Multicore architectures as platform to extend database engine functionality

A dissertation submitted to the
ETH Zurich

for the degree of
Doctor of Sciences

presented by
IONUT-EMANUEL SUBASU
Dipl. Ing. in Computer Science - Technical University of Cluj-Napoca
born 16 May 1982
citizen of Romania

accepted on the recommendation of:
Prof. Dr. Gustavo Alonso, examiner
Prof. Dr. Donald Kossman, co-examiner
Prof. Dr. Moira Norrie, co-examiner
Prof. Dr. Michael Böhlen, co-examiner

2012
Acknowledgments

First of all I would like to thank my supervisor Prof. Dr. Gustavo Alonso. Without his constant support and guidance this thesis would not have been possible. I admire his talent to lead the group and the way he always manages to find time for everybody. I have learned many things about research while working with him.

I am grateful to the examining committee, Prof. Dr. Donald Kossman, Prof. Dr. Moira Norrie and Prof. Dr. Michael Böhlen, for their valuable feedback and advice in writing this thesis.

I would also like to thank all my colleagues from the Systems Group for all the great discussions and help. Special thanks go to Tudor and Jana for the long nights and weekends spent while working at the Multimed System – I think we have built quite a good team. My coffee mates Alex, Adrian, Michael, Nihal, Patrick, Thomas and Simon L. for their friendliness, inspiration and the great working atmosphere. Jena, Eva and Simonetta for all their support, great trips and retreats organized during the years.

I will always remember Prof. Dr. Klaus Dittrich and my colleagues from the in the DBTG Group at and the great people from IFI, University of Zurich.

Thanks also to all my friends for their encouragement, great discussions on various topics, and for being there when I needed them.

Last but not least I would like to thank Florica for her love and patience. She encouraged me and managed to get along with me when I was stressed and in a bad mood. Many thanks go to my family which guided me during the years and always provided moral support.
Abstract

This thesis is centered around the topic of database engines running on multicores. For the last years, the increasing processor frequencies inherently improved the performance of existing software. As the current trend in processor technology moves towards an increased number of cores in future multicore CPUs, software used in infrastructure systems such as operating systems and databases face a substantial challenge. Further improvement in performance stands in the ability to exploit the new resources available in modern multicore processors. Such software typically evolves slower than the underlying hardware, and with multicore it faces structural limitations that can be solved only with radical architectural changes.

Research in operating systems suggests that multicore architectures should be treated as distributed systems rather than trying to hide the parallel nature of the hardware. This thesis proposes to use a similar approach in the case of database engines and take advantage of existing work on cluster-based replication systems.

We start with an overview of current challenges introduced by modern multicore processors to software used in infrastructure systems. We discuss how recent research from operating systems and databases, has shown that the main performance deterrents are locking, memory locality, load interaction, and the extensive use of shared data structures. We analyze the limitations of database engines when running on multicores using MySQL and PostgreSQL as examples.

Based on the related work and results of our analysis, we notice that the effects from sharing and locking can be reduced by using multiple database replicas. For this reason, we have developed a system, Multimed, used to deploy a cluster-based database replication system on a single multicore machine. We, show how to deploy several replicated engines within a single multicore machine to achieve better scalability and stability than a single database engine operating on all cores.

The proposed system can be extended to manage different kind of situations and specializations that lead to increased stability and performance of the system. We
further extend the system to support extensions to the existing database engine, such as new operators or specialized indexes. These extensions are deployed as specialized replicas that increase the initial engine’s robustness and flexibility.

This thesis proposes to use the multicore architecture as a platform to deploy a set of specialized replicas to improve the performance of the initial database engine and extend its functionality with specialized functionality that was not supported by the initial database engine. The resulting system avoids a complete redesign of the database engine while providing a low overhead alternative with significant performance gains for an important class of workloads.

The evaluation of the system shows that by using Multimed we are able to improve the performance of existing database engines when running on multicores. Deploying a set of replicas inside a single multicore machine replicates the shared data structures and synchronization primitives while distributing the load reduces the contention over the system. Extending the system with optimized operators and specializing satellites to improve performance of long running queries increases memory locality and removes interaction with concurrent queries in the system.
Zusammenfassung


Forschungsarbeiten im Bereich Betriebssysteme schlagen vor, Multikernprozessoren eher als verteilte Systeme zu behandeln statt zu versuchen, die parallele Struktur der Hardware zu überdecken. Diese Arbeit schlägt einen ähnlichen Ansatz in Bezug auf Datenbanksysteme vor, wobei die Ergebnisse von bereits vorhandenen Arbeiten zu clusterbasierten Replikationssystemen genutzt werden.


Basierend auf den Ergebnissen bereits vorhandener Arbeiten und eigenen Analysen stellen wir fest, dass die Effekte von geteilten Datenstrukturen und Sperren verringert werden können, indem mehrere Datenbankreplikate genutzt werden. Zu diesem Zweck entwickelten wir ein System, Multimed, das ein clusterbasiertes Datenbankreplikation-
system auf einem Mehrkernprozessorsystem einsetzen kann. Schliesslich zeigen wir, wie man einzelne replizierte Datenbankmanagementsysteme auf einem Mehrkernprozessorsystem einsetzen kann, um bessere Skalierbarkeits- und Stabilitätswerte zu erhalten als mit nur einem einzelnen Datenbankmanagementsystem das auf allen Kernen arbeitet.

Das vorgeschlagene System kann erweitert und seine Stabilität und Performance erhöht werden, um verschiedenen Situationen und Spezialisierungen gerecht zu werden. Weiterhin bauen wir das System so aus, dass Erweiterungen am vorhandenen Datenbankmanagementsystem, wie neue Operatoren, spezialisierte Indizes und spezialisierte Replikate unterstützt werden, die schliesslich die Robustheit und die Flexibilität des ursprünglichen Systems erhöhen.


### Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1SR</td>
<td>One-Copy Serializability</td>
<td>25</td>
</tr>
<tr>
<td>2PC</td>
<td>Two-Phase Commit</td>
<td>37</td>
</tr>
<tr>
<td>2PL</td>
<td>Two-Phase Locking</td>
<td>38</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
<td>29</td>
</tr>
<tr>
<td>CMP</td>
<td>Chip Multiprocessor</td>
<td>2</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
<td>2</td>
</tr>
<tr>
<td>HT</td>
<td>Hyper Threading</td>
<td>13</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
<td>16</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
<td>12</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
<td>33</td>
</tr>
<tr>
<td>M-CMP</td>
<td>Multiple Chip Multiprocessor</td>
<td>15</td>
</tr>
<tr>
<td>NDB</td>
<td>Network Database</td>
<td>37</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-uniform Memory Access</td>
<td>45</td>
</tr>
<tr>
<td>OLAP</td>
<td>Online Analytical Processing</td>
<td>19</td>
</tr>
<tr>
<td>OLTP</td>
<td>Online Transaction Processing</td>
<td>19</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
<td>1</td>
</tr>
<tr>
<td>Perm</td>
<td>Provenance Extension of the Relational Model</td>
<td>130</td>
</tr>
<tr>
<td>RAC</td>
<td>Real Application Clusters</td>
<td>37</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
<td>11</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
<td>1</td>
</tr>
<tr>
<td>RSI-PC</td>
<td>Remote Snapshot Isolation with Primary Copy</td>
<td>22</td>
</tr>
<tr>
<td>RT</td>
<td>Response Time</td>
<td>47</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>SAN</td>
<td>Storage Area Network</td>
<td>37</td>
</tr>
<tr>
<td>SCN</td>
<td>Snapshot Count Number</td>
<td>75</td>
</tr>
<tr>
<td>SI</td>
<td>Snapshot Isolation</td>
<td>5</td>
</tr>
<tr>
<td>SMT</td>
<td>Simultaneous Multi-threading</td>
<td>2</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
<td></td>
</tr>
<tr>
<td>WAL</td>
<td>Write Ahead Log</td>
<td>35</td>
</tr>
<tr>
<td>WIPS</td>
<td>Web Interactions per Second</td>
<td>43</td>
</tr>
</tbody>
</table>
## Contents

1 Introduction .................................................. 1
   1.1 Problem Statement ........................................... 2
   1.2 Summary and Contributions .................................. 4
   1.3 Thesis Structure ............................................. 6

2 Background .................................................... 9
   2.1 Hitting the Memory Wall ...................................... 10
   2.2 Towards Multicore Architectures ............................ 12
      2.2.1 Challenges Introduced by Multicore Architectures ...... 14
      2.2.2 Designing Systems for Multicores ....................... 16
   2.3 Database Systems on Multicores ............................. 18
   2.4 Snapshot Isolation .......................................... 22
      2.4.1 Overview ............................................... 23
      2.4.2 Systems using SI ....................................... 24
   2.5 Replication ................................................ 25
      2.5.1 Overview of Replication Systems ......................... 25
      2.5.2 Architecture of Replication Systems .................... 29
   2.6 RSI-PC .................................................. 30
   2.7 Real Replication Systems .................................... 31
      2.7.1 Research Replication Systems ............................ 32
      2.7.2 Open-Source Replication Systems ....................... 35
Relational Database Management Systems (RDBMSs) have played an important role in business applications for more than 20 years [75]. With time they became a center piece of current systems and had to provide more features and to support a variety of workloads. As a consequence database performance started to play an important role in today’s software used in infrastructure systems. At the same time database engines needed to enable access to more users while maintaining the consistency of fast changing data.

The performance of database engines improved in the last years due to the higher clock frequencies offered by processors with every new generation. It is now widely accepted that the new shift towards multicore processors poses new challenges for database architectures and design.

The next section introduces some of the main problems imposed by current multicore processors to current infrastructure software such as Operating Systems (OSs) and database engines and mentions some of the current approaches and solutions. Section 1.2 presents a short summary and the main contributions of this thesis.
Chapter 1. Introduction

1.1 Problem Statement

Until recently, every year introduced higher processor speeds with improved single thread performance and higher instruction level parallelism[1, 93, 94]. In 2002 the processors reached the 3GHz limit and the high clock frequency created heat dissipation problems and high energy consumption. These issues have set a cap on future evolution of processor frequencies [1, 8].

After four decades, Moore’s Law [59] is still valid and the number of transistors within a single chip doubles every couple of years. To further improve performance of current processors, the manufactures came up with the solution to add more processing cores on a single die [89, 92]. As a consequence, processor technology is shifting towards Simultaneous Multi-threading (SMT) and Chip Multiprocessors (CMPs) as an alternative to highly complex single stream Central Processing Units (CPUs) [8]. Producers leave the race for speed and focus on increased parallelism. Analyzing the processor evolution in the last years we notice that the number of cores on a processor doubles almost every second year, and the research community predicts that the number will continue to increase [16].

Increasing clock frequencies, in single core processors, inherently improved the performance of existing software. Current advances in processor technology go towards increased degrees of parallelism. Taking advantage of new processors and the increased degree of parallelism requires architectural changes and redesign of existing applications. Infrastructure software such as OSs, web servers or database engines are the first ones to suffer from the scalability problems due to complex multicore architectures.

The operating systems research community has a long history on improving scalability on Unix-like OSs using shared-memory multiprocessors. It is now widely accepted that these systems scale poorly due to coarse-grained locks. The large number of cores introduces more parallelism due to increased contention over the shared data structures.

The complex nature of multicore architectures drives new OS designs to focus on improving scalability by introducing new concepts and technologies. Current research proposals start to treat multicore architectures as distributed systems and use shared-memory for fast message passing [13]. This transition introduced new systems that aim towards improved data locality and load balancing of tasks between cores [13, 19]. Meanwhile new solutions such as distributed memory management, scalable locking, wait free synchronization [49], and multiprocessor schedulers have already made their
1.1. Problem Statement

Way into new Linux kernels [19].

Starting from early days, the focus in database engine research has been driven towards performance and reliability under heavy loads. On the one hand, reliability and consistency is usually achieved using locking mechanisms on shared data structures. On the other hand, performance on current multicore architectures is achieved by exploiting the higher degree of parallelism. Trying to get the best out of these two worlds poses a significant challenge. Most relational database engines are based on old designs optimized for disk I/O bottlenecks and designed to run on single stream processors.

Database systems rely on concurrency to achieve high throughput. Concurrency is achieved through threads and/or processes and ideally they should scale well with the increased number of hardware contexts, especially in read-mostly workloads. Practically, to maintain data consistency and durability, database engines make extensive use of shared data structures and synchronization primitives. The extended use of shared read-write data structures to ensure that updates are performed atomically ends up producing a large synchronization overhead. As a consequence, in a highly parallel environment, atomic updates to a frequent value can lead to serialization of all threads [26].

Due to the large number of hardware contexts, parallel threads end up competing for resources, having as effect an increased number of cache misses, added latency of cache coherence protocols and main memory access. Current strategies for choosing the right locking strategies, such as blocking versus spinning are no longer valid [51]. The increased level of parallelism leads to a more heterogeneous workload, putting more pressure on read-write synchronization primitives in index-based data structures [88]. All these concerns present performance bottlenecks for current database applications running on multicore[97]. In conclusion, locking primitives [51], shared data structures and the read-write contention in index structures [88] are some of the main scalability problems of databases on multicores. As a result the research community has proposed lots of different approaches and solutions to redesign the engine or create new architectures. Systems that use pure main memory scans [88], use helper cores to pre-fetch the data and avoid cache misses [69], modularize the engine into a sequence of stages that work independently as a self-contained set of components, improving data locality or removing contention from within the storage engine [67, 53] have appeared and have been shown to perform well on current multicore systems.

Even if all the performance bottlenecks have been sorted out, and a way to exploit parallelism has been found for the problem at hand, parallelizing database operations
to a high degree, supporting an ever increasing number of cores is not a trivial problem. The concept of parallel programming is very complex and comes with new problems such as race conditions and deadlocks that are not obvious to debug and appear nondeterministically.

The increased number of resources (hardware contexts and memory) should allow database systems to support more complex workloads. However, the abundance of resources and the increased amount of sharing leads to higher contention on shared data structures which in turn leads to performance degradation due to updates and scans from parallel transactions [88, 75].

In spite of the intense research in the area, improving performance of relational database engines running on current multicore architectures proves to be a challenging task. While current proposals imply radical changes to the database storage engine, there are a few practical solutions out there that allow a flexible deployment of databases over multicores. The work in this thesis proposes an alternative solution to improve performance of relational database deployed on modern multicore architectures. We propose to use the multicore architecture as a platform to improve the performance of existing database engines. The solution relies on well-known distributed systems techniques and proves to be widely applicable for many workloads without requiring changes to the database engine. Due to the distributed nature of the system, it can be extended and specialized to improve or add extra functionality to the existing database engine. Designed for multicores, our approach allows deployment of fine-tuned configurations within a single multicore machine. The flexible configuration aims to achieve the best mapping between the system’s components and the existing hardware and computational resources.

1.2 Summary and Contributions

Inspired by recent work presented in modern operating systems [11, 94, 18] and database systems designed for multicores [69, 53, 88, 88], this thesis proposes to treat multicore systems in the same way we treat distributed systems, and deploy the database as a collection of distributed replicas coordinated through a middleware layer that manages consistency, load balancing and query routing. In other words, rather than opting for redesigning the engine, the thesis proposes to partition the multicore machine and allocate an unmodified database engine to each partition.

The resulting system, Multimed and presented in [79], is based on techniques used
in LANs as part of computer clusters. Inspired from the Ganymed system [74], and designed to run on multicore machines, Multimed uses a primary copy (master database) and multiple replicas (satellite databases) where each one runs on a subset of resources (CPU cores, memory). The master database receives the entire update load and asynchronously propagates the changes to satellite databases. The satellites store copies of the database and run on non-overlapping subsets of cores. These satellites are kept in sync with the master copy (with some latency) and are used to execute the read-only load (queries). The system guarantees global consistency in the form of Snapshot Isolation (SI) and can be extended to enable alternative consistency guarantees.

As in the case of cluster replication systems, this approach provides a practical solution to scale up performance of existing database engines on multicore without changing the engine’s architecture. Moreover, the Multimed system allows deployment of specialized satellites that are finely tuned to execute specific queries enabling a seamless extension of the original engine with new functionality.

The Multimed project aims to provide a solution to scale existing database engines on multicore systems without major changes to the database engine. Usually software evolves at a slower pace than the underlying hardware; the project goals are directed towards finding practical solutions that enable existing software to exploit the abundance of resources offered by current multicore architectures. The initial evaluation and implementation of Multimed [79], was joint work in the Systems Group at ETH Zurich. The first prototype analyzed the applicability of our approach in a transactional workload using PostgreSQL with small database sizes. I designed the analytical model of the system and used the insights derived from the model to improve Multimed configurations and deployments. I broadened the system’s applicability and created new configurations to cover complex analytical workloads. From this point, I extended and enabled the system to support other database engines. I expanded the performance analysis and fine-tuned the system to cover larger database sizes and explored wider range of system characteristics and configurations. Another contribution to this project is the added support for extensions and specialized satellites. I have deployed and analyzed the applicability of the approach in the context of improving the robustness of complex workloads.

The proposed approach aims to achieve the following goals:

- **Improve the performance and scalability of database engines running on multicore**. One of the main goals of this thesis is to show that a shared nothing design, commonly used in cluster-based systems can be successfully applied to
multicore machines. The thesis is going to present experiments and evaluations on two of the most popular open source relational database engines PostgreSQL and MySQL running on multicores.

- **Extend the functionality of existing database engines.** Taking advantage of the increased parallelism and available resources enables our system to run specialized satellites, optimized for complex queries and add new functionality that is not supported by the initial engine. Further on, these extensions can add new specialized algorithms to improve the performance of specific queries such as Skyline Operator [17], Data Provenance [37], Keyword Search and Ranking Operators enabling a seamless integration without any changes to the initial engine. As a result the system has an extended functionality as well as improved performance.

- **Improve robustness of complex workloads** by separating long running queries that have a disruptive effect over the existing workload. Long running queries create contention over shared resources and limit the scalability properties of traditional database engines. Our approach aims at performance improvements due to specialized satellites, load separation and increased memory locality.

- **No changes to the underlying engine architecture.** The thesis proposes to exploit the parallelism offered by multicore systems by combining the performance of multiple unmodified database engines running in a single multicore machine rather than through optimizing a single engine to run on multiple cores.

Like any replication-based system, the approach presented in this thesis is not suitable for all possible use cases, but it can support a wide range of workloads. This approach performs best for data warehousing and business intelligent workloads, while write intensive workloads suffer a minor loss in performance due to the replicated nature of the system.

### 1.3 Thesis Structure

The next chapter presents and discusses related work from different research fields and projects that have a close connection with the work presented in this thesis. First we describe the current architectural trends in modern CPUs, and present some of the main problems introduced by existing multicore architectures. We continue with a short overview of known scalability problems in operating systems and database engines running on multicores. As we argue that, in the case of database systems, multicore
1.3. Thesis Structure

Architectures should be treated as distributed systems; this chapter discusses different replication systems and techniques and motivates which one is more suitable in the context of multicores. Chapter 3 presents a performance evaluation of database engines running on multicores and analyzes the problems introduced by current multicore architectures to existing database engines. Chapter 4 introduces the Multimed system and describes its main components. The next chapter evaluates the system’s ability to improve database scalability on multicores. Chapter 6 takes the idea one step further and uses Multimed to extend the functionality of database engines with specialized satellites, that add new functionality not supported by the initial database engine. Finally, Chapter 7 concludes the thesis.
The work presented in this thesis is based on concepts and ideas from different research areas such as operating systems, database systems, distributed systems and applies them in the context of multicores. This chapter presents work that contributed to the development of the system presented in this thesis.

We argue in this thesis that current architectural changes in modern multicore CPUs affect existing applications in many ways and existing assumptions about memory locality and concurrency no longer apply but rather lead to performance degradation. To understand why data locality is important the next section gives a short overview of the memory architecture. Section 2.2 explains the transition from single stream CPUs to multicore architectures and presents some of the challenges introduced by current multicore CPUs.

We continue with a presentation of related work on database engines and operating systems designed to run on multicore machines. We also describe other systems and related work that either contributed to our efforts or can be seen as a parallel line of work.
Chapter 2. Background

2.1 Hitting the Memory Wall

Memory in current CPU architectures is organized in layers. Each layer has the role of reducing access latency by mirroring a part of the main memory. It is not possible to have fast memory of large sizes, due to the costs and technological limitations. This leads to a layered organization of the memory architecture where each layer has specific characteristics. Figure 2.1 shows the typical memory hierarchy with typical latency values and data sizes. The layers between the CPU and the storage represent the memory hierarchy.

![Figure 2.1: Memory hierarchy](image)

Optimizing memory accesses plays an important role in database engines and achieving efficient data access with low latency plays an important role in obtaining high throughput rates.

Memory access is in the order of nanoseconds while disk accesses are in the order of milliseconds, making the system I/O bound and resulting in CPU stalling while waiting for data. For database engines to achieve a high throughput, one must reduce data access latency. This requires the introduction of a memory level between the CPU and the disk to ensure low latency access to the frequently accessed data.

To better understand the challenges of designing software for multicore architectures, we have to take a look at the evolution of the memory hierarchy of these systems. In recent years, CPUs architectures have evolved faster than memory systems widening the memory-gap. The increasing amount of parallel threads, enable more work to be executed in the same time and puts more pressure on the memory and I/O bus.
First generations of CPUs used only on chip registers to buffer frequently accessed data and compensate for main memory or disk access latency. When CPU frequencies started to increase by 75% per year, memory speeds could not keep up and introduced an imbalance between memory bandwidth and CPU speeds [56]. On the one hand, newer CPUs required larger memory sizes since they could process more data in shorter times. On the other hand, the increased memory size added access latency, making the CPU hit the “memory wall” [96].

In an attempt to bridge the memory gap, a small memory of a few kB called the cache, was added to the CPUs to keep the most frequently accessed instructions and data. This new layer of memory allowed fast access to regularly accessed data and alleviated the memory bottleneck. The cache memory is located on the chip and enables fast access with low latency to frequently accessed data due to the close proximity to the CPU. To get a better understanding of the problem, let us take a look at the equation of the average memory access time [96].

\[
T = T_c \times P_c + (1 - P_c) \times T_{mem} \tag{2.1}
\]

where \( T \) is the average access time, \( T_c \) and \( T_{mem} \) are the cache and memory access times and \( P_c \) is the probability of a cache hit.

Best performance is achieved when the average access time is minimal. To get the most out of the cache, we need to obtain high cache hit ratios and low latency access times on the cache memory, so that \((1 - P_c)\) will tend towards 0. If we consider the average access time to the cache memory at 4\( ns \) and the time to access main memory at 100\( ns \) and compute the equation, we notice that in the case of a 96% cache hit ratio 
\[
T_{avg} = 7.84\text{ns}
\]

this is an impressive gain.

With time, the gap between CPU and memory speeds got even higher and the cache memory could no longer compensate, moving the bottleneck back to the memory system. Due to physical constraints increasing the cache size would lead to an increased latency. To be useful for the CPU, the cache needed to have a small latency (up to a few clock cycles) thus the size could only grow to a few kB. The solution was to add an extra cache layer. This layer would trade off a larger size with longer access times.

