High-performance main memory database management systems

by

Spyridon Blanas

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2013

Date of final oral examination: 07/09/2013

The dissertation is approved by the following members of the Final Oral Committee:

Jignesh M. Patel, Professor, Computer Sciences
David J. DeWitt, Emeritus Professor, Computer Sciences
AnHai Doan, Associate Professor, Computer Sciences
Jonathan T. Eckhardt, Associate Professor, Management and Human Resources
Jeffrey F. Naughton, Professor, Computer Sciences
David A. Wood, Professor, Computer Sciences

To Katerina, for all she went through.

Acknowledgments

This six-year endeavor would not have been as didactic had I not been surrounded with outstanding friends, peers and mentors that embraced the uncertainty associated with research, promoted excellence and helped me distinguish what is important from what is urgent. This section represents an incomplete list of the people who helped create this unique environment that allowed me to grow both personally and professionally.

I thank my advisor, Jignesh Patel, for teaching me perseverance, supporting me in mistakes, never using his absolute power of authority to resolve a conflict or dictate a direction, and —most importantly— for being patient with my volatile Mediterranean temperament. David DeWitt financially supported this endeavor, and frequently dispensed his unique mix of deeply personal advice with sharp and timely criticism. Jeff Naughton, AnHai Doan, and Chris Ré made me appreciate the breadth of the data management field and sharpened my critical thinking through numerous discussions. David Wood, Mark Hill, and Mike Swift kept me connected with the computer architecture and operating systems communities, and welcomed me to tap into the tangible, but more importantly the intangible resources of the Multifacet group. I thank them all for their wonderful advice throughout the years.

During my studies, I was extremely fortunate to extensively interact with researchers and engineers in industry. I thank Alan Halverson, Hideaki Kimura, Willis Lang, Rimma Nehme, Eric Robinson, Srinath Shankar,

Nikhil Teletia, Dimitris Tsirogiannis, Dhongui Zhang, and Melody Bakken at the Microsoft Jim Gray Systems Lab for the wonderful daily interactions, and their unique industrial perspective. Cristian Diaconu, Craig Freedman and Mike Zwilling helped make the summer of 2009 memorable, productive and enjoyable. After working with them for a few months, I learned to respect the effort involved in building working, production-quality systems. Paul Larson helped me realize the benefits of being organized and taking strategic small steps every day. I also thank Ravi Ramamurthy, Arvind Arasu, Ken Eguro and Raghav Kaushik for teaching teamwork through example in the summer of 2012. Finally, Jun Rao, Vuk Ercegovac and Eugene Shekita introduced me to industrial research, and they deserve credit for convincing me to stay in the Ph.D. program at the end of my first year. Working with all these talented individuals made me appreciate the simple and practical solutions to complex research problems, and I thank them for that lesson.

The most memorable and invaluable part of my education has been the amazing network of current and former students who were always eager to provide advice, critical comments and support. This network includes Arkaprava Basu, Victor Bittorf, Craig Chasseur, Fei Chen, Jaeyoung Do, Yasuko Eckert, Avrilia Floratou, Dan Gibson, Chaitanya Gokhale, Cindy Rubio González, Yeye He, Allison Holloway, Derek Hower, Asim Kadav, Arun Kumar, Willis Lang, Yinan Li, Jiexing Li, David Malec, Ian Rae, Somayeh Sardashti, Mohit Saxena, Rathijit Sen, Warren Shen, Swaminathan Sundararaman, Khai Tran, Haris Volos, Ba-Quy Vyong, and Chen Zeng. I would have learned far less in graduate school had it not been for the countless interruptions and impromptu discussions with these students.

In addition, I would also like to thank Tycho Andersen, Polina Dudnik, Tristan Ravitch, Dana Vantrease, Haris Volos, and Dalibor Zelený, for tolerating me during a hard transition from the relaxed, warm Greek island of Crete to the sub-zero temperatures of Wisconsin during the first winter.

The regular social activities of the following Greek students in Madison helped remind me of home, and prevented my Greek from deteriorating. I would like to thank Aris Avgoustis, Michalis Bachtis, Avrilia Floratou, Nikos Georgiou, George Manoussakis, Kostas Mavrakakis, Charalambos Michael, Loizos Solomou, Giorgos Stratis, Mina Syrika, Joanne Tsarouha, Haris Volos, and Andreas Vlachos for all our fun outings.

Looking back, I would have never considered relocating to a different continent to pursue a graduate degree without the full support of my family, Stathis, Maria and Konstantina. Thank you for tirelessly encouraging me to stay curious and pursue my dreams.

Finally, and most importantly, a special thank you goes to Katerina, who reminded me of the existence of other priorities in life every time I had nearly convinced myself that there were none. Katerina, thank you for all the sacrifices you have made for me to reach this point.

Although all these individuals contributed to this dissertation in varying degrees and in different ways, all errors and omissions herein are my sole responsibility.

Contents

Contents **v**

List of Tables **vii**

List of Figures **viii**

Abstract **x**

1 Introduction **1**

1.1 Opportunities for the next generation of database management systems 2

1.2 Outline 7

2 Redesigning the hash join algorithm for single-socket, multi-core CPUs **9**

2.1 Introduction 10

2.2 Related work 13

2.3 The multi-core landscape 15

2.4 Different hash join variants 16

2.5 Experimental evaluation 24

2.6 Experimenting with a different implementation 47

2.7 Concluding remarks 53

3 Equi-join algorithms for memory-resident data **55**

3.1 Introduction 56
3.2 Recent related work 59
3.3 Join algorithms 59
3.4 Evaluation methodology 71
3.5 Experimental results 74
3.6 Concluding remarks 95

4 Concurrency control for main memory databases 97
4.1 Introduction 98
4.2 Related work 99
4.3 Multi-version storage engine 101
4.4 Optimistic transactions 117
4.5 Pessimistic transactions 126
4.6 Experimental results 137
4.7 Concluding remarks 147

5 Conclusions and future work 149

Bibliography 152

List of Tables

2.1 Platform characteristics. 22
2.2 Shorthand notation used when presenting the results 32
2.3 Hardware events for the uniform dataset 33
2.4 Hardware events for the high skew dataset 34
2.5 Impact of simultaneous multi-threading (Intel Nehalem) 37
2.6 Impact of simultaneous multi-threading (Sun UltraSPARC T2) . 38
2.7 Overhead of materialization 42
2.8 Sensitivity to join input cardinalities 43
2.9 Radix partitioning efficiency 49
2.10 Cycle breakdown during radix partitioned hash join. 52

3.1 Join query plans that are considered 72
3.2 Time breakdown per operator, for the uniform dataset. 78
3.3 Wide-tuple dataset properties 91

4.1 Version visibility outline 111
4.2 Version visibility when version's `Begin` field is not a timestamp 112
4.3 Version visibility when version's `End` field is not a timestamp . . 114
4.4 Throughput at higher isolation levels 142
4.5 TATP results. 147

List of Figures

1.1 Performance per thread from TPC-C and TPC-H results 3
1.2 Inflation-adjusted price of DRAM memory per megabyte 6

2.1 Cycles per output tuple for the uniform dataset. 23
2.2 Cycles per output tuple for the low skew dataset. 27
2.3 Cycles per output tuple for the high skew dataset. 31
2.4 Speedup over single threaded execution, uniform dataset. 36
2.5 Time breakdown of the radix join. 39
2.6 Performance on Intel Nehalem with uniform dataset and $|R|=|S|$. 44
2.7 Sensitivity to join selectivity. 46

3.1 Results when S is in random order, uniform dataset 77
3.2 Results when S is sorted in join key order, uniform dataset . . . 82
3.3 Results when S is in random order, skewed dataset 85
3.4 Results when S is sorted in join key order, skewed dataset . . . 89
3.5 Join response time with wider tuples. 93

4.1 Example account table with one hash index. 105
4.2 State transitions for each transaction. 108
4.3 Possible transaction validation outcomes. 121
4.4 Scalability under low contention. 139
4.5 Scalability under high contention. 141
4.6 Impact of read-only transactions on throughput (low contention). 143

4.7 Impact of read-only transactions on throughput (high contention).144
4.8 Update throughput with long read transactions. Each update transaction performs ten reads and two updates. 145
4.9 Read throughput with long read transactions. Each long read transaction performs one million reads (10% of the database). . 146

Abstract

Decision makers today want to analyze constantly evolving datasets of unprecedented volume and complexity in real time. This poses a significant challenge for the underlying data management system. In the past, data processing could scale to meet the growing demand with few changes to the individual software components mainly due to a sustained improvement in single-threaded processor performance. Because of fundamental technological limitations, however, single-processor performance has recently been increasing much more slowly than in the past.

