
4   SELECT w_street_1...2, w_city, w_state, w_zip, w_name
5       FROM warehouse
6           WHERE w_id = ?;
7   UPDATE d_ytd
8       FROM district
9           WHERE d_w_id = ? AND d_id = ?;
10  SELECT d_street_1...2, d_city, d_state, d_zip_d_name
11      FROM district
12          WHERE d_w_id = ? AND d_id = ?;
13  if
14      SELECT count(c_id)
15          FROM customer
16              WHERE c_last = ? AND c_d_id = ? AND c_w_id;
17      SELECT c_first, c_middle, c_id, c_street_1...2, c_city,
18          c_state, c_zip, c_phone, c_credit, c_credit_lim,
19          c_discount, c_balance, c_since
20          FROM customer
21              WHERE c_w_id = ? AND c_d_id = ? AND c_last = ?;
22  else
23      SELECT c_first, c_middle, c_id, c_street_1...2, c_city,
24          c_state, c_zip, c_phone, c_credit, c_credit_lim,
25          c_discount, c_balance, c_since
26          FROM customer
27          WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;
28  endif
29  if
30      SELECT c_data
31          FROM customer
32               WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;
33      UPDATE c_balance, c_data
34          FROM customer
35              WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;
36  else
37      UPDATE c_balance
38          FROM customer
39              WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;
40  endif
```

```
41 INSERT
42     INTO history (h_c_d_id, h_c_w_id, h_c_id, h_d_id,
43         h_w_id, h_date, h_amount, h_data)
44         VALUES (?, ?, ?, ?, ?, ?, ?, ?);
```

The original payment transaction does not respect the RbW pattern, since it starts by updating a single row of table *warehouse* (line 1) before any other operations, and also updating an additional row of table *district* (line 7) between two SELECT operations. Again, since no dependencies exist between these WRITE operations and the following SELECT operations, reordering is straightforward, by simply moving the WRITE operations after SELECT operations. However, an additional SELECT operation is used inside a condition branch (line 30). Once again, since it does not depend on any preceding WRITE operation it can be reordered before all WRITE operations.

Thus, the application was modified to allow the reordering of operations, as presented in Listing 6.24. Again, note that the relative order of WRITE operations is preserved.

Listing 6.24: Reordered TPC-C payment transaction.

```
1  # READ PHASE
2  SELECT w_street_1...2, w_city, w_state, w_zip, w_name
3      FROM warehouse
4          WHERE w_id = ?;
5  SELECT d_street_1...2, d_city, d_state, d_zip_d_name
6      FROM district
7          WHERE d_w_id = ? AND d_id = ?;
8  if
9      SELECT count(c_id)
10         FROM customer
11             WHERE c_last = ? AND c_d_id = ? AND c_w_id;
12     SELECT c_first, c_middle, c_id, c_street_1...2, c_city,
13         c_state, c_zip, c_phone, c_credit, c_credit_lim,
14         c_discount, c_balance, c_since
15         FROM customer
16             WHERE c_w_id = ? AND c_d_id = ? AND c_last = ?;
17 else
18     SELECT c_first, c_middle, c_id, c_street_1...2, c_city,
19         c_state, c_zip, c_phone, c_credit, c_credit_lim,
20         c_discount, c_balance, c_since
21         FROM customer
22         WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;
23 endif
24 if
25     SELECT c_data
26         FROM customer
27              WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;
28 endif
29
30 # WRITE PHASE
31 UPDATE w_ytd
32     FROM warehouse
33         WHERE w_id = ?
34 UPDATE d_ytd
35     FROM district
36         WHERE d_w_id = ? AND d_id = ?;
37 if
```

```
38      UPDATE c_balance, c_data
39          FROM customer
40              WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;
41  else
42      UPDATE c_balance
43          FROM customer
44              WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;
45  endif
46  INSERT
47      INTO history (h_c_d_id, h_c_w_id, h_c_id, h_d_id,
48          h_w_id, h_date, h_amount, h_data)
49          VALUES (?, ?, ?, ?, ?, ?, ?, ?);
```

6.2.2.3 Delivery

Listing 6.25 presents the database interactions made by the *delivery* transaction. The reordering process for this transaction required additional application modifications, due to the iterative nature of the transaction, but, from the view point of the database interaction it is also fairly straightforward since no dependencies exist between WRITE and SELECT operations. Like in the previous transactions, WRITE operations modify the database based on values previously read by SELECT operations or previously defined by the application.

Listing 6.25: TPC-C delivery transaction.

```
1   for 1  ... 10
2       SELECT no_o_id
3           FROM new_order
4               WHERE no_d_id = ? AND no_w_id = ?;
5       for each no_o_id
6           DELETE
7               FROM new_order
8                   WHERE no_i_id = ? AND no_w_id = ? AND no_o_id = ?;
9       endfor
10      if
11          SELECT o_c_id
12              FROM orderr
13                  WHERE o_id = ? AND o_d_id = ? AND o_w_id = ?;
14          UPDATE o_carrier_id
15              FROM orderr
16                  WHERE o_id = ? AND o_d_id = ? AND o_w_id = ?;
17          UPDATE ol_delivery_d
18              FROM order_line
19                  WHERE ol_o_id = ? AND ol_d_id= ? AND ol_w_id = ?;
20          SELECT sum(ol_amount)
21              FROM order_line
22                  WHERE ol_o_id = ? AND ol_d_id = ? AND ol_w_id = ?;
23          UPDATE c_balance, c_delivey_cnt
24              FROM customer
25                  WHERE c_id = ? AND c_d_id = ? AND c_w_id = ?;
26      endif
27  endfor
```

The original delivery transaction does not respect the RbW pattern, since two SELECT operations (line 11 and 20) may execute after the execution of WRITE operations. Again, since no dependencies exist between these SELECT operations and preceding WRITE operations, reordering is straightforward, by simply moving the WRITE operations after SELECT operations.

Due to the iterative nature of the transaction, it required careful modifications to the application for guaranteeing transaction semantics, especially since the conditional part of the transaction depends on previously read values within the same iteration. Thus, for reordering the transaction a loop was needed for the read phase, storing for each iteration the read values, and an additional loop was used for the write phase, using the previously read values, as presented in Listing 6.26. Again, note that the relative order of WRITE operations is preserved.

Listing 6.26: Reordered TPC-C delivery transaction.

```
1  # READ PHASE
2  for 1  ... 10
3      SELECT no_o_id
4          FROM new_order
5              WHERE no_d_id = ? AND no_w_id = ?;
6      SELECT o_c_id
7          FROM orderr
8              WHERE o_id = ? AND o_d_id = ? AND o_w_id = ?;
9      SELECT sum(ol_amount)
10         FROM order_line
11             WHERE ol_o_id = ? AND ol_d_id = ? AND ol_w_id = ?;
12 endfor
13
14 # WRITE PHASE
15 for 1  ... 10
16     foreach no_o_id
17         DELETE
18             FROM new_order
19                 WHERE no_i_id = ? AND no_w_id = ? AND no_o_id = ?;
20     endfor
21     if
22         UPDATE o_carrier_id
23             FROM orderr
24                 WHERE o_id = ? AND o_d_id = ? AND o_w_id = ?;
25         UPDATE ol_delivery_d
26             FROM order_line
27                 WHERE ol_o_id = ? AND ol_d_id= ? AND ol_w_id = ?;
28         UPDATE c_balance, c_delivey_cnt
29             FROM customer
30                 WHERE c_id = ? AND c_d_id = ? AND c_w_id = ?;
31     end if
32 endfor
```

6.3 Performance with Modified Transactions

We evaluated our hypothesis that modifying the order of operations inside transactions, by moving reads to the begin and writes to the end, can help improve application performance. To this end, we run the TPC-C benchmark using the original version and the modified one.

For this experiment we used our modified HSQLDB storage engine, without latches, and the serializable isolation level, and ran TPC-C benchmark with original transactions, denoted as *Original*, and the RbW reordered version, denoted as *RbW*.

6.3.1 8-92 workload

Figure 6.1 compare the throughput for the 8-92 workload. Under this update-intensive workload, the RbW version presents very bad scalability, with a rapid drop in performance as the number of clients increase.

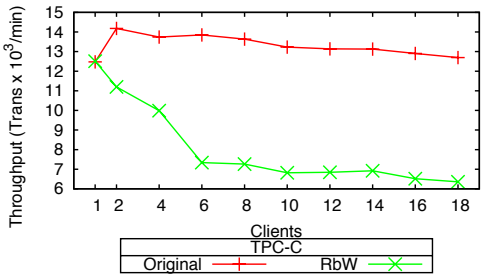


Figure 6.1: RbW TPC-C performance (8-92 workload).

This drop in performance results from the RbW modification allowing all transactions to execute their read phase concurrently even if they, later on, have to abort due to conflicting updates. This harms throughput since it uses CPU time for executing transactions that end up aborting, thus preventing non-conflicting transactions from using the same CPU time. Additionally, RbW increases dependencies and possible deadlock situations, due to concurrent transactions sharing read locks on data item before trying to acquire a write lock on a previously locked table.