Therefore access to memory started to be organized in a hierarchical model. The first level was named L1 cache and is usually divided in two parts, one for data and another one for instructions, while the new level was named L2. This memory has higher access latency but it is faster than Random Access Memory (RAM) access, and compensates almost entirely for L1 misses.
The new cache level has the following effect on the average access time to main memory:

\[
T = T_{L1c} * P_{L1c} + (1 - P_{L1c}) * T_{L2,mem} \tag{2.2}
\]

\[
T_{L2,mem} = T_{L2c} * P_{L2c} + (1 - P_{L2c}) * T_{mem} \tag{2.3}
\]

where \( T_{L2,mem} \) is the average memory access time with the new cache level, \( T_{L2c} \) and \( P_{L2c} \) represent the access time and the probability of accessing data from the L2 cache memory. Introducing equation 2.3 in 2.2, we obtain the following equation to compute the average access time:

\[
T = T_{L1c} * P_{L1c} + (1 - P_{L1c}) * (T_{L2c} * P_{L2c} + (1 - P_{L2c}) * T_{mem}) \tag{2.4}
\]

Considering a 96% L1 cache hit ratio and a 65% L2 cache hit ratio with a 4ns latency time to access the L1 cache and 16ns for the L2 and a 100 ns to access the memory. The average memory latency would be 6.07 ns. The higher CPU speeds lead to an increased number of requests thus the initial case would have been much worse.

Even if the L2 cache memory can grow to larger sizes, it is not sufficient to compensate for the further increase of the cpu-memory gap. Therefore current processors introduce a new cache level (L3) added to further reduce the gap.

## 2.2 Towards Multicore Architectures

Starting in the early 2000’s, power constraints and thermal dissipation problems have set a limit on further development of highly complex single core CPUs reducing the possibility to continue the improvement of clock frequencies in modern processors. At the same time, single core CPUs are unable to further exploit parallelism in sequential code. Moore’s law [59] is still valid and transistors are getting smaller, every year allowing a higher degree of integration in the same chip. To increase the processing power of current CPUs and take advantage of the current advances in transistor technology, CPU producers have started to add more processing units on the same chip to enable higher degrees of parallelism.

Figure 2.2 shows the transition from single core to multicore CPUs and the architectural changes in the process. The first architectures were relying on the improvement of existing CPUs frequencies and focused on exploiting the Instruction Level Parallelism (ILP). ILP exploits the potential overlap of among instructions and evaluate
2.2. Towards Multicore Architectures

similar instructions in parallel. This was followed by SMT, which introduces additional processing units and attempts to run multiple instructions in the same time over a set of independent threads, achieving a more optimal utilization. This approach allows the processor to take advantage of both instruction level and thread level parallelism. The first processor to implement this technology was the Intel Pentium 4 processor, released in 2002 and Intel calls this technology Hyper Threading (HT). Since then, more processors have emerged supporting similar technologies, starting with Sun T1, IBM Power 5 and others. Although there is just a single physical processor where threads share execution resources and caches, for the OS, the SMT processors appears as a set of two CPUs, enabling the OS to schedule multiple execution threads at the same time. In the case of SMT processors, if one thread stalls or waits for resources, the other thread can continue working, leading to better resource utilization. As threads compete for resources or evict each other’s caches some applications can degrade in performance when running on SMT processors. In consequence, designing applications to run on this class of processors requires careful design and attention to data sharing and resource utilization. Treating each hardware thread as an independent processor is no longer the best design solution [25].

The next evolutionary step was to further increase the level of parallelism by placing two processing cores on the same chip, leading to the first generation of CMPs. Initially each core had independent caches. The increasing number of cores and diminished returns from larger caches have led to the introduction of different architectures of CMPs that share different cache levels.

The evolution with time of frequency and number of cores inside a CPU according to the Stanford CPU database [41] is shown in Figure 2.3. Starting with 2004, the CPU
frequency is no longer improving, while the number of cores per die doubles almost every year. It is now clear that future architectures trends go towards an increasing number of cores.

Modern multicore architectures such as the Intel Nehalem family offers on demand frequency scaling (Turbo Boost) to further improve the performance of single threaded applications. To further exploit the existing level of parallelism and better utilize the processing power each core uses SMT or HT technologies.

### 2.2.1 Challenges Introduced by Multicore Architectures

Previous sections presented major changes and architectural trends of current multicore CPUs. These changes introduce new challenges for existing software and in the next sections we will present some of the problems and existing solutions.

First generations of multicore processors, with a small number of cores, had independent caches for each core. Newer processors with an increased number of cores have shared low-level caches. The reduced complexity of the cache coherence mechanism, improved data sharing options, and a more efficient usage of the last level cache favors cache sharing as the winning approach for the next generations of multicore processors. This approach allows every core to use the entire low level cache. While large caches improve single thread performance, more threads sharing the same cache increases the number of cache misses. The added latency introduced by large cache memories restricts the possibility of increasing the cache sizes proportionally with the number of cores.
available cores. Hence the cache management problem will get worse in future generations of multicore CPUs that will support even more cores per chip. The increased number of hardware contexts will allow an increasing number of threads that will end up competing for the same disk, cache or memory resources. Thus, the bottleneck for current multicores is no longer the CPU but rather I/O and memory accesses.

Multicores rely on cache coherence protocols to communicate shared data structures and locks between cores. Whenever a core reads shared data, the cache coherence protocol needs to find it, mark it as referenced and fetch a copy to the new core. In the case of updating data cached by other cores, the cache coherence protocol must invalidate the shared data from all the cores that reference the data. These operations are complex, expensive, and take around 100 clock cycles which makes them comparable with main memory accesses.

Current multicore architectures have a complex cache hierarchy as an example; Figure 2.4(a) depicts the cache hierarchy of an AMD Opteron Magny-Cours processor. In the case of this processor, each one of the 12 cores has its own local L1 and L2 cache while all share the L3 cache level. As a consequence, all core individual caches need to be kept in sync with the data from the other cores and every write or read to a shared data structure that is cached implies a high overhead to ensure cache coherence between cores.
Multiple Chip Multiprocessor (M-CMP) architectures consist of a series of CMP processors interconnected with each other, as shown in Figure 2.4(b). This configuration requires a new level of cache coherence protocols meant to handle coherence between the CMPs. The hierarchical structure of the caches and of the cache coherence protocols increases the design complexity of future applications, as a more careful design is needed to allow them to benefit from an ever increasing level of parallelism offered by multicores. Without proper thread scheduling and attention to data locality, the cache coherence protocols can easily become a bottleneck. With an increasing number of cores, HW cache coherence protocols will get very complex and expensive since they serialize simultaneous access to the same data. The increasing number of Hardware (HW) contexts accessing shared data structures and locks between different processors will eventually saturate the processor’s interconnect.

2.2.2 Designing Systems for Multicores

In the previous section, we have seen some of the features and design challenges of the multicore architectures. These systems bring an increased level of hardware contexts, while reducing the clock frequency for each core. To take full advantage of existing multicores one must be able to exploit the increased level of parallelism offered by these systems. In consequence Amdahl’s [4] law plays an important role in designing applications for modern multicore architectures.

Almost five decades ago Amdahl mentioned that if we are given an application which has a fraction \( P \) of the work that can be done in parallel, then the maximum speedup \( S_{\text{max}} \) given the number of cores \( N \) that can be achieved is given by the following formula:

\[
S_{\text{max}}(N) = \frac{1}{(1 - P) + \frac{P}{N}}
\] (2.5)

Since \( (1 - P) \) is the amount of work that cannot be done in parallel and thus cannot benefit from the increased number of hardware contexts available today, we found ourselves limited by the performance of the serial code. To exemplify this, assume that we have only 10% of the application that is serial and cannot be executed in parallel. Then if we assume that \( N \) tends to infinity \( S_{\text{max}} \) can be at most 10.

To take full advantage of current multicores one must exploit as much parallelism as possible from today’s applications. Unfortunately most of the applications are not embarrassingly parallel problems. A more detailed analysis of Amdahl’s law in the context of modern multicore architectures taking into consideration different types of
processors and technologies available today is presented in [50].

The previous analysis does not take into consideration other limitations such as memory latency, shared data structures and synchronization costs, but rather, presents the best case scenario.

The OS research community was one of the first to fight the scalability issues of current systems running on multicore. The first alarm signals were drawn when operating systems started to face scalability problems due to locking in shared data structures such as file systems, kernel modules and shared-memory management [19].

The extensive use of coarse-grained locking was one of the first reasons for performance degradation on modern multicore architectures. As a consequence, with the increased number of hardware contexts the time to acquire a lock started to grow. On the one hand, the transition to fine-grained locks of existing applications can lead to deadlocks due to missing dependencies. On the other hand, choosing the right lock granularity is not an easy task due to performance implications of frequent locking and unlocking as well as the higher chance of introducing circular deadlocks [94]. Without careful design and an increasing number of cores, the increased lock contention starts to dominate execution times. A detailed explanation of locking problems and their implications in the case of current operating systems is described in [93].

The increased number of hardware contexts available in modern multicore architectures requires more attention to task management and data locality. Currently, threads are distributed around the entire set of available cores without taking into consideration any data locality characteristics. The OS is scheduling threads to cores based on the current load of each core, and as a consequence, data is shuffled around the cores. As a result, the cache hit ratio starts to decrease once more threads share the same caches and execute more tasks that work on different data [93].

Core to core communication, data sharing and locking is currently achieved using shared memory. As the number of cores increases so does the concurrent access to these structures, putting more pressure on the cache coherence protocols and the processor's interconnect bus.

As a solution to these problems, the OS research community recommends the following principles for designing systems running on multicore:

- **Improve data locality** by scheduling tasks according to their data locality patterns. Each core should handle different data with the effect that cache interference is reduced. This approach tries to add space multiplexing as a new dimension to the design perspective to the traditional time multiplexing approach [93].
• **Reduce the use of shared data structures and synchronization primitives.** Use message passing for communication between application running on different cores. [93]

• **Reuse existing ideas from network distributed systems** such as local data caching, replication, lazy updates and others. These systems had to handle similar problems in the past due to network latencies and were forced to reduce the amount of communication between nodes to achieve performance [13].

• **Push the job of sharing and resource allocation at the application level.** Applications are more aware of what resources they need, and can do a better job in organizing tasks to cores [18, 93, 12].

Modern OSs such as Barrellfish [12], Corey [18], Disco [20, 39], fos [93] embrace these guidelines, and started to design the future OS using the multikernel model or as a collection of services. Due to modularization and reduced contention on resources and shared data structures, these systems show improved scalability with the increasing number of cores.

Taking a step back and to look at the big picture we can observe that multicore systems tend to look more like a network system [13]. In consequence current multicore start to resemble the same problems in regards of caching, latency effects, and more recently heterogeneity [13, 93, 18].

The next sections show that similar problems are also present in database systems. These systems heavily rely on locking and synchronization primitives and their performance is highly dependent on efficient memory access.

### 2.3 Database Systems on Multicores

In 1992 DeWitt and Jim Gray were saying:

"The ideal database machine would have a single infinitely fast processor with an infinite memory with infinite bandwidth and it would be infinitely cheap free." [30]

Two decades later, the idea of an “ideal database machine” has no chance of being developed as current processor trends have shifted towards multicore processors.

Even from the early times database engineers were looking for ways to further improve database engine performance by exploiting the increased level of parallelism in
2.3. Database Systems on Multicores

mainframes, clusters and grids. The challenge was to find a way to build an architecture that can achieve near-linear speedups and handle a wide range of workloads, from complex analytical queries such as Online Analytical Processing (OLAP) to Online Transaction Processing (OLTP).

Stonebraker noticed that these architectures can be grouped in three categories shared-nothing, shared-memory or shared-disk [84]. Shared-memory systems suffer from interference. As multiple CPUs share the same memory, it is a challenge to manage skew and build efficient load balancing techniques in a scalable way. In both shared-memory and shared-disk architectures, the interconnection between the processors or disks cannot provide enough bandwidth to keep up with the increasing number of processors or disks required by demanding workloads. Shared-nothing architectures have proved to be the winning choice [29]. Having a winning architecture, systems and algorithms were developed to manage the increased degree of parallelism [66, 30, 10] in the new cluster-based systems. Techniques for transparent parallelism, data partitioning replication, and parallelization of relational operators are used for a long time and show their efficiency in commercial systems such as IBM Parallel DB2 [9] or Oracle Parallel Server [43].

While shared-nothing architectures proved to scale well with the number of extra processing nodes, the new CMP processors are based on resource sharing at different architectural levels. In the case of multicore processors, each core can have local cache resources and start to share resources from one of the lowest cache levels, to memory, network and disks. From this perspective current multicore architectures, start to adopt a share-everything architecture. As the number of cores per die increases and multiple processors are used inside a single machine current systems no longer resemble the shared-nothing architecture. Deploying existing parallel database systems on multicores will bring an improvement over traditional database engines, due to their parallel nature. But it is not guaranteed that they will natively scale due to the inherent sharing in modern multicore processors.

The increasing degree of parallelism combined with sharing, comes, as in the case of OS, with data and control dependencies. As a consequence, further improvement in the performance of database engines is limited to their ability to use the increasing level of native parallelism. In [46, 25], the authors give a detailed overview of the advantages as well as current problems, solutions and challenges of database engines running on CMP processors. In this section, we are going to present only the ones relevant to our work.
In database engines, transactions can be processed by independent threads in parallel. As the number of hardware contexts increases, naive parallelism starts to be the performance limiting factor. As more independent queries compete for resources, problems like synchronization and concurrent access over shared data structures lead to performance degradation.

Some of the main factors for performance degradation for database engines running on multicore architectures are locking, interaction between concurrent queries, access to main memory and excessive use of shared data structures. Locking has been shown to be a major deterrent for scalability with the number of cores [53]. One must ensure that the extra cores are used efficiently and threads do not spin while waiting for a lock if others wait to use the blocked resources [51].

With the increased number of cores, interaction between concurrent queries when updates or whole table scans are involved can have a severe impact on overall performance [88].

For a very long time, memory operations have been one of the major factors that limit performance of database engines [2]. To alleviate this problem, one can use the increased level of parallelism and create a set of helper threads that pre-fetch data needed by the worker threads from main memory [97, 70]. As presented in [97] there are different strategies to use the increased level of hardware contexts available in multicores. While the naive parallelism approach provides a small improvement, it is by far the one most affected by cache conflicts due to the independent workloads.

Although data sharing might improve locality and reduce the pressure on the memory system, it also has an important negative effect: it can lead to a reduced amount of parallelism thus leading to performance degradation on highly parallel architectures [52]. Aggressive sharing between queries can easily become a bottleneck by serializing the workload and saturating the machine with long running queries [52]. As a consequence, current research proposes to join queries in sharing groups and partition the resources among them, enabling the system to exploit the increased parallelism and the work sharing benefits.

To solve the scalability problems of database engines running on multicores, a great deal of work proposed either ways to modify the engine or to completely redesign the architecture. To summarize a few examples, there are proposals to replace existing engines with pure main memory scans [88]; to use dynamic programming optimizations to increase the degree of parallelism for query processing [44, 45]; to use helper cores to efficiently pre-fetch data needed by working threads [97, 70]; to modularize the engine into a sequence of stages, obtaining a set of self-contained modules, which improve
2.3. Database Systems on Multicores

data locality and reduce cache problems [48, 47]; or to remove locking contention from the storage engine [53, 51].

Commercially, the first engines that represent a radical departure from the established architecture are starting to appear in niche markets. This trend can best be seen in the several database appliances that have become available (e.g., TwinFin of IBM/Netezza, and SAP Business Data Warehouse Accelerator; see [3] for a short overview).

Current research and commercial proposals point out and manage to solve with great success many of the problems in current database systems running on multicores. Due to the complex nature of the solutions, most of the approaches require a major engine redesign. The transition to new engine designs that enable database systems to take advantage of the increasing number of resources and hardware contexts, is needed but will take time.

One of the first methods to scale up existing operating systems on multicores is by using virtualization of hardware resource. This approach widely explored in systems like Disco [20] and Cellular Disco [39]. These systems split the available resources inside a multicore machine within virtual clusters. As key benefits of these systems we can mention a correct allocation of resources with better results than a hardware partitioning scheme presenting a better solution to scale existing off-the-shelf OSs. They achieve these results by adding a software layer between the HW and the OS. Our approach takes a similar strategy applied to database engines, it adds a middleware layer between the OS and the database engine to manage the database replicas. We do not use virtualization of hardware, but rather provide a mechanism to partition the resources using functionality provided by existing off-the-shelf OSs. The middleware layer deploys the database replicas and manages the communication between them.

Current research in modern operating systems addresses the scalability problems of multicores by redesigning the OS in a modular fashion. On the one hand, modularization in modern OSs like fos [93, 94], Barreelfish [13, 12] or Corey [18] is used to avoid the use of hardware locks and reduce the pressure on the cache coherence protocol. On the other hand, specialized services or kernels that run on a set of resources are used to reduce latency and improve memory locality. The use of message based communication and state replication inspired from the distributed system world is aimed to further reduce resource and data sharing while increasing data locality and improving data access latency. As a final approaches try to improve the performance of the OS having as effect an improvement of the applications that run on top of them. One of the main goals of this thesis is to provide a practical approach that enables existing
Chapter 2. Background

database engines to scale on multicore architectures. Existing modern OSs are still in an early stage of development. In the same time current database engines implement their own memory allocation and scheduling techniques. To take advantage of the new features offered by modern OSs they have to go through a major redesign. For this reason, in our approach we reuse ideas and concepts that were explored in modern OSs and apply them in the context of database engines running on multicores. We also show that this approach allows existing database engines running on traditional OSs to take advantage of existing resources and efficiently exploit the resources in current multicore architectures.

Based on techniques used in LANs as part of computer clusters in the Ganymed system [74] and ideas from modern operating systems this thesis proposes to partition the resources available in a multicore machine and allocate an unmodified database engine to each partition, treating the whole as a distributed database, rather than redesigning the engine.

One of the main goals of this thesis is to create a flexible solution to scale database engines on multicores without a complete engine re-design. Scalability should rather be achieved with a limited set of changes to the applications and to the database engine. For this purpose we hide the distributed nature of the underlying system under a middleware layer. This layer takes care of the user requests and forwards them to the underlying nodes hiding the distributed nature of the system.

The next sections present an overview of existing replication systems and motivate our choice for the Remote Snapshot Isolation with Primary Copy (RSI-PC) replication protocol presented in the Ganymed system [74] as the underlying replication mechanism used to deploy the database system on multicores.

2.4 Snapshot Isolation

To get a better understanding of the following sections we need to describe the Snapshot Isolation (SI) protocol. This is an important prerequisite as most commercial and open source database systems today such as Oracle, Microsoft SQL, IBM DB2 or PostgreSQL and MySQL use SI to manage the concurrent executions of database transactions. At the same time, many replication systems use different adaptations of SI to handle transaction management between the replicas.
2.4. Snapshot Isolation

Introduction in [14], the SI protocol is a multi-version concurrency control mechanism [91, 15] widely used in database engines. ANSI SQL [5] categorizes the Isolation Levels in terms of phenomena as follows: Dirty Reads, Non-Repeatable Reads, and Phantoms. Based on these phenomena ANSI SQL defines four isolation levels read uncommitted, read committed, repeatable read and serializable. SI avoids all the phenomena presented in ANSI SQL but it is not serializable, it is a special isolation level that lies between read committed and repeatable read. Due to its characteristics SI has become very popular and has been introduced in a wide range of database engines starting with commercial systems such as Oracle [64], IBM DB2 and Microsoft SQL Server [82], to some of the most important open source engines such as PostgreSQL [76], MySQL [60] and others. Different approaches in software transactional memory and specialized systems such as Google Percolator [71] have adopted SI as their base concurrency control mechanism.

Similar to multi-version control mechanisms presented in [15], under SI each transaction \( T_i \) runs on its own version of the data called snapshot \( S_i \) that is taken at the start of the transaction. Every transaction sees all the commits that happened before the start timestamp of \( T_i \) \( (BOT_i) \) but nothing after that point. Due to this fact, SI does not suffer from inconsistent reads and there are no read/write conflicts as in the case of 2 phase locking(2PL) algorithms. One of the main features of SI is that no locking is required for read transactions. In the case of write transactions, each transaction \( T_i \) will store the updated values in a local writeset \( WS_i \) as presented in Figure 2.5(a). In this way, any future reads of the data by the same transaction will see the latest data up to the start of the transaction. When transaction \( T_i \) commits, it is assigned an end of transaction timestamp \( (EOT_i) \) which is larger than any start or committed timestamp.

Figure 2.5: Transactions running under SI

2.4.1 Overview

(a) Non conflicting transactions

(b) First committer wins
A transaction $T_i$ commits successfully if there is no other transaction $T_j$ that has performed a commit while $T_i$ was running $EOT_j \not\in \{BOT_i, EOT_i\}$ and has not updated the same data as $T_i$, $WS_j \cap WS_i \neq \phi$ (Figure 2.5(a)). A successful commit means that all the updates from $WS_i$ are applied. In the case of a conflict the first committer wins rule prevents lost updates. Meaning that the transaction that committed last, in our case $T_i$ will be aborted (Figure 2.5(b)).

As mentioned in [14] even if SI avoids all the ANSI SQL phenomena, it is non-serializable because reads can come at one instance and writes at another. Consequently, there are a few anomalies but as mentioned in [72] and [22] there are solutions to solve these problems in real applications.

### 2.4.2 Systems using SI

SI is very popular for a wide range of systems that need to manage concurrent read and write accesses to data. SI enables readers and writers to work independently without blocking. This feature makes it suitable for a wide range of applications that need to support long running read queries and in the same time they must allow frequent updates on existing data without a major impact in performance. For this reasons SI has been applied in various domains starting with operating systems [83], replication systems [72, 55], database engines [61], federated systems [80], search engines [71] to mention just a few.

The diversity of the systems that implement SI leads to a variety of implementations. The straightforward approach to implementing SI is to store multiple versions of the objects and once they are persisted they can be removed when they are no longer needed (e.g. PostgreSQL). A more detailed description of the way PostgreSQL implements SI can be seen in [85]. Another way is to write the data in place and keep a history of the changes to allow reconstruction of a specific state in case of a transaction abort. This is mainly the approach taken by Oracle [78] and MySQL using the InnoDB engine.

SI has been adopted in a wide range of applications, starting with transactional memory to web applications. One of the recent use cases of SI is web search systems that manage large indexes that are continuously updated, but at the same time they must handle long running read transactions [71]. SI will allow read transactions to run on the snapshot while the write queries can continue to update the index with new values.

In the case of replication systems, especially for middleware-based replication systems, the main advantage of SI is the fact that the middleware does not have to know about the data that is touched by the read and write transactions. The system has
2.5. Replication

to route SQL statements to the replicas that contain the right snapshot. One of the main advantages of this approach is that the system does not have to send information about current reads. Still it must make sure that the update transactions are applied in the same order by all other replicas. Cluster-based replication systems have been around for a long time and detailed descriptions of SI based replications systems can be found existing literature such as the works of Kemme and Plattner [74, 72, 54, 24].

2.5 Replication

The main challenge of a replication process consists in sharing and maintaining consistency of data between a set of redundant resources. Some of the main objectives of replication are to improve reliability, accessibility, fault tolerance or performance.

In the context of this thesis, database systems running on multi-cores, we will use replication as a means to improve the system’s performance. This goal is achieved by replicating shared data structures and reducing contention over locking primitives as fewer HW contexts are allocated for each database replica.

The rest of this section presents an overview of replication in database systems, and motivates the choice for RSI-PC as the replication protocol used in our system.

2.5.1 Overview of Replication Systems

Replicated database systems use multiple copies of data called replicas. Depending on the way data is replicated, we can have full replication where each replica contains full copies of the initial database, or partial replication where replicas contain just parts of the initial database. Each replicated database engine relies on a concurrency control mechanism and needs a mechanism to coordinate access to different replicas. The main correctness criterion for a replicated database system is One-Copy Serializability (1SR) [15]. 1SR guarantees that the effect of the transactions executed by the clients in a replicated system is equivalent to running the transactions sequentially on a single database engine.