It is not uncommon today for a single database server to be able to concurrently execute instructions from hundreds of threads and store terabytes of data in main memory. Commercial database management systems, however, have not been designed for such hardware; they treat main memory as a vast software-controlled cache, and commonly rely on multiple concurrent requests to fully utilize a modern system. My thesis is that we can improve data processing efficiency by one order of mangitude if we redesign the data processing kernel to better leverage existing hardware.

This dissertation makes three contributions to main memory database management systems. The first contribution is a simple non-partitioned hash join for memory-resident data that has comparable performance with much more sophisticated hash join methods. The second contribution is demonstrating that hash join plans are commonly advantageous over sort-merge join plans in a main-memory setting because they commonly have

shorter query response times while reserving less working memory. The third contribution is the design and implementation of two multi-version concurrency control schemes that are optimized for main memory storage, and can achieve throughputs of millions of transactions per second without sacrificing transactional atomicity, isolation or durability.

This dissertation points to promising directions for future performance improvements in the database system kernel, and identifies key open problems in the areas of query execution, transaction processing and query optimization.

Chapter 1

Introduction

Data today is being generated and collected at an unprecedented rate, with data-driven decision making largely replacing the ad-hoc models of the past in industries such as retailing, healthcare, disaster management, finance, and more. Decision makers today demand real-time answers to queries that involve an exponentially increasing amount of data. The performance of database management systems had, until recently, been improving exponentially to match this demand. Jim Gray notes that transaction processing performance has achieved an average improvement of 68% per year, slightly higher than Moore's law [39].

The primary technique used to process data quickly and cost-effectively has been to scale out, that is, distribute data and computation among multiple computer systems. Fueled by the economies of scale, corporations and governments are aggressively building data centers containing hundreds of thousands of computing nodes. Further data center expansion, however, is likely to encounter significant headwinds, primarily due to high energy consumption, and thus deliver diminishing returns in the future. As a consequence, using a single system efficiently is becoming a crucial consideration.

In the past, the data processing pipeline could scale to meet the growing demand with few major changes to the underlying software components.

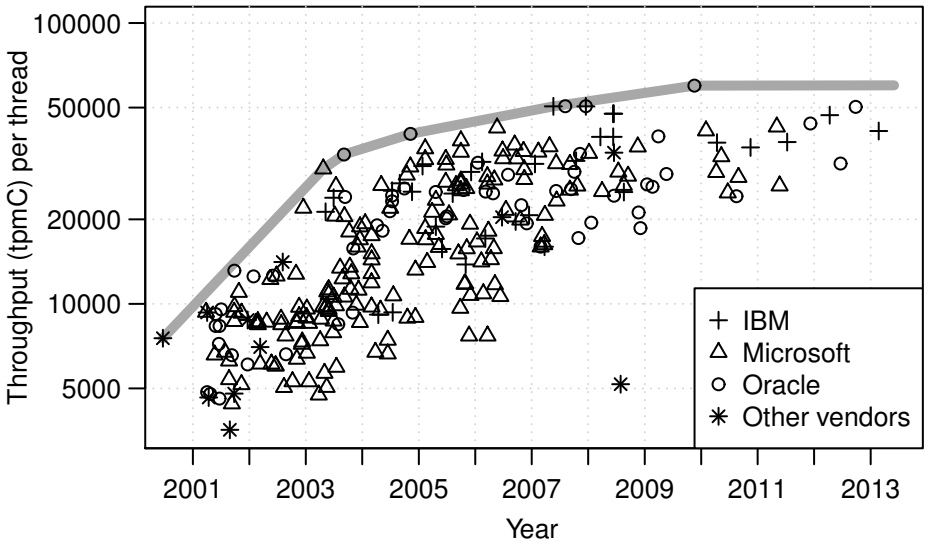
Two factors mainly contributed to this phenomenon. The first was a sustained 52% yearly performance improvement in performance that was powered by innovative computer architecture research and an impressive 40% growth in clock rates per year [44]. The second factor was the constantly diminishing price per gigabyte of hard disk storage. As a consequence, computer manufacturers could deliver a new generation of computer systems every year with faster CPUs and more aggregate disk bandwidth for the same cost. Such a system could complete last year's data processing tasks in a fraction of the time, and without any changes to the software. In the last few years, however, physical limitations, such as the heat dissipation rate, have caused the clock rate growth to virtually stall. Single-processor performance has increased much more slowly than in the past, only about 22% per year [44].

We are now facing the challenge that bigger data centers with newer computer systems running old software will fail to deliver answers at the cost and speed that decision makers expect. This dissertation explores a different approach to improve the efficiency of the data processing pipeline: *redesign the data processing software to better exploit existing hardware.*

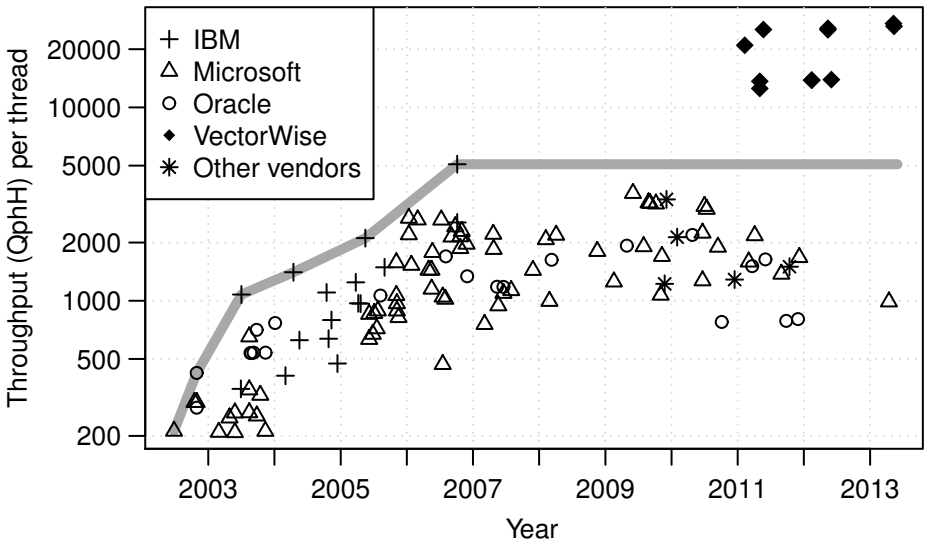
In this evolving architectural landscape, database management systems have a unique opportunity. The relational data model allows user applications to interact with data through a declarative query language that captures intent, leaving the system free to choose the optimal execution strategy. Innovation in this area can have a transformative impact on the entire big data ecosystem.

1.1 Opportunities for the next generation of database management systems

One potential solution to improve performance in light of these fundamental changes in hardware would be to adjust the various configuration knobs



(a) Transaction processing performance per thread, measured in TPC-C transactions completed per minute.



(b) Decision support performance per thread, measured in TPC-H queries answered per hour. VectorWise is a new database system that has been designed from scratch to better utilize modern hardware [80].

Figure 1.1: Performance per thread for transaction processing and decision support workloads. The thick gray line denotes peak performance per thread among the three established database software vendors.

found in the existing database management systems. If fully utilizing existing hardware was a matter of tuning, this would be reflected in the performance results of the standard TPC benchmarks which are highly regarded by the industry. Such results, after all, are submitted by teams working in close cooperation with the original software and hardware vendors, and each submission is independently audited for adherence to the benchmarking standards.

Unfortunately, an analysis of the published TPC results indicates that reconfiguring existing database management systems has yielded diminishing returns in the past. In fact, peak single-threaded performance among the established database software vendors has hardly improved in the past five years. This holds for both transactional and decision support workloads, which are shown in Figures 1.1(a) and 1.1(b), respectively. The introduction of VectorWise [80] in 2011 (shown in the upper right corner of Figure 1.1(b)) demonstrates that a database system that has been designed to better utilize existing hardware can improve its single-threaded performance by one order of magnitude.