As an example consider transactions *Payment* and *New Order*, which, combined make approximately 88% of all transactions executed in this workload. The *original* Payment transaction starts by updating table *warehouse*, while the *original* New Order transaction reads from the same table. If New Order succeeds locking table *warehouse*, then Payment will block waiting for an exclusive lock on the same table. Since Payment transaction does not hold any locks, it allows other concurrent New Order transactions to complete.

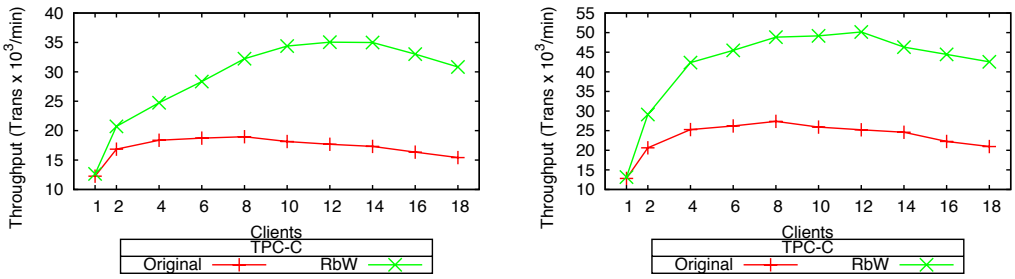
However, under *RbW* both transactions are allowed to acquire the shared lock for table *warehouse*. Thus, when initiating their write phase, Payment will bock trying to acquire the exclusive lock for table *warehouse*, since New Order has a shared lock for that table, while New Order will block trying to acquire the exclusive lock for table *district*,

since Payment has a shared lock for that table. This leads to a deadlock situation that needs to be dealt by the transaction manager, and will force one of the transactions to abort.

This has a considerable performance impact on the overall performance. Additionally, since the deadlock resolution process is performed in mutual exclusion, it also contributes to the observed reduction in concurrency, since possible non conflicting transactions also block during this process.

6.3.2 Other workloads

As expected, the throughput for the RbW modification increases with the the ratio of read-only transactions. This is shown in Figures 6.2(a) and 6.2(b), for the 50-50 and 80-20 workloads respectively.



(a) 50-50 workload (b) 80-20 workload

Figure 6.2: RbW TPC-C performance (50-50 and 80-20 workloads).

Under these workloads, the RbW modification allows an increase in performance of up to 2× of the original version, for both 50-50 and 80-20 workloads, as presented in Figure 6.3. These are expected results since RbW pattern allows read-only transactions to execute to completion concurrently with update transactions, during their read phase. Also, the increasing ratio of read-only transactions reduces contention created by update transactions, also reducing interference between them.

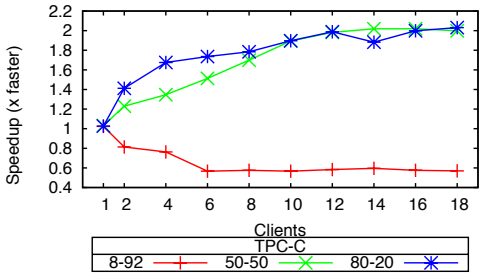


Figure 6.3: RbW speedup over Original TPC-C.

6.4 Read before Writes with early lock release

Contrarily to the original TPC-C version, lock acquisition under *RbW* always follows the same pattern, where all read operations have been executed before executing any update operation. Thus, under the RbW pattern, no new read (shared) lock is acquired after acquiring the first write (exclusive) lock.

As implied by 2PL, serializability is enforced if no lock is obtained after the first lock is released. Thus, it is possible to release all shared locks after acquiring all exclusive locks.

To study the implications of this approach, we have modified our latch-free version of HSQLDB, so that an application can specify the locks it needs before each transaction begins. This allows the database to have an *a priori* knowledge of each transaction lock set. Since, OLTP applications have pre-defined transactions, it is reasonable to assume that lock information at table and attribute levels can be extracted by a simple analysis of transaction operations.

Additionally, the database lock manager has also been modified, so that, when obtaining the first exclusive lock, it will try to obtain all the necessary exclusive locks for completing the transaction, followed by the release of all previously acquired shared locks that have not been upgraded. This modification is expected to increase concurrency for update intensive workloads, since the early release of shared locks allows otherwise blocked write operations to execute.

Unlike some systems that use information about read and write sets to improve concurrency during transaction execution [Ren+12; Tho+12], our approach only requires coarse grain information - the name of the table and attributes accessed. This information can be easily obtained by a simple transaction analysis of the programs. This contrasts to schemes using low granularity information, which is very hard or impossible to obtain statically - e.g. when a read/write depends upon a value read during the same transaction.

6.4.1 Evaluation

To evaluate this modification we compared the RbW results with the throughput obtained when combining the RbW pattern with the early release of read locks, referred to as RbW-RL.

6.4.1.1 8-92 Workload

The results for the 8-92 workload, presented in Figure 6.4, show some improvements for update-intensive workloads. As expected the early release of shared locks allows otherwise blocked update transaction to execute, thus increasing the concurrency for update transactions.

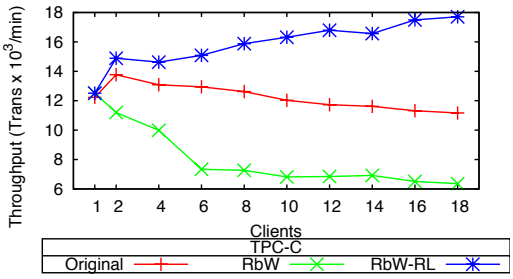
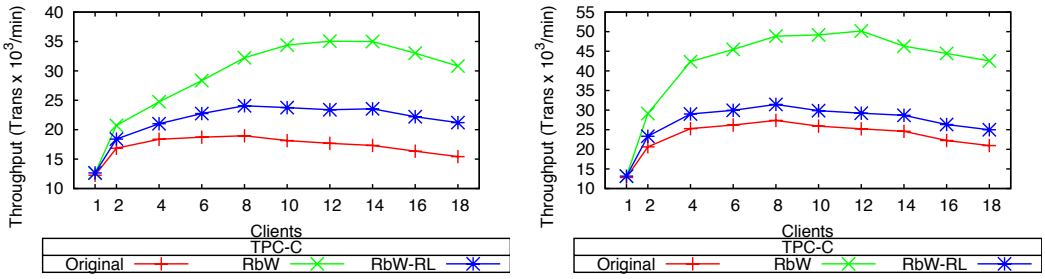


Figure 6.4: RbW early release of read lock (8-92 workload).

6.4.1.2 50-50 and 80-20 Workloads

Moderate read and moderate update workloads, such as 50-50 and 80-20, suffer a significative performance penalty, compared to the RbW modification, as presented in Figure 6.5(a) and 6.5(b). This can be explained by the fact that, to preserve isolation semantics, the early release of shared locks can only occur after the early acquisition of all exclusive locks. However, this early acquisition of exclusive locks, increases the period locks are held, thus reducing concurrency of read-only transactions.



(a) 50-50 workload (b) 80-20 workload

Figure 6.5: RbW early release of read lock (50-50 and 80-20 workloads).

Next we compare the RbW TPC-C modification when combined with A2L and our modified storage engine.

6.5 Combining RbW with A2L

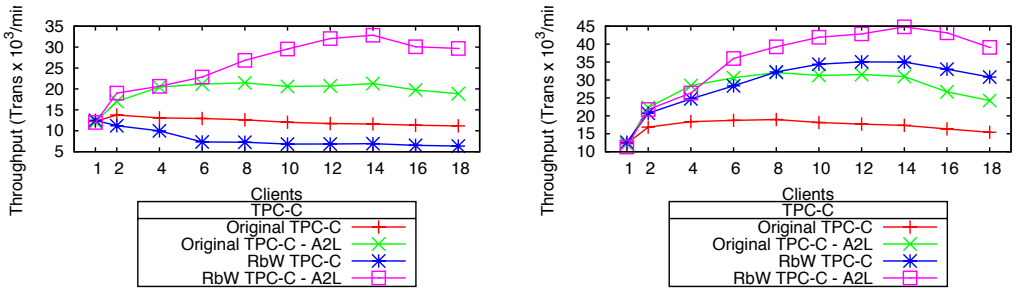
In Section 5.2, we have shown that A2L enforces serializability when using 2PL mechanisms. Thus, since the reordering of operation inside transactions according to a RbW pattern does not compromises 2PL, it can be safely combined with A2L, providing serializable semantics. We now show the evaluation results obtained when combining these techniques, and compare them to both the original HSQLDB engine, and the original TPC-C implementation.

For this evaluation we compared the performance of the original TPC-C version, running on both the original HSQLDB and our A2L version of HSQLDB, referred as *Original*

TPC-C and *Original TPC-C - A2*, with the RbW modified version of TPC-C, running on our A2L version of HSQLDB, referred as *A2L - RbW TPC-C*.