Grey et al. in [40] categorized the replication systems based on two characteristics. The first one is transaction location or (where), and the second is the replica synchronization strategy or (when).

Read transactions can be executed at every replica as they do not lead to any data in-
consistencies. Therefore the "Where" component defines the location of update transactions and here we can identify two cases. First, update transactions take place at only one location also called (the primary), and the changes are propagated to the other replicas (secondaries) this approach is called primary-copy. Second, if the replication system allows update transactions to be executed at any replica the system is called update-anywhere or multi-primary.

The "When" component describes the synchronization strategy. To ensure consistency, replicas can apply the updates asynchronously or lazily, after a transaction commit has been sent to the client. Another way is to let each wait until each replica has applied the update transaction before we send the commit response to the client, this approach is also called eager replication.

To summarize, the existing replication systems can be grouped in four categories, which are visually represented in Figure 2.6 . As we will see in the next subsections each category has its own advantages and disadvantages.

This thesis provides just an overview of the main building blocks of replication database systems, a more detailed description can be found in the current related work [24].

**Primary-copy vs. Update-anywhere**

Primary-copy replication uses a single replica to process the updates, this replica is also known as a primary replica. Once the updates are applied at the primary they will
be propagated and applied in the same order at all the secondary sites. This approach solves all the update conflicts at the primary copy providing a very simple concurrency mechanism. In most primary-copy replication systems, the concurrency control is the same as in the non-replicated system [24]. This ensures that the concurrency management is done locally and no extra work is needed at the other replicas in the case of a transaction abort. On the one hand, primary-copy replication systems have the disadvantage of limiting the access options of update transactions by forcing them to be applied at one site. On the other hand, the primary node becomes a bottleneck in the case of a heavy update workloads, the best performance in heavy write workloads is the same as the performance of a single database. However, it has been shown that these systems are well suited for read intensive workloads [57, 87].

Update-anywhere replication systems allow the execution of update transactions at any replica. Once the updates are committed at one of the replicas, the concurrency mechanisms will apply the same update at all the other sites. This approach gives more flexibility to the clients and can improve the performance of the system due to the close proximity of the replicas to the clients. In the same time the decentralization of the update transactions puts more pressure on the concurrency system. To ensure that replicas converge to the same state while running concurrent update transactions update-anywhere replication systems must use a more complex concurrency control mechanisms that lead at performance degradation in workloads with an increased amount of update transactions.

**Eager vs. Lazy**

Eager systems are also called *synchronous* since once the client has received the commit confirmation, all the replicas have converged to the same state. Due to the extra time needed for replica communication and convergence these systems have longer response times. These systems ensure 1SR, strongest serialization level [15].

Lazy replication systems are *asynchronous*, because replica updates are propagated outside of the transaction boundaries. Once an update transaction is committed at the primary copy, the client receives the confirmation for a successful transaction. The next step is to extract the changes produced by the transaction on the primary replica and lazily propagate them to the other nodes. For this reason these replication systems can encounter delayed conflict resolution. In the case of lazy-update-anywhere replication systems more conflicting situations can occur and transactions can be aborted even if they were reported as successful to the client [40].
Discussion of Replication Systems

In a distributed replication system that requires flexibility and 1SR, the suitable replication system would be eager-update-anywhere. As in the case of update-anywhere replication systems, the main disadvantage of this approach is the complexity of conflict and deadlock detection algorithms. The eager-primary-copy systems can reuse the existing concurrency mechanism making them easier to implement. Due to the eager approach, clients have to wait until all the secondary replicas converge. These systems are suitable for reliability, but as the number of replicas increases so does the latency. The need for locks and the increased latency of eager systems makes them known to have performance and scalability problems even with a small number of replicas [40, 72, 54]. To maintain serialization these systems must use extra communication increasing transaction response times. While eager replication systems rely on locking schemes to achieve serialization, making them prone to deadlock, lazy replication systems typically rely on multi-version concurrency systems such as SI to solve non-serializable situations [40].

Lazy-update-anywhere systems allow update transactions to be executed at every replica without any communication between replicas. Due to conflicting write transactions that can modify the same data in two different replicas, these systems can break fundamental properties of transactions such as durability or atomicity therefore not guaranteeing serialization or isolation. As in the case of update-anywhere conflict resolution is a very complex problem, and is highly dependent on the application semantics. Lazy replication systems asynchronously propagate the data to all the other nodes, and send the commit message before the data is committed to all the nodes. One of the main advantages of these systems is the reduced response times for the read transactions, which comes at the cost of stale data between the replicas. To avoid consistency problems and stale data the clients must be able to cope with stale data and the update rate must be low. For these reasons, these replication systems are best suited for very well controlled environments with an exact knowledge of the application and its behavior [24].

Lazy-primary-copy replication systems can easily ensure a serial execution as all the update transactions are performed at one replica, namely the primary-copy. The lazy protocol allows little communication overhead with the other replicas leading to a very simple replication approach. Once the transaction has been applied at the primary replica, it can be seen as committed and the changes are extracted producing the so called writesets and propagated asynchronously to the other replicas. Due to the write-set extraction and propagation, this approach provides weaker consistency guarantees
2.5. Replication

compared to the eager approaches, especially in update intensive scenarios. In these situations the secondary copies cannot keep up with the high update rates ending up with stale data and reducing the overall system performance.

As we can see in Figure 2.6, each option in building a replication system is a trade-off between performance, flexibility and consistency. The most problematic issue in the case of replication systems is the execution of update transactions.

To deploy a replication system inside a modern multicore machine, we need to find a light weight algorithm that uses little resources. This way the computational resources are used to execute transactions, thus doing useful work, rather than coordinating the consistency protocol. A simple communication protocol between the replicas will reduce the communication overhead between the replicas and cut back on the amount of cache misses due to the amount of data that is exchanged between the cores.

2.5.2 Architecture of Replication Systems

According to [24] there are two ways of implementing replication protocols, as a kernel-based or white box approach or outside the database as a middleware layer. In the first approach, the replication is integrated into the database engine as a module, in which case we have a replication aware database engine. Due to the tight integration, the client can connect directly to any database replica which then will coordinate independently with all the other replicas. In the second approach, the middleware layer presents itself to the clients as a database engine. The clients will connect to the middleware which will take care of the read write query routing between the replicas.

The kernel-based replication systems are tightly integrated with the database engines. Therefore, they rely on information from the database engine to manage consistency among the replicas. While they can be implemented very efficiently, access to the database engine internals or source code is not always an option. Most of the time they are very specialized systems.

The middleware replication systems manage the routing and implement their own concurrency control mechanism. This approach can be implemented without any changes or access to the database engine source code or internal Application Programming Interface (API). Middleware replication systems are non-intrusive and portable. Due to the extra layer and loose integration they have an increased latency.
2.6 RSI-PC

A suitable transaction scheduler that can be used to deploy a replicated system inside a modern multicore machine is the RSI-PC [72]. The scheduler hides the replicated nature of the underlying system and presents the system to the user as a usual relational database engine. This light weight scheduler does not need any information about the data that is going to be touched, it only disseminates between read and write transactions. It is based on the lazy-primary-copy replication strategy so it can scale with an increasing number of replicas without a significant penalty in performance for a wide range of workloads. In the same time it behaves as an eager replication strategy solving the problems of stale data. The limitations of the lazy replication approach are solved by the RSI-PC scheduler which also offers SI consistency guarantees.

This section presents only the main idea of the RSI-PC scheduler, more details and correctness proofs are presented in the PhD thesis of Christian Plattner [72].

The main goal of this scheduling algorithm is to improve the performance of SI database engines by scheduling transactions over a series of replicas that support SI, combining the advantages of lazy and eager strategies.

The algorithm uses a middleware to manage the transaction scheduling over the replicas the main components of the system can be seen in Figure 2.7. This approach allows the middleware to look like a database that offers SI to the connecting clients and act as a eager system. In the background the RSI-PC scheduler is a lazy system with-primary copy that uses two types of replica nodes: a master and a set of satellite databases.

The scheduler will route the update transactions to the master database which is the primary-copy and the read-only transactions to the secondaries also called satellites. Due to the middleware based approach, the RSI-PC scheduler can hide inconsistencies from the clients. In the particular case depicted in Figure 2.7, we have an RSI-PC scheduler that uses one master database with three satellite nodes. As update transactions can also contain read queries and the scheduler keeps track of transactions, the clients must specify at the beginning of each transaction the transaction type (read-only or write).

The transaction scheduler will route all the write transactions to the master node. Once the update transactions have been committed on the master node, the changes are extracted and the writesets propagated to the satellites. Each satellite will apply the writesets in the same order as they were applied on the master node, thus guaranteeing that in the end they will converge to the same state as the master node. The read-only
transactions will be routed to the satellite nodes wherever this is possible. Read-only transactions are routed to the satellites that are up-to-date. If none of the satellites can provide the required level of freshness, the read query is going to be executed at the master, which is always up-to-date. As the satellites are using SI, the readers and writers will not interfere, allowing the nodes to apply the writesets without any performance losses.

The RSI-PC algorithm does not need to do any SQL statement parsing to keep track of the transactions or rely on row level locks in the database. Doing only routing and scheduling, it alleviates the need for any other complex concurrency mechanism. Thus it can be easily implemented as a light weight middleware layer. One of the main advantages of RSI-PC is the lack of locking, shared data structures and the reduced amount of communication between the nodes, making it suitable for the high degree of parallelism offered by current multicore architectures.

The next section will present a short overview of the more important existing replication systems.

2.7 Real Replication Systems

This section presents an overview of some of the most popular replication systems. It is out of the scope of this thesis to present a full survey of existing replication systems.
Instead we focus on the main advantages and disadvantages of the solutions and try to assess the applicability to our system.

### 2.7.1 Research Replication Systems

Replication is one of the oldest and important topics in the database research community, where problems and solutions to improve different aspects of distributed replication system have been studied for a long time.

**Posgres-R**

Kernel-based replication systems such as Postgres-R [54] or lately PostgreSQL-R(SI) [95] use a decentralized approach to scalability. To ensure conflict resolution between replicas, they rely on group communication. Their main problem is the high overhead generated by the increased degree of communication in the case of update intensive workloads. Postgres-R lacks a load balancing mechanism between the replicas leaving this job to the client applications or assumes that clients are uniformly distributed over the replicas. In worse case scenarios these systems can produce lower response times than the initial database engines due to the increased communication overhead.

Kernel systems rely on extensions to the database engine to ensure communication and conflict resolution. These extensions are tightly coupled to the database engine and usually require applications to use specific APIs, resulting in changes to both the engine as well as to the application. Being engine specific, they rely on internal characteristics and implementation details of the systems. Due to the tight coupling they are hard to integrate with other engines. Even more, they do not support partial replication or a different database schema between replicas.

**C-JDBC**

One of the first middleware replication systems C-JDBC [23] introduced a flexible middleware for database engines. C-JDBC deploys a Read-one Write-Many replication mechanism relying on table locking to guarantee consistency and group communication to ensure synchronization and reliable ordering of messages. The main goals of C-JDBC are to improve flexibility and performance of clustered database engines. For this reason it presents itself as a single database engine to the clients, hiding the clustered nature of the system. C-JDBC can integrate any database engine that provides a
Java Database Connectivity (JDBC) interface. The standard interface allows a flexible integration with various database engines. The limitations of the JDBC interface forces C-JDBC to duplicate database functionality in the middleware level. While JDBC allows a flexible way to interact with the database engine, it is not designed as a way to gain fine grained access to the database engine. As a consequence, recovery and locking mechanisms need to be duplicated in the C-JDBC controller. Propagation of updates between replicas is achieved by broadcasting SQL statements since the JDBC driver is not designed to handle writeset extraction and propagation. The limited amount of information provided by the SQL statements forces the controller to work with table level locks.

Even if it provides a flexible way to achieve a replicated system by integrating various database engines C-JDBC was not designed to run on multicore machines. The increased amount of communication between replicas, the coarse-grained locking and the fact that each replica needs to evaluate the same update statements can lead to memory locality and locking problems.

Tashkent/Tashkent+

Tashkent \[33\] and the newer extension Taskent+\[34\], are middleware replication systems that use generalized SI \[35\] to manage concurrency. Tashkent allows update-anywhere replication, while Taskent+ uses a load balancer to send the requests from the clients to the replicas. Both systems use two main components to manage replication: a certifier to detect write-write conflicts and a set of symmetric replicas. Conflicts are detected by comparing writesets of different transactions by the certifier to ensure that two different replicas do not produce a conflict. Each replica communicates directly with the certifier before committing transactions to ensure that there are no write-write conflicts. Since read-only transactions do not have to be certified (do not have any writesets) and they are assigned to a snapshot (which can be out of date) they can read stale data. Taskent+ uses the load balancing component as an intermediate layer between the clients and the underlying replicas to extend the functionality of Tashkent. This approach allows the system to use various load balancing techniques. For example, it can group transactions in Transaction groups based on different requirements and characteristics (e.g. memory requirements) to improve data locality and performance. The authors have implemented a prototype using PostgreSQL. To improve performance, they needed to make small changes to the engine in order get the commit order and avoid serialization of commits.
Ganymed

Ganymed [72] is a lightweight middleware system designed to schedule transactions over a set of replicas. Based on the RSI-PC scheduling algorithm, the system is designed to improve performance of SI database engines using replication. Ganymed shows almost linear scalability for e-commerce workloads (e.g. TPCW [57]). It achieves this by using a primary-copy lazy replication mechanism but hides the lazy nature of the system underneath the middleware showing an eager behavior to the clients. The Ganymed middleware layer presents itself to the clients as a normal SI database engine and clients can access functionality using a basic JDBC interface. Ganymed communicates with the underlying replicas by means of a JDBC interface, and can use a flexible mechanism to extract writesets (e.g. triggers or custom engine functions).

Underneath, Ganymed separates read and write transactions and runs them on different replicas. All the update transactions are routed to the master node and the writesets are propagated in a lazy fashion to the secondary replicas (satellite nodes). Each satellite applies the writesets in the same order as they have were applied at the master, ensuring that the nodes will converge to the same state (snapshot). Read-only transactions are scheduled on replicas that have the latest version of the data. If no such replica is available, then the read transaction is going to be routed to the master which is always up-to-date. Since all replicas use SI, read transactions will not interfere with write transactions, thus the satellites can apply the writesets without loss in performance due to locking.

The middleware does not need to keep track of conflicting writesets or read transactions. There is no communication between replicas and it does not involve any parsing of SQL statements. To differentiate between read and write transactions, the client must mention the transaction type (read-only or update) using a standard JDBC statement property. Due to the simple scheduling strategy, there is no need for low level or table locking at the scheduler level.

The authors have proved the flexibility and performance of the system by using a wide range of engines (e.g. PostgreSQL, Oracle, IBM DB2) and used it in a wide range of configurations and scenarios [73]. As another feature the system allows extensions to the initial engine’s functionality [72].

In the context of multicores, the lightweight middleware approach presented by Ganymed will not introduce a large overhead, while the reduced amount of communication and locking will not create any extra contention allowing.
2.7.2 Open-Source Replication Systems

Most of the open-source solutions for database replication are engine specific, thus not allowing integration of different engines. Even more, using partial replicas or different database schema between replicas is not supported by most of the systems. Lack of support for partial replication reduces the amount of satellites that can be supported in a single multicore system. The high dependency on the underlying engine reduces the flexibility of the system and limits the extension capability of the system.

Some of the most important open-source relational database engines systems have native support for replication. Due to the open nature of the systems and availability of their source code, a variety of systems and tools have been developed to enable more extended replication features for these engines. This subsection will present a short overview of existing replication systems available for MySQL and PostgreSQL.

PostgreSQL Streaming Replication

Recent versions of PostgreSQL have native support for replication, and it is called streaming replication. This approach enables replication between multiple instances of PostgreSQL servers using propagation of Write Ahead Log (WAL) records. Write-set extraction is done by analyzing transaction logs. Transaction log keep a history of all the transactions executed by the engine and are used to ensure the ACID in the case of failures. The PostgreSQL Streaming Replication streams the logs from the primary to the replicas asynchronously, in the same order as they are generated at the primary. Thus there is a small delay until the replicas are up-to-date. According to PostgreSQL documentation, the sync delay is typically under one second, if the secondary can manage to process the load. Until the secondary becomes up-to-date and catches up with applying the WAL records the system will have a period time in which the replica will contain stale data. As a consequence, any queries executed in this time frame will read old data.

The native replication included services in PostgreSQL are designed to improve high availability and reliability rather than improve performance with an increasing number of replicas or extend the database functionality with custom replicas. This approach is not suitable if the database engines do not have the same version. It is also limited to full replication, the secondary node cannot have more indexes or a different schema than the primary.
Slony-I and others

Slony is a trigger-based replication system for PostgreSQL designed for single-master replication supporting cascading and slave promotion. The system aims at achieving high availability by supporting high redundancy and load balancing between replicas. Making use of triggers to extract writesets, this replication approach supports integration of different PostgreSQL engine versions. Conflict resolution is done using table level locking, thus long running transactions can impact performance of Slony-I. While PostgreSQL uses SI to avoid conflicts between readers and writers, Slony-I uses internal locks and shared data structures to keep track of updated rows and the state of replicas in the system. Thus each update to a replicated table will add a new row in an internal Slony-I table generating an increased overhead.

One of the most mature systems on the available for PostgreSQL replication, Slony-I is a very complex system allowing a fine grained control. Due to the trigger-based approach that extracts data from the replicated tables on insert update or delete, Slony-I allows a flexible integration of different database versions and schema. While the extra bookkeeping, sharing and locking will create problems in the context of multicores.

There are also other systems such as Londiste and Bucardo that try to offer simpler replication mechanisms with fewer features and easier to deploy. Due to their simplicity and early stages of development, they fail to offer a fully fledged replication system.

MySQL

One of the most popular engines MySQL also has native support for replication. The main focus is to improve availability and reliability. MySQL supports asynchronous replication, allowing the slaves to be disconnected from the master managing interrupted or high latency connections. It also provides a more flexible replication mechanism, allowing the integration of secondary replicas that use different MySQL storage engines, thus allowing the application to use the right engine based on the workload such as the use of MyISAM engines where no transaction support is required. It can also support hierarchical replication, or replication of different databases from the master node to different slaves. Regarding the synchronization mechanism, MySQL uses binary logs that can be configured to log on statement based, on row changes, as a way to sync the replicas.

While MySQL supports a flexible replication, it does not support any load balancing nor does it employ any consistency control mechanism, pushing this problem to the
application layer. Recent versions of MySQL, starting with 5.5, can provide higher integrity by using a semi-synchronous replication. In this case the primary replica waits for a confirmation from one of the slaves that it has applied the binary log before returning the commit acknowledgment to the client.

**MySQL Cluster**

MySQL Cluster is an improved version of MySQL designed for distributed computing environments and supports asynchronous replication. It achieves this by using a specialized engine, Network Database (NDB) to ensure cluster functionality. NDB uses Two-Phase Commit (2PC) to ensure conflict resolution and relies on the replication support provided by standard MySQL. 2PC is a distributed algorithm that coordinates all the participants in a distributed transaction to achieve a commit or rollback of the transaction.

Once updates are replicated on two nodes they are considered durable, but there is no guarantee that they have been flushed to disk. The engine is also not supporting statement-based replication and requires a primary key on every replicated table to avoid duplicate entries.

In addition to these replication approaches there are also a set of other 3rd party replication systems for MySQL such as Tungsten Replicator or Galera Replication which are in early stages of development. Tungsten Replicator relies on the MySQL binary logs and uses its own replication protocol. Tungsten converts the binary logs of MySQL in standard SQL queries allowing integration with different database engines as satellite replicas. Galera is designed as a replication plug-in and supports only use the InnoDB engine of MySQL. One drawback of these approaches is the tight coupling with the internals of the MySQL engine.

### 2.7.3 Commercial Replication Systems

Most commercial database systems such as Oracle, IBM or Microsoft SQL server offer replication solutions. For example, Oracle provides Oracle Real Application Clusters (RAC) that is based on SI. To ensure performance scalability and consistency it relies on a shared storage system such as a Storage Area Network (SAN), or specialized network infrastructure. The Integrated Cluster Environment from DB2 is similar. Microsoft supports replication, as a part of their active standby server solution, and can be extended to keep in sync a master node and multiple replicas. Microsoft has
Chapter 2. Background

implemented the support for SI starting with the MS SQL Server 2005 allowing both Two-Phase Locking (2PL) replication as well as SI guarantees. 2PL is a widely used concurrency control method used to guarantee serialization. When compared with 2PL is more prone to deadlocks an increased response time. While running in 2PL a transaction must acquire all the locks need and retains them until the transaction commits or rolls back the changes. Being very specialized systems, they rely heavily on the underlying database engine, and cannot be reused on commodity engines.

2.8 Discussion

This chapter presented the current trends in CPU architectures and problems that arise with the advent of current large multicore architectures. On the one hand, systems running on current multicores do not gain any extra performance with an increasing number of cores due to locking and synchronization primitives. On the other hand, the performance can even drop due the increased time spend on waiting and delays due to contention on shared data structures, one.

To take advantage of the increased level of parallelism, different research areas have produced solutions and guidelines that enable systems to scale on multicore architectures. Solutions from the OS community recommend dividing the system in functional units [93], or multi kernels [11] that replicate data structures as a way to improve scalability, since the data is closer to the cores and coherence mechanism puts less pressure on the interconnect. Similar approaches [36, 7] show that replicated data structures reduce the access latencies on multicore architectures.

Contention over non-scalable locks produce high interconnect traffic, a core that holds a lock can be slowed down with an amount proportional with the number of waiting cores [19]. Scalable systems on multicores need to reduce the use of shared data structures and synchronization primitives. This can be achieved using well known algorithms and protocols from distributed replicated systems. For example, message passing is already used in operating systems such as Barrelish [13] or FOS [93] to reduce the communication overhead.

In the case of current database engines running on multicores, most of the problems mentioned in the OS community still hold. Therefore, various solutions have been proposed and focus on optimizing the access times in shared data structures [52] or locking primitives [51] or aim at complete engines redesigns [47, 88].

While these approaches are needed, and a complete engine redesign is the way in
which future database engines will benefit to the full extent of the new features offered by modern multicore architectures, database systems are part of the critical path in current systems and architectural changes on existing engines will take time. We propose a new system that does not require any changes to the database engine.

Taking ideas from the operating systems world on scaling systems on multicores, and combining them with work done in distributed replication system, we propose to treat the multicore machine as a special distributed system and run a set of replication engines.

To achieve high performance, we need to make sure that the middleware level is not the bottleneck. For this reason, we were inspired by the Ganymed approach [74] and decided to use the RSI-PC as our replication algorithm of choice and run it inside a single multicore machine.
One of the goals of this thesis is to improve the performance of existing database engines running on modern multicore systems. This chapter evaluates the performance of existing database engines with an increasing number of clients while varying the amount of available cores. The main objectives of this evaluation are to evaluate the scalability properties of existing database engines and find any performance bottlenecks. Once the performance deterrents are found we use this information to assess possible solutions.

To evaluate the performance of traditional databases running on large multicore architectures, we have performed extensive benchmarks on two of the most popular open source relational database engines PostgreSQL and MySQL. The open nature of these systems allowed a detailed evaluation and code level mapping of their performance bottlenecks.