This dissertation explores what performance gains a relational database management system can achieve if it is designed for two key hardware trends. The first trend is the abundance of on-chip and across-chip parallelism, as demonstrated by the rising number of cores per die and sockets per computer. The second trend is the continuing drop of memory price per gigabyte, which now makes it possible to store very large datasets entirely in main memory. The next two sections briefly outline the opportunities associated with each trend.

Eliminate serial bottlenecks

Processors today already have four to eight cores, and for the past few years CPU manufacturers have been introducing two more cores roughly every 18 months. In addition, even low-end servers today are increasingly using a

two-socket configuration. This new level of hardware integration in a single chip leads to architectural changes with deep impact.

Beginning about three decades ago, the data management community thoroughly examined how a database management system can use various forms of parallelism. These forms of parallelism include pure shared-nothing, shared-memory, and shared disk architectures [74]. If the modern multi-core architectures resembled any of these architectural templates, then we could simply adopt the methods that have already been designed. To a large extent, this is the approach that database system designers have taken towards dealing with multi-socket, multi-core machines. At a high level, query processing today involves variations of techniques that were developed for parallel shared-nothing architectures [26], but was adapted for shared multi-processor (SMP) machines.

Most commercial database management systems simply treat a single multi-core processor as many independent processors and break up the task of a single operation, such as an join, into independent work units to allow each core to work on each part independently. The drawback of this parallelization strategy is that it does not take advantage of the opportunities for quick communication and data sharing via the shared cache hierarchy that processors today commonly have.

Most open-source database management systems, on the other hand, have largely ignored multi-core processing and generally rely on request parallelism: they use each core to run a different query. Although this strategy is trivial to implement, the disadvantage is that a single query will only utilize a small fraction of the available resources of a modern system.

Memory is more than a cache

At the other side of the motherboard, a quiet change has been underfoot. Most database system designs trace their origins back to the times when the memory capacity of a computer system was a few megabytes, or about

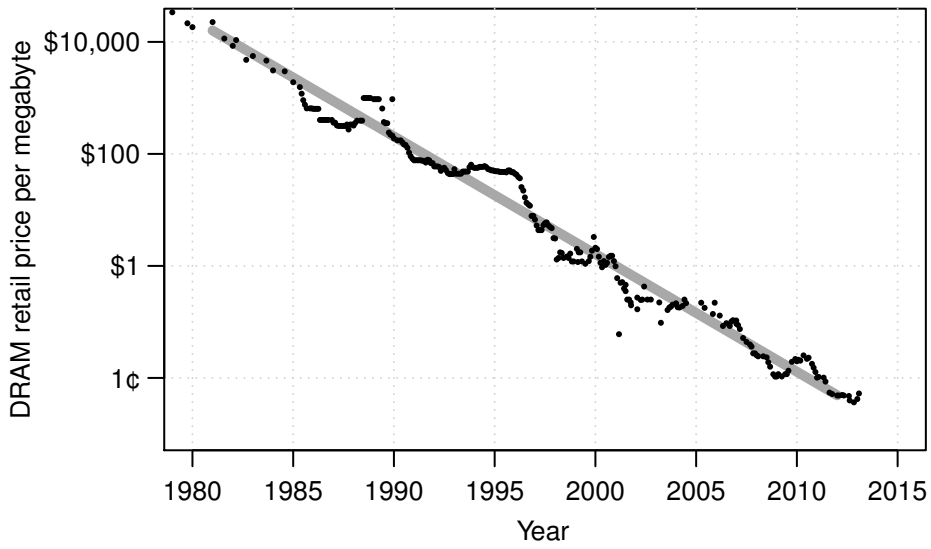


Figure 1.2: Inflation-adjusted historical DRAM memory price per megabyte, in 2013 US dollars. Data by John C. McCallum [60].

the size of the last level cache of a current CPU. One megabyte of memory, however, has been getting 10× cheaper approximately every 5 years, during the last three decades (see Figure 1.2). With one terabyte of memory costing as little as $8,000, the working set of many databases today can cheaply and efficiently be stored entirely in main memory.

Under the resource-limited environment of the past, database system designers were forced to be very careful about how and when each record would be brought in memory, processed and written back to non-volatile storage. As a result, a central component of most database managements systems is the buffer pool, which is a large software cache that temporarily stores records while they are being manipulated. In addition, in order to amortize the latency cost of accessing a single record from non-volatile storage, multiple records are physically packed together in pages.

As a result, the execution path for many database management systems

has been optimized for structures that have a disk-friendly data layout. An application that can store its entire dataset in memory today gets penalized by the existing functionality. It has been shown that in a main-memory environment, the buffer pool adds significant overhead [42], and the page layout is far from optimal [3, 69].

To summarize, all server systems now come in multi-socket multi-core configurations, providing abundant opportunities to exploit parallelism. In addition, due to a steady decline in the price per gigabyte of memory, main memory is now abundant and relatively cheap. It is not uncommon for a single database server today to be able to concurrently execute instructions from hundreds of threads and store terabytes of data in main memory. Commercial database management systems, however, have not been designed for such hardware, and as a consequence they do not utilize the available hardware to its full potential. They treat main memory as a vast software-controlled cache and largely rely on multiple concurrent requests to utilize all the processors of a system. This dissertation carefully reconsiders the implications of these design choices for data processing.

1.2 Outline

This document is structured as follows. Chapter 2 studies how to optimize the hash join operator for a main-memory setting. We discover that optimizing for low cache miss rates in a single-socket multi-core CPU is only a single dimension of a much more complex problem. In fact, we find that we have reached a point where optimizing only for cache misses yields diminishing returns. We then consider optimal execution strategies for an entire query in Chapter 3. We find that the physical organization of the inputs plays a key role in deciding on the optimal parallel query execution plan. In addition, we find that total memory consumption is an important optimization metric, as it limits the number of queries a main memory database

system can admit at any given time.

The performance of query execution is largely determined by the performance of the individual data retrieval operations to the underlying data store. Chapter 4 looks at how to efficiently guarantee the atomicity, isolation and durability of transactional operations on memory-resident data. In that chapter, we design and implement three concurrency control algorithms that have been optimized for main memory. One algorithm is a memory-optimized version of single-version two-phase locking. We then compare this scheme with two multi-version schemes, one optimistic that does validation and one pessimistic that requires synchronization. We find that the single version scheme is extremely efficient under short update-heavy transactions, but its performance deteriorates in the presence of long-running read queries. Multi-versioning implementations prove to be more robust, even when there is high contention. A unique characteristic of the proposed design is that transactions running with either multi-version scheme can co-exist with each other and thus the database system can dynamically choose the optimal execution strategy. Finally, Chapter 5 outlines the contributions of the dissertation and discusses future work.

Chapter 2

Redesigning the hash join algorithm for single-socket, multi-core CPUs

The focus of this chapter is on investigating efficient hash join algorithms for modern multi-core processors in main memory environments. We dissect each internal phase of a typical hash join algorithm and considers different alternatives for implementing each phase, producing a family of hash join algorithms. Then, we implement these main memory algorithms on two radically different modern multi-processor systems, and carefully examine the factors that impact the performance of each method.

Our analysis reveals some interesting results – a very simple hash join algorithm is very competitive to the other more complex methods. This simple join algorithm builds a shared hash table and does not partition the input relations. Its simplicity implies that it requires fewer parameter settings, thereby making it far easier for query optimizers and execution engines to use it in practice. Furthermore, the performance of this simple algorithm improves dramatically as the skew in the input data increases, and it quickly starts to outperform all other algorithms. Based on our

results, we propose that database implementers consider adding this simple join algorithm to their repertoire of main memory join algorithms, or adapt their methods to mimic the strategy employed by this algorithm, especially when joining inputs with skewed data distributions.

2.1 Introduction

Processors today already have four or more cores, and for the past few years Intel has been introducing two more cores per processor roughly every 15 months. At this rate, it is not hard to imagine running database management systems (DBMSs) on processors with hundreds of cores in the future. In addition, memory prices are continuing to drop. Consequently, many databases now either fit entirely in main memory, or their working set is main memory resident. As a result, many DBMSs are becoming CPU bound.