6.5.1 8-92 and 50-50 workloads

We start by presenting the results for the 8-92 and 50-50 workloads, in Figures 6.6(a) and 6.6(b) respectively. These results show that the RbW modification, when combined with our A2L engine, offers increased performance improvements over the other configurations, under both workloads.

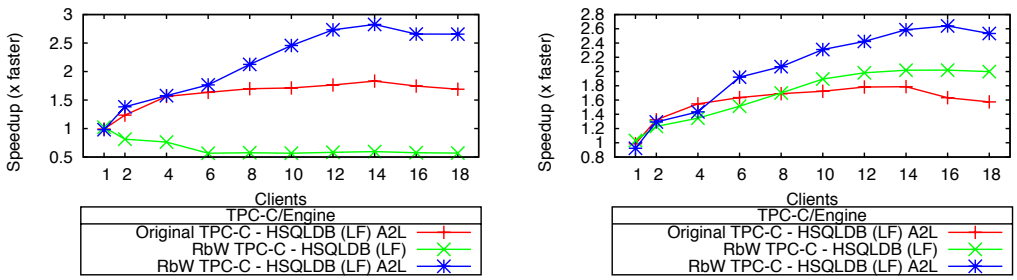


(a) 8-92 workload (b) 50-50 workload

Figure 6.6: Comparing performance of proposed modifications (8-92 and 50-50 Workloads).

This is put into evidence when comparing the speedup of the differente configurations over the original HSQLDB engine for the 8-92 and 50-50 workloads, presented in Figures 6.7(a) and 6.7(b) respectively.

The RbW/A2L combination achieves up to 2.8× and 2.5×, when compared to the original HSQLDB engine running the original TPC-C version, for the 8-92 and 50-50 workloads respectively. This combination also offer a 1.4× and 1.2× performance improvement, when compared to the original TPC-C version running on our A2L engine, for the same workloads.

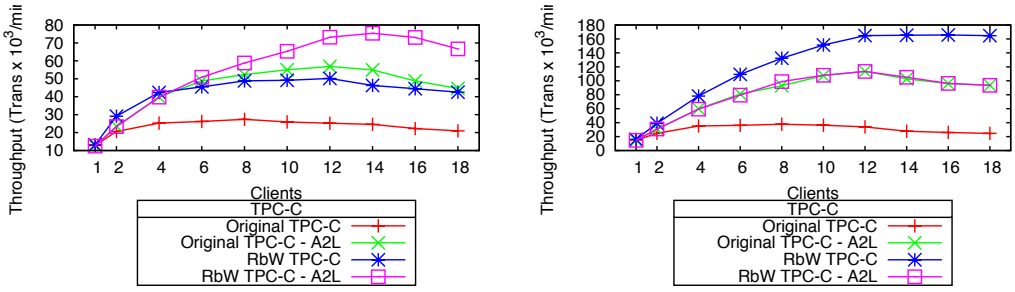


(a) 8-92 workload (b) 50-50 workload

Figure 6.7: Comparing speedups of proposed modifications (8-92 and 50-50 Workloads).

6.5.2 80-20 and 100-0 Workloads

The performance obtained for the 80-20 workloads, presented in Figures 6.8(a) and 6.8(b), present identical improvements. These results show that the RbW modification, when combined with our A2L engine, offers increased performance improvements over the other configurations, under both workloads.

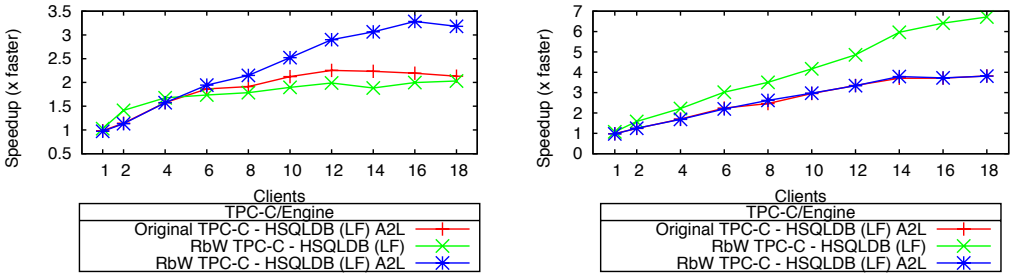


(a) 80-20 workload (b) 100-0 workload

Figure 6.8: Comparing performance of proposed modifications (80-20 and 100-0 Workloads).

Like before, this is put into evidence when comparing the speedup of the differente configurations over the original HSQLDB engine for the 80-20 and 100-0 workloads, presented in Figures 6.9(a) and 6.9(b) respectively.

The RbW/A2L combination achieves up to 3.5× and 4×, for the 80-20 and 100-0 workload respectively, when compared to the original HSQLDB engine running the original TPC-C version. Also, it offers a 1.5× performance improvement, for the 80-20 workload respectively, when compared to the original TPC-C version running on the same engine.



(a) 80-20 workload (b) 100-0 workload

Figure 6.9: Comparing speedups of proposed modifications (80-20 and 100-0 Workloads).

6.5.3 Scalability

The combination of the RbW TPC-C version and the A2L engine offers scalable performance, with the number of clients, up to approximately 12 to 14 clients, compared to the original TPC-C running on the original HSQLDB engine. This is put in evidence

in Figure 6.10, where we show the *RbW TPC-C - A2L* speedup over the original TPC-C version running on the original HSQLDB engine, for the different workloads.

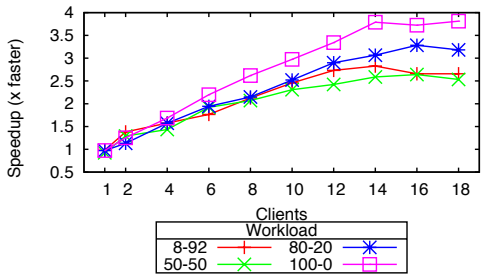


Figure 6.10: A2L RbW TPC-C Speedup over Original TPC-C on HSQLDB.

6.5.4 Additional discussion

Contrarily to RbW, the early release of read locks (RbW-RL) when combined with the A2L scheme does not translate into an increase in performance. This occurs for two reasons: First, under A2L most TPC-C transactions end up conflicting only with themselves, due to attribute dependencies. This is a situation that the early release of read lock does not improve. Second, the early release of read locks increases locking overhead, as put into evidence by the decrease in performance under the 50-50 and 80-20 workload, as presented in Figures 6.5(a) and 6.5(b).

Since transaction locking algorithm is itself a contention point, the early release of read locks further increases its overhead, resulting in a decrease in performance. This is also put into evidence with the decrease in performance of A2L under read-only workloads, as presented in Figure 5.5(b). Again, we believe these limitations could be addressed by increasing concurrency of lock management algorithms with some of the ideas presented in recent works [Jun+13; Ren+12; Tu+13].

6.6 Correctness

For the correctness of the RbW and RbW-RL modifications, it is necessary to guarantee that transactions executed under these access patterns respect serializability semantics, i.e., that the result of concurrently executing two or more transactions, under RbW and RbW-RL, is identical to some sequential order of those same transactions. This proof is identical to the ones for A2L.

Proposition 5. *RbW and RbW-RL guarantees serializability of concurrent transactions.*

Proof. Two-phase locking (including predicate locks) guarantees serializability [Ady+00; Ber+95], regardless of access pattern. As the RbW approach uses two-phase locking it guarantees serializability. Thus, we only need to show that the early release of locks, i.e., RbW-RL, is serializable. Since, under RbW-RL, read locks are only releases after all write

locks have been acquired, and no new locks are acquired after this, RbW-RL preserves the 2PL locking scheme, thus guaranteeing its correctness. This is also true under A2L, since locking patterns under A2L are identical to 2PL, even when early releasing read locks. Additionally, since both approaches lock entire tables, this prevents concurrent transaction from updating any row, including adding new rows, thus achieving the same effect of predicate locking. □

For proving the correctness of RbW and RbW-RL, we further need to prove that it can run with a modified IMDB that includes no concurrency control on the data structures used for maintaining data.

Proposition 6. *RbW and RbW-RL preserves consistency with data structures that include no concurrency control.*

Proof. 2PL and A2L locking schemes acquire shared and exclusive locks on an item before accessing it, thus prevent any concurrent operation from acquiring an exclusive lock on the same item for modifying it. This prevents concurrent read/write and write/write operations on the same data items. Additionally, A2L exclusive locks all attributes of a table before executing inserting or deleting an item. This prevents concurrent read operation from acquiring shared locks on any attribute, thus preventing transversals while structural changes occur in the data structures. Since both RbW and RbW-RL preserve A2L locking behavior, and A2L can be combined with a lock free storage engine, then RbW and RbW-RL can be combined with a lock free storage engine. □

6.7 Summing up

From the presented results we can conclude that applications can have a considerable influence in database performance. This impact, as expected, is most noticeable at moderate read and update workloads (50-50 and 80-20 workloads), since operation reordering, following the *RbW* pattern, allows for increased concurrency between read-only and update transactions.