In this section we use two popular benchmarks, TPC-W and TPC-H, to evaluate the performance of database engines running on large multicore architectures. TPC-W is an OLTP benchmark, simulating a book shop. This benchmark simulates a wide range of situations with a varying amount of read and write transactions. One of the most popular benchmarks for replication systems, it evaluates the performance of database engines under various loads. These loads are designed to put pressure on different com-
ponents of the database engine such as the scheduling or locking mechanisms. The TPC-H benchmark is an OLAP benchmark, simulating large analytical tasks that involve complex operations. The large aggregations and sorting operations in this benchmark put a high pressure on the memory system. In this kind of workloads data locality is an important performance factor. These benchmarks cover a wide range of scenarios, exposing different problems and aspects of database engines running on multicores.

The final results complement and confirm the results obtained by similar studies done on a variety of database engines running on multicores [52, 46, 70, 19]. Experiments presented in the related work, either analyze the performance of very specific components of existing storage managers [46, 52] or analyze the performance of existing OSs and use database engines as reference applications [70, 19]. Systems in the first category analyze in high detail the performance of specific components of open source database engines such as PostgreSQL and MySQL with varying number of threads. With a focus on the the memory and thread management systems. Their results show that the main performance degradation factors are memory management, locking and synchronization primitives. The second category [19, 70] analyzes the performance of existing applications running on current operating systems. For the evaluation they use a mix of popular application and analyze their performance characteristics. Using this information they search for bottlenecks that prevent scalability of the OS and applications that run on modern multicore architectures. We focus our experiments on exiting open source engines and their performance while varying the number of cores and clients. In the same time we analyze the effects of the workload on the scalability properties of the database engine.

The main objectives of our evaluation aim to find the point where database engines stop gaining performance with increasing number of performance and in the same time to define the causes that lead to the degradation. Both in our experiments as well as in the ones presented in related work, we observe that some of the most popular open source database engines don’t scale with increasing number of resources. The main performance deterrents are contention over locking and synchronization primitives as well as workload interaction.

### 3.1 Benchmarks and Objectives

To evaluate the performance of current database engines we have decided to use a transactional (TPC-W) and an analytical (TPC-H) benchmark. On the one hand,
3.1. Benchmarks and Objectives

TPC-W has a mix of read and write transactions which creates pressures on locks and shared data structures. This benchmark allowed us to stress the consistency mechanism and observe the effect of load interaction under an increasing level of parallelism. On the other hand, TPC-H is an analytical benchmark which does full table scans and complex aggregations over large tables putting pressure on the memory system and exposing the importance of data locality.

The objectives of these benchmarks are to stress the database engines in different scenarios and produce comparable results between different engines. These benchmarks evaluate the performance of database engines running on systems with a varying number of concurrent clients and the database sizes. In this thesis, we add an extra parameter; the number of cores used by the database engines and use the same benchmarks to evaluate the performance of database engines with different number of cores and varying number of clients. This approach presents the scalability properties of current database engines with an increasing number of cores.

3.1.1 TPC-W

The TPC Benchmark W is a transactional web benchmark developed by the Transaction Processing Council [57], and is one of the most popular benchmarks used to test the performance of replication systems [73, 33, 34]. The benchmark simulates the specific activities of an on-line bookstore and is composed of 14 web interactions each consisting of one or more transactions. The performance metric for this benchmark is the number of Web Interactions per Second (WIPS) managed by the server.

To test different situations and stress different components, the benchmark defines three different workload types each one with a different mix of read (browse) and write (order) interactions. The browsing mix is intended to simulate a scenario where the users mostly browse through the website but there are a few buy orders. In this case the benchmark consists of 95% browsing and just 5% ordering transactions. The ordering mix simulates a heavy ordering scenario, where 50% of the transactions are order requests and 50% are browsing. Finally, the shopping mix is somewhere in the middle having 80% browsing and 20% ordering transactions.

For our experiments we want to stress only the database back-end, and therefore the clients connect directly to the database and issue queries according to the TPC-W mix. To generate the load, we used a Java implementation of the TPC-W emulated browsers. The load generator creates a set of parallel connections to the database engine and generates the TPC-W queries on the fly. Once a transaction has been
executed, a new transaction is assigned to the client without any wait time. Since we were not using a web interface, we implemented the shopping cart as a table in the database, rather than storing it in a web-server session.

Unless otherwise specified, the TPC-W clients do a warm-up run of 20 minutes and perform full table scans over each table to ensure that the buffers are hot and the database is in main memory. The benchmark queries are executed for 30 minutes after the warm-up phase, without any client disconnect.

3.1.2 TPC-H

The TPC-H benchmark [87] uses ad-hoc complex queries that do large table scans or have complex computations over large amounts of data. The benchmark simulates the decision support queries needed in a business that sells or distributes products worldwide.

The benchmark consists of 8 tables and 22 queries that have a wide range of complexity, starting from aggregations over large scans to complex joins and sorts over multiple tables. It also has two refresh functions that update the data at specific intervals. Each client is assigned an ad-hoc permutation of the 22 queries, thus creating complex workload which is hard to anticipate by current database engines.

3.2 Experimental Setup

Figure 3.1 shows an overview of the testing framework used in the experimental evaluation. The main components are the clients that generate the workload, the system under test, which is a dedicated server for one of the database engines and the test coordinator which ensures client synchronization and coordination as well as the aggregation of the final results.

The database server is a dedicated machine having a 4 way AMD Opteron 6174 Magny Cours Processor with 48 cores, 256GB of RAM and two 146GB 15k RPM Seagate Savvio disks in RAID1.

Figure 3.2(a) presents an overview of the AMD Magny Cours CPU. Each CPU consists of two dies, with 6 cores per die totaling 12 cores per CPU. Each core has a local L1 (128KB) and L2 cache (512KB). Each die has a shared L3 cache with 6MB on each die totaling up to 12MB. The dies within a processor are connected with two HT links.
3.2. Experimental Setup

between each other, each has two additional HT links. Each die has 2 memory channels. In this architecture, each CPU creates two Non-uniform Memory Access (NUMA) nodes, due to the separate memory controllers on each die. In our system, we used 4 CPUs that use the Max I/O configuration as presented in Figure 3.2(b). This configuration is designed to ensure uniform memory access distribution. A detailed description of the AMD Magny Cours and information about the implications of the interconnect and cache hierarchies is given by [28].

At the end of each experimental run, fresh copies of the database were installed on the system under test, to prevent erroneous results due to database evolution. Each database has been tuned and has the required indexes to perform at its best capacity.

To generate the testing clients, we used 4 machines each with 16 cores and 32 GB of main memory. In this way, each machine can generate up to 100 clients without any delays on the client side. Each client stores local statistics and counters to measure throughput and response time.

The OS used on the machines is a 64-bit Ubuntu 10.04 LTS Server and the load generator is a Java application that runs on Sun Java SDK 1.6. The database engines running on the system under test are PostgreSQL 8.3.7, MySQL 5.1.
Chapter 3. Evaluation of Database Engines

The following sections will present the performance results of two database engines PostgreSQL and MySQL running on multicores. Although they are two open source relational engines that support SI, they have significant architectural differences.

Both engines make intensive use of shared data structures and synchronization primitives and store the data in a set of files which are accessed concurrently by internal processes. PostgreSQL starts one process for each connection, while MySQL uses a pool of threads which are assigned to concurrent connections.

PostgreSQL and MySQL make extensive use of threads to manage requests from multiple client connections. Consequently they should scale well with the increasing level of real parallelism offered by multicores. Unfortunately, the extensive use of shared files, data structures and locking primitives leads to performance degradation on large multicore architectures.

### 3.3.1 OLTP Performance

For the OLTP evaluation we used our Java based TPC-W implementation and generated a database with a size of 20GB using the following TPC-W scaling parameters:
100000 Items, 2880000 customers and 4000 Emulated Browsers. The database server was a 4 way AMD Opteron 6167 "Magny Cours" with 48 cores, presented here.

The database engine was pinned to a specific number of cores using `taskset`, a tool offered by the operating system to set the CPU affinity. The CPU affinity is a operating system scheduler property that "bonds" a process to a given set of CPUs on the system. The operating system will have to honor the CPU affinity and not run the processes or the child processes generated by the main process on other cores than the ones assigned [62]. Although these experiments were executed on a Linux OS, similar tools exist on other operating systems (eg. Windows or MacOS).

In the following experiments the load generators were connected to the database engine through the specific JDBC type 4 driver of PostgreSQL and MySQL. Three machines were used to generate up to 100 TPC-W clients each, for a total of 300 clients. The number of cores allocated to the database engine is varied from 8 to 48.

**PostgreSQL**

The first part of the evaluation establishes a performance baseline for the PostgreSQL engine. We have measured the performance of the engine while varying the number of clients and the number of cores allocated to the database engine. Before each experiment we have a 20 minutes warm-up phase. Each experiment was run over a period of 40 minutes and the measurement interval was of 60 seconds resulting a total of 40 samples for the entire experimental run. All transactions were executed under the SERIALIZABLE isolation level in PostgreSQL. Prior to the experiments indexes, memory and buffer settings were tuned for best performance. Multiple runs have been done to ensure a low deviation of the results.

PostgreSQL behavior with an increasing number of clients and cores running the TPCW-B benchmark is presented in Figure 3.3. As previously mentioned in Section 3.1.1, this benchmark is composed of 95% read and just 5% update transactions. The experiments capture two main performance metrics of database engines throughput (Figure 3.3(a)) and response time (Figure 3.3(b)). Both values were measured independently on the clients and aggregated. The presented plots show the average values over the entire clients and the error bars present the standard deviation of the results. In Figure 3.3 the x-axis shows the number of clients used to generate the TPC-W load. For Figure 3.3(a) the y axis shows the average number of WIPS executed by the engine while in Figure 3.3(b) it represents the average Response Time (RT) for a web interaction in seconds.
Chapter 3. Evaluation of Database Engines

TPCW-B  Taking a closer look at the throughput results from Figure 3.3(a) we notice that the overall performance of the PostgreSQL engine degrades with an increasing number of concurrent clients for a fixed number of cores. Since PostgreSQL creates a new thread for every connection this is expected behavior. One would expect this problem to be solved with an increased number of cores. But experiments show that
the throughput drops with an increasing number of cores which points out the engine’s scalability problems on current multicore architectures.

Figure 3.3(a) shows that PostgreSQL has scalability problems on a small number of clients as well as on a large number of clients. On the one hand, with a small number of clients the engine cannot fully benefit from the extra number of cores and resources to further improve the performance. On the other hand, the added number of cores introduces an increased level of real parallelism, which leads in performance degradation when running with a larger number of clients. Taking a look at the throughput when running with 300 clients, we notice that the performance on 48 cores is lower than the performance on 8 cores. We can observe that the extra number of cores do not produce performance improvements, but rather degrades the overall system’s performance.

The same conclusions as above can be drawn while taking a look at the evolution of the average response time shown in Figure 3.3(b). As the number of cores and clients increases the average transaction response time grows. Therefore higher number of cores degrades the average execution time for each query.

In the case of the TPCW-B benchmark, we can conclude that PostgreSQL can scale up to 12 cores for all client configurations, while for a smaller number of clients performance improvements can be seen up to 24 cores. Once these limits are reached the system will start to degrade in performance. In the end, at 48 cores the performance drops below the initial baseline when PostgreSQL is running on 8 cores.

**TPCW-S** With an increased number of updates, the TPCW-S (80% browsing and 20% ordering transactions) shows the same performance problems as in the previous case. Taking a closer look at Figure 3.4(a) we notice that the reduced number of long running read intensive transactions allows the system to achieve a better performance, as the increased number of short lived write transactions leads to an increased overall system throughput.

The same effects can be seen by analyzing the response evolution presented in Figure 3.4(a). As the number of cores increase, the database engine starts to degrade in performance with an increasing number of clients.

PostgreSQL running TPCW-S can scale up to 24 cores, on all client configurations. While for a small number of clients, the system can scale up to 36 cores with minor improvements. Larger numbers of concurrent clients create performance degradation, as in the case of the TPCW-B benchmark. We notice the same behavior when we look at the average query response time, since with an increasing number of cores and
Chapter 3. Evaluation of Database Engines

Figure 3.4: PostgreSQL with 20GB database running TPCW-S

clients the increased number of cores does not bring any improvements.

TPCW-O The write intensive version of the benchmark TPCW-O (50% browsing and 50% ordering transactions) is mostly composed of short lived point update transactions. In the case of PostgreSQL, these transactions can benefit from the extra number
3.3. Performance Evaluation

![Graph showing Throughput and Response time](image)

With an increasing number of cores, we notice that the limiting factor is the contention on the disk I/O. The system starts to become unstable as the number of clients increases and more pressure is put on the disk.

Summing up the results of the three benchmarks when varying the number of cores and the increased level of parallelism available in today's multicore machines. With an increasing number of cores, we notice that the limiting factor is the contention on the disk I/O. The system starts to become unstable as the number of clients increases and more pressure is put on the disk.

Figure 3.5: PostgreSQL with 20GB database running TPCW-O
and clients for PostgreSQL, we can conclude by saying that long running transactions in mixed workloads produce performance degradation due to workload interaction. The increasing number of cores introduces an increased level of real concurrency and the fact that PostgreSQL forks a new thread for every connection only makes this problem worse. The increased number of connections and hardware contexts leads to more workload interaction in the context of current multicore architectures. The extra resources will degrade the performance of PostgreSQL on multicores rather than improve it.

**MySQL**

For the evaluation of the MySQL engine, we took a similar approach. MySQL can use different storage engines and for these experiments, we used the InnoDB storage engine. This is a transaction-safe storage engine that uses row-level locking. Furthermore, this is the most stable engine and uses Snapshot Isolation to improve multi-user concurrency and performance. As opposed to PostgreSQL, this engine uses a fixed number of storage threads to operate over the data, thus acting like a queuing system. InnoDB does not rely on the operating system cache buffer, but rather implements its own disk buffer and manages the disk activity on its own. As a consequence, MySQL shows increased disk activity over the case of PostgreSQL.

To assess the best ratio of the number of threads to CPUs used in the InnoDB engine to the number of cores, we evaluated the performance of MySQL while varying the number of cores and threads. As a result of these experiments, we observed that the best performance is obtained when running one thread per core. In all the experiments presented below the number of storage threads in the storage engine is equal to the number of threads assigned to the MySQL engine.

**TPC-W-B** Figure 3.6 shows the performance of MySQL with an increasing number of TPC-W-B clients and cores. To ensure fairness, the experiments were performed under the same conditions. The x axis presents the number of clients used to generate the load. For best performance the clients are connected to the database using the MySQL Type 4 JDBC driver. The y axis in Figure 3.6(a) presents the average throughput over the entire run while, in Figure 3.7(b), it presents the average query response time in seconds.

Analyzing the throughput plot (Figure 3.6(a)), we notice that MySQL has a different behavior than PostgreSQL. Due to the different architecture of the engine, which re-
lies on queuing, once MySQL reaches the saturation point, it will not show a drop in throughput since the number of threads in the engine remains constant. As a consequence, the response time (Figure 3.7(b)) shows a linear increase. Each line in the plot represents MySQL running with a different number of cores.

Varying the number of cores, and implicitly increasing the number of threads in the storage engine, adds no improvement in throughput or response time. Even worse, the extra number of cores and the increased degree of parallelism creates performance problems for MySQL. In the case of TPCW-B (Figure 3.6), we notice that MySQL can scale up to 12 cores and, after this point, the performance starts to degrade with the increasing number of cores.

**TPCW-S** In the case of TPCW-S (Figure 3.7), the reduced number of long running transactions allows the execution of more short lived transactions, thus the system’s throughput increases. We notice that up to 12 cores the throughput increases. From that point, the increased number of threads starts to increase the amount of locking and degrade the overall system performance. We can conclude that the increased number of available storage engine threads cannot be efficiently exploited by the MySQL engine. As the throughput increases for a small number of clients and with an increased amount of update transactions (in the case of TPCW-S), we can conclude that, in both cases (TPCW-B, TPCW-S) the performance bottleneck is not the disk I/O but rather contention over shared data structures.

**TPCW-O** In the case of the ordering benchmark of TPCW (Figure 3.8), the system is limited by the contention over the disk I/O. The performance does not increase nor degrades with an increasing number of clients showing a contention over the disk.

Summarizing the current results, we notice performance degradation only when increasing the degree of hardware parallelism. We can conclude that MySQL shows scalability problems in all OLTP scenarios varying from read most workloads to write most scenarios. The generality of this statement is shown by the fact that in the case of small databases MySQL presents a similar behavior [79].
Chapter 3. Evaluation of Database Engines

![Graph showing throughput and response time for MySQL with 20GB database running TPCW-B](image)

(a) Throughput

(b) Response time

Figure 3.6: MySQL with 20GB database running TPCW-B
3.3. Performance Evaluation

![Graph showing throughput and response time for MySQL with 20GB database running TPCW-S](image)

(a) Throughput

(b) Response time

Figure 3.7: MySQL with 20GB database running TPCW-S
Figure 3.8: MySQL with 20GB database running TPCW-O
3.3. Performance Evaluation

3.3.2 Load Interaction

To understand the reasons behind the performance degradation, we took a closer look to the response time. The first step was to analyze which queries from the TPCW mix show an increased response time with the increasing number of cores. By analyzing the response time breakdown presented in Figure 3.9 we observe that one query dominates the entire response time and it also shows an increase in response time with an increase in the number of cores. This query is the Best Sellers query.

The Best Sellers query is an analytical query that returns the 50 most sold books from the last 3333 orders. The query is one of the most expensive queries in the benchmark, since it needs to aggregate and perform scans over the items, orders and customers data which is one of the largest and frequently accessed tables. On the one hand, the query needs to lock and access a large amount of resources and, while doing this it, creates a large amount of cache misses. On the other hand, all the other queries must wait for it to release the locked resources.

When running the Best Sellers query alone we noticed a lower throughput than in the mix due to the interference among concurrently running queries and, again, a degradation in performance as the number of cores increases as shown in Figure 3.10.

Once we removed this query from the workload we noticed that the throughput increases and the database engine can take advantage of the number of cores as shown in Figure 3.11. While this query is just 11% of the TPCW-B benchmark, it has a large impact over the performance of the entire system.

---

Figure 3.9: Detailed RT of PostgreSQL with 20 GB database running the TPCW-B mix

---
The reason behind this behavior is the fact that conventional database engines assign threads to operations and do one query at a time optimizations. The execution plan is optimized for individual queries and does not take into account the other queries running concurrently in the system. As a result, concurrent transactions significantly interfere with each other. This effect is minor in single CPU machines where real concurrency
3.3. Performance Evaluation

![Graph showing throughput and response time for PostgreSQL with 20 GB database running TPCW-B mix without the Best-Sellers transaction.]

Figure 3.11: PostgreSQL with 20 GB database running TPCW-B mix without the Best-Sellers transaction

among threads is limited. In multicores, the larger number of hardware contexts leads to more transactions running in parallel which in turn amplifies load interaction.

The degradation effect of load interaction arises from the competition for resources, which becomes worse as the larger number of cores allows more queries to be started
concurrently.

Similar effects have been observed on MySQL and were presented in [88], when using full table scans. These database operations require a large memory bandwidth slowing down all the concurrent operations.

### 3.3.3 Contention

One of the reasons of load interaction is contention. Database engines rely on shared data structures and locking primitives to ensure consistency and enforce ACID properties. A higher number of threads will increase the concurrent access on these structures, leading to longer wait times and contention.

To further analyze the causes of performance degradation with the increasing number of cores and clients, we have profiled PostgreSQL using a runtime system profiler [63].

One of the main effects of contention over shared and locking data is an increased amount of caches misses. On the one hand, every time a thread acquires a lock or updates a value in a shared data structure, the cache coherence protocol must ensure that the other caches are updated thereby producing larger delays with the increased number of hardware contexts. On the other hand, queries that go over large amounts of data end up evicting data used by other queries that run on the same processors thereby increasing the amount of cache misses. When long running queries end up waiting for locks, threads are suspended and can be rescheduled on other cores or processors. The impact on thread migration on cache misses is even worse.

The L2 data cache miss ratio is the portion of data accesses that missed the L2 data cache. For best performance, the L2 data cache miss ratio should be as small as possible.

To compute the L2 data cache miss ratio ($L_{2DC}$), we have used the direct method presented in the AMD Developer Manual [31]. The AMD Opteron processors provide the following CPU counters: $L2_{miss}(L_{2DM})$, $L2_{fill\_write}(L_{2CF})$ and $L2_{requests}(L_{2CR})$. Where ($L_{2DM}$) is the number of requests not found in the L2 cache, the L2 cache fills ($L_{2CF}$) is the number of L2 cache fill/write-back requests and the ($L_{2CR}$) is the number of L2 cache requests. We compute the L2 cache miss ratio with the following formula:

$$L_{2DC} = \frac{100 * L_{2DM}}{L_{2CR} + L_{2CF}}$$

(3.1)

To analyze the correlation between the effects of cache misses, database engine
3.3. Performance Evaluation

performance and number of cores, we analyzed the evolution of level 2 cache misses while running the workloads presented in Section 3.3.2.

The effect of cache misses with a varying number of cores can be seen in Figure 3.12(a). The plot shows a comparison of the L2 cache misses when running PostgreSQL on 8 cores and 48 cores with 225 clients with TPCW-B, TPCW-B without the Best Sellers query and just the Best Sellers query. When running the TPCW-B benchmark without the Best Sellers query the cache miss ratio is only marginally influenced by the increasing number of cores. In the case of the Browsing and just Best Sellers the increasing number of cores produces an increase of the cache miss ratio even if the number of clients is constant.

After analyzing the OProfile traces we have observed that most of the cache misses...
Figure 3.13: Comparison of cache misses when running PostgreSQL with 20 GB database while varying the number of TPCW-B clients.

are due to the s_lock function of PostgreSQL. This function implements a hardware-dependent implementation of spinlocks in PostgreSQL. Figure 3.12(b) shows that from the total amount of cache misses are due to locking and contention over shared data structures. The effects of interaction have a higher impact when we increase the amount of large scan queries in the workload and allocate more resources to the engine. In this case 80% of the time in for the TPCW-B and close to 90% for the Bestsellers workload are is spent in managing locking and contention over shared data structures. The problems of locking and shared data structures of PostgreSQL have been confirmed by other research in the area of operating systems such as [19].

The evolution of cache misses while varying the number of clients and the number of cores for PostgreSQL running TPCW-B as presented in Figure 3.13, shows that
the number of clients has a different impact on the cache misses while allocating the same number of cores. The overall amount of L2 data cache misses increases slightly between 100 and 225 clients when we allocate 8 cores to the database system as presented in Figure 3.13(a). The highest performance degradation appears once we increase the number of clients and increase the number of cores allocated to the engine. As a consequence, the cache miss ratio doubles from almost 25% to more than 50%. In conclusion, the increasing number of hardware contexts and clients are the main cause of contention over shared resources.

Contention over locking primitives (\texttt{e.lock}) is presented in figure 3.13(b). As the number of available cores increases, the amount of cache misses due to locking increases dramatically. This means that, with an increasing number of cores, more time is spent waiting for locks. We would expect to see an increase with the number of clients, but not with the number of cores. The reason behind this effect is the real parallelism offered by multicores: more hardware contexts allow more threads to compete for the locks.

We can conclude that, in the case of PostgreSQL, the increasing number of cores creates contention over shared data structures and synchronization primitives due to contention and load interaction. The increased number of cores allows more threads to compete for shared data structures leading to more contention.

While MySQL has a different architecture and also does not manifest performance degradation when running the \emph{BestSellers} query, it still has the same scalability problems due to synchronization and locking primitives.

For this reason, when analyzing the MySQL engine, we have monitored the number of times a thread waited for a lock until it was rescheduled by the OS. A thread yields to the scheduler ever time it blocks when waiting for a lock. The longer the threads need to wait, the more contention over synchronization primitives is in the system, leading to an increased amount of thread yields to the scheduler.