In this evolving architectural landscape, DBMSs have the unique opportunity to leverage the inherent parallelism that is provided by the relational data model. Data is exposed by declarative query languages to user applications and the DBMS is free to choose its execution strategy. Coupled with the trend towards impending very large multi-cores, this implies that DBMSs must carefully rethink how they can exploit the parallelism that is provided by the modern multi-core processors, or DBMS performance will stall.

A natural question to ask then is whether there is anything new here. Beginning about three decades ago, at the inception of the field of parallel DBMSs, the database community thoroughly examined how a DBMS can use various forms of parallelism. These forms of parallelism include pure shared-nothing, shared-memory, and shared disk architectures [74]. If the modern multi-core architectures resemble any of these architectural templates, then we can simply adopt the methods that have already been

designed.

In fact, to a large extent this is the approach that DBMSs have haven taken towards dealing with multi-core machines. Many commercial DBMSs simply treat a multi-core processor as a symmetric multi-processor (SMP) machine, leveraging previous work that was done by the DBMS vendors in reaction to the increasing popularity of SMP machines decades ago. These methods break up the task of a single operation, such as an equijoin, into disjoint parts and allow each processor (in an SMP box) to work on each part independently. At a high-level, these methods resemble variations of query processing techniques that were developed for parallel shared-nothing architectures [25], but adapted for SMP machines. In most commercial DBMSs, this approach is reflected across the entire design process, ranging from system internals (join processing, for example) to their pricing model, which is frequently done by scaling the SMP pricing model. On the other hand, open-source DBMSs have largely ignored multi-core processing and generally dedicate a single thread/process to each query.

The design space for modern high performance main memory join algorithms has two extremes. Reflecting on the previous work in this area, one can observe that the database community has focused on optimizing query processing methods to reduce the number of processor cache and TLB misses. One extreme of this design space focuses on minimizing the number of processor cache misses. The radix-based hash join algorithm [16] is an example of a method in this design class. The other extreme is to focus on minimizing processor synchronization costs. In this chapter we propose a "no partitioning" hash join algorithm that does not partition the input relations to embody an example of a method in this later design space.

A crucial question that we answer is *what is the impact of these two extreme design points in modern multi-core processors for main memory hash join algorithms*. A perhaps surprising answer is that for modern multi-core architectures, in many cases the right approach is to focus on reducing the

computation and synchronization costs, as modern processors are very effective in hiding cache miss latencies via simultaneous multi-threading. For example, in our experiments, the "no partitioning" hash join algorithm far outperforms the radix join algorithm when there is skew in the data (which is often the case in practice), even while it incurs many more processor cache and TLB misses. Even with uniform data, the radix join algorithm only outperforms the "no partitioning" algorithm on a modern Intel Xeon when the parameters for the radix join algorithm are set at or near their optimal setting. In contrast, the non-partitioned algorithm is "parameter-free", which is another important practical advantage.

This chapter makes three main contributions. First, we systematically examine the design choices available for each internal phase of a canonical main memory hash join algorithm – namely, the partition, build, and probe phases – and enumerate a number of possible multi-core hash join algorithms based on different choices made in each of these phases. We then evaluate these join algorithms on two radically different architectures and show how the architectural differences can affect performance. Unlike previous work that has often focused on just one architecture, our use of two radically different architectures lets us gain deeper insights about hash join processing on multi-core processors. To the best of our knowledge, this is the first systematic exploration of multiple hash join techniques that spans multi-core architectures.

Second, we show that an algorithm that does not do any partitioning, but simply constructs a single shared hash table on the build relation often outperforms more complex algorithms. This simple "no-partitioning" hash join algorithm is robust to sub-optimal parameter choices by the optimizer, and does not require any knowledge of the characteristics of the input to work well. To the best of our knowledge, this simple hash join technique differs from what is currently implemented in existing DBMSs for multi-core hash join processing, and offers a tantalizingly simple, efficient, and

robust technique for implementing the hash join operation.

Finally, we show that the simple "no-partitioning" hash join algorithm takes advantage of intrinsic hardware optimizations to handle skew. As a result, this simple hash join technique often benefits from skew and its relative performance increases as the skew increases! This property is a big advancement over the state-of-the-art methods, as it is important to have methods that can gracefully handle skew in practice [29].

The remainder of this chapter is organized as follows: The next two section cover background information. The hash join variants are presented in Section 2.4, and experimental results are described in Section 2.5. Section 2.6 examines how sensitive our findings are with respect to the implementation of the radix-partitioned join. Finally, Section 2.7 contains our concluding remarks.

2.2 Related work

There is a rich history of studying hash join performance for main memory database systems, starting with the early work of DeWitt et al. [27]. A decade later Shatdal et al. [72] studied cache-conscious algorithms for query execution and discovered that the probe phase dominates the overall hash join processing time. They also showed that hash join computation can be sped up if both the build and probe relations are partitioned so as to fit in the cache.

Ailamaki et al. [4] studied where the time is spent when executing four different commercial databases systems by assigning fixed latency penalties to architectural events. The study reveals two prominent areas that developers can focus on to improve performance of database code: they should try to do data placement to avoid cache misses and optimize branches so that branch mispredictions are minimized.

Manegold et al. [58] inspected the time breakdown for a hash join oper-

ation, and singled out cache and TLB misses as the two primary culprits for suboptimal performance in main memory hash join processing. A follow-up paper [59] presented a cost model on how to optimize the performance of the radix join algorithm on a uniprocessor [16].

Ross [70] presented a more efficient way to improve the performance of hash joins by using cuckoo hashing [66] and SIMD instructions. More recently, Fan et al. [30] described one approach for making cuckoo hashing safe for concurrent modifications. Garcia and Korth [33] have studied the benefits of using simultaneous multi-threading for hash join processing.

Cieslewicz and Ross [23] studied how to efficiently partition a relation with a multi-core processor. The main insight from this work is that when the number of created partitions is small, then it is best to use independent per-core output buffers to avoid lock contention. If more partitions are desired, the working set for the independent output buffer algorithm exceeds the cache size and performance degrades. The best option then becomes to share output buffers among all threads and synchronize accesses either through blocking or atomic operations. We leverage these two findings in our work.

With the advent of multi-core CPUs, there was a renewed interest in parallel join algorithms. Kim et al. [54] compared a parallel sort-merge join with a parallel radix join. Their study relies on an analytical model and raises interesting issues about the impact of SIMD register lengths on the relative performance of the sort-merge join versus the hash-join in the future. As the SIMD registers have doubled in size with the introduction of the AVX extension, revisiting the use of SIMD for join processing is a promising direction that complements our work. Balkesen et al. [9] extended this work and demonstrated that hardware-conscious implementations of the non-partitioned and radix partitioned join algorithms can greatly improve performance, while Albutiu et al. [5] introduced a NUMA-aware sort-merge algorithm that is designed for a modern multi-socket server

Finally, there has been prior work in handling skew during hash join processing. The experiments with a high number of partitions that we will present in Section 2.5.4 are an extension of an idea by DeWitt et al. [29] for a main memory, multi-core environment.

2.3 The multi-core landscape

In the last few years alone, more than a dozen different multi-core CPU families have been introduced by CPU vendors. These new CPUs have ranged from powerful dual-CPU systems on the same die to prototype systems of hundreds of simple RISC cores.

This new level of integration has lead to architectural changes with deep impact on algorithm design. Although the first multi-core CPUs had dedicated caches for each core, we now see a shift towards more sharing at the lower levels of the cache hierarchy and consequently the need for access arbitration to shared caches within the chip. A shared cache means better single-threaded performance, as one core can utilize the whole cache, and more opportunities for sharing among cores. However, shared caches also increase conflict cache misses due to false sharing, and may increase capacity cache misses, if the cache sizes don't increase proportionally to the number of cores.

One idea that is employed to combat the diminishing returns of instruction-level parallelism is simultaneous multi-threading (SMT). Multi-threading attempts to find independent instructions across different threads of execution, instead of detecting independent instructions in the same thread. This way, the CPU will schedule instructions from each thread and achieve better overall utilization, increasing throughput at the expense of per-thread latency.