Additionally, combining RbW scheme with the early release of read locks, i.e., RbW-RL, allows for some performance improvement for update-intensive workloads (8-92 workload), although at the cost of a performance decrease under moderate read and write workloads (50-50 and 80-20 workloads). This is expected since RbW-RL, not only increases the locking overhead, but also limits concurrency for read operations. This results from write locks being acquired all at the same time, for guaranteeing isolation semantics, and from being held for longer periods of time. This increase in lock duration has a negative impact on concurrency, thus decreasing the overall performance.

Finally, combining the RbW modifications with A2L offers significative performance improvements, not only due to the less interfering nature of RbW, but also, when combined with A2L transaction tend to conflict only with themselves. This allows our system to scale under every workload, as presented by the results in Figure 6.10.

Chapter 7

Conclusions

Multicore systems offer increased computational power by providing multiple processing cores. Exploring this computational power is challenging, as it requires using multiple processors concurrently. This dissertation focus on improving performance and scalability of general purpose in-memory databases on multicore systems. More specifically, we proposed techniques for improving the scalability of IMDBs while enforcing strong isolation semantics (i.e., Serializability).

As the first contribution of our work, we have presented a performance study of two general purpose IMDBs running on a multicore system. This study showed that both databases are unable to scale with the number of clients, independently of the type of workload or isolation level used. Additionally, we have identified the *Storage* subcomponent as a major scalability bottleneck of the database. More specifically, the lack of scalability of these systems is a direct result of the contention created by the concurrency control mechanisms (latches) used by the index data structures to maintain data.

We have explored different approaches for addressing this problem. First, we started by trying to answer the question of whether it is possible to improve scalability without modifying the database engines and the applications. To address this question, we designed and implemented a system, MacroDB, that explores database replication in a single multicore machine. MacroDB treats multicore machines as extremely low latency clusters extended with some shared memory, and builds on the knowledge of distributed and replicated databases, for creating a system that combines multiple database replicas that can run concurrently at different cores. This allows to partially address the scalability problem by distributing and balancing load among different replicas.

Our experiments show that this approach is able to offer performance benefits raging from 40% to 180% over standalone database engines under read-intensive workloads. For

update-intensive ones, it suffers from 5% to 14% overhead. We have also shown that the memory used by the system is not directly proportional to the number of replicas, thus making this approach practical. The result of update-intensive workloads show that the performance of a single replica is a key limitation factor to the scalability of the overall system.

Second, we tried to answer the question of how to scale IMDBs by modifying the database engine, while minimizing changes to the architecture of these systems. In this approach, we delved into the database engine and tried to fix the major performance bottlenecks. We started by proposing the removal of concurrency control mechanisms (latches) in low level data structures, and a new locking scheme for supporting different isolation levels, while preventing concurrent read/write and write/write access in the storage data structures.

Our experiments show that this approach is able to outperform the original database performance by a factor of up to 7× for the TPC-C benchmark under read-intensive workloads, offering scalable performance. Also, and contrarily to the previous approach, moderate update workloads benefit from 1.5× to 4× performance improvement, although with limited scalability.

Next, we tried to address the problem for update-intensive workloads. For allowing increased concurrency, we proposed the use of attribute level locking (A2L). In A2L, operations lock table attributes, i.e., pairs (table,column), instead on locking the complete table. This reduces contention by allowing transactions to execute concurrently on different attributes, i.e., as long as no concurrent read/write or write/write operations execute on the same table attribute.

Efficient support of A2L required modifying the storage component, since database update operations are commonly implemented as remove/insert operation pairs at the index level. Thus, we modified the database to perform direct in-place updates for non key attribute updates, while maintaining the original behavior when updating key attributes.

Running the TPC-C benchmark on our modified engine shows this approach improves performance under all types of workloads, scaling up to 2× the performance of the original engine in standard TPC-C workload (92% updates). Performance under read-intensive workloads also improved over the original engine, scaling up to 3.5×.

Finally, we tried to address the question of what is the impact of the application code in the performance of the database, and whether it is possible to change the application code to improve scalability while retaining the same semantics. We started by analyzing how operation order inside transactions influence the database performance, and then studied how transactions can be modified in a database friendly way, i.e., for reducing interference. Finally, we have shown how to modify database to take advantage of these modifications to further improve database performance.

This study showed that modifying transactions using a *read before write* (RbW) pattern, where all read operations execute prior to all write operations, is able to increase performance up to 2.5×, when compared to the original unmodified transactions, for update

intensive workloads.

Additionally, lock acquisition under RbW allows for early releasing of read lock. We further modified the HSQLDB database for early releasing read locks. The results with the modified version of the database showed some performance improvements for update intensive workloads, at the expense of performance decrease for read intensive workloads.

Finally, we combined the RbW modification with the A2L locking scheme. Running the modified TPC-C benchmark on our modified engine showed performance improvements under all workloads, scaling up to 3× the performance of the original engine in standard TPC-C workload (92% updates). Performance benefits, under read-intensive workloads also improved, scaling up to 3.5× the performance of the original engine. Additionally, the proposed modifications allowed the modified engine to scale on multicore systems, under all workloads.

7.1 Future Work

While the proposed database modifications have improved database performance, update-intensive workloads have not achieved the same improvement as read-intensives. Additionally, the modification for improving these workloads, compromised the improvement for read-intensive ones. Thus, a possible research direction, would be to further address the scalability of update-intensive workloads, by allowing SQL INSERT and DELETE operations to execute concurrently with other operations.

We believe this could be achieved by further modifying the index data-structures, so that SQL INSERT and DELETE operations would not result in complex state changes. A straightforward modification to the HSQLDB engine would be to prevent the immediate rotation of the AVL index tree, when inserting or removing nodes, performing these state changes periodically. This modification would require additional modifications to the Lock Manager, for preserving transaction isolation.

An additional direction would be to study the benefits of delaying the effects of write operations until the end of a transaction. The purpose would be to reduce the periods that transactions hold exclusive locks, thus increasing concurrency. Under such scenario, rather than acquiring locks when executing write operations, these would only be applied at commit time. Isolation is still preserved since locking pattern would be preserved. Also, whenever update operations depend on previously read values, the database can be modified so that the value is only calculated internally, and not returned to the applications. This ideia follows self increment values, that automatically increment the value for a given attribute when inserting a new table row. Contrarily to these, this idea is to allows developers to define operations and corresponding arguments, so that the final values are calculated internally by the database. This should reduce conflicts, for instance when inserting new rows based on previously read values, where concurrent inserts would try to insert the same value.

Finally, current conflict detection mechanisms used by optimistic or multi-version concurrency control tend to operate at the row level. This allows concurrent operations on the same table, as long as these access different rows. However, the granularity for conflict detection can be reduced to the attribute level, which would allow conflicting operations on the same table and on the same row, as long as these access different attributes. An additional research path would be to study the implications of reducing concurrency control granularity to the attribute level.

Other directions include, integrate the database modifications into MacroDB to provide improved performance, also to further modify the MacroDB to provide improved fault-tolerance.

Bibliography

[Aba+12] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. "The Design and Implementation of Modern Column-Oriented Database Systems". In: *Foundations and Trends® in Databases* 5.3 (2012), pp. 197–280. ISSN: 1931-7883. DOI: 10.1561/1900000024. URL: http://dx.doi.org/10.1561/1900000024.

[Ady+00] A. Adya, B. Liskov, and P. O'Neil. "Generalized isolation level definitions". In: *Data Engineering, 2000. Proceedings. 16th International Conference on*. 2000, pp. 67–78. DOI: 10.1109/ICDE.2000.839388.

[Agr+87] R. Agrawal, M. J. Carey, and M. Livny. "Concurrency Control Performance Modeling: Alternatives and Implications". In: *ACM Trans. Database Syst.* 12.4 (Nov. 1987), pp. 609–654. ISSN: 0362-5915. DOI: 10.1145/32204.32220. URL: http://doi.acm.org/10.1145/32204.32220.

[AC09] I. Anjo and J. Cachopo. "Jaspex: Speculative parallel execution of java applications". In: *Proceedings of the Simpósio de Informática (INFORUM)*. Lisboa, Portugal, 2009.

[Ast+76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. "System R: Relational Approach to Database Management". In: *ACM Trans. Database Syst.* 1.2 (June 1976), pp. 97–137. ISSN: 0362-5915. DOI: 10.1145/320455.320457. URL: http://doi.acm.org/10.1145/320455.320457.

[Att+11] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. "Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 487–498. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926442. URL: http://doi.acm.org/10.1145/1926385.1926442.

[Avi85] A. Avizienis. "The N-Version Approach to Fault-Tolerant Software". In: *IEEE Trans. Softw. Eng.* 11.12 (Dec. 1985), pp. 1491–1501. ISSN: 0098-5589. DOI:

10.1109/TSE.1985.231893. URL: http://dx.doi.org/10.1109/TSE.1985.231893.