Figure 3.14 shows the average amount of thread yields to the scheduler for a transaction. For this analysis we used three different core configurations (12, 24 and 48) and three different scenario levels for each. The \emph{under-load} scenario is done with 12 TPCW-B clients, \emph{perfect load} uses the same number of clients as cores while \emph{overload} uses 200 clients.

Analyzing the lock contention, we notice that, the number of clients does not influence the time to acquire the locks as long as we use a reduced number of hardware contexts, namely 12 cores for the analysis presented in Figure . Once we allocate more cores to the database engine, the contention over locking primitives starts to increase producing
Chapter 3. Evaluation of Database Engines

Figure 3.14: MySQL contention for locks with a 20 GB database running TPCW-B

performance degradation. Due to the design of the engine’s architecture, which allocates a fixed number of internal storage engine threads, we can see that increasing the number of clients does not add any extra thread yields. Further, analyzing the causes of contention, we notice that the dominant cause is the wait for row level read latches and general mutexes that are used in MySQL through the entire code base. The wide spread of the locking primitives makes the attempts of improving performance of the engine a challenging task.

We can conclude by saying that MySQL as well as PostgreSQL face contention and load interaction problems when running on multicores. With an increasing number of cores, these effects start to produce performance degradation due to the longer wait time on synchronization and locking primitives or contention due to shared data structures. The validity of our experimental evaluation is also confirmed by experiments done in related works [47, 53, 19, 46] previously presented in Chapter 2.

Our previous experiments [79] show the same effects are present on smaller database sizes and larger numbers of clients while using the same database engines. Conse-
3.3. Performance Evaluation

Figure 3.15: NUMA Nodes in the 4 way AMD Opteron Magny Cours

sequently, we can state that this is a general behavior in a wide range of configurations, database sizes and number of concurrent clients.

3.3.4 OLAP Performance

To evaluate the performance of database engines in an OLTP scenario we analyzed the performance of each TPC-H query with different of clients when running on 48 cores using PostgreSQL. Due to the analytical nature of the TPC-H benchmark there is no contention over locking or synchronization primitives. By the means of this benchmark we want to address the problem of contention over memory resources, memory locality and thread scheduling.

In M-CMP architectures memory access time depends on the memory location relative to a processor. Hence, each processor has its own local memory which is faster to access compared to non-local memory. This memory design is called NUMA, and each local memory region is called NUMA Node. The AMD Opteron used in our experiments has 4 processors, each one consisting of two dies with 6 cores each. Each die has its own local-memory controller resulting in a total number of 8 NUMA nodes. Each NUMA node has a memory size of 16 GB (Figure 3.15), resulting a total memory for each CPU of 32GB.

The OS schedules various threads on different cores based on the utilization level of each core. Without taking into consideration contention over memory resources, two threads can run on the same CPU and, in the case of multicores, they can evict
Figure 3.16: Evolution of individual TPCH query response time with an increasing number of clients when using a 10GB TPC-H database (SF=10) and PostgreSQL on 48 cores

each other’s caches. Since the CPU fetches the same data from main memory multiple times, more pressure is put on the memory system.

We experimented with a TPC-H database that has the scaling factor 10 (10 GB of real data) and applied all the necessary indexes (final database size 24 GB) to improve query execution times for this experiment. As the system used to run the database engine has 48 cores and PostgreSQL creates a new thread for every connection, the average response time should stay constant if the OS schedules threads efficiently and they do not interfere with each other. We used up to 48 TPC-H clients for the test and ran 20 full TPC-H runs to compute the average result times for each query. We have analyzed the memory requirements, execution plans for each individual query to optimize the PostgreSQL memory and buffers. The TPC-H benchmark is composed of read-only queries that aggregate large amounts of data.

Figure 3.16 shows the evolution of the average response time for each TPC-H query with a varying number of clients. With the increasing number of clients, we notice that the response time of almost all the queries, and especially the queries that need to execute nested loop joins (Q18, Q9, Q21, a.o.) starts to degrade with an increasing number of clients. The reason behind this degradation is contention over memory resources and poor memory locality. This is also confirmed by the fact that the effect takes place even if we still have available hardware contexts and CPUs. On the one hand, the OS is not aware of the application requirements; therefore the OS as well as
the PostgreSQL engine do not allocate data close to the cores that process it. It is also
the case that threads waiting for resources are suspended and wake up on other cores,
in which case the memory blocks must be re-fetched. On the other hand, the database
engine is not designed to take advantage of the available resources or the hardware
topology. Once the number of concurrent threads starts to increase, the OS scheduler
assigns them to the cores that have less load. This placement leads to memory locality
problems as threads need to access memory from remote NUMA and combined with
delays due to remote memory access, it adds up to an increased overall latency.

3.4 Discussion

This section has presented a performance evaluation of two of the most popular open
source database engines running on multicore machine. We have measured the per-
formance of the engines with varying the number of cored and clients. In the same
time we have used both OLTP as well as OLAP benchmarks to assess the scalability
properties of the engines.

As a conclusion to our experimental evaluation we notice that the two database en-
gines face scalability problems both with number of cores as well as with the number
of clients. The main scalability problems were traced to load interaction as well as con-
tention over shared primitives and locking data structures. In conventional database
systems concurrent transactions interfere with each other, since the engines are doing
one query at a time optimizations. This effect is not noticed with a small number of
cores. Once we deploy the database engine on a multicore machine the larger number
of hardware contexts leads to an performance degradation due to the amplified load in-
teraction effect. Database engines rely on shared data structures and locking primitives
to ensure consistency and enforce ACID properties. With the increase of available com-
putational cores, the increased number of concurrent threads running in the engine are
exerting more pressure on the synchronization primitives. Contention over shared data
structures presents itself as an increase in cache misses and longer times to acquire
locks. In the case of the OLAP evaluation we notice that the database engines have
scalability problems due to data locality and memory access times. The OS schedules
various threads on different cores based on the utilization level of each core, without
taking into consideration data locality or contention over memory resources.

Finer grained locks, improved scheduling techniques and data locality are the main
objectives to achieve scalability on current database engines running on modern mul-
ticores. To increase the performance of existing engines without a complete engine redesign, we propose to use a middleware replication system. The replication system distributes the load over a series of database engines, thus reducing contention over shared data structures and locking primitives. To achieve this goal, we propose to design a middleware replication system that will hide the replication mechanism and act like a normal database engine that offers SI isolation level. SI is well known for the performance advantages due to the reduced amount of locking. Using these approaches, we have little contention over the engines as we replicate the locking and synchronization primitives as well as the shared data structures. To improve scalability of current application running on modern multicore machines, the resource management needs to be done at the application level [18, 93, 12]. By using a replication system as the underlying framework, we can satisfy this requirement by pinning replicas to specific cores.

The next chapter will present the resulting system, called Multimed, with its internal components and architecture. We present the building blocks of the system as well as an analytical model that can be used as a guideline to obtain optimal Multimed deployments.
The Multimed system was presented in [79] and is part of a joint work with Tudor Salomie and Jana Giceva. The next sections are going to present the architecture of the system and its components. Multimed is the framework used to show how we can achieve scalability and extensibility of current database engines running on multicores.

Multimed is a middleware replication system based on RSI-PC [72] designed for multicores. To the clients it acts like a usual database engine that offers SI and hides the replicated nature of the underlying system. The Multimed system can be used with any database engine that provides a JDBC interface. As in the case of Ganymed [74], the Multimed middleware layer coordinates the execution of transactions across a set of replicated database engines.

We start by presenting the main design decisions needed for the transitions to multicores from the cluster-based replication systems. Section 4.2 shows details about the system architecture and its components. Next we introduce the analytical model of the system and we conclude with a discussion about its features and characteristics.
4.1 Transition to Multicores

Multicore architectures are shared everything architectures, as opposed to shared-nothing in the case of cluster systems. Thread and data placement plays an important role. In the case of multicores contention over shared resources (memory, disk, I/O a.o.) can easily lead to performance degradation. This is just one of the reasons why Multimed uses a different architecture than in the case of Ganymed.

Ganymed uses a separate machine to act as the transaction scheduler (dispatcher) and each database replica resides on a separate cluster machine that has its own local adapter. The dispatcher talks only with the adapter and not directly with the database engines. Adapters are also used to handle errors or failures that might arise, add extra features or handle query rewriting [72].

Ganymed manages a set of replicas distributed over a set of cluster machines. In contrast, Multimed is deployed inside a single multicore machine. This change imposes architectural challenges and requires careful resource management.

The Multimed middleware has fewer available resources because it is collocated on the same multicore machine as the database replicas. To scale on multicores, it needs to take advantage of increased parallelism, reduce the communication between the nodes and have a small memory footprint. To achieve these goals, the middleware does not use separate adapters, but groups the functionality into one package that is deployed on a set of cores. High throughput is achieved by using a few resources for the middleware and running more database replicas on the remaining resources. This approach also reduces the access paths to the components, while keeping the data local to the cores.

Another aspect of Multimed not present in Ganymed due to its cluster-based architecture, is the need for resource management and allocation inside the multicore machine. To scale on multicores, Multimed must deploy the replicas in an efficient way to preserve data locality and reduce the concurrent access to shared data structures and synchronization primitives.

Multimed is deployed on a single multicore machine. For this reason, it does not consider communication latencies or failure of nodes due to network interruptions. These are cases that occur often in distributed systems. In the case of multicore architectures, CPU failure is a critical problem since most of the times it leads to a full system crash. Solutions to handle system failure use parallel schedulers or backup satellites to ensure dynamic generation of replicas. By not handling these cases, the complexity of the Multimed middleware layer is reduced.
4.2. Multimed Architecture

In conclusion, to efficiently scale on multicores, the Multimed system has to efficiently use the highly threaded nature of the architecture, manage the available hardware resources between the replicas, enforce data locality and take advantage of the fast communication between the nodes that now takes place locally inside a single machine.

The next sections present the architecture and main components of the Multimed system.

4.2 Multimed Architecture

Multimed is built as a client server architecture. It is composed from a client interface which acts as a communication front-end, and a backend that connects to a set of database engines. The communication component provides a JDBC Type 3 interface to the clients. This interface hides the replication nature of the underlying system, and presents itself to the clients as a usual database engine that offers SI level serialization. Multimed does not rely on specific functionality in the database engine and uses standardized communication interfaces with the clients and databases engines. As a consequence, it can be deployed on any OS and without any programming language limitations, OS or database engine constraints. The backend connects to a set of database engines and implements the RSI-PC replication protocol presented in [72]. Communication with the underlying database engines is done using JDBC drivers specific to the database engines. This non-intrusive approach enables a flexible way of using and extending the Multimed middleware, without the need to change the existing application or the underlying database engines.

From the operational point of view, the system has two phases: the setup phase and the operational phase. The setup phase consists of the allocation of components to specific hardware resources. This phase also includes the fine tuning phase of the system to match the underlying hardware architecture and workload specific properties. In the operational phase, the Multimed middleware coordinates the execution of different transactions on different nodes, while monitoring the amount of free resources available at each node (CPU, memory) to ensure an even load between replicas.

Presented in Figure 4.1, the Multimed architecture is based on three main components the communication component, the dispatcher and the computational nodes. Communication with clients is implemented in the communication component as an asynchronous server. The transaction scheduler and load balancer is implemented in the Dispatcher. Transactions are scheduled over a set of computational nodes. The set
of Computational Nodes represent the System Model which manages the deployment and mapping of database engines to hardware resources.

The next section presents the replication model implemented in the Dispatcher, which ensures that each transaction is executed under high consistency guarantees according to ACID principles.
4.2.1 Replication model

The replication model implemented by Multimed is Lazy Update with Primary Copy as presented in [40]. This model allows update transactions to a single master database and distributes the read transactions over the slave replicas. The master node ensures durability and consistency and executes all the specific triggers, stored procedures and user defined functions. The slave nodes are lazily kept in sync with the master nodes, therefore the replicas can contain old data.

Multimed uses the RSI-PC replication protocol [72] which keeps track of the status of each replica and always assigns transactions to the up-to-date nodes. If none of the satellite nodes are up-to-date, the master node is always up-to-date. The scheduler does not need any extra information about the underlying data or to implement a row or table locking mechanism. It keeps track of the status of each satellite and makes sure that the updates are applied in the same order at each node as on the master node. The isolation level enforced by Multimed is SI. Thus, each query is guaranteed to see all the changes committed before the start of the transaction. If a satellite replica is not up-to-date, or the snapshot is out of date, new transactions are not scheduled to the node, thereby allowing it to catch up while the existing transactions do not interfere with the incoming updates. The use of SI allows Multimed to update nodes even if read transactions are currently executing on the node with a small impact on performance.

The amount of data that needs to be propagated to the satellite node can be reduced by using partial replicas. This approach introduces an overhead due to the extra complexity introduced in the transaction scheduler. Full replication allows the system to schedule transactions and propagate updates to every node as they all have the same schema. Partial replicas allow a smaller memory footprint and reduced contention over memory resources as well as an increased number of satellites inside the multicore machine. To improve performance, Multimed allows the use of both full and partial replicas as well as mixed combinations.

4.3 Multimed Components

Multimed has three main components, the client communication interface, the dispatcher and the computational nodes (all the nodes define the system model). Each component has specific roles and are packaged as a single deployment unit.

The main design goals for each component were to reduce the amount of shared
data, communication and memory footprint as much as possible. This approach allows Multimed to run efficiently on just two cores for up to 400 clients. As shown in [79], the Multimed engine can efficiently provide enough computational power to support an increased amount of clients without loss in performance.

4.3.1 Communication Component

The Communication Component implements the communication subsystem between the clients and the Multimed middleware.

On the client side, the communication component exposes a JDBC interface. This is a standardized interface for database-independent connectivity between an application and a wide range of databases. This interface allows database clients to interact with a database engine and do operations such as open connections to a database, send an SQL query, get metadata about the database engine, etc. The main goal of JDBC is to provide a “Write Once, Run Anywhere” solution for Java applications. On that account, any application written in Java for a database engine can add the Multimed driver and run without major adaptations to the application side [65].

This component is one of the critical parts for the overall system performance. For the system to perform at its best, the communication component needs to support a large number of connections and manage a high number of requests. To prevent contention over the communication component, the communication subsystem has been implemented as an asynchronous server using the Java Native I/O (Java NIO) libraries. To further improve the performance of the system, we have based our implementation on the Apache Mina 2.0 framework [6]. For this reason, the server component of the Multimed system is a non-blocking message processing system, enabling it to support a large number of concurrent client connections without an extra overhead. On the server side, the main role of the component is to keep track of the open client connections and their requests. It needs to efficiently pass the client requests to the dispatcher and from the dispatcher to the computational nodes. Once the nodes have finished processing the requests, the communication component has to send the replies back to the client.

4.3.2 Dispatcher

The Dispatcher has three main functions: schedule transactions, collect statistics on the underlying satellites and ensure load balancing of transactions between the computational nodes. It also has to coordinate and manage the replication mechanism,
4.3. Multimed Components

which consists of the writeset extraction and propagation from the master node to the replicas as well as ensuring a consistent view over the replicas.

Transactions are received by the Dispatcher from the communication component and scheduled on the underlying computational nodes. All update transactions are routed to the master node, while all read transactions are scheduled on one of the satellite nodes. The Dispatcher needs to differentiate between the read and write transactions, as the JDBC driver offers a standardized method on the Connection object to specify whether the transaction is Read-Only. As this functionality can be easily achieved in Java by calling the `Connection.setReadOnly(true)` method for every Read-Only transaction, we decided to use this lightweight implementation rather than increasing the complexity of the middleware. If the client does not mark its transactions accordingly, we assume that it is an update transaction and the Dispatcher will route the transaction to the master node. Consistency is still guaranteed but will have the effect of an increased load on the master node.

The load balancing function of the dispatcher takes care that the satellite nodes are used efficiently to improve performance. The dispatcher can schedule transactions on the satellites in a round robin fashion, based on the current load of the node (CPU utilization, number of active transactions running on the satellite, etc.). Computing the current load can be done based on information available in the dispatcher or communication component. The middleware is implemented as a single package, so we do not have to worry about how to send the load statistics over the network and increased latencies. Satellites are eligible to run transactions if two conditions are met: first they must have the available resources to execute the transaction and second they must have a fresh copy of the data. As resources, we consider the right schema (in the case of partial replication) and available computational resources. If the satellite cannot answer the query, because it contains stale data or it is overloaded, the dispatcher can delay the transaction for a certain amount of time and try to rebind it or can execute it directly on the master node.

A consistent state over the entire system is ensured by the replication mechanism. With the commit of an update transaction on the master node, all the changes produced in that transaction are stored in a writeset and propagated to the satellite nodes. The writesets are applied in the same order on the satellite nodes as they have been committed on the master. This ensures that at every point in time a satellite contains a consistent snapshot of the master. As in the case of Ganymed [72], every time a transaction commits on the master node, the dispatcher receives the writeset and increases an internal counter that stores the Snapshot Count Number (SCN). Each computational
node has a local SCN \((SCN_{Ni})\) that is increased every time the writesets are applied. The master node SCN is also used to tag transactions as they enter the dispatcher \((SCN_{Ti})\). A satellite is allowed to execute a read transaction if it has applied all the writesets prior to the read transaction. Or in other words \(SCN_{Ti} \leq SCN_{Ni}\).

### 4.3.3 System Model

The system model describes the mapping of resources (CPU, memory, database buffer sizes, ports) to computational nodes. In the current version, the model defines a static partition of the resources without taking into consideration a dynamic repartitioning of the system at runtime.

The main components of the system model are computational nodes and the nodes manager. The model is used in both operational phases of Multimed. In the setup phase, the node manager deploys the computational nodes on the hardware resources and creates the connections between the nodes and the Multimed middleware based on the system description. Figure 4.2 shows a possible deployment of Multimed using one master node and 4 satellite nodes deployed on a 4 way AMD Opteron Magny Cours multicore machine.

In the operational phase, the system model provides specific node information to the dispatcher such as the node type (master, or satellite), replication scheme (full, partial), supported transactions and state.
4.3.4 Computational Nodes

Computational nodes coordinate a set of hardware (CPU, memory, etc.) and software (database engine, buffers, connection pools) resources. The dispatcher schedules transactions to computational nodes. Nodes forward the transactions to the underlying database engines and send the replies back to the clients using the communication component. Another role of the computational nodes is to manage local transaction execution. There are two types of computational nodes, master nodes and satellite nodes. Master nodes manage the execution of query and update transactions as well as the writeset extraction. Satellite nodes handle the query execution and make sure that the writesets are applied correctly. The dispatcher assigns each transaction (which can contain multiple queries) to one computational node based on its capabilities (computational resources, available connections, data freshness, node type, etc.).

To ensure future scalability of the system, each computational node runs as a separate thread. The main operations of each node require local transactions to be executed and the results returned to the clients. The separation ensures that threads do not compete for resources or block over shared data structures.

Master Node

The master node manages the master database engine. The dispatcher will route to the master all the update transactions. The main role of the master is to ensure consistency and persistency. As in any Lazy Primary Copy replication system, the master manages the concurrency control, and the updates are then propagated to the satellite nodes. Once the updates are committed on the master database engine, the writesets are extracted by the master node and sent to the dispatcher. At this moment, the Dispatcher increases the SCN and propagates the writesets to all the other satellite nodes. The communication with the underlying database engine is done using a JDBC driver. Since the driver does not provide a way of obtaining information about the transaction commit number, the write transactions on the master node need to be serialized to ensure an atomic increase of the SCN.

Satellite Nodes

Read transactions are assigned to satellite nodes. When a read transaction arrives at the Dispatcher, it is assigned a timestamp which in our case is the current SCN. The
Dispatcher will schedule the transaction to the first available satellite that has applied all the writesets up to the transaction’s timestamp. Since the master node ensures persistency and durability in the system, the satellite nodes do not have to ensure the same guarantees. As opposed to the master nodes, the satellite nodes can reside in main memory, and do not have to persist the changes to disk. This optimization allows us to alleviate the pressure on disks and improve the performance of the satellite node. The increased contention on disks, was not present in cluster systems as each satellite had a dedicated machine with its own set of resources. In the case of a satellite node failure, no data is lost as everything is durably committed on the master node.

As in the case of the master node, communication with the underlying database engines is done using the JDBC driver specific to the database engine. To ensure consistency over the satellite node, we enforce a total order when applying the writesets to the satellite node. The writesets are applied in the same order in which the generating transactions were committed on the master node. For partially replicated satellites, only the required writesets are propagated to reduce the amount of communication between the dispatcher and the underlying nodes.

As we will see in Chapter 5, the key to scale on multicores is to use more satellite nodes running on fewer resources. In this way we run multiple database engines close to their best performance. Chapter 6 will show how specialized satellites can be used to extend database functionality by adding new operators and extensions to the master database engine.

4.3.5 Writeset Extraction and Propagation

Extraction and propagation of updates from the master to the satellite nodes is one of the most important components of our system. To ensure a total ordering of the update transactions, the master needs to serialize the commit order of update transactions. Once the transaction has been committed at the master node, the effects produced by the update transaction are collected in a writeset and applied on each satellite node in the same order as they were applied on the master node.

Serialization of update transactions can pose a bottleneck in write intensive workloads, however from Section 3.3.1 we notice that database engines suffer more from scalability problems in workloads with mixed workloads. As more read transactions can be executed in parallel on the satellite nodes, the introduced latency is hidden by the faster execution of the read transactions. The performance of update queries is also improved at the master node as all the read transactions are no longer collocated inside
4.3. Multimed Components

the same engine and no longer compete for resources, or interact with each other.

One way to propagate update transactions from the master node to the satellite nodes is to send the committed transactions to the satellite nodes as SQL statements. This is the easiest approach, but has performance and consistency drawbacks. First, the update transactions can contain complex statements and, as such, each satellite node will have to reevaluate and execute every transaction locally producing a high overhead. Second, SQL statements can contain different functions that return timestamps or random numbers. Due to the lazy nature of the replication system, at the time of the update, these values are going to differ from one computational node to the other creating consistency problems. Due to its simplicity this approach is mostly used in research systems [58, 54].

Another way is to extract the writesets from the master node and propagate it as a set of updates to all the satellite nodes. As seen in Section 2.7, existing replication systems take different approaches to extract and propagate updates from one replica to the other. These solutions range from extensions to the engine to simpler trigger based solutions that capture the changes of update, insert or delete statements inside a transaction. Each one of these approaches has its own advantages and disadvantages. Engine extensions ensure an efficient way of extracting the updates, but due to tight integration they sacrifice portability and are highly integrated. Extracting writesets using database engine triggers or custom functions introduces an execution overhead [72] but allows a generic approach.

Aiming for a generic and flexible approach, Multimed implements writeset extraction using custom functions for the PostgreSQL engine and triggers installed in the database engine for the MySQL engine. We needed different implementations since custom functions in MySQL cannot create global accessible data structures. As we will see in the evaluation sections, both solutions produce good results as the advantages obtained by reducing load interaction and contention surpass the introduced overheads. The custom functions and triggers present a generic approach and are independent of the underlying database schema. At deployment time, the System model reads the database schema from the underlying master node and installs the triggers accordingly. Multimed uses row-level insert, delete and update triggers in the master database engine on the union of the tables replicated in all the satellites that are fired after a transaction is committed. Whenever a row is modified, the triggers record the old and new values of the row (old and current snapshot). A server side function will aggregate the changes and send them to the dispatcher. The server side function is specific to the underlying database engine but is a standard feature offered by today’s systems. The dispatcher
interprets the results and passes them to each satellite node to be enqueued in the writeset queue.