We briefly consider two modern architectures that we subsequently use for evaluation. At one end of the spectrum, the Intel Nehalem family is an

instance of Intel's latest microarchitecture that offers high single-threaded performance because of its out-of-order execution and on-demand frequency scaling (TurboBoost). Multi-threaded performance is increased by using simultaneous multi-threading (HyperThreading). At the other end of the spectrum, the Sun UltraSPARC T2 has 8 simple cores that all share a single cache. This CPU can execute instructions from up to 8 threads per core, or a total of 64 threads for the entire chip, and extensively relies on simultaneous multi-threading to achieve maximum throughput.

2.4 Different hash join variants

In this section, we consider the anatomy of a canonical hash join algorithm, and carefully consider the design choices that are available in each internal phase of a hash join algorithm. Then using these design choices, we categorize various previous proposals for multi-core hash join processing. In the following discussion we also present information about some of the implementation details, as they often have a significant impact on the performance of the technique that is described.

A hash join operator works on two input relations, R and S. We assume that $|R| < |S|$. A typical hash join algorithm has three phases: partition, build, and probe. The partition phase is optional and divides tuples into distinct sets using a hash function on the join key attribute. The build phase scans the relation R and creates an in-memory hash table on the join key attribute. The probe phase scans the relation S, looks up the join key of each tuple in the hash table, and in the case of a match creates the output tuple(s).

Before we discuss the alternative techniques that are available in each phase of the join algorithm, we briefly digress to discuss the impact of the latch implementation on the join techniques. As a general comment, we have found that the latch implementation has a crucial impact on the

overall join performance. In particular, when using the pthreads mutex implementation, several instructions are required to acquire and release an uncontended latch. If there are millions of buckets in a hash table, then the hash collision rate is small, and one can optimize for the expected case: latches being free. Furthermore, pthread mutexes have significant memory footprint as each requires approximately 40 bytes. If each bucket stores only a few <key, record-id> pairs, then the size of the latch array may be greater than the size of the hash table itself. These characteristics make mutexes a prohibitively expensive synchronization primitive for buckets in a hash table. Hence, we implemented our own 1-byte latch for both the Intel and the Sun architectures, using the atomic primitives *xchgb* and *ldstub*, respectively. Protecting multiple hash buckets with a single latch to avoid cache thrashing did not result in significant performance improvements even when the number of partitions was high.

2.4.1 Partition phase

The partition phase is an optional step of a hash join algorithm, if the hash table for the relation R fits in main memory. If one partitions both the R and S relations such that each partition fits in the CPU cache, then the cache misses that are otherwise incurred during the subsequent build and probe phases are almost eliminated. The cost for partitioning both input relations is incurring additional memory writes for each tuple. Work by Shatdal et al. [72] has shown that the runtime cost of the additional memory writes during partitioning phase is less than the cost of missing in the cache – as a consequence partitioning improves overall performance. Recent work by Cieslewicz and Ross [23] has explored partitioning performance in detail. They introduce two algorithms that process the input once in a serial fashion and do not require any kind of global knowledge about the characteristics of the input. Another recent paper [54] describes a parallel implementation of radix partitioning [16] which gives impressive performance improvements

on a modern multi-core system. This implementation requires that the entire input is available upfront and will not produce any output until the last input tuple has been seen. We experiment with all of these three partitioning algorithms, and we briefly summarize each implementation in Sections 2.4.1 and 2.4.1.

In our implementation, a partition is a linked list of output buffers. An output buffer is fully described by four elements: an integer specifying the size of the data block, a pointer to the start of the data block, a pointer to the free space inside the data block and a pointer to the next output buffer that is initially set to zero. If a buffer overflows, then we add an empty output buffer at the start of the list, and we make its next pointer point to the buffer that overflowed. Locating free space is a matter of checking the first buffer in the list.

Let p denote the desired number of partitions and n denote the number of threads that are processing the hash join operation. During the partitioning phase, all threads start reading tuples from the relation R, via a cursor. Each thread works on a large batch of tuples at a time, so as to minimize synchronization overheads on the input scan cursor. Each thread examines a tuple, then extracts the key k, and finally computes the partitioning hash function $h_p(k)$. Next, it then writes the tuple to partition $R_{h_p(k)}$ using one of the algorithms we describe below. When the R cursor runs out of tuples, the partitioning operation proceeds to process the tuples from the S relation. Again, each tuple is examined, the join key k is extracted and the tuple is written to the partition $S_{h_p(k)}$. The partitioning phase ends when all the S tuples have been partitioned.

Note that we classify the partitioning algorithms as "non-blocking" if they produce results on-the-fly and scan the input once, in contrast to a "blocking" algorithm that produces results after buffering the entire input and scanning it more than once. We acknowledge that the join operator overall is never truly non-blocking, as it will block during the build phase.

The distinction is that the non-blocking algorithms only block for the time that is needed to scan and process the smaller input, and, as we will see in Section 2.5.3, this a very small fraction of the overall join time.

Non-blocking algorithms

The first partitioning algorithm creates p shared partitions among all the threads. The threads need to synchronize via a latch to make sure that the writes to a shared partition are isolated from each other.

The second partitioning algorithm creates $p * n$ partitions in total and each thread is assigned a private set of p partitions. Each thread then writes to its local partitions without any synchronization overhead. When the input relation is depleted, all threads synchronize at a barrier to consolidate the $p * n$ partitions into p partitions.

The benefit of creating private partitions is that there is no synchronization overhead on each access. The drawbacks, however, are (a) many partitions are created, possibly so many that the working set of the algorithm no longer fits in the data cache and the TLB; (b) at the end of the partition phase some thread has to chain n private partitions together to form a single partition, but this operation is quick and can be parallelized.

Blocking algorithm

Another partitioning technique is the parallel multi-pass radix partitioning algorithm described by Kim et al. [54]. The algorithm begins by having the entire input available in a contiguous block of memory. Each thread is responsible for a specific memory region in that contiguous block. A histogram with $p * n$ bins is allocated and the input is then scanned twice. During the first scan, each thread scans all the tuples in the memory region assigned to it, extracts the key k and then computes the exact histogram of the hash values $h_p(k)$ for this region. Thread $i \in [0, n-1]$ stores the number of tuples it encountered that will hash to partition $j \in [0, p-1]$ in

histogram bin $j*n+i$. At the end of the scan, all the n threads compute the prefix sum on the histogram in parallel. The prefix sum can now be used to point to the beginning of each output partition for each thread in the single shared output buffer. Finally, each thread performs a second scan of its input region, and uses h_p to determine the output partition. This algorithm is recursively applied to each output partition for as many passes as requested.

The benefit of radix partitioning is that it makes few cache and TLB misses, as it bounds the number of output destinations in each pass. This particular implementation has the benefit that, by scanning the input twice for each pass, it computes exactly how much output space will be required for each partition, and hence avoids the synchronization overhead that is associated with sharing an output buffer. Apart from the drawbacks that are associated with any blocking algorithm when compared to a non-blocking counterpart, this implementation also places a burden on the previous operator in a query tree to produce the compact and contiguous output format that the radix partitioning requires as input. Efficiently producing a single shared output buffer is a problem that has been studied before [24].

2.4.2 Build phase

The build phase proceeds as follows: If the partition phase was omitted, then all the threads are assigned to work on the relation R. If partitioning was done, then each thread i is assigned to work on partitions $R_{i+0*n}, R_{i+1*n}, R_{i+2*n}$, etc. For example, a machine with four cores has $n = 4$, and thread 0 would work on partitions $R_0, R_4, R_8, ...$, thread 1 on $R_1, R_5, R_9, ...$, etc.

Next, an empty hash table is constructed for each partition of the input relation R. To reduce the number of cache misses that are incurred during the next (probe) phase, each bucket of this hash table is sized so that it fits on a few cache lines. Each thread scans every tuple t in its partition,

extracts the join key k, and then hashes this key using a hash function $h(\cdot)$. Then, the tuple t is appended to the end of the hash bucket $h(k)$, creating a new hash bucket if necessary. If the partition phase was omitted, then all the threads share the hash table, and writes to each hash bucket have to be protected by a latch. The build phase is over when all the n threads have processed all the assigned partitions.

2.4.3 Probe phase

The probe phase schedules work to the n threads in a manner similar to the scheduling during the build phase, described above. Namely, if no partitioning has been done, then all the threads are assigned to S, and they synchronize before accessing the read cursor for S. Otherwise, the thread i is assigned to partitions S_{i+0*n}, S_{i+1*n}, S_{i+2*n}, etc.