[AK84] A. Avizienis and J. P. J. Kelly. "Fault Tolerance by Design Diversity: Concepts and Experiments". In: *Computer* 17.8 (Aug. 1984), pp. 67–80. ISSN: 0018-9162. DOI: 10.1109/MC.1984.1659219. URL: http://dx.doi.org/10.1109/MC.1984.1659219.

[Bau+09] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. "The Multikernel: A New OS Architecture for Scalable Multicore Systems". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 29–44. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629579. URL: http://doi.acm.org/10.1145/1629575.1629579.

[Ber+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. "A Critique of ANSI SQL Isolation Levels". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. San Jose, California, USA: ACM, 1995, pp. 1–10. ISBN: 0-89791-731-6. DOI: 10.1145/223784.223785. URL: http://doi.acm.org/10.1145/223784.223785.

[BN97] P. Bernstein and E. Newcomer. *Principles of Transaction Processing: For the Systems Professional*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-415-4.

[Ber+87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987. ISBN: 0-201-10715-5.

[Bon+08] P. A. Boncz, M. L. Kersten, and S. Manegold. "Breaking the Memory Wall in MonetDB". In: *Commun. ACM* 51.12 (Dec. 2008), pp. 77–85. ISSN: 0001-0782. DOI: 10.1145/1409360.1409380. URL: http://doi.acm.org/10.1145/1409360.1409380.

[Cam+07] L. Camargos, F. Pedone, and M. Wieloch. "Sprint: A Middleware for High-performance Transaction Processing". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 385–398. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273036. URL: http://doi.acm.org/10.1145/1272996.1273036.

[Car+94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. "Shoring Up Persistent Applications". In: *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*.

SIGMOD '94. Minneapolis, Minnesota, USA: ACM, 1994, pp. 383–394. ISBN: 0-89791-639-5. DOI: 10.1145/191839.191915. URL: http://doi.acm.org/10.1145/191839.191915.

[CL02a] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery". In: vol. 20. 4. New York, NY, USA: ACM, Nov. 2002, pp. 398–461. DOI: 10.1145/571637.571640. URL: http://doi.acm.org/10.1145/571637.571640.

[CL02b] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery". In: *ACM Trans. Comput. Syst.* 20.4 (Nov. 2002), pp. 398–461. ISSN: 0734-2071. DOI: 10.1145/571637.571640. URL: http://doi.acm.org/10.1145/571637.571640.

[Cec+08] E. Cecchet, G. Candea, and A. Ailamaki. "Middleware-based Database Replication: The Gaps Between Theory and Practice". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 739–752. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376691. URL: http://doi.acm.org/10.1145/1376616.1376691.

[Cha+01] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. "Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems". In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 181–190. ISBN: 1-55860-804-4. URL: http://dl.acm.org/citation.cfm?id=645927.672375.

[Cha+81] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. "A History and Evaluation of System R". In: *Commun. ACM* 24.10 (Oct. 1981), pp. 632–646. ISSN: 0001-0782. DOI: 10.1145/358769.358784. URL: http://doi.acm.org/10.1145/358769.358784.

[Che+95] C. Chekuri, W. Hasan, and R. Motwani. "Scheduling Problems in Parallel Query Optimization". In: *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '95. San Jose, California, USA: ACM, 1995, pp. 255–265. ISBN: 0-89791-730-8. DOI: 10.1145/212433.212471. URL: http://doi.acm.org/10.1145/212433.212471.

[CA78] L. Chen and A. Avizienis. "N-version programming: A Fault-tolerance approach to reliability of software operation". In: *Proceedings of FTCS-8*. Toulouse, France: IEEE Computer Society, 1978, pp. 3–9.

[Che+94] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. "RAID: High-performance, Reliable Secondary Storage". In: *ACM Comput. Surv.* 26.2 (June 1994), pp. 145–185. ISSN: 0360-0300. DOI: 10.1145/176979.176981. URL: http://doi.acm.org/10.1145/176979.176981.

[CR08] J. Cieslewicz and K. A. Ross. "Data Partitioning on Chip Multiprocessors". In: *Proceedings of the 4th International Workshop on Data Management on New Hardware*. DaMoN '08. Vancouver, Canada: ACM, 2008, pp. 25–34. ISBN: 978-1-60558-184-2. DOI: 10.1145/1457150.1457156. URL: http://doi.acm.org/10.1145/1457150.1457156.

[Cie+10] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. "Automatic Contention Detection and Amelioration for Data-intensive Operations". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 483–494. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807221. URL: http://doi.acm.org/10.1145/1807167.1807221.

[Cle+13] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 1–17. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522712. URL: http://doi.acm.org/10.1145/2517349.2522712.

[Cod70] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685.

[Com79] D. Comer. "Ubiquitous B-Tree". In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137. ISSN: 0360-0300. DOI: 10.1145/356770.356776. URL: http://doi.acm.org/10.1145/356770.356776.

[CB09] T. M. Connolly and C. E. Begg. *Database Systems: A Practical Approach to Design, Implementation and Management*. 5th. USA: Addison-Wesley Publishing Company, 2009. ISBN: 0321523067.

[CK85] G. P. Copeland and S. N. Khoshafian. "A Decomposition Storage Model". In: *SIGMOD Rec.* 14.4 (May 1985), pp. 268–279. ISSN: 0163-5808. DOI: 10.1145/971699.318923. URL: http://doi.acm.org/10.1145/971699.318923.

[Cou12] T. P. P. Council. *TPC-C on-line transaction processing benchmark*. 2012. URL: http://www.tpc.org/tpcc/.

[Dia+13] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. "Hekaton: SQL Server's Memory-optimized OLTP Engine". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 1243–1254. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2463710. URL: http://doi.acm.org/10.1145/2463676.2463710.

[Dic+13] D. Dice, D. Hendler, and I. Mirsky. "Lightweight Contention Management for Efficient Compare-and-swap Operations". In: *Proceedings of the 19th International Conference on Parallel Processing*. Euro-Par'13. Aachen, Germany: Springer-Verlag, 2013, pp. 595–606. ISBN: 978-3-642-40046-9. DOI: 10.1007/978-3-642-40047-6_60. URL: http://dx.doi.org/10.1007/978-3-642-40047-6_60.

[Esw+76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. "The Notions of Consistency and Predicate Locks in a Database System". In: *Commun. ACM* 19.11 (Nov. 1976), pp. 624–633. ISSN: 0001-0782. DOI: 10.1145/360363.360369. URL: http://doi.acm.org/10.1145/360363.360369.

[F+12] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. "SAP HANA Database: Data Management for Modern Business Applications". In: *SIGMOD Rec.* 40.4 (Jan. 2012), pp. 45–51. ISSN: 0163-5808. DOI: 10.1145/2094114.2094126. URL: http://doi.acm.org/10.1145/2094114.2094126.

[Fre60] E. Fredkin. "Trie Memory". In: *Commun. ACM* 3.9 (Sept. 1960), pp. 490–499. ISSN: 0001-0782. DOI: 10.1145/367390.367400. URL: http://doi.acm.org/10.1145/367390.367400.

[Gar+11] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. "OS Diversity for Intrusion Tolerance: Myth or Reality?" In: *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*. DSN '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 383–394. ISBN: 978-1-4244-9232-9. DOI: 10.1109/DSN.2011.5958251. URL: http://dx.doi.org/10.1109/DSN.2011.5958251.

[Gas+07] I. Gashi, P. Popov, and L. Strigini. "Fault Tolerance via Diversity for Off-the-Shelf Products: A Study with SQL Database Servers". In: *IEEE Trans. Dependable Secur. Comput.* 4.4 (Oct. 2007), pp. 280–294. ISSN: 1545-5971. DOI: 10.1109/TDSC.2007.70208. URL: http://dx.doi.org/10.1109/TDSC.2007.70208.

[Gia+12] G. Giannikis, G. Alonso, and D. Kossmann. "SharedDB: Killing One Thousand Queries with One Stone". In: *Proc. VLDB Endow.* 5.6 (Feb. 2012), pp. 526–537. ISSN: 2150-8097. DOI: 10.14778/2168651.2168654. URL: http://dx.doi.org/10.14778/2168651.2168654.

[Gra03] G. Graefe. "Sorting And Indexing With Partitioned B-Trees". In: *CIDR*. 2003. URL: http://www-db.cs.wisc.edu/cidr/cidr2003/program/p1.pdf.

[Gra+12] G. Graefe, H. Kimura, and H. Kuno. "Foster B-trees". In: *ACM Trans. Database Syst.* 37.3 (Sept. 2012), 17:1–17:29. ISSN: 0362-5915. DOI: 10.1145/2338626.2338630. URL: http://doi.acm.org/10.1145/2338626.2338630.

[Gra+75] J. N. Gray, R. A. Lorie, and G. R. Putzolu. "Granularity of Locks in a Shared Data Base". In: *Proceedings of the 1st International Conference on Very Large Data Bases*. VLDB '75. Framingham, Massachusetts: ACM, 1975, pp. 428–451. ISBN: 978-1-4503-3920-9. DOI: 10.1145/1282480.1282513. URL: http://doi.acm.org/10.1145/1282480.1282513.