If we take the generic aspect aside, one can easily implement engine specific solutions to allow concurrent write transactions on the master node, or use engine specific tools or interfaces such as the solutions presented in Section 2.7 to further improve the performance of the master node. As Multimed is a research prototype that shows that a distributed replication systems can scale on multicores, we do not specialize the system any further.

4.4 Configuring Multimed

Multimed is designed to be deployed on a single multicore machine and can be configured in different ways to use the various resources and hardware configurations offered by the multicore architectures. As opposed to a traditional database engines, the computational node in Multimed can be configured independently to ensure a fine-tuned deployment and execution of the workload.

This section will provide an overview over possible Multimed configurations as well as some of the optimizations used in the evaluation. Detailed configurations as well as performance results will be presented in the next chapters.

The first optimization was done at the communication component. Messages from the client to the server side are small (under 100 bytes). They are packaged to efficiently use the network bandwidth, but this approach also introduces latency. We have disabled this option in the Java Sockets by setting the \texttt{TPC\_NODELAY} option. This optimization reduces the RTT for messages at the cost of a higher number of packets in the network. Communication with the underlying databases is done using Type 4 JDBC Drivers that use native protocols. These drivers ensure the best performance and increased functionality. Opening and closing connections to the database engine introduces a large overhead. Therefore, Multimed's computational nodes use internal connection pools to communicate with the underlying database engines.

As mentioned in Section 4.3.4, satellite nodes do not need to ensure that every update is persisted on disk. To obtain better performance, Multimed disables the synchronous commit of transactions on satellite nodes.

Multimed deploys computational nodes on a set of resources and each node can be deployed independent of each other. Some possible configurations are presented in [79]. Nodes can be deployed in a \textit{naive approach}, where all nodes are fully replicated
and data is stored on disk. As contention over synchronization primitives and shared data structures is now distributed over a series of nodes, the system provides improved performance but creates contention on disk. *In-memory deployment* assumes that the master node is deployed on disk while the satellite nodes use full replication and reside in main memory, thereby reducing disk contention. However to avoid contention over memory resources, the number of satellites is going to be reduced. Using an *in-memory with partial replication* deployment allows Multimed to use an increased amount of satellites in main memory, as some satellites can use partial replication. For the Dispatcher to know how to schedule transactions, the System model needs to describe the replicated tables on each satellite node. The memory savings are consistent, especially for complex queries that use just a part of large databases, for example to create a replica that can handle the *BestSellers* query we need just 5.6GB as compared with the initial size of 20GB.

Other configurations can use different combinations of in-memory and on-disk as well as full and partial replicas. If the multicore machine has network attached storage, Multimed can add an increased number of satellites. One can see these different types of satellites as different specializations. For example, in-memory satellites ensure best performance on complex long running transactions, while disk-based satellites can produce contention on disk and are more suitable for short, fast running transactions.

As Stonebreaker wrote, there is no "One size fits all" database [86] and this is reflected in the fact that there is no general configuration for Multimed. The system deployment and the resource allocation to its satellites is specific to the workload type. As in the case of database engines, the system must be configured and optimized according to a workload’s characteristics.

### 4.4.1 Deployment Guidelines

Designed to take advantage of multicore architectures, Multimed deployments need to take into consideration the underlying hardware architecture. To achieve maximum performance, deployments should follow the design guidelines mentioned in Section 2.2.2. Deployment of computational nodes should be aimed towards improving data locality. For this reason, all the deployments try to reduce the amount of sharing of computational resources between different processors. In our experiments, the satellites try to use processors that belong to the same NUMA node, to ensure memory locality and reduced overhead due to the cache coherence protocol. Computational resources (CPU, memory) must be assigned to the computational nodes based on the load, type
of queries and number of connections that they must handle. The best configuration can only be achieved in an experimental way as different hardware resources and multicore architectures have different characteristics such as frequency, bandwidth, latency architecture of the interconnect, or cache coherency protocol.

A poor deployment of Multimed will not take into consideration the architecture of the multicore machine. Figure 4.3 shows such a deployment. Here each node is bound over multiple cores that run in different CPUs. This configuration will cause contention over memory resources as cores will have to access memory in different locations. Furthermore, a lot of pressure will be put on the cache coherence protocol due to the increased amount of cache sharing. A good deployment of Multimed will try to avoid cache sharing, and improve data locality. Such a deployment is presented in Figure 4.2.

Other deployment guidelines address the problem of load interaction. Transactions that produce load interaction or create contention on shared resources will be executed on separate satellites. This approach removes load interaction and increases data locality. Satellites can be further specialized to improve the performance of the query (such as adding extra indexes, increased buffer size etc.). They can also implement extra functionality that is not supported by the initial database engine. This extra functionality can be new operators, such as the Skyline operator [17], or can have specialized indexes (e.g., to support text search). It is well known that index maintenance is a very expensive operation in the database engines that creates locking and slows down the update transactions. On the one hand, by adding the extra indexes on specialized satellites, we remove the work from the master node and improve its performance. On
the other hand, queries that use the extra indexes will run faster on the specialized satellite, improving response time and throughput.

In the case of OLTP workloads, Multimed can use optimized satellites for different query classes. For example, one can create replicas that are optimized for short running queries and others for long running analytical queries. Short queries require fewer resources while the others need large sort buffers and extra indexes.

As Multimed just schedules transactions over different computational nodes that communicate with underlying database engines using a well-defined interface (the JDBC Driver), the system is flexible and allows future extensions. Even if the extension can be implemented in the database engine, contention over shared resources produces performance degradation both on the entire workload, as well as on the query running the specialized operator.

4.5 Analytical Model

For a better understanding of the scalability properties of the Multimed system, we present an analytical model. This model tries to determine the scalability properties of Multimed compared with a traditional database system. We base our analytical model on a replication-based model previously presented in [81] which analyzes the scale-out properties of replicated systems using Partial Replication and 1-Copy-Snapshot-Isolation. As Multimed uses a different replication model and has a different architecture, we adapt the model to our approach.

Multimed is composed of a set of replica nodes consisting of the master node \( M \) and multiple satellite nodes \( S \) where \( S = \{1..n\} \). Each satellite node can hold a full or a partial replica of the database. To model the system more accurately we assume that the database consists of a set objects \( O = \{O_1,..,O_m\} \) which can be columns, rows or any characteristic used to achieve the partitioning. Since we use a primary master replication protocol, the Multimed master satellite contains a full copy of the database. In the case of Multimed, the master node executes all the updates and the satellite nodes apply the resulting writesets.

The workload, as presented in [81], \( L \) can be defined as a pair \((T_i, U_i)\), where \( T_i \) is the proportion of transactions (queries and updates) that access an object \( O_i \) per unit of time. \( U_i \) is the proportion of update transactions that change an object \( O_i \). Therefore for \( m \) objects we have \( T = \{T_1,..,T_m\} \) and \( U = \{U_1,..,U_m\} \). As each satellite contains a set of objects and executes a subset of read transactions we can further decompose
\( T_i = \{ t_{i1}, t_{i2}, \ldots, t_{in} \} \). The update transactions are only executed at the master node and the writeset propagation will generate an overhead at the satellite nodes.

The Multimed replication model allows for both fully and partially replicated satellites, to express this we use a function \( r : S \times O \to \{0,1\} \) to define the schema. If the object \( O_j \) exists at the satellite \( S_i \) then \( r(i,j) = 1 \) otherwise \( r(i,j) = 0 \). Similar to [81], a full replication model ensures that each satellite node contains a full database replica therefore \( \forall i \in \{1..n\}, \forall j \in \{1..m\} | r(i,j) = 1 \). A heterogeneous replication model has at least one satellite that holds a partial replica, \( \exists i \in \{1..n\}, \exists j \in \{1..m\} | r(i,j) = 0 \).

The analytical model measures the scalability property of the Multimed system as the ratio between the processing capacity of Multimed (\( C_{MM} \)) and the capacity of the database engine (\( C_{DB} \)) when running on the same given set of resources and workload.

\[
\text{scalability} = \frac{C_{MM}}{C_{DB}}
\]  

(4.1)

We assume that each database engine has a processing capacity \( C \) which is defined by the workload and the available underlying resources. The initial capacity can be evaluated experimentally for each database engine, as previously shown in Section 3.3. We can say that the capacity of Multimed \( C_{MM} \) is the sum of the capacities of all the nodes, as follows:

\[
C_{MM} = C_M + \sum_{i=1}^{n} C_{Si}
\]  

(4.2)

where \( C_M \) is the capacity of the master node and \( C_{Si} \) is the capacity of the satellite node \( i \). In the case of Multimed, we use part of the processing capacity from the satellite nodes to apply the writesets from the master node. Each satellite node uses a part of the processing capacity to apply the writesets \( R \). As each satellite node can hold a different schema, the performance overhead due to the writesets application is specific to the satellite node. Therefore, the capacity of the satellite node has two components: the capacity used to manage read-only queries \( L \) and the capacity used to apply the writesets \( R \) and \( C_{Si} = L_{Si} + R_{Si} \). The useful processing capacity of a satellite node \( L \) is as follows:

\[
L_{Si} = C_{Si} - R_{Si}
\]  

(4.3)

From this equation, we notice that the best performance is obtained from a satellite node when no updates are applied. Thus, we do not have any writesets from the master node. To achieve the highest throughput, we have to maximize the value of \( L_{Si} \). For best performance, we must ensure that each satellite node has the best resource allocation for the current workload.
As each satellite executes a part of the read workload, the amount of work done by a satellite can be computed as the total sum of read accesses to the objects it replicates, and is expressed by the following equation:

\[ L_{Si} = \sum_{j=1}^{m} C_{Si} \cdot r(i, j) \cdot t_{ij} \cdot (1 - u_{ij}), \forall i = 1..n \]  \hspace{1cm} (4.4)

As \( L_{Si} \) is the processing capacity of the satellite node used to process requests from the workload, maximizing \( L_{Si} \) is a key factor for higher performance in the case of the Multimed system. Using partial replication removes load interaction and allows higher values for \( C_{Si} \).

Each writeset extracted from the master node needs to be applied at each satellite node that contains the specific objects. The satellite node uses a fraction of the processing power to manage the writesets and creates a write overhead \( wo \).

\[ R_{Si} = wo \cdot \sum_{j=1}^{m} r(i, j) \cdot t_{ij} \cdot u_{ij} \]  \hspace{1cm} (4.5)

Partially replicated satellite nodes incur fewer updates as they hold a limited set of the replicated objects, leading to a smaller write overhead. A reduced amount of write transactions in the workload results in a smaller overhead to the satellite nodes.

The processing capacity of a satellite node is computed by replacing \( L_{Si} \) and \( R_{Si} \) from 4.4 and 4.5 in equation 4.3

\[ \sum_{j=1}^{m} C_{Si} \cdot r(i, j) \cdot t_{ij} \cdot (1 - u_{ij}) = C_{Si} - wo \cdot \sum_{j=1}^{m} r(i, j) \cdot t_{ij} \cdot u_{ij}, \forall i = 1..n \]  \hspace{1cm} (4.6)

To compute the scalability of Multimed compared to the initial database, we must compute the processing capacity of the system which is the sum of all the node capacities and subtract the processing overhead incurred when applying the writesets at the satellite nodes:

\[ C_{MM} = C_{M} + \sum_{i=1}^{n} C_{Si} - \sum_{i=1}^{n} R_{Si} \]  \hspace{1cm} (4.7)

Replacing \( R_{Si} \) from equation 4.5 in 4.7, we obtain the following formula to compute the processing capacity of the entire Multimed engine:

\[ C_{MM} = C_{M} + \sum_{i=1}^{n} C_{Si} - wo \cdot \sum_{i=1}^{n} \sum_{j=1}^{m} r(i, j) \cdot t_{ij} \cdot u_{ij} \]  \hspace{1cm} (4.8)
Chapter 4. Multimed

The master node can manage both read and write transactions. Each update transaction produces an overhead due to the writeset extraction. The processing capacity of the master node has two components $L_M$ and $R_M$.

$$C_M = L_M + R_M$$ (4.9)

The local processing capacity $L_M$ is used to process all the write requests and the part of the read requests that cannot be managed by the satellite nodes. The residual processing capacity $R_M$ is the capacity used to process the writeset extraction, and it manifests as a processing overhead that varies with the efficiency of the writeset extraction method used in the system (triggers, custom functions, etc.). The writeset extraction function uses a fraction from the processing power of the engine $we$, $0 \leq we \leq 1$.

$$L_M = \sum_{j=1}^{m} C_M \cdot t_j$$ (4.10)

$$R_M = we \cdot \sum_{j=1}^{m} t_j \cdot u_j$$ (4.11)

$$C_M = \sum_{j=1}^{m} C_M \cdot t_j + we \cdot \sum_{j=1}^{m} t_j \cdot u_j$$ (4.12)

The best performance of the master node is achieved when the entire processing capacity is used to process the workload. To achieve this, we need reduced processing costs for the writeset extraction process.

The scalability of Multimed is given by the amount of work executed at each node divided by the processing capacity of the initial database engine when using the same amount of resources.

$$scalability = \frac{L_M + \sum_{i=1}^{n} L_{Si}}{C_{DB}}$$ (4.13)

To obtain better scalability, the total performance of the Multimed nodes must be higher than the performance of the initial database engine when running on the same amount of resources. Chapter 3 showed that the performance of existing database engines degrades with the increased number of available resources. Using Multimed to deploy a set of nodes that run at the best performance of the database engine should produce almost linear scalability with the number of nodes, as long as the write extraction and propagation overhead is not a dominant part of the processing capacity of the nodes.
4.5. Analytical Model

Replacing equation 4.10 and 4.4 in 4.13, we obtain the following equation:

\[ \text{scalability} = \frac{\sum_{j=1}^{m} C_M \cdot t_j + \sum_{i=1}^{n} \sum_{j=1}^{m} C_{Si} \cdot r(i, j) \cdot t_{ij} \cdot (1 - u_{ij})}{C_{DB}} \]  \hspace{1cm} (4.14)

From equation 4.14, we observe that another factor that influences the scalability of the Multimed system is the proportion of write transactions in the workload. The proportion of write transactions in the workload affects the processing capacity of the satellite nodes. As satellite nodes execute only read transactions, an increased amount of write transactions will not make use of the processing power at the satellite nodes.

In the case of a write intensive workload, the master node needs to process a large amount of update transactions and the Multimed system can show performance degradation due to the following reasons:

1. the residual overhead due to writeset extraction and propagation over all the satellites starts to increase with a higher number of satellite nodes
2. a small proportion of read transactions leads to an underutilization of the processing capacity offered by the satellite nodes.

Satellite nodes must apply the writesets extracted from the master node. The use of partial replication reduces the amount of computational resources used to apply writesets as each satellite will apply updates for a reduced set of objects.

Given a Multimed configuration with \( S \) satellite nodes and a replication schema \( r \) and a workload \( (T, U) \), we must look for the values of \( C_M, C_{Si}, t_{ij}, u_{ij} \) that maximize the scalability properties of Multimed running on the same set of resources as the reference database engine. Achieving the best Multimed configuration given a set of processing resources and given a workload is reduced to solving the following optimization problem:

\[ \max \frac{1}{C_{DB}} \cdot \left\{ \sum_{j=1}^{m} C_M \cdot t_j + \sum_{i=1}^{n} \sum_{j=1}^{m} C_{Si} \cdot r(i, j) \cdot t_{ij} \cdot (1 - u_{ij}) \right\} \]  \hspace{1cm} (4.15)

From equation 4.15, we observe that the Multimed engine will show an improvement in performance as long as the performance of the database engine degrades or does not scale with the increasing amount of available resources. The use of multiple replicas of the database engine running at its top performance will produce better scalability of the Multimed engine. If the processing costs of writeset extraction and propagation are not higher than the processing power used to manage workload transactions, the
Multimed engine manifests better scalability properties than the initial engine when using the same available resources. The replicated satellite nodes allow the Multimed system to use the extra computational resources available in the system. In a read-intensive scenario, the performance of the Multimed engine will scale up linearly with the number of satellites, as the computational resources used for writeset extraction and propagation have a small impact over the overall performance of the engine. The use of partially replicated satellites reduces the writeset propagation overhead leading to improvements in performance.

4.6 Discussion

The Multimed system implements a distributed replicated system inside a single multicore machine. For this purpose, it adapts techniques from cluster-based replication systems and modern operating systems to achieve scalability on current multicore architectures. Multimed is a middleware based replication system that employs RSI-PC thus hiding the complexity of the replication system and exposing itself to the clients as usual database engine. The use of satellites nodes assigned to a set of resources allows Multimed to reduce contention and load interaction between queries by reducing the amount of concurrent requests on each node and replicating the shared and locking data structures. Instead of allocating all the resources to a single database engine, Multimed distributes the load over a set of smaller engines that run at their best capacity. By removing contention and load interaction Multimed’s satellites can answer queries faster than a single engine that faces contention and locking problems due to the increased amount of hardware contexts and concurrent transactions.

The performance of Multimed highly depends on the number of available satellite nodes. Due to the replicated nature of Multimed, it requires extra resources for each replica in the system, thus the number of satellites in Multimed is limited to the amount of available memory and the initial database size. Partial replication is a solution to increase the number of satellites in the system. By pushing the job of sharing resources to the application level rather than leaving it to the OS, Multimed can efficiently use the available resources by custom deployments. This approach enables deployments tailored for a specific multicore architecture, reducing the amount of cache sharing and data locality.

Multimed satellite nodes can be used to extend the functionality of the initial database engine. Such extensions can add new database operators, user defined functions, extra
indexes or extending the system’s functionality without any changes to the initial engine. Similar approaches have been used in cluster-based systems such as Ganymed [74]. The next chapters will cover the use of extensions and satellite specialization inside Multimed and their effects on performance.

As with any replication system, Multimed performs best in read-intensive workloads such as TPCW-B, TPCW-S or TPCH, while in write-intensive workloads the update extraction and propagation will create a significant overhead.

The next chapter evaluates Multimed’s scalability properties, while Chapter 6 evaluates the use of specialized satellites to further extend functionality of the satellite nodes and improve the performance.
Improving Scalability

One of the main goals of this thesis is to build and design a system that can efficiently use the resources offered by Multicores. In this chapter, we evaluate Multimed’s scalability with an increasing number of cores and clients. We use as reference databases systems PostgreSQL and MySQL and compare their performance with Multimed using the same engines as satellite nodes. Multimed allows extra flexibility when it comes to satellites location, thus we also compare the effects of different satellite configurations on the scalability properties of the system. For a fair evaluation we only compare the performance of different Multimed configurations with the performance of the underlying engine. Therefore, in our experiments we only compare the performance of PostgreSQL and MySQL with Multimed running on top of PostgreSQL and MySQL.

To gain more insights about the applicability of our approach, we evaluate the performance of the underlying database engines and Multimed under different workloads. We start with a series of transactional workloads and vary the amount of read in write transactions in the mix. Each workload is designed to stress different components of the database engine, starting with the memory system and down to the disk I/O. Multimed can manage on-disk, in-memory or partial replicated satellites. To achieve best performance, Multimed must use the right configuration for the workload. These workloads show the ability of Multimed to resolve problems such as load interaction and con-
tention over shared data structures by using replication. We continue the performance evaluation, with a complex analytical workload. For this workload, Multimed deploys configurations which take advantage of memory locality using tailored resource allocation. Improved memory locality, enables the system to achieve better performance when compared with the underlying database engine.

All the experiments in this section have executed on the same 4 way AMD Opteron 6147 CPU with 256GB of RAM presented in the evaluation of existing database engines Chapter 3.

5.1 Experimental Setup

For all benchmarks, OLTP and OLAP, we use a specific load generator that creates the client connections and executes the queries on the databases engines, a more detailed description of the load generator and testing framework was described in Section 3.2. We use the TPC-W benchmark to evaluate how different Multimed configurations can alleviate load interaction and contention over shared data structures and locking primitives in OLTP workloads. The main advantage of TPC-W is the possibility to compare multiple workload scenarios, varying from read intensive to write intensive workloads, stressing different components of the database engine. The read intensive analytical queries put pressure on the memory system, while the mixed workloads increase the pressure on locking, synchronization primitives and disk I/O.

TPC-H is the OLAP benchmark of choice, as it contains 22 read-only queries that have a wide range of complexity, starting from aggregating data over large scans to complex joins and sorts over multiple tables. TPC-H puts a large pressure on the memory system and we use this benchmark to assess the performance of Multimed configurations designed to remove contention over shared memory resources.

5.2 Scalability in OLTP Workloads

To determine the scalability of MySQL and PostgreSQL with the number of cores we have measured the throughput in WIPS and the average query RT in seconds for PostgreSQL and MySQL. The load was generated by 200 parallel TPC-W clients and we have varied the number of cores assigned to the database engine. For the evaluation of Multimed, we used the same number of clients, and deployed different configura-
5.2. Scalability in OLTP Workloads

The Multimed engine offers different configuration options when it comes to the location and type of satellite nodes (on disk, in-memory, fully replicated, partially replicates, etc.). Different configurations produce effects on the scalability properties of the system. By comparing the performance of various configurations we can estimate the system performance with different multicore architectures (such as larger memories, faster disks, larger number of cores, etc.).

In all the experiments, the emulated clients connect to the Multimed middleware and generate the load according to the benchmark’s specification. For a fair comparison, we used the same experimental setup and load generator as in the performance evaluation of the database engines, as presented in Section 3.2 and 3.1.1.

5.2.1 TPCW-B

This section presents the experimental evaluation of Multimed while running on PostgreSQL and MySQL with varying number of cores and clients using the TPCW-B mix. The experiments with varying number of cores evaluate the scalability properties of Multimed with an increasing number of resources. While varying number of number of clients enables us to evaluate the system’s scalability with increasing load.

PostgreSQL

Figure 5.1 shows the scalability with the number of cores of PostgreSQL compared to different configurations of Multimed with a TPCW-B load generated by a constant number of 200 concurrent TPCW clients. The database size used in these experiments is 20GB and a fresh copy was installed at every run (point in the plot) to ensure that results are not affected by database evolution. The x-axis shows the number of cores used by PostgreSQL or the Multimed configurations as well as the number of satellites used by Multimed.

The first Multimed deployment uses 2 cores for the Multimed’s middleware and 6 cores for the master node. The master node always resides on disk as it must guarantee persistency of the data in the case of a failure. When we compare with the performance of PostgreSQL on the same amount of cores we notice a slight drop in performance.
Chapter 5. Improving Scalability

Figure 5.1: Scalability with the number of cores of Multimed running on PostgreSQL with 20 GB database, using 200 TPCW-B clients
This is due to several factors: the master database engine runs on fewer resources than PostgreSQL, the serialization of the write transactions by the middleware and the extra latency introduced by the added level of indirection. Even so, the small drop in performance shows that Multimed has a small overhead compared with the initial database engine.

In the next steps, we added satellites to the Multimed engine and compared the performance gain to the initial engine. The AMD Magny Cours processor connects two 6 core dies on a single CPU for a total of 12 cores per processor. To reduce the amount of resource sharing, the first satellite runs on 4 cores, so it will be collocated with the master node and the Multimed system on the same CPU. Each one of next satellites will use 6 cores, collocating 2 satellite nodes in a single CPU, for a total of 7 satellite nodes in the system. To give a broader overview of how different configurations affect the performance of Multimed, we will use three different satellite configurations. The first configuration will deploy a set of fully replicated satellites on disk, the second configuration will deploy the satellites in main memory while the third configuration will use partial replicated satellites in main memory. Due to the limited amount of available main memory, in the case of in-memory deployment, the last two satellites reside on disk. In the case of partial replication, we use three fully replicated satellites, while the rest of the partial replicas have the required tables to answer the BestSellers transaction. In all cases the load balancer distributes the queries in a round robin fashion to all the up-to-date satellites.