During the probe phase, each thread reads every tuple s from its assigned partition and extracts the key k. It then checks if the key of each tuple r stored in hash bucket $h(k)$ matches k. This check is necessary to filter out possible hash collisions. If the keys match, then the tuples r and s are joined to form the output tuple. If the output is materialized, it is written to an output buffer that is private to the thread.

Notice that there is parallelism even inside the probe phase: looking up the key for each tuple r in a hash bucket and comparing it to k can be parallelized with the construction of the output tuple, which primarily involves shuffling bytes from tuples r and s. (See Section 2.5.10 for an experiment that explores this further.)

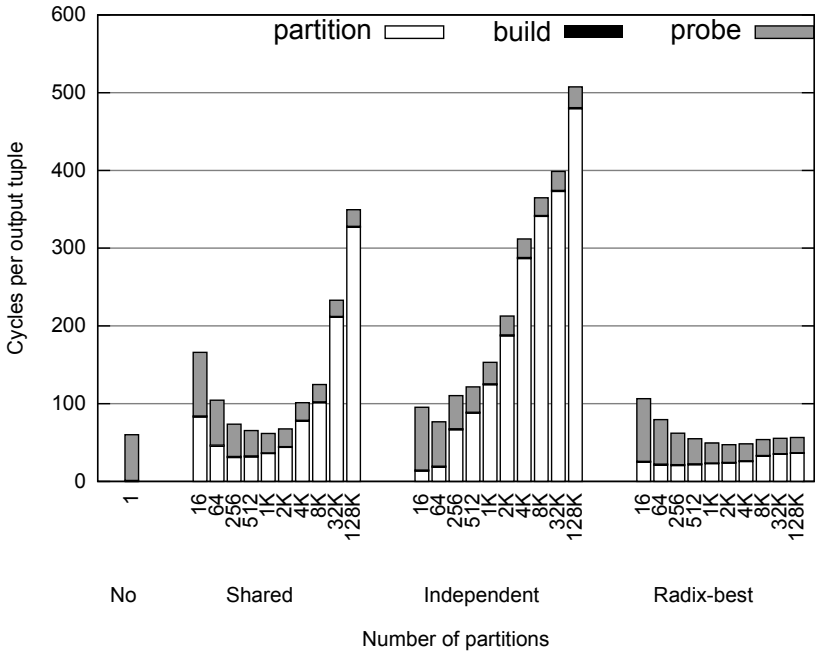
2.4.4 Hash join variants

The algorithms presented above outline an interesting design space for hash join algorithms. In this paper, we focus on the following four hash join variations:

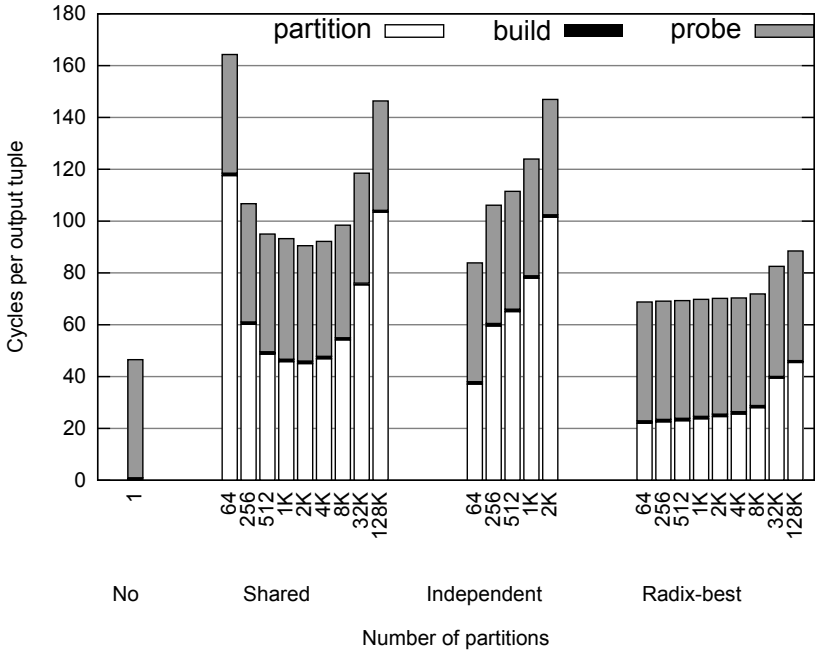
	Intel Nehalem
CPU	Xeon X5650 @ 2.67GHz
Cores	6
Contexts per core	2
Cache size, sharing	12MB L3, shared
Memory	3x 4GB DDR3
	Sun UltraSPARC T2
CPU	UltraSPARC T2 @ 1.2GHz
Cores	8
Contexts per core	8
Cache size, sharing	4MB L2, shared
Memory	8x 2GB DDR2

Table 2.1: Platform characteristics.

1. **No partitioning join**: An implementation where partitioning is omitted. This implementation creates a shared hash table in the build phase.
2. **Shared partitioning join**: The first non-blocking partitioning algorithm of Section 2.4.1, where all the threads partition both input sources into shared partitions. Synchronization through a latch is necessary before writing to the shared partitions.
3. **Independent partitioning join**: The second non-blocking partitioning algorithm of Section 2.4.1, where all the threads partition both sources and create private partitions.
4. **Radix partitioning join**: An implementation where each input relation is stored in a single, contiguous memory region. Then, each thread participates in the radix partitioning, as described in Section 2.4.1.



(a) Intel Nehalem



(b) Sun UltraSPARC T2

Figure 2.1: Cycles per output tuple for the uniform dataset.

2.5 Experimental evaluation

We have implemented the hash join algorithms described in Section 2.4.4 in a stand-alone C++ program. The program first loads data from the disk into main memory. Data is organized in memory using traditional slotted pages. The join algorithms are run after the data is loaded in memory. Since the focus of this work in on memory-resident datasets, we do not consider the time to load the data into main memory and only report join completion times.

For our workload, we wanted to simulate common and expensive join operations in decision support environments. The execution of a decision support query in a data warehouse typically involves multiple phases. First, one or more dimension relations are reduced based on the selection constraints. Then, these dimension relations are combined into an intermediate one, which is then joined with a much larger fact relation. Finally, aggregate statistics on the join output are computed and returned to the user. For example, in the TPC-H decision support benchmark, this execution pattern is encountered in at least 15 of the 22 queries.

We try to capture the essence of this operation by focusing on the most expensive component, namely the join operation between the intermediate relation R (the outcome of various operations on the dimension relations) with a much larger fact relation S. To allow us to focus on the core join performance, we initially do not consider the cost of materializing the output in memory, adopting a similar method as previous work [27, 54]. In later experiments (see Section 2.5.8), we consider the effect of materializing the join result – in these cases, the join result is created in main memory and not flushed to disk.

We describe the synthetic datasets that we used in the next section (Section 2.5.1). In Section 2.5.2 we give details about the hardware that we used for our experiments. We continue with a presentation of the results in Sections 2.5.3 and 2.5.4. We analyze the results further in Sections 2.5.5

through 2.5.7. We present results investigating the effect of output materialization, and the sensitivity to input sizes and selectivities in Sections 2.5.8 through 2.5.10.

2.5.1 Dataset

We experimented with three different datasets, which we denote as *uniform*, *low skew* and *high skew*, respectively. We assume that the relation R contains the primary key and the relation S contains a foreign key referencing tuples in R. Furthermore, as R contains the primary key, we assume that R can be accessed in sorted order on the primary key. In all the datasets, we fix the cardinalities of R to 16M tuples and S to 256M tuples[1]. We picked the ratio of R to S to be 1:16 to mimic common decision support settings. We experiment with different ratios in Section 2.5.9.

In our experiments both keys and payloads are eight bytes each. Each tuple is simply a <key, payload> pair, so tuples are 16 bytes long. Keys can either be the values themselves, if the key is numeric, or an 8-byte hash of the value in the case of strings. We chose to represent payloads as 8 bytes for two reasons: (a) Given that columnar storage is commonly used in data warehouses, we want to simulate storing <key, value> or <key, record-id> pairs in the hash table, and (b) make comparisons with existing work (i.e. [54, 23]) easier. Exploring alternative ways of constructing hash table entries is not a focus of this work, but has been explored before [70].