[Gra78] J. Gray. "Notes on Data Base Operating Systems". In: *Operating Systems, An Advanced Course*. London, UK, UK: Springer-Verlag, 1978, pp. 393–481. ISBN: 3-540-08755-9. URL: http://dl.acm.org/citation.cfm?id=647433.723863.

[GR92] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601902.

[Gra+96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. "The Dangers of Replication and a Solution". In: *SIGMOD Rec.* 25.2 (June 1996), pp. 173–182. ISSN: 0163-5808. DOI: 10.1145/235968.233330. URL: http://doi.acm.org/10.1145/235968.233330.

[Gro12] T. H. D. Group. *HyperSQL*. 2012. URL: http://hsqldb.org/.

[H212] H2. *H2 Database Engine*. 2012. URL: http://www.h2database.com/html/main.html.

[HR83] T. Haerder and A. Reuter. "Principles of Transaction-oriented Database Recovery". In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: http://doi.acm.org/10.1145/289.291.

[Ham+97] L. Hammond, B. A. Nayfeh, and K. Olukotun. "A Single-Chip Multiprocessor". In: *Computer* 30.9 (1997), pp. 79–85. ISSN: 0018-9162. DOI: http://doi.ieeecomputersociety.org/10.1109/2.612253.

[HL09] W.-S. Han and J. Lee. "Dependency-aware Reordering for Parallelizing Query Optimization in Multi-core CPUs". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09.

Providence, Rhode Island, USA: ACM, 2009, pp. 45–58. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559853. URL: http://doi.acm.org/10.1145/1559845.1559853.

[Har+07] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. "Database Servers on Chip Multiprocessors: Limitations and Opportunities." In: *CIDR*. www.cidrdb.org, 2007, pp. 79–87. URL: http://dblp.uni-trier.de/db/conf/cidr/cidr2007.html#HardavellasPJMAF07.

[Har+08] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. "OLTP Through the Looking Glass, and What We Found There". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 981–992. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376713. URL: http://doi.acm.org/10.1145/1376616.1376713.

[Hel+96] A. A. Helal, B. K. Bhargava, and A. A. Heddaya. *Replication Techniques in Distributed Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 1996. ISBN: 0792398009.

[HM93] M. Herlihy and J. E. B. Moss. "Transactional Memory: Architectural Support for Lock-free Data Structures". In: *SIGARCH Comput. Archit. News* 21.2 (May 1993), pp. 289–300. ISSN: 0163-5964. DOI: 10.1145/173682.165164. URL: http://doi.acm.org/10.1145/173682.165164.

[HS08] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123705916, 9780123705914.

[HW90] M. P. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: http://doi.acm.org/10.1145/78969.78972.

[Hoa74] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept". In: *Commun. ACM* 17.10 (Oct. 1974), pp. 549–557. ISSN: 0001-0782. DOI: 10.1145/355620.361161. URL: http://doi.acm.org/10.1145/355620.361161.

[HR79] H. B. Hunt and D. J. Rosenkrantz. "The Complexity of Testing Predicate Locks". In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. SIGMOD '79. Boston, Massachusetts: ACM, 1979, pp. 127–133. ISBN: 0-89791-001-X. DOI: 10.1145/582095.582115. URL: http://doi.acm.org/10.1145/582095.582115.

[ISDLS92] A. N. S. for Information Systems Database Language SQL. *ANSI X3.135-1992*. 1992.

[Joh+09a] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki. "A New Look at the Roles of Spinning and Blocking". In: *Proceedings of the Fifth International Workshop on Data Management on New Hardware*. DaMoN '09. Providence, Rhode Island: ACM, 2009, pp. 21–26. ISBN: 978-1-60558-701-1. DOI: 10.1145/1565694.1565700. URL: http://doi.acm.org/10.1145/1565694.1565700.

[Joh+09b] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. "Shore-MT: A Scalable Storage Manager for the Multicore Era". In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '09. Saint Petersburg, Russia: ACM, 2009, pp. 24–35. ISBN: 978-1-60558-422-5. DOI: 10.1145/1516360.1516365. URL: http://doi.acm.org/10.1145/1516360.1516365.

[JMHH07] M. S. Joseph M. Hellerstein and J. Hamilton. "Architecture of a Database System". In: *Foundations and Trends® in Databases* 1.2 (2007), pp. 141–259. ISSN: 1931-7883. DOI: 10.1561/1900000002. URL: http://dx.doi.org/10.1561/1900000002.

[Jun+13] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. "A Scalable Lock Manager for Multicores". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 73–84. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465271. URL: http://doi.acm.org/10.1145/2463676.2465271.

[Kal+08] R. Kallman, H. Kimura, J. Natkins, A. Pedro, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. "H-store: A High-performance, Distributed Main Memory Transaction Processing System". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1496–1499. ISSN: 2150-8097. DOI: 10.14778/1454159.1454211. URL: http://dx.doi.org/10.14778/1454159.1454211.

[KA10] B. Kemme and G. Alonso. "Database Replication: A Tale of Research Across Communities". In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 5–12. ISSN: 2150-8097. DOI: 10.14778/1920841.1920847. URL: http://dx.doi.org/10.14778/1920841.1920847.

[Kem+10] B. Kemme, R. Jimenez-Peris, and M. Patino-Martinez. *Database Replication*. Vol. 2. 1. 2010, pp. 1–153. DOI: 10.2200/S00296ED1V01Y201008DTM007. eprint: http://dx.doi.org/10.2200/S00296ED1V01Y201008DTM007. URL: http://dx.doi.org/10.2200/S00296ED1V01Y201008DTM007.

[Kis+12] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. "KISS-Tree: Smart Latch-free In-memory Indexing on Modern Architectures". In: *Proceedings of the Eighth International Workshop on Data Management on New Hardware*.

DaMoN '12. Scottsdale, Arizona: ACM, 2012, pp. 16–23. ISBN: 978-1-4503-1445-9. DOI: 10.1145/2236584.2236587. URL: http://doi.acm.org/10.1145/2236584.2236587.

[Kri+09] K. Krikellas, M. Cintra, and S. Viglas. *Multithreaded query execution on multicore processors*. Tech. rep. The University of Edinburgh School of Informatics, 2009.

[KR79] H. T. Kung and J. T. Robinson. "On Optimistic Methods for Concurrency Control". In: *Proceedings of the Fifth International Conference on Very Large Data Bases - Volume 5*. VLDB '79. Rio de Janeiro, Brazil: VLDB Endowment, 1979, pp. 351–351. URL: http://dl.acm.org/citation.cfm?id=1286711.1286749.

[Lah+13] T. Lahiri, M.-A. Neimat, and S. Folkman. "Oracle TimesTen: An In-Memory Database for Enterprise Applications." In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 6–13.

[Lam78] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[Lam98] L. Lamport. "The Part-time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[Lar+11] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. "High-performance Concurrency Control Mechanisms for Main-memory Databases". In: *Proc. VLDB Endow.* 5.4 (Dec. 2011), pp. 298–309. ISSN: 2150-8097. DOI: 10.14778/2095686.2095689. URL: http://dx.doi.org/10.14778/2095686.2095689.

[Lea00] D. Lea. "A Java Fork/Join Framework". In: *Proceedings of the ACM 2000 Conference on Java Grande*. JAVA '00. San Francisco, California, USA: ACM, 2000, pp. 36–43. ISBN: 1-58113-288-3. DOI: 10.1145/337449.337465. URL: http://doi.acm.org/10.1145/337449.337465.

[LY81] P. L. Lehman and s. B. Yao. "Efficient Locking for Concurrent Operations on B-trees". In: *ACM Trans. Database Syst.* 6.4 (Dec. 1981), pp. 650–670. ISSN: 0362-5915. DOI: 10.1145/319628.319663. URL: http://doi.acm.org/10.1145/319628.319663.

[LC86] T. J. Lehman and M. J. Carey. "A Study of Index Structures for Main Memory Database Management Systems". In: *Proceedings of the 12th International Conference on Very Large Data Bases*. VLDB '86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 294–303. ISBN: 0-934613-18-4. URL: http://dl.acm.org/citation.cfm?id=645913.671312.

[Lev+13a] J. Levandoski, D. Lomet, and S. Sengupta. "LLAMA: A Cache/Storage Subsystem for Modern Hardware". In: *Proc. VLDB Endow.* 6.10 (Aug. 2013), pp. 877–888. ISSN: 2150-8097. DOI: 10.14778/2536206.2536215. URL: http://dx.doi.org/10.14778/2536206.2536215.

[Lev+14] J. Levandoski, D. Lomet, S. Sengupta, A. Birka, and C. Diaconu. "Indexing on Modern Hardware: Hekaton and Beyond". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 717–720. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2594536. URL: http://doi.acm.org/10.1145/2588555.2594536.