In the case of TPCW-B, we can see that with an increasing number of cores the performance of the PostgreSQL engine starts to degrade as the increased number of hardware contexts start to put more pressure on the engine, leading to contention over shared data structures and synchronization primitives. In the case of Multimed, the throughput increases as the number of cores is increased. Comparing different configurations, we can clearly see that the performance bottleneck for our system is the disk I/O. Even if the satellites delay the writes to the disk, they do write eventually, creating contention over the disk. Once satellites are deployed in main memory the system starts to scale linearly with the number of cores.

Introducing partial replication and deploying replicas optimized for expensive queries such as Best Sellers, we remove the problem of load interaction inside the satellite nodes. This is achieved by separating long running queries and assigning them to partially replicated satellites.

To evaluate the scalability properties of Multimed with an increasing number of clients, we used a configuration with 3 fully replicated nodes, for the on disk and in main mem-
Figure 5.2: Scalability with the number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database
ory deployment. For those experiments, the Multimed system used 4 cores, the master node used 8 cores and each satellite node was deployed on 12 cores. The partial replication configuration deploys 2 fully replicated satellites, one using 12 cores and the other 6, and 3 partially replicated satellites in main memory each one running on 6 cores. Figure 5.2 shows a comparison of throughput and response time of Multimed and PostgreSQL while varying the number of clients. The results clearly point out that the increased number of clients produces scalability problems for PostgreSQL and we even have a performance drop with the increasing number of cores. In the case of Multimed, the load distribution over the satellite nodes improves the overall system performance. As the contention on disk is removed, the system starts to manifest a better performance, while load separation through partial replication produces further improvements.

MySQL

We repeated the experiments with the MySQL engine. As MySQL allows multiple storage engines, we used the InnoDB engine for our experiments as it provides SI and is one of the most used and stable engines available.

Figure 5.3 shows the scalability with the number of cores of MySQL compared to different configurations of Multimed with a TPCW-B load generated by a constant number of 200 concurrent TPCW clients. The database size used in these experiments is 20GB and a fresh copy was been installed for every run (point in the plot) to ensure that results were not affected by database evolution. The x-axis shows the number of cores used by MySQL or the Multimed configurations. In the case of Multimed, it also shows the number of satellites coordinated by Multimed.

The MySQL experiments show that the system cannot scale with the number of cores. In contrast, the Multimed deployments can take advantage of the increased number of available cores in the system. As MySQL implements its own disk buffer and does not rely on the OS cache buffer, Multimed starts to content on disk much faster than in the case of PostgreSQL. For this reason, Multimed's scalability when running on MySQL stops at about 6 nodes. This problem can be solved by adding faster disks or removing the disk bottleneck by deploying the satellites in main memory. Introducing partially replicated satellites does not bring any improvements as the bottleneck is the disk I/O and the master node is saturated with update transactions.

Scalability with the number of clients can be seen in Figure 5.4. In contrast to PostgreSQL, MySQL uses a pool of internal threads to handle requests from clients. Con-
Figure 5.3: Scalability with the number of cores of Multimed running on MySQL with 20 GB database, using 200 TPCW-B clients
5.2. Scalability in OLTP Workloads

![Diagram showing scalability with the number of TPCW-B clients of Multimed running on MySQL with 20 GB database.](image)

Figure 5.4: Scalability with the number of TPCW-B clients of Multimed running on MySQL with 20 GB database
Chapter 5. Improving Scalability

Subsequently, the engine acts as a queuing system and does not show performance degradation with an increasing number of clients, but rather an increase in response time.

The experimental evaluation shows that MySQL does not scale with an increasing number of clients, but the increasing number of hardware contexts degrades the engine’s performance. MySQL performs best at 12 cores. After this threshold, contention over synchronization primitives (i.e., mutexes) that are used by MySQL throughout its entire code start to produce performance degradation.

The three configurations used to evaluate the performance of Multimed show that by using replication, our solution can outperform standalone MySQL until it reaches the disk I/O bound. The on-disk configuration of Multimed performs close to the performance of MySQL even if we have 4 nodes sharing the same disk. Using in-memory replicated satellites removes the disk contention from Multimed and increases performance. As the disk bottleneck is already hit at the master node, using partial replication adds only minor improvement to the system. The system is already disk bound and load interaction does not influence MySQL for this workload. To improve performance in this scenario, a faster disk or lower I/O latency would be needed.

5.2.2 TCPW-S

This section presents the experimental evaluation of Multimed while running on PostgreSQL and MySQL with varying number of cores and clients using the TPCW-S mix. The objectives of these experiments are to determine the scalability properties of the Multimed system in a write-intensive workload. For this purpose we use the TPCW-S mix and evaluate the performance of Multimed with varying number of cores and analyze the ability of the system to use the increasing number of resources. The scalability with an increasing load is achieved by running experiments with varying number of number of clients that issue queries according to the TPCW-S mix.

PostgreSQL

Scalability results while varying the number of cores assigned to PostgreSQL and Multimed with an increased number of write transactions are shown in Figure 5.5. Due to the increased amount of write transactions in the workload, the master node will have to handle an increased load. To achieve better performance, the master node is deployed on 8 cores; the Multimed middleware uses 4 cores, while each satellite node is deployed on 6 cores.
5.2. Scalability in OLTP Workloads

![Graphs showing scalability with the number of cores of Multimed running on PostgreSQL with 20 GB database, using 200 TPCW-S clients.](image)

Figure 5.5: Scalability with the number of cores of Multimed running on PostgreSQL with 20 GB database, using 200 TPCW-S clients
Figure 5.6: Scalability with the number of TPCW-S clients of Multimed running on PostgreSQL with 20 GB database
5.2. Scalability in OLTP Workloads

As in the previous scenario, PostgreSQL shows scalability problems with the number of cores in the case of the Shopping mix. Multimed can scale up to 4 satellites, using an in-memory deployment. After this point, the disk becomes the bottleneck and the performance remains constant. As read transactions are taken away from the master node, the increased amount of write transactions start to put pressure on the disk, limiting the increase in performance. Even so, Multimed shows a more stable performance with the number of cores while PostgreSQL shows performance degradation with the increased amount of hardware contexts.

The on-disk deployment shows performance degradation over PostgreSQL because all the satellites write to disk. Removing contention from the disk with the in-memory deployment and reducing the amount of writesets by using partial replication prove that the problem was the contention on disk I/O. By using a faster disk, Multimed can further improve over the performance of PostgreSQL.

Scalability with an increasing number of clients of Multimed compared with PostgreSQL is shown in Figure 5.6. For the on disk and in-memory experiments, the Multimed system used 4 cores, the master node used 8 cores and each satellite node was deployed on 12 cores. The partial replication configuration deploys 2 fully replicated satellites each one using 12 cores, the number of partially replicated satellites has been reduced to one as the amount of concurrent long running queries in the system is smaller due to the workload's characteristics.

For a small number of clients, Multimed performs worse than PostgreSQL. This effect is due to the reduced amount of read transactions that can be distributed over the satellite nodes. For this reason, a reduced number of clients puts more pressure on the master node. Once the number of clients increases Multimed can efficiently distribute more load to the satellite nodes, alleviating the problems of load interaction and contention on shared and locking primitives that limits the scalability of PostgreSQL.

MySQL

We repeated the same experiments using MySQL as the underlying engine for Multimed.

Figure 5.7 shows the scalability of Multimed compared with MySQL while varying the number of cores with a load generated by 200 clients that issue queries according to the TPCW-S mix. For MySQL, we notice that the engine scales up to 12 cores and once we add more hardware contexts, the engine’s performance starts to degrade. Multimed scales up to 16 cores, at which point the throughput flattens. At this point, by analyzing
Figure 5.7: Scalability with the number of cores of Multimed running on MySQL with 20 GB database, using 200 TPCW-S clients
5.2. Scalability in OLTP Workloads

![Graph of scalability with number of TPCW-S clients of Multimed running on MySQL with 20 GB database](image)

Figure 5.8: Scalability with the number of TPCW-S clients of Multimed running on MySQL with 20 GB database.
the system performance counters, we noticed that the disk I/O is the limiting factor.

Scalability with the number of clients shows, presented in Figure 5.8, the same characteristics as in the case of TPCW-B. Multimed deployments can scale up to the point when the master becomes the bottleneck due to contention over disk I/O. We can observe an increased performance for in-memory deployment, but once the disk becomes a limiting factor, adding partial replication does not add any performance improvement.

Discussion

In the previous section, we have shown that the increased amount of cores and clients produces performance degradation for both workloads and database engines. On the one hand, the increased amount of hardware contexts leads to a higher synchronization overhead. Thus, the database ends up spending more time in thread management and synchronization rather than doing useful work. On the other hand, deploying a replication system over the same amount of resources replicates the locking and synchronization primitives. As each replica needs to handle a smaller number of concurrent connections the overall improvement turns out to be higher than the overhead of managing the replica management system.

Reducing contention over disk and deploying the replicas in main memory bring the biggest improvements, especially in workloads with an increased amount of write transactions. Further, specializing the system and deploying partial replicas tailored for special queries, reduces memory requirements and improves the overall performance due to load separation and increased memory locality.

5.3 OLAP Performance

Section 3.3.4 showed that, in the case of TPCH, the increased number of resources available in multicore architectures does not ensure an improved response time of long running analytical queries due to memory locality problems. PostgreSQL assigns a thread to each query, and the OS schedules the threads on the available cores without taking into consideration the data that they need to access. This approach leads to increased memory latency due to an increased amount of memory accesses to remote NUMA nodes. Related work has showed that placing data close to working threads will reduce contention on the critical path [67].

Multimed allows deployment of satellites on a set of resources, and by binding the
5.3. OLAP Performance

satellite nodes to a set of cores; we restrict the threads generated by the underlying database engine to run only on those cores. In the same way, once the database engine has been deployed on a set of cores, it will allocate memory from its local NUMA node thus improving the memory locality. An efficient deployment of Multimed for TPC-H takes into consideration a partition over the queries and the data they access. Therefore, we have created a deployment with four nodes (Figure 5.9), each running on 12 cores, to evaluate if partitioning the memory resources improves the memory locality and the performance of the TPC-H. Each satellite resides on disk but during the warm-up phase all the data from the satellites is loaded in the database buffers.

We did a clustering of the TPCH queries based on the data and operations used by each individual and query. We have grouped queries that do large table scans over similar tables (Q1,Q6,Q3,Q18,Q21) and queries that are highly affected by the increased number of clients to another satellite (Q4,Q5,Q6,Q10). The other two satellites have a generic purpose and are configured to run all TPC-H queries. These satellites are used as backup in case the other satellites are overloaded. The current clustering was done in an experimental way, and it is not the best possible clustering, but, the results presented in Figure 5.10 show improvements of response times for each TPC-H query when using Multimed over the initial PostgreSQL runs. Therefore using memory locality and collocation of similar queries improves the performance of all the TPC-H queries when using 24 clients.
Figure 5.10: Evolution of individual TPCH query response time running on PostgreSQL and Multimed, using a 10GB TPC-H database (SF=10), PostgreSQL and Multimed running on 48 cores with 24 TPC-H clients

5.4 Discussion

This section has addressed the problem of making existing database engines run efficiently on multicore machines without changing the internal architecture of the engine. Multimed achieves this goal by using techniques from distributed replicated systems and modern operating systems designed for multicores.

As shown in this section, the main scalability problems of current database engines running on modern multicore architectures are contention over memory, shared data structures, locking and synchronization primitives as well as load interaction between queries.

Multimed proves to be a solution for a wide range of workloads. Due to its replicated nature, its performance is limited by the number of updates that can be performed at the master node. As the experiments show, the best performance can be achieved in the case of the browsing and shopping mixes of the TPC-W benchmark. In the case of the ordering mix, as we hit the disk bottleneck, the system can only offer similar performance to the reference database engine. Multimed can be used to linearly scale read dominated loads such as those found in business intelligence applications and data warehousing. For instance, it is possible to show linear scale up of Multimed by simply assigning more satellites to complex analytical queries. As a general rule the
5.4. Discussion

Workloads with high update rates and less complex read transactions are not the best case for Multimed – as in any primary copy approach – because the entire system becomes limited by the performance of the master node. Cluster-based replication systems solve this problem by increasing the resources on the master machine. The same approach could be used in Multimed by assigning more cores to the master node and adding a faster disk to the machine, or deploying the master database on a high speed network attached storage system.

Multimed manages to remove the contention problems by using multiple replicas of the database engine and distributing the transactional load, thus reducing the number of threads that concurrently access each replica. Using different satellites designed to run queries that produce load interaction, the Multimed engine alleviates this problem through load separation and replication of shared resources.

The effects of this approach on the response time of each query in the TPCW-B mix can be seen in Figure 5.11. In the case of PostgreSQL the response time increases with the number of number of cores. Deployed on the same time of resources Multimed the response time of the BestSellers query is significantly reduced in the same time, the increasing number of clients has a smaller impact on performance.

Through careful resource partitioning, Multimed exhibits a more stable and better performance on multicore architectures than the underlying database engines. From the scalability evaluation, we can clearly see, that as the number of cores, and the amount of available memory increases, Multimed can take advantage of the extra resources.
and manifests an improved performance. Faster disks or network attached storage will further improve the performance of Multimed due to the reduced contention on disks. With the advent of new multicores architectures that employ heterogeneous computational units, Multimed can use specialized replicas specialized to take advantage of specific characteristics from the underlying cores.

The next chapter will present the performance of Multimed using specialized replicas designed to manage complex queries and extend the functionality of existing database engines using specialized operators.
Multimed satellites can also be used to offer optimizations and specializations to existing database engines. As in the case of Ganymed [72] satellite database engines can be used to provide extra functionality without affecting the performance of the master engine. The Multimed satellites are designed to improve performance of long running queries and can be used as well as extensions to commercial database engines.

In the case of Multimed the extensions provide extra functionality by deploying specialized satellites inside the multicore machine. These satellites are used to improve the performance of long running queries, add new functionality that is not supported by the database engine. Reduced load interaction and improved performance of the specialized queries, leads to an overall increase in performance. Figure 6.1 shows a Multimed deployment that we will use in this section to show the applicability of this approach. The deployment contains two fully replicated satellites, one partially replicated optimized to run the BestSellers query and one specialized satellite. The specialized satellite is going to be extended and optimized to run complex queries and specialized operators.

Specialized operators are used in long running queries that need to perform complex aggregations over large amounts of data, skyline, ranking or provenance queries are just a few use cases. The next sections of this chapter are going to explore different
6.1 Fine Tuned Satellites

Queries using aggregation operators or TOP-N queries can trick the PostgreSQL query optimizer in doing full table scans and nested loop joins rather than using the indexes to improve the overall query performance. Such a query is the Best Sellers query presented below:

```
SELECT * FROM ( 
   SELECT i_id, i_title, a_fname, a_lname, SUM(ol_qty) AS orderkey 
   FROM item, author, order_line 
   WHERE i_id = ol_i_id AND i_a_id = a_id 
   AND ol_o_id > (SELECT MAX(o_id)-3333 FROM orders) 
   AND i_subject = 'CHILDREN' 
   GROUP BY i_id, i_title, a_fname, a_lname 
   ORDER BY orderkey DESC 
) LIMIT 50
```

Improving performance of such a query in PostgreSQL requires either a query rewrite or to instruct the engine to disable nested loops and always do hash joins. While the first approach is not generic, the second approach is applied engine wide thereby degrading optimizations and various extension satellites and their effect over the entire workload as well as the performance impact to the specialized queries.
the performance of the entire workload as some of the queries benefit from the use of nested loop joins.

Multimed can use a specialized satellite to achieve fast execution of the *BestSellers* queries without an impact to the existing workload. This satellite is specially optimized to run the *BestSellers* query, as only the *BestSellers* queries are executed on this satellite the other queries do not suffer from performance degradation due to the special optimization. The *BestSellers* query is an analytical query in a transactional benchmark. This query aggregates data from the most accessed tables in the database.

Similar queries that can be executed on specialized satellites are long running analytical queries that create contention over memory and shared data structures. To avoid load interaction, these complex queries should be separated from the initial workload and executed in a specialized satellite. In the same time, executing them on the same set of resources will improve performance due to memory locality and reduced contention over shared resources.

This is not a general problem in all database engines, as most commercial systems use optimizer hints to instruct it to generate the right query plan. Nevertheless the specialized satellite adds extra functionality that was not available in the initial database engine, and can be generalized to other engine specific functionality that has a similar effect.

### 6.1.1 Impact of Options to the Workload

The performance of the TPC-W workload with an increasing number of cores and clients for the PostgreSQL engine running with the nested loops functionality disabled is presented in Figure 6.2 (marked in the plot as PostgreSQL BS-OPT). Comparing the throughput and RT results with the values of the initial PostgreSQL, we notice that the overall performance of the engine has decreased due to this change. On the one hand, the optimizer flags improve the performance of the *BestSellers* query. On the other hand, removing the nested loops option degrades the performance of all the other queries in the mix.

The Multimed system can offer the advantages from both cases with the use of a specialized satellite for designed to run the *BestSellers* query. This satellite will execute only the *BestSellers* query and will not allow the engine to use the nested loop join algorithm. In the same time all the other queries in the benchmark will be executed as before, on satellites that allow nested loop joins, thus we will not have performance degradation on the queries in the mix. This way Multimed separated the *BestSellers*
Figure 6.2: The effect of disabling nested loops for the PostgreSQL engine when running TPC-WB with a 20GB database
query from the initial workload and further improves the performance of the query with fine-tuned optimizations. Compared with the initial approach this optimization comes with no side effects to the initial workload. Furthermore we achieve this performance improvement without any changes to the initial query or indexes.

### 6.1.2 Experimental Setup

To evaluate the improvement introduced by this specialized satellite we have used the in-memory configuration presented in Section 5.2.1. In this configuration we have assigned 4 cores to the Multimed system and 8 cores to master node. We have deployed 2 fully replicated satellites, one using 12 cores, the other one 6 cores and 3 in-memory partially replicated satellites each one running on 6 cores. The partial replicated satellites have the option to run nested loops disabled. Just the BestSellers web interactions are executed on the optimized satellites, all the other interactions in the workload are executed on the other satellites. This deployment allows a fast execution of the BestSellers web interaction as all the specialized satellites reside in-memory and are highly tuned for this specific analytical transaction.

### 6.1.3 Multimed with a Specialized BestSellers Engine

As we can see in Figure 6.3, compared with the previous configuration the overall performance of Multimed is improved as the BestSellers web interaction is executed faster. By deploying a specialized satellite to handle the Best Sellers query and disabling the nested loops functionality on the engine, we manage to obtain the best of both worlds. On the one hand we optimize the satellite for the BestSellers query to its best performance. On the other hand, separating the query reduces load interaction and introduces further gains in performance.

To improve performance of the overall workload we need to have a critical mass of BestSellers transactions running in the system, for this reason we notice that we have only a marginal p over the initial deployments of Multimed when we use a small number of clients. As the number of clients increase so does the amount of BestSellers queries running in the same time in the system. At this point the specialized version of Multimed outperforms the initial system. Once the disk bottleneck is reached, the system can no longer achieve scalability with the number of clients and the throughput line starts to flatten out.
Figure 6.3: Optimization of BestSellers - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database
Further gains in performance without any hardware improvements, can be achieved with more fine-tuned optimizations and increasing the number of in-memory satellites.

### 6.2 Heavy Query Optimization – Keyword Search

A frequent operation in web applications and especially online bookstores is a full text keyword search. We design this operation as two phases: the first step consists of generating all the keywords from the item names and the item descriptions, while the second phase requires storing and maintaining the keywords index. These two operations require access over large amount of data, and pose a challenge in keeping the index up-to-date.

#### 6.2.1 Experimental Setup

Building a text search index is usually an operation that takes place once a day and it is not computed in real time for every request. For the scope of this experiment we have created an external client that issues the query at a specific interval. The increased amount of hardware contexts available in the multicore machines should provide sufficient resources to run the **KeywordSearch** query without a major drop in performance, especially when we have a single client issuing the request.

Our implementation of the **KeywordSearch** is an based on the **BestSellers** query and generates a keyword index over the most recently bought items. Thus, it collects all the items and their description that appear in the most recent orders. An external application will take these results, extract the keywords from the `i_desc` column and produce a set of tuples of the following form `{i_id, keyword}`.

```sql
SELECT i_id, i_desc FROM item
WHERE i_id in
    (SELECT DISTINCT(ol_i_id) FROM order_line
    WHERE ol_o_id >= (SELECT MIN(o_id)
    FROM (SELECT o_id-3333 AS o_id
    FROM orders ORDER BY o_id DESC) AS a))
```

As the query needs to aggregate results over the three most accessed and largest tables (**items**, **orders** and **order_lines**), the query’s execution time is about 5 seconds
when executed alone in the database engine. This is not the most optimal implementation, but it represents a class of applications that collect a set of data and then do the processing off-site. Such use cases can be found in the commercial systems when generating monthly or yearly reports.

### 6.2.2 Impact of Keyword Search

To evaluate the performance impact of the *KeywordSearch* query over existing workloads running on multicore architectures. We have used the same setup from the TPCW experiments. The *KeywordSearch* application connects to the database engine as a usual client called *KW/Search* that issues the query continuously once the previous one has finished.

We use 10 concurrent *KW/Search clients* to evaluate the performance degradation when using a large number of applications issuing heavy queries.

As the number of available hardware contexts increases in current multicore architectures, these queries should manifest an improved performance, especially in the case of engines such as PostgreSQL that assign threads to new connections. As in the case of the *BestSellers* query, this is a read-only query that scans over a large set of data.

Figure 6.4 shows the average throughput and response time of the TPCW-B benchmark when running one *KW/Search* client and Figure 6.5 shows the same experiment using 10 concurrent *KW/Search* clients. As we can clearly see from the overall throughput and the *KeywordSearch* query response time, going from 24 to 48 cores brings an insignificant improvement.

With the increased number of clients, the *Keyword Search* query starts to degrade in performance and, as threads start to compete for resources, the response time of the *KW/Search* client starts to increase. In the case of MySQL, the performance degradation is more severe due the thread pool based architecture of the MySQL engine.

### 6.2.3 Multimed with Keyword Search Satellites

The Multimed experiments use an in-memory configuration with the following deployment: Multimed system used 4 cores, the master node used 8 cores. We have deployed 2 fully replicated satellites, one using 12 cores and the other 6 cores and 3 partially replicated satellites in main memory. Two of the partially replicated satellites were used to run the *BestSellers* web interaction, while the third one was specialized to handle the
Figure 6.4: 1 KW Search Client - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database
Chapter 6. Extending Functionality

Figure 6.5: 10 KW Search Clients - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database
The KW_Search client(s) connects to the Multimed middleware and all the requests for the Keyword Search are routed to the specialized satellite. The experimental results, presented in Figure 6.4, show that separating the Keyword Search query from the main workload reduces its execution time to the run time of the query running alone in the database engine. Increasing the number of KW_Search clients to 10 shows no significant increase in the response time due to the fact that the query runs locally thereby benefiting from data and memory locality as presented in Figure 6.4.

In both cases we notice an improved RT when we run the query in a specialized satellite. Even if the satellite uses a reduced amount of resources the query benefits from improved data locality and contention over shared locking and synchronization primitives.