For the uniform dataset, we create tuples in the relation S such that each tuple matches every key in the relation R with equal probability. For the skewed datasets, we added skew to the distribution of the foreign keys in the relation S. (Adding skew to the relation R would violate the primary key constraint.) We created two skewed datasets, for two different s values of the Zipf distribution: low skew with $s = 1.05$ and high skew with $s = 1.25$. Intuitively, the most popular key appears in the low skew dataset 8% of

[1]Throughout the chapter, M=2^{20} and K=2^{10}.

the time, and the ten most popular keys account for 24% of the keys. In comparison, in the high skew dataset, the most popular key appears 22% of the time, and the ten most popular keys appear 52% of the time.

In all the experiments, the hash buckets that are created during the build phase have a fixed size: they always have 32 bytes of space for the payload, and 8 bytes are reserved for the pointer that points to the next hash bucket in case of overflow. These numbers were picked so that each bucket fits in a single, last-level cache line for both the architectures. We size the hash table appropriately so that no overflow occurs.

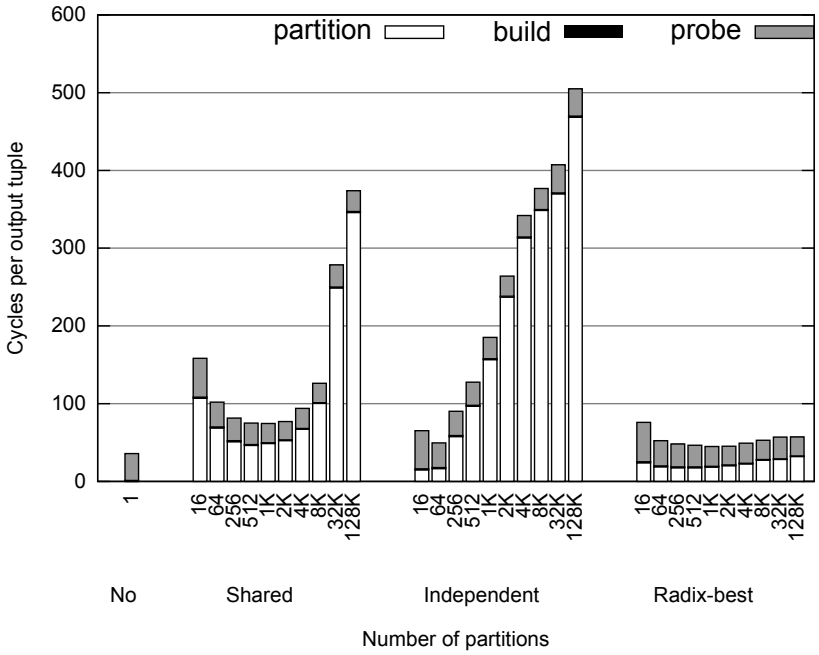
2.5.2 Platforms

We evaluated our methods on two different architectures: the Intel Nehalem and the Sun UltraSPARC T2. We describe the characteristics of each architecture in detail below, and we summarize key parameters in Table 2.1.

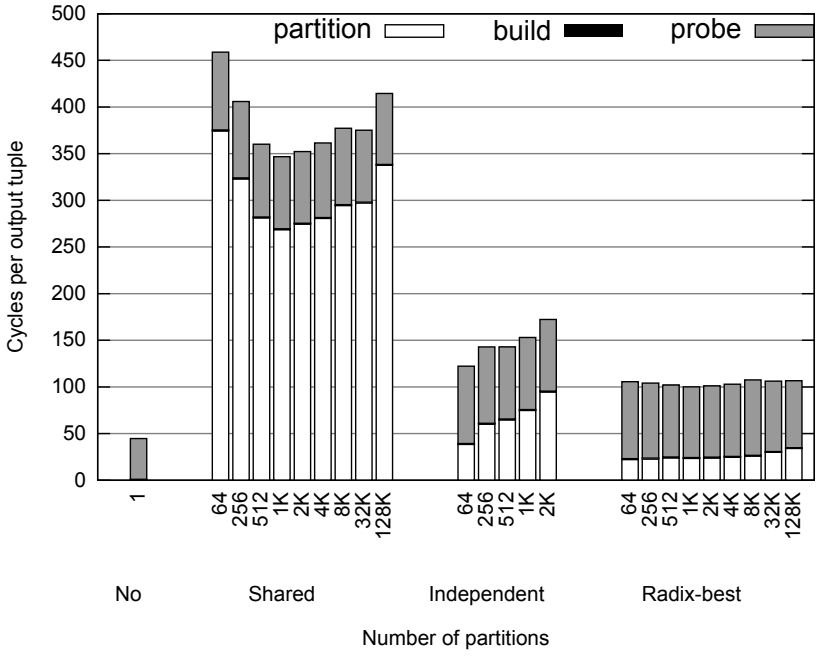
The Intel Nehalem microarchitecture is the successor of the Intel Core microarchitecture. All Nehalem-based CPUs are superscalar processors and exploit instruction-level parallelism by using out-of-order execution. The Nehalem family supports multi-threading, and allows two contexts to execute per core.

For our experiments, we use the six-core Intel Xeon X5650 that was released in Q1 of 2010. This CPU has a unified 12MB, 16-way associative L3 cache with a line size of 64 bytes. This L3 cache is shared by all twelve contexts executing on the six cores. Each core has a private 256KB, 8-way associative L2 cache, with a line size of 64 bytes. Finally, private 32KB instruction and data L1 caches connect to each core's load/store units.

The Sun UltraSPARC T2 was introduced in 2007 and relies heavily on multi-threading to achieve maximum throughput. An UltraSPARC T2 chip has eight cores and each core has hardware support for eight contexts. UltraSPARC T2 does not feature out-of-order execution. Each core has a single instruction fetch unit, a single floating point unit, a single memory



(a) Intel Nehalem



(b) Sun UltraSPARC T2

Figure 2.2: Cycles per output tuple for the low skew dataset.

unit and two arithmetic units. At every cycle, each core executes at most two instructions, each taken from two different contexts. Each context is scheduled in a round-robin fashion every cycle, unless the context has initiated a long-latency operation, such as a memory load that caused a cache miss, and has to wait for the outcome.

At the bottom of the cache hierarchy of the UltraSPARC T2 chip lies a shared 4MB, 16-way associative write-back L2 cache, with a line size of 64 bytes. To maximize throughput, the shared cache is physically split into eight banks. Therefore, up to eight cache requests can be handled concurrently, provided that each request hits a different bank. Each core connects to this shared cache through a non-blocking, pipelined crossbar. Finally, each core has a 8KB, 4-way associative write-through L1 data cache with 16 bytes per cache line that is shared by all the eight hardware contexts. Overall, in the absence of arbitration delays, the L2 cache hit latency is 20 cycles.

2.5.3 Results

We start with the uniform dataset. In Figure 2.1, we plot the average number of CPU cycles that it takes to produce one output tuple, without actually writing the output, for a varying number of partitions. (Note that to convert the CPU cycles to wall clock time, we simply divide the CPU cycles by the corresponding clock rate shown in Table 2.1). The horizontal axis shows the different join algorithms (bars "No", "Shared", "Independent"), corresponding to the first three hash join variants described in Section 2.4.4. For the radix join algorithm, we show the best result across any number of passes (bars marked "Radix-best"). Notice that we assume that the optimizer will always be correct and pick the optimal number of passes.

Overall, the build phase takes a very small fraction of the overall time, regardless of the partitioning strategy that is being used, across all architectures (see Figure 2.1). The reason for this behavior is two-fold. First

and foremost, the smaller cardinality of the R relation translates into less work during the build phase. (We experiment with different cardinality ratios in Section 2.5.9.) Second, building a hash table is a really simple operation: it merely involves copying the input data into the appropriate hash bucket, which incurs a lot less computation than the other steps, such as the output tuple reconstruction that must take place in the probe phase. The performance of the join operation is therefore mostly determined by the time spent partitioning the input relations and probing the hash table.