[Lev+13b] J. J. Levandoski, S. Sengupta, and W. Redmond. "The BW-Tree: A Latch-Free B-Tree for Log-Structured Flash Storage." In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 56–62.

[Lin+13] J. Lindström, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila. "IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability." In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 14–20.

[Lom+13] D. B. Lomet, S. Sengupta, and J. J. Levandoski. "The Bw-Tree: A B-tree for New Hardware Platforms". In: *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*. ICDE '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 302–313. ISBN: 978-1-4673-4909-3. DOI: 10.1109/ICDE.2013.6544834. URL: http://dx.doi.org/10.1109/ICDE.2013.6544834.

[MAH08] K. Maabreh and A. Al-Hamami. "Increasing database concurrency control based on attribute level locking". In: *Electronic Design, 2008. ICED 2008. International Conference on*. 2008, pp. 1–4. DOI: 10.1109/ICED.2008.4786747.

[Man+09] S. Manegold, M. L. Kersten, and P. Boncz. "Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct". In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1648–1653. ISSN: 2150-8097. DOI: 10.14778/1687553.1687618. URL: http://dx.doi.org/10.14778/1687553.1687618.

[Mao+12] Y. Mao, E. Kohler, and R. T. Morris. "Cache Craftiness for Fast Multicore Key-value Storage". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: ACM, 2012, pp. 183–196. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168855. URL: http://doi.acm.org/10.1145/2168836.2168855.

[Mar+10] P. Mariano, J. Soares, and N. Preguiça. "Replicated Software Components for Improved Performance". In: *InForum2010: Proceedings of InForum 2010*. Universidade do Minho, Sept. 2010.

[Mar+13] H. R. L. Martins, J. Soares, J. M. Lourenço, and N. Preguiça. "Replicação Multi-nível de Bases de Dados em Memória". In: *INForum 2013 — Proceedings of INForum Simpósio de Informática*. INForum. Universidade de Évora, Sept. 2013, pp. 190–201.

[Meh+09] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. "Parallelizing Sequential Applications on Commodity Hardware Using a Low-cost Software Transactional Memory". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 166–176. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542495. URL: http://doi.acm.org/10.1145/1542476.1542495.

[Mic15] Microsoft. *SQL Server*. 2015. URL: http://www.microsoft.com/en-us/server-cloud/products/sql-server/.

[MN09] T. Mishima and H. Nakamura. "Pangea: An Eager Database Replication Middleware Guaranteeing Snapshot Isolation Without Modification of Database Servers". In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 1066–1077. ISSN: 2150-8097. DOI: 10.14778/1687627.1687747. URL: http://dx.doi.org/10.14778/1687627.1687747.

[Moo98] G. Moore. "Cramming More Components Onto Integrated Circuits". In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85. ISSN: 0018-9219. DOI: 10.1109/JPROC.1998.658762.

[Nar+14] N. Narula, C. Cutler, E. Kohler, and R. Morris. "Phase Reconciliation for Contended In-memory Transactions". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 511–524. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685088.

[Niu+13] N. Niu, L. D. Xu, and Z. Bi. "Enterprise Information Systems Architecture-Analysis and Evaluation". In: *Industrial Informatics, IEEE Transactions on* 9.4 (2013), pp. 2147–2154. ISSN: 1551-3203. DOI: 10.1109/TII.2013.2238948.

[Olu+96] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. "The Case for a Single-chip Multiprocessor". In: *SIGOPS Oper. Syst. Rev.* 30.5 (Sept. 1996), pp. 2–11. ISSN: 0163-5980. DOI: 10.1145/248208.237140. URL: http://doi.acm.org/10.1145/248208.237140.

[Pan+10] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. "Data-oriented Transaction Execution". In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 928–939. ISSN: 2150-8097. DOI: 10.14778/1920841.1920959. URL: http://dx.doi.org/10.14778/1920841.1920959.

[Pan+11] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. "PLP: Page Latch-free Shared-everything OLTP". In: *Proc. VLDB Endow.* 4.10 (July 2011), pp. 610–621. ISSN: 2150-8097. DOI: 10.14778/2021017.2021019. URL: http://dx.doi.org/10.14778/2021017.2021019.

[Pan+08] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. "Software Engineering for Multicore Systems: An Experience Report". In: *Proceedings of the 1st International Workshop on Multicore Software Engineering*. IWMSE '08. Leipzig, Germany: ACM, 2008, pp. 53–60. ISBN: 978-1-60558-031-9. DOI: 10.1145/1370082.1370096. URL: http://doi.acm.org/10.1145/1370082.1370096.

[Pap79] C. H. Papadimitriou. "The Serializability of Concurrent Database Updates". In: *J. ACM* 26.4 (Oct. 1979), pp. 631–653. ISSN: 0004-5411. DOI: 10.1145/322154.322158. URL: http://doi.acm.org/10.1145/322154.322158.

[Pap+08] K. Papadopoulos, K. Stavrou, and P. Trancoso. "HelperCoreDB: Exploiting multicore technology to improve database performance". In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. 2008, pp. 1–11. DOI: 10.1109/IPDPS.2008.4536288.

[PA04] C. Plattner and G. Alonso. "Ganymed: Scalable Replication for Transactional Web Applications". In: *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*. Middleware '04. Toronto, Canada: Springer-Verlag New York, Inc., 2004, pp. 155–174. ISBN: 3-540-23428-4. URL: http://dl.acm.org/citation.cfm?id=1045658.1045671.

[Pla09] H. Plattner. "A Common Database Approach for OLTP and OLAP Using an In-memory Column Database". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 1–2. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559846. URL: http://doi.acm.org/10.1145/1559845.1559846.

[Por+15] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. "Designing Distributed Systems Using Approximate Synchrony in Data Center Networks". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Oakland, CA: USENIX Association, 2015, pp. 43–57. ISBN: 978-1-931971-218. URL: http://dl.acm.org/citation.cfm?id=2789770.2789774.

[Pre+08] N. Preguiça, R. Rodrigues, C. a. Honorato, and J. a. Lourenço. "Byzantium: Byzantine-fault-tolerant database replication providing snapshot isolation". In: *Proceedings of the Fourth conference on Hot topics in system dependability*. HotDep'08. San Diego, California: USENIX Association, 2008, pp. 9–9.

[Pro+13] A. Prokopec, P. Bagwell, and M. Odersky. "Lock-Free Resizeable Concurrent Tries". English. In: *Languages and Compilers for Parallel Computing*. Ed. by S. Rajopadhye and M. Mills Strout. Vol. 7146. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 156–170. ISBN: 978-3-642-36035-0. DOI: 10.1007/978-3-642-36036-7_11. URL: http://dx.doi.org/10.1007/978-3-642-36036-7_11.

[RR99] J. Rao and K. A. Ross. "Cache Conscious Indexing for Decision-Support in Main Memory". In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 78–89. ISBN: 1-55860-615-7. URL: http://dl.acm.org/citation.cfm?id=645925.671362.

[RR00] J. Rao and K. A. Ross. "Making B+- Trees Cache Conscious in Main Memory". In: *SIGMOD Rec.* 29.2 (May 2000), pp. 475–486. ISSN: 0163-5808. DOI: 10.1145/335191.335449. URL: http://doi.acm.org/10.1145/335191.335449.

[Ren+12] K. Ren, A. Thomson, and D. J. Abadi. "Lightweight Locking for Main Memory Database Systems". In: *Proc. VLDB Endow.* 6.2 (Dec. 2012), pp. 145–156. ISSN: 2150-8097. DOI: 10.14778/2535568.2448947. URL: http://dx.doi.org/10.14778/2535568.2448947.

[RS77] D. R. Ries and M. Stonebraker. "Effects of Locking Granularity in a Database Management System". In: *ACM Trans. Database Syst.* 2.3 (Sept. 1977), pp. 233–246. ISSN: 0362-5915. DOI: 10.1145/320557.320566. URL: http://doi.acm.org/10.1145/320557.320566.

[Rod+01] R. Rodrigues, M. Castro, and B. Liskov. "BASE: Using Abstraction to Improve Fault Tolerance". In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP '01. Banff, Alberta, Canada: ACM, 2001, pp. 15–28. ISBN: 1-58113-389-8. DOI: 10.1145/502034.502037. URL: http://doi.acm.org/10.1145/502034.502037.

[RO92] M. Rosenblum and J. K. Ousterhout. "The Design and Implementation of a Log-structured File System". In: *ACM Trans. Comput. Syst.* 10.1 (Feb. 1992), pp. 26–52. ISSN: 0734-2071. DOI: 10.1145/146941.146943. URL: http://doi.acm.org/10.1145/146941.146943.

[Ros+78] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. "System Level Concurrency Control for Distributed Database Systems". In: *ACM Trans. Database Syst.* 3.2 (June 1978), pp. 178–198. ISSN: 0362-5915. DOI: 10.1145/320251.320260. URL: http://doi.acm.org/10.1145/320251.320260.

[Sal+11] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. "Database Engines on Multicores, Why Parallelize when You Can Distribute?" In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: ACM, 2011, pp. 17–30. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966448. URL: http://doi.acm.org/10.1145/1966445.1966448.