6.3 Optimizing Expensive Operators

Starting with version 8.3, PostgreSQL offers native support for text search. It is often the case in web applications that search results must be ordered according to their relevance. For this reason, PostgreSQL offers a ranking operator. The ranking operator attempts to measure how relevant documents are to a particular query so that, when there are many matches, the most relevant ones can be shown first. Producing the ranking results is an expensive operation, as every new matching element requires a re-evaluation of the existing results, which can lead to contention over CPU and memory resources[77].

PostgreSQL offers a cover density ranking function (ts_rank_cd) which computes the result's ranking based on the location of the terms in the documents. Presented by Clarke et al. in [27], this ranking function considers that if most of the query terms are in close proximity, the results are likely to be more relevant.

6.3.1 Experimental Setup

For the experimental evaluation, we have used the same experimental setup as in the previous section. The ranking clients connect to the database engine as usual clients and issue the following ranking query without any wait time between transactions:
SELECT i_id, (i_title || ' ' || i_desc) ,
    ts_rank_cd(to_tsvector(i_title || ' ' || i_desc),query,2) AS rank
FROM item, to_tsquery('regular_expression') query
WHERE query @@ to_tsvector(i_desc)
ORDER BY rank DESC LIMIT 10

This query simulates a search request and returns the 10 best ranked items that match the user’s request. The match function is given as a regular expression, while the matching will be done on the item title and description fields.

We use 1 and 10 concurrent ranking clients to evaluate the performance degradation with an increased number of TPCW-B concurrent clients.

### 6.3.2 Impact on PostgreSQL Performance

The results presented in Figure 6.6 and Figure 6.7 show that, for a constant number of ranking clients increasing number of cores produces just a marginal improvement in response time. We notice an improvement from 8 to 24 cores while at 48 cores there is only a marginal benefit, despite the increased amount of resources. At the same time, increasing the number of ranking clients to 10 shows an increase in the response time of the ranking query up to 10 times. As the queries access the same data, the decrease in performance is mainly caused by the interaction between the queries and contention over memory resources.

This is a typical case of load interaction and contention over shared resources. In the case of Multimed the straightforward solution to this problem is to route the ranking query to a specialized satellite.

### 6.3.3 Ranking Operator as Multimed Satellite

We use the same configuration as in the previous case for Multimed experiments. This is an in-memory configuration where the Multimed system uses 4 cores, the master node uses 8 cores, we have 2 fully replicated satellites, one using 12 cores and the other one 6 cores, and 3 partially replicated satellites. Two of the partially replicated satellites are used to run the BestSellers web interaction while the third one executes the Ranking queries.

Analyzing Figures 6.6 and 6.7 in more detail, we notice that Multimed improves the response time of the Ranking query by removing load interaction through load separa-
6.3. Optimizing Expensive Operators

Figure 6.6: 1 Ranking Client - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database
Figure 6.7: 10 Ranking Clients - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database
tion and removing contention over the available resources. Even if we allocate a limited set of resources to the ranking satellite, by removing the query from the initial workload and routing it to a specialized satellite we manage to reduce the contention over shared resources. As a further advantage, the response time of the ranking is not affected by the increasing number of TPCW-B clients or Ranking clients.

6.4 Extending with New Operators – Skyline Satellites

Another class of specialized satellites is the use of specialized operators that are not supported by database engines. The main objective of this experiment is to show another design property of Multimed, extending the engine with new functionality.

The main use of specialized operators in database engines is to improve performance on complex queries using specialized algorithms. Such an operator is the Skyline Operator presented in [17]. As mentioned in [17], the skyline operator filters interesting points from a potentially large set of data points. Assume a user asks the following query to a travel agency web site: “I want to find the hotels from a specific location that are closer to a beach but also have a low room price and offer the highest comfort”. This query does not have an exact answer as many constraints can be contradictory, but it is possible to present a set of interesting hotels that fit the search criteria. These hotels present the Skyline. Computing the Skyline is a very expensive task as the number of constraints increase.

Adding support for the Skyline operator required internal changes at different levels of the PostgreSQL engine. First, we needed to modify the parser to support the syntax of the new operator. Once this was done, we implemented multiple algorithms and created new data structures to compute the Skyline operator. The modified PostgreSQL engine can support Skyline queries with the syntax presented in [17]:

```
SELECT ... FROM ... WHERE ...
GROUP BY ... HAVING ...
SKYLINE OF [DISTINCT] d1 [MIN | MAX | DIFF], ..., dm [MIN | MAX | DIFF]
ORDER BY ...
```

As mentioned in [17], one way to execute Skyline queries over an existing engine is through query rewriting, as one can create an equivalent query in SQL. However, the resulting queries are very expensive as they cannot be un-nested [42]. The most
efficient way of handling skyline queries is to extend the database engine and integrate new support and algorithms for the skyline Operator.

The experimental section uses the Block Nested Loop implementation (BNL) presented in [17] as we only have a small number of conditions. For more complex queries, the user can choose one of the other supported algorithms such as Sort First Skyline (SFS) [68], and Linear Elimination Sort for Skyline (LESS) [38].

### 6.4.1 Experimental Setup

The experimental setup used in this section has the same characteristics as in the case of Keyword Search. The Skyline clients connect to the database engine as usual clients and issue the following skyline query without any wait time between transactions:

```
SELECT o_id, o_total, SUM (ol_qty) AS qty_sum
FROM orders, order_line
WHERE o_id = ol_o_id
AND ol_o_id >
  (SELECT o_id-500 FROM orders
   ORDER BY o_id DESC LIMIT 1)
GROUP BY o_id, o_total
SKYLINE OF o_total MAX, qty_sum MIN ORDER BY o_total
```

We used 1 and 10 concurrent Skyline clients to evaluate the performance degradation with an increased number of TPCW-B concurrent clients.

### 6.4.2 Impact of Skyline Operator

Figure 6.8 presents the response time of one Skyline client with a varying number of TPCW-B clients. We can observer that the response time of the Skyline degrades with the increasing number of clients.

For the first experiment, we used one Skyline client and varied the number of TPCW-B clients and the number of cores allocated to the database engine. The increasing number of cores allows the database engine to use a larger number of hardware contexts. This should lead to an improvement in the performance of the skyline queries. As we can see in Figure 6.8, once we reach 24 cores, there is no improvement to the response time of the Skyline query. The skyline operators aggregate large amounts of
data in main memory and are highly CPU intensive tasks. The response time degra-
dation is due to contention over shared computational resources between the Skyline
queries and the TPCW-B clients.

Increasing the number of Skyline clients to 10 further degrades the performance of
the Skyline query due to increased load interaction and contention over shared re-
sources.

### 6.4.3 Multimed with Skyline Satellites

The Multimed experiments use an in-memory configuration with the following deploy-
ment: Multimed system with 4 cores, the master node with 8 cores. We further deploy
2 fully replicated satellites, one using 12 cores and the other one 6 cores, and 3 par-
tially replicated satellites in main memory. Two of the partially replicated satellites were
used to run the BestSellers web interaction while the third one has a modified Post-
greSQL engine that offers support for Skyline queries. The specialized satellite for the
Skyline queries replicates only the tables needed to run this query, namely orders and
order_line.

The results of the same experiments for Multimed are presented in Figure 6.8 and
Figure 6.9. Here, we notice that the Skyline queries show an improvement in response
time. When we use 10 skyline clients the Skyline response time has a slight increase,
but it is still much smaller than in the case of PostgreSQL.

Furthermore, Multimed manages to keep a constant response time to the Skyline
Client. By separating the heavy Skyline query and adding an engine that is extended
to efficiently support the Skyline query, Multimed can provide a stable response time
to complex analytical queries and remove the load interaction with the other concurrent
queries in the system.

### 6.5 Adding new Functionality – Data Provenance

In the case of OLAP workloads data provenance queries are one example of specialized
queries that are hard to manage without extending the database is engine. The process
of data provenance tracks the origins of the data and describes to the used how the
data was produced [37]. Computing provenance is a very expensive operation. On the
one hand, computing provenance leads to complex query plans. On the other hand,
the result size of the provenance queries is much bigger than the initial query, requiring
Figure 6.8: 1 Skyline Client - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database
6.5. Adding new Functionality – Data Provenance

Figure 6.9: 10 Skyline Clients - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database
large amounts of memory to keep the intermediate results. For example, the result sizes for TPC-H queries without provenance varies between 1 and 10,000 tuples. Queries with provenance lead to result sizes between hundreds and hundreds million tuples.

The Perm project develops a novel provenance management system called Provenance Extension of the Relational Model (Perm) that is capable of computing, storing and querying provenance for relational databases. Perm generates provenance by rewriting transformations (queries). For a given query, it generates a single query that produces the same result as the initial query but extended with additional attributes used to store provenance data [37].

Perm is implemented as an extension of PostgreSQL DBMS and operates on the internal query tree representation of a query, taking advantage of the internal query optimizations of the PostgreSQL engine. Provenance computation for a query is enabled by adding the Provenance keyword to it.

```
SELECT PROVENANCE col1,...,coln FROM t1,r2...;
```

The large amount of data that is processed while executing the provenance queries can create contention over shared resources in the same time memory locality plays an important role in performance. For this reason Multimed can further improve the performance of the provenance queries by deploying a specialized satellite that is modified and to manage data provenance queries.

### 6.5.1 Experimental Setup

To test the impact of the provenance query on the TPC-H benchmark we selected a TPC-H Query and computed provenance over it while running the normal TPC-H mix. We have selected Q9, as it is not one of the most computationally expensive queries in the mix. The generation of data provenance on this query takes about 30 minutes to finish, which is a reasonable execution time as provenance queries can even run several days.

Perm is implemented as an extension to the same version of the PostgreSQL engine used in our experiments. Hence, we can directly compare the effects of the Perm engine to the TPC-H workload. We have used the same experimental setup as in the TPC-H experimental section (Section 3.3.4) but extended it by adding the provenance computation over Query 9.
6.5. Adding new Functionality – Data Provenance

6.5.2 Perm impact on PostgreSQL performance

As we can see in Figure 6.11, without provenance the Q9 query of the TPC-H benchmark executes in 745 seconds, on a TPC-H database with scaling factor 10 and 48 concurrent TPC-H clients. If we run the same query and compute its provenance, we notice that the execution time increases dramatically to almost 2000 seconds. Even more, the execution time of all the other queries starts to increase, even if we have sufficient hardware contexts as our multicore machine has 48 cores. The main reason for this effect is contention over memory resources. Having 48 concurrent clients, we could only allocate 2.5 GB of sorting and work memory to every PostgreSQL connection. As the database engine does not differentiate between different connections and query requirements and allocate memory accordingly, the clients running the provenance query ended up swapping temporary data to disk, thereby increasing latency times and decreasing the overall performance.

At the same time, as in the case of TPC-H, problems with memory locality also contribute to performance degradation of all the concurrent queries in the system.

6.5.3 Perm as Multimed satellite

A Multimed configuration for this scenario needs to improve data locality for the standard TPC-H queries and ensures enough memory resources for the Provenance query. Figure 6.10 shows the Multimed configuration used to improve the performance of the
provenance query and remove the interaction with other queries. To achieve better performance, we have modified the previous configuration used for the general TPC-H benchmark. We have reduced the number of computational resources to one of the general purpose satellites and deployed a new satellite optimized to handle the Provenance query.

In Multimed, we can customize special satellites and allocate more memory to them. In our case, we have deployed a specialized satellite that offers support for Provenance queries. We deployed this satellite on 6 cores and configure it to use 7GB of main memory for sorting and managing temporary data. At the same time, this node has a reduced connection pool (5 concurrent connections) to limit the total amount of possible used memory. In this way, the expensive queries can benefit from the increased amount of local memory and finish faster. In the same time this approach can create problems if the frequency of the queries is very high, in which case one must assess the correct sizing of the memory pools and connections according to their workload. The TPC-H benchmark has 21 queries, and the execution frequency of Q9 is not very high. As a result, trading the number of possible concurrent connections running in the same time in the Multimed system for more memory is not a limiting factor for our scenario. Of course, if the heavy query is very frequent, the Multimed Dispatcher will queue them on the specialized node, thus increasing the latency, in such a case, we need to change
the configuration and allocate more resources to the specialized satellite.

As we can see in Figure 6.11, using this configuration improves the performance of both the Provenance query and the concurrent TPC-H query. Even if the new configuration has as effect an increased response time over the other TPC-H queries, due to the reduced computational resources the overall TPC-H performance is increased by almost a factor of three. The main advantage of the Multimed system for these scenarios is the flexible deployment schemes that can be tailored for specific workload, having as a consequence an improved data locality and optimal resource allocation to manage heavy analytical queries.

6.6 Discussion

This chapter shows the potential of Multimed to improve the performance of complex analytical queries from both OLTP and OLAP workloads. Due to load interaction and contention over memory resources these queries start to degrade in performance when running on database engines deployed on modern multicores. The main reason for the degradation is the fact that the OS schedules the database engine threads without taking into account any information about data locality. Threads executing these queries end up accessing memory from remote NUMA nodes thereby increasing load interaction, latency and degrading the overall system performance. With an increased number of concurrent clients and available hardware contexts remote memory accesses become more frequent and threads start evicting each other’s caches.

By partitioning the hardware resources to specialized satellites, Multimed manages to increase data locality as database engines use memory from their local NUMA node. At the same time, as threads executing queries that access the same data incur a higher cache hit ratio. Moreover, the specialized satellites can work on a small replica and produce a large improvement to the overall system performance.

The main advantage of Multimed is the ease of extending the engine with new specialized satellite nodes that provide new functionality and configure the Dispatcher to route the transactions accordingly. This approach allows the use of multiple engine extensions without any changes to the initial database engine.

We have shown in this section multiple ways and use cases for improving database performance by using specialized satellites that extend the engine’s functionality at different levels. Multimed can take advantage of both engine specific options as well as new operators that are not supported by the database engine or are designed to
manage complex data. In all cases, Multimed has proved to produce an increased performance over the specialized queries and the collocated workload.
The new CPU architectures create new challenges for database engines as they must be able to exploit the increased number of hardware contexts and available resources to further increase their performance. The increased degree of parallelism offered by current multicore processors presents both an advantage as well as a disadvantage for current database engines. On the one hand, database engines can exploit the inherent parallelism offered by the new CPUs to increase throughput of highly concurrent workloads. On the other hand, the new multicore architectures pose new challenges to existing database engines. The higher degree of parallelism increases the pressure on the concurrency control and locking mechanism. The increased amount of queries running at the same time in the engine amplifies the problem of load interaction and contention over shared resources.

This thesis presents an overview of the current challenges posed by existing multicore processors to software used in infrastructures systems (e.g., database engines and operating systems). As presented in Chapter 2, designing operating systems that can exploit the resources offered by multicore architectures is an ongoing challenge. Modern operating systems treat the multicore architecture as a distributed system to reduce scalability problems due to data locality and contention over shared data structures. Similar problems have been encountered in the case of database engines as they
heavily rely on locking, shared data structures and data locality to achieve consistency, high throughput rates and improved performance. New database engines and architectures have emerged in an attempt to solve scalability problems in existing database engines when running on multicores. Chapter 3 presents a performance evaluation of the two most popular open source database engines. From the performance evaluation, we notice that, even if the engines have different internal architectures, they both have scalability problems with the number of cores as well as with the number of clients.

As in the case of operating systems, we propose to consider the multicore architectures as a distributed system and use existing techniques from cluster-based replicated systems to improve the performance of existing database engines. The resulting system, called Multimed and presented in Chapter 4, is a departure from existing work solving the database scalability problem for a wide range of workloads without having to modify the engine. Based on existing work in replicated database systems, Multimed deploys a set of replicated database engines over a multicore machine treating it as a distributed system.

The performance evaluation of Multimed, presented in Chapter 5, shows that Multimed exhibits a better and more stable performance than the underlying engines. Moreover, the Multimed engine can take advantage of the increased amount of resources available in current multicore architectures, showing an improved performance with the increasing number of cores.

As presented in Chapter 6, Multimed’s satellites can be used to extend the functionality of existing database engines. Specialized satellites are used to improve the performance of existing workloads as well as improving the response times of complex queries by using specialized operators that are not provided by existing database engines. These satellites are used to remove load interaction and improve the robustness of complex workloads by separating disruptive queries from existing workloads.

Further improvements to the Multimed system can be added to the load balancer. The current light weight load balancing techniques included in the dispatcher are designed for speed and performance. A more complex load balancer might be needed to enable support for large multicore architectures with more cores and available resources.

One of the shortcomings of Multimed is the extra amount of memory and storage needed for the replicas. While storage and memory prices dropped in the last years, the increased amount of storage needed for storing the satellite nodes can be seen as a limiting factor for very large databases. We argue that replication of locking, and shared data structures are needed to scale on modern multicore architectures. Finding solu-
tions to enable fine-grained replication will reduce the memory and storage footprint. We have already done a first step in this direction by using partial replicas. Next steps can make use of specialized satellites or in memory engines, to cut back on memory requirements. Optimizing satellites for in-memory operations could even improve the overall performance of the Multimed system.

Current Multimed configurations are workload specific and done in a static way. To obtain efficient configurations, a thorough performance evaluation and fine tuning of the underlying database engines was needed. The same approach is also used for database engines, such as Oracle RAC or IBM DB2, which are tuned for specific application workloads before placed in production. Introducing a way to dynamically allocate computational resources to the underlying database engines, based on their current workload and requirements, can turn Multimed into a more flexible and easy to use system.

Recent research in cloud and multi-tenant databases has brought up similar topics and show promising results. One approach proposes to improve resource allocation by minimizing the resource allocation overhead in virtualized environments [90]. This approach shows that it can quickly adapt resources to workload changes at runtime. They achieve this by caching resource allocation preferences at runtime and using a small set of workload classes. Another approach, proposes to use results from running queries in isolation to achieve performance prediction and meet performance expectations [32]. The experimental evaluation was done top of PostgreSQL with the TPC-H benchmark and shows promising results in predicting query latency. With the introduction of very large multicore architectures, these approaches will be more and more needed to obtain optimal deployments and configurations. However, a more detailed analysis is needed to determine the applicability of these approaches and estimate the introduced overhead in the context of Multimed.

Better insights about the performance characteristics of Multimed, can be obtained by understanding the effect of in-memory database engines or column-stores in the context of the system. Multimed can make use of virtualization techniques to enable dynamic deployment of satellites and resource allocation. However we have not evaluated the impact of virtualization over the performance of Multimed. Multimed uses techniques from cluster based replication systems. Designing and deploying a distributed version of Multimed in deployed over multiple multicore systems, will open new questions and research directions.

With the advent of new processors and emerging virtualization technologies, infrastructure software needs to define new ways of exploiting the multiplicity of computa-
tional, communication, and storage resources to obtain scalable systems that are pre-
dictable and measurable [21]. Designed to take advantage of the diversity resources
offered by the multicore processors, Multimed enables a flexible deployment and config-
urations based on the available resources. Two of the key aspects of Multimed are the
independence from the underlying database engine and the ability to take advantage
of current hardware developments, something that is not always the case for alterna-
tive approaches. With the advent of new technologies and larger number of cores and
increase in memory size of future multicore systems, Multimed will only have to gain.
Further performance gains can be obtained the use of network attached storage, and
SSD/Flash storage. In addition, it is in a better position to cope with the impending
heterogeneity of multicore machines by allowing asymmetric replicas of the database
that can be specialized to the characteristics of the underlying cores.
List of Figures

2.1 Memory hierarchy .............................................. 10
2.2 The shift from single cores to multicores ..................... 13
2.3 Evolution of CPU frequencies and number of cores with time .... 14
2.4 Cache coherence levels for modern multicores ................. 15
2.5 Transactions running under SI .................................. 23
2.6 Categorization of replication systems .......................... 26
2.7 An RSI-PC scheduler executing transactions, an adaptation from [72] ............................... 31

3.1 Components of the testing framework .......................... 45
3.2 Architecture and a 4 way Max I/O interconnect for an AMD Opteron 6174
  Magny Cours .............................................. 46
3.3 PostgreSQL with 20GB database running TPCW-B .................. 48
3.4 PostgreSQL with 20GB database running TPCW-S .................. 50
3.5 PostgreSQL with 20GB database running TPCW-O .................. 51
3.6 MySQL with 20GB database running TPCW-B ........................ 54
3.7 MySQL with 20GB database running TPCW-S ........................ 55
3.8 MySQL with 20GB database running TPCW-O ........................ 56
3.9 Detailed RT of PostgreSQL with 20 GB database running the TPCW-B mix ............... 57
3.10 PostgreSQL with 20 GB database running just the BestSellers transaction ......... 58
3.11 PostgreSQL with 20 GB database running TPCW-B mix without the Best-
  Sellers transaction .............................................. 59
List of Figures

3.12 Cache misses in PostgreSQL with 20 GB database running different vari-
ations of TPCW-B with varying number of cores and 225 clients ........ 61
3.13 Comparison of cache misses when running PostgreSQL with 20 GB
database while varying the number of TPCW-B clients ................... 62
3.14 MySQL contention for locks with a 20 GB database running TPCW-B .... 64
3.15 NUMA Nodes in the 4 way AMD Opteron Magny Cours ............... 65
3.16 Evolution of individual TPCH query response time with an increasing
number of clients when using a 10GB TPC-H database (SF=10) and
PostgreSQL on 48 cores ........................................... 66

4.1 The architecture of the Multimed system ............................. 72
4.2 A possible deployment of the Multimed system ....................... 76
4.3 A bad deployment of the Multimed system ............................. 82

5.1 Scalability with the number of cores of Multimed running on PostgreSQL
with 20 GB database, using 200 TPCW-B clients .......................... 94
5.2 Scalability with the number of TPCW-B clients of Multimed running on
PostgreSQL with 20 GB database ......................................... 96
5.3 Scalability with the number of cores of Multimed running on MySQL with
20 GB database, using 200 TPCW-B clients ................................. 98
5.4 Scalability with the number of TPCW-B clients of Multimed running on
MySQL with 20 GB database ................................................ 99
5.5 Scalability with the number of cores of Multimed running on PostgreSQL
with 20 GB database, using 200 TPCW-S clients .......................... 101
5.6 Scalability with the number of TPCW-S clients of Multimed running on
PostgreSQL with 20 GB database ......................................... 102
5.7 Scalability with the number of cores of Multimed running on MySQL with
20 GB database, using 200 TPCW-S clients .................................. 104
5.8 Scalability with the number of TPCW-S clients of Multimed running on
MySQL with 20 GB database ............................................. 105
5.9 Multimed deployment for the TPC-H Benchmark ....................... 107
5.10 Evolution of individual TPCH query response time running on PostgreSQL and Multimed, using a 10GB TPC-H database (SF=10), PostgreSQL and Multimed running on 48 cores with 24 TPC-H clients ........................................ 108
5.11 Effects of Multimed with 20 GB database running TPCW-B on PostgreSQL109
6.1 Multimed deployment with specialized satellites .......................... 112
6.2 The effect of disabling nested loops for the PostgreSQL engine when running TPC-WB with a 20GB database .......................... 114
6.3 Optimization of BestSellers - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database .......... 116
6.4 1 KW Search Client - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database ........ 119
6.5 10 KW Search Clients - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database .......... 120
6.6 1 Ranking Client - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database .......... 123
6.7 10 Ranking Clients - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database .......... 124
6.8 1 Skyline Client - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database .......... 128
6.9 10 Skyline Clients - Performance with varying number of TPCW-B clients of Multimed running on PostgreSQL with 20 GB database .......... 129
6.10 Multimed deployment with the Perm Satellite .......................... 131
6.11 Evolution of individual TPCH query response time running on PostgreSQL and Multimed, using a 10GB TPC-H database (SF=10), PostgreSQL and Multimed running on 48 cores with 24 TPC-H clients .......... 132
Bibliography