[Sch90] F. B. Schneider. "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167. URL: http://doi.acm.org/10.1145/98163.98167.

[Sew+11] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. "PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors". In: *Proc. VLDB Endow.* 4.11 (Aug. 2011), pp. 795–806. ISSN: 2150-8097.

[Sik+12] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. "Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, pp. 731–742. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213946. URL: http://doi.acm.org/10.1145/2213836.2213946.

[Sil+06] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. 5th ed. New York, NY, USA: McGraw-Hill, Inc., 2006. ISBN: 0072958863, 9780072958867.

[SZ97] A. Silberschatz and S. Zdonik. "Database Systems-Breaking out of the Box". In: *SIGMOD Rec.* 26.3 (Sept. 1997), pp. 36–50. ISSN: 0163-5808. DOI: 10.1145/262762.262768. URL: http://doi.acm.org/10.1145/262762.262768.

[Sil+91] "Database Systems: Achievements and Opportunities". In: *Commun. ACM* 34.10 (Oct. 1991). Ed. by A. Silberschatz, M. Stonebraker, and J. Ullman, pp. 110–120. ISSN: 0001-0782. DOI: 10.1145/125223.125272. URL: http://doi.acm.org/10.1145/125223.125272.

[SP15] J. a. Soares and N. Preguiça. "Database Engines on Multicores Scale: A Practical Approach". In: *Proceedings of the 30th ACM/SIGAPP Symposium On Applied Computing (SAC 2015)*. Salamanca, Spain: ACM, 2015.

[Soa+13a] J. a. Soares, J. a. Lourenço, and N. Preguiça. "MacroDB: Scaling Database Engines on Multicores". In: *Proceedings of the 19th International Conference on Parallel Processing*. Euro-Par'13. Aachen, Germany: Springer-Verlag, 2013, pp. 607–619. ISBN: 978-3-642-40046-9. DOI: 10.1007/978-3-642-40047-6_61. URL: http://dx.doi.org/10.1007/978-3-642-40047-6_61.

[SP12] J. Soares and N. Preguiça. "Improving Application Fault-Tolerance with Diverse Component Replication". In: *1st Euro-TM Workshop on Transactional Memory (WTM 2012)*. 2012.

[Soa+13b] J. Soares, J. Lourenço, and N. Preguiça. "Software Component Replication for Improved Fault-Tolerance: Can Multicore Processors Make It Work?" English. In: *Dependable Computing*. Ed. by M. Vieira and J. Cunha. Vol. 7869. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 173–180. ISBN: 978-3-642-38788-3. DOI: 10.1007/978-3-642-38789-0_15. URL: http://dx.doi.org/10.1007/978-3-642-38789-0_15.

[Son+11] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. "A Case for Scaling Applications to Many-core with OS Clustering". In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: ACM, 2011, pp. 61–76. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966452. URL: http://doi.acm.org/10.1145/1966445.1966452.

[Sto81] M. Stonebraker. "Operating System Support for Database Management". In: *Commun. ACM* 24.7 (July 1981), pp. 412–418. ISSN: 0001-0782. DOI: 10.1145/358699.358703. URL: http://doi.acm.org/10.1145/358699.358703.

[SW13] M. Stonebraker and A. Weisberg. "The VoltDB Main Memory DBMS". In: *Bulletin of the Technical Committee on Data Engineering* 36.2 (June 2013).

[Sto+86] M. Stonebraker, P. Kreps, E. Wong, and G. Held. "The INGRES Papers: Anatomy of a Relational Database System". In: ed. by M. Stonebraker. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. Chap. The Design and Implementation of INGRES, pp. 5–45. ISBN: 0-201-07185-1. URL: http://dl.acm.org/citation.cfm?id=4161.4162.

[Sto+07] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. "The End of an Architectural Era: (It's Time for a Complete Rewrite)". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 1150–1160. ISBN: 978-1-59593-649-3. URL: http://dl.acm.org/citation.cfm?id=1325851.1325981.

[Sto+05] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. "C-store: A Column-oriented DBMS". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 553–564. ISBN: 1-59593-154-6. URL: http://dl.acm.org/citation.cfm?id=1083592.1083658.

[Sut05] H. Sutter. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* 30.3 (2005).

[Tho+12] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. "Calvin: Fast Distributed Transactions for Partitioned Database Systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, pp. 1–12. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213838. URL: http://doi.acm.org/10.1145/2213836.2213838.

[Tiw+10] D. Tiwari, S. Lee, J. Tuck, and Y. Solihin. "MMT: Exploiting fine-grained parallelism in dynamic memory management". In: *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470428. URL: http://dx.doi.org/10.1109/IPDPS.2010.5470428.

[Tu+13] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. "Speedy Transactions in Multicore In-memory Databases". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 18–32. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522713. URL: http://doi.acm.org/10.1145/2517349.2522713.

[Tul+95] D. M. Tullsen, S. J. Eggers, and H. M. Levy. "Simultaneous Multithreading: Maximizing On-chip Parallelism". In: *SIGARCH Comput. Archit. News* 23.2 (May 1995), pp. 392–403. ISSN: 0163-5964. DOI: 10.1145/225830.224449. URL: http://doi.acm.org/10.1145/225830.224449.

[Unt+09] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. "Predictable Performance for Unpredictable Workloads". In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 706–717. ISSN: 2150-8097. DOI: 10.14778/1687627.1687707. URL: http://dx.doi.org/10.14778/1687627.1687707.

[Vac+05] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, and D. Connors. "Chip Multi-processor Scalability for Single-threaded Applications". In: *SIGARCH Comput. Archit. News* 33.4 (Nov. 2005), pp. 44–53. ISSN: 0163-5964. DOI: 10.1145/1105734.1105741. URL: http://doi.acm.org/10.1145/1105734.1105741.

[Val90] L. G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782. DOI: 10.1145/79173.79181. URL: http://doi.acm.org/10.1145/79173.79181.

[VDG74] J. Van Doren and J. Gray. "An Algorithm for Maintaining Dynamic AVL Trees". English. In: *Information Systems*. Ed. by J. Tou. Springer US, 1974, pp. 161–180. ISBN: 978-1-4684-2696-0. DOI: 10.1007/978-1-4684-2694-6_8. URL: http://dx.doi.org/10.1007/978-1-4684-2694-6_8.

[Van+07] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. "Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 59–72. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294268. URL: http://doi.acm.org/10.1145/1294261.1294268.

[Wam+13] J.-T. Wamhoff, C. Fetzer, P. Felber, E. Rivière, and G. Muller. "FastLane: Improving Performance of Software Transactional Memory for Low Thread Counts". In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '13. Shenzhen, China: ACM, 2013, pp. 113–122. ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442528. URL: http://doi.acm.org/10.1145/2442516.2442528.

[WS05] M. Wiesmann and A. Schiper. "Comparison of database replication techniques based on total order broadcast". In: *Knowledge and Data Engineering, IEEE Transactions on* 17.4 (2005), pp. 551–566. ISSN: 1041-4347. DOI: 10.1109/TKDE.2005.54.

[Wie+00] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. "Database Replication Techniques: A Three Parameter Classification". In: *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*. SRDS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 206–. ISBN: 0-7695-0543-0. URL: http://dl.acm.org/citation.cfm?id=829525.831077.

[Xu11] L. D. Xu. "Enterprise Systems: State-of-the-Art and Future Trends". In: *Industrial Informatics, IEEE Transactions on* 7.4 (2011), pp. 630–640. ISSN: 1551-3203. DOI: 10.1109/TII.2011.2167156.

[Ye+11] Y. Ye, K. A. Ross, and N. Vesdapunt. "Scalable Aggregation on Multicore Processors". In: *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. DaMoN '11. Athens, Greece: ACM, 2011, pp. 1–9. ISBN: 978-1-4503-0658-4. DOI: 10.1145/1995441.1995442. URL: http://doi.acm.org/10.1145/1995441.1995442.

[Zha+15] H. Zhang, G. Chen, B. Ooi, K. Tan, and M. Zhang. "In-Memory Big Data Management and Processing: A Survey". In: *Knowledge and Data Engineering, IEEE Transactions on* 27.7 (2015), pp. 1920–1948. ISSN: 1041-4347. DOI: 10.1109/TKDE.2015.2427795.

[Zha+07] L. Zhang, C. Krintz, and P. Nagpurkar. "Language and Virtual Machine Support for Efficient Fine-Grained Futures in Java". In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. PACT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 130–139. ISBN: 0-7695-2944-5. DOI: 10.1109/PACT.2007.45. URL: http://dx.doi.org/10.1109/PACT.2007.45.

[Zha+13] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. "Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 276–291. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522729. URL: http://doi.acm.org/10.1145/2517349.2522729.

[Zho+05] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. "Improving Database Performance on Simultaneous Multithreading Processors". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 49–60. ISBN: 1-59593-154-6. URL: http://dl.acm.org/citation.cfm?id=1083592.1083602.