As can be observed in Figure 2.1(a) for the Intel Nehalem architecture, the performance of the non-partitioned join algorithm is comparable to the optimal performance achieved by the partition-based algorithms. The shared partitioning algorithm performs best when sizing partitions so that they fit in the last level cache. This figure reveals a problem with the independent partitioning algorithm. For a high number of partitions, say 128K, each thread will create its own private buffer, for a total of $128\text{K} * 12 \approx 1.5$ million output buffers. This high number of temporary buffers introduces two problems. First, it results in poor space utilization, as most of these buffers are filled with very few tuples. Second, the working set of the algorithm grows tremendously, and keeping track of 1.5 million cache lines requires a cache whose capacity is orders of magnitude larger than the 12MB L3 cache. The radix partitioning algorithm is not affected by this problem, because it operates in multiple passes and limits the number of partition output buffers in each pass.

Next, we experimented with the Sun UltraSPARC T2 architecture. In Figure 2.1(b) we see that doing no partitioning is at least 1.5X faster compared to all the other algorithms. The limited memory on this machine prevented us from running experiments with a high number of partitions for the independent partitioning algorithm because of the significant memory overhead discussed in the previous paragraph. As this machine supports nearly five times more hardware contexts than the Intel machine, the mem-

ory that is required for bookkeeping is five times higher as well.

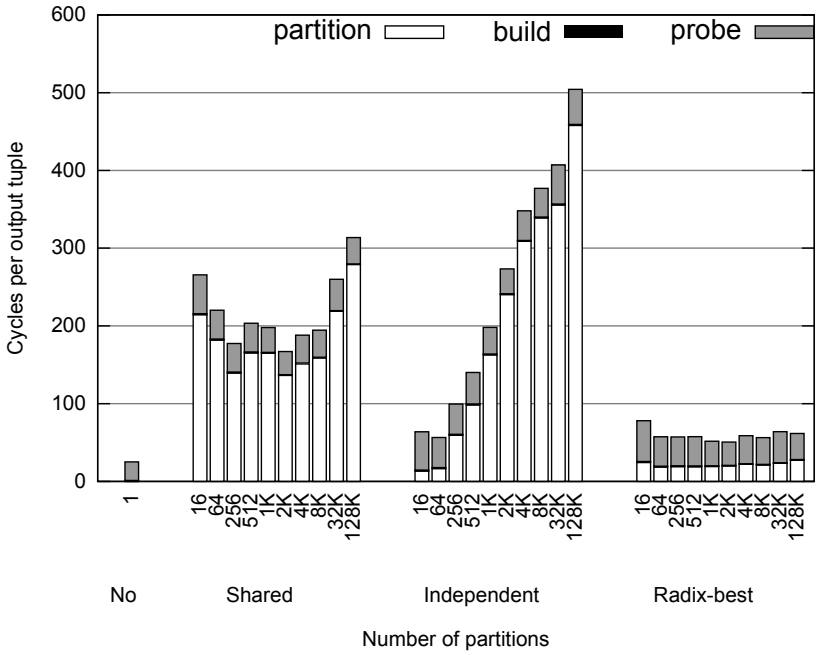
To summarize our results with the uniform dataset, we see that on the Intel architecture the performance of the no partitioning join algorithm is comparable to the performance of all the other algorithms. For the Sun UltraSPARC T2, we see that the no partitioning join algorithm outperforms the other algorithms by at least 1.5X. Additionally, the no partitioning algorithm is more robust, as the performance of the other algorithms degrades if the query optimizer does not pick the optimal value for the number of partitions.

2.5.4 Effect of skew

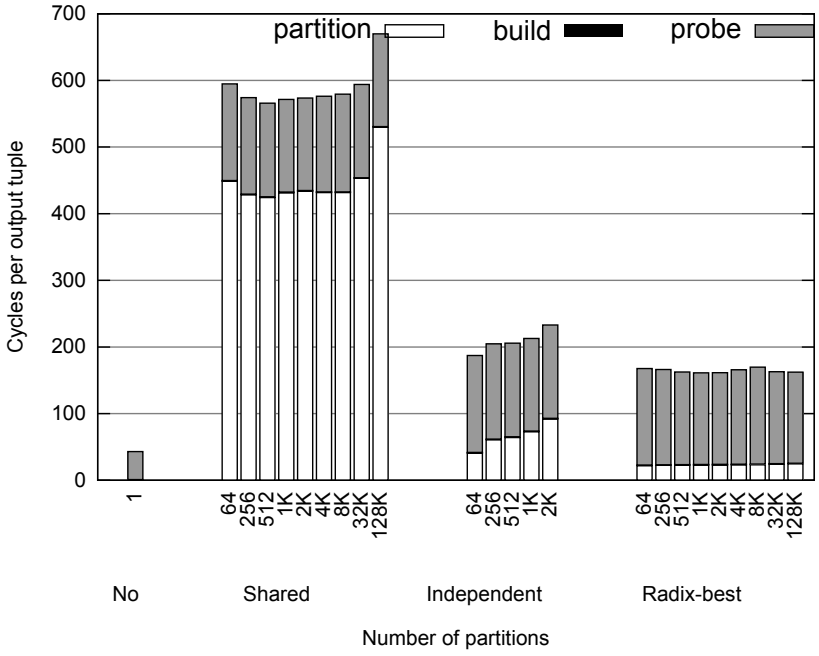
We now consider the case when the distribution of foreign keys in the relation S is skewed. We again plot the average time to produce each tuple of the join (in machine cycles) in Figure 2.2 for the low skew dataset, and in Figure 2.3 for the high skew dataset.

By comparing Figure 2.1 with Figure 2.2, we notice that, when using the shared hash table (bar "No" in all graphs), performance actually improves in the presence of skew! On the other hand, the performance of the shared partitioning algorithm degrades rapidly with increasing skew, while the performance of the independent partitioning and the radix partitioning algorithms shows little change on the Intel Nehalem and degrades on the Sun UltraSPARC T2. Moving to Figure 2.3, we see that the relative performance of the non-partitioned join algorithm increases rapidly under higher skew, compared to the other algorithms. The non-partitioned algorithm is generally 2X faster than the other algorithms on the Intel Nehalem, and more than 4X faster than the other algorithms on the Sun UltraSPARC T2.

To summarize these results, skew in the underlying join key values (data skew) manifests itself as partition size skew when using partitioning. For the shared partitioning algorithm, during the partition phase, skew causes latch contention on the partition with the most popular key(s). For all



(a) Intel Nehalem



(b) Sun UltraSPARC T2

Figure 2.3: Cycles per output tuple for the high skew dataset.

	Intel Nehalem	Sun UltraSPARC T2
NO	No / 1	No / 1
SN	Indep. / 16	Indep. / 64
L2-S	Shared / 2048	Shared / 2048
L2-R	Radix / 2048	Radix / 2048

Table 2.2: Shorthand notation and corresponding partitioning strategy / number of partitions.

partitioning-based algorithms, during the probe phase, skew translates into a skewed work distribution per thread. Therefore, the overall join completion time is determined by the completion time of the partition with the most popular key. (We explore this behavior further in Section 2.5.7.) On the other hand, skew improves performance when sharing the hash table and not doing partitioning for two reasons. First, the no partitioning approach ensures an even work distribution per thread as all the threads are working concurrently on the single partition. This greedy scheduling strategy proves to be effective in hiding data skew. Second, performance increases because the hardware handles skew a lot more efficiently, as skewed memory access patterns cause significantly fewer cache misses.

2.5.5 Performance counters

Due to space constraints, we focus on specific partitioning configurations from this section onward. We use "NO" to denote the no partitioning strategy where the hash table is shared by all threads, and we use "SN" to denote the case when we create as many partitions as hardware contexts (join threads), except we round the number of partitions up to the next power of two as is required for the radix partitioning algorithm. We use "L2" to denote the case when we create partitions to fit in the last level cache, appending "-S" when partitioning with shared output buffers, and "-R" for

		Cycles	L3 miss	Instruc--tions	TLB load miss	TLB store miss
NO	partition	0	0	0	0	0
	build	322	2	2,215	1	0
	probe	15,829	862	54,762	557	0
SN	partition	3,578	18	29,096	6	2
	build	328	8	2,064	0	0
	probe	21,717	866	54,761	505	0
L2-S	partition	11,778	103	31,117	167	257
	build	211	1	2,064	0	0
	probe	6,144	35	54,762	1	0
L2-R	partition	6,343	221	34,241	7	237
	build	210	1	2,064	0	0
	probe	6,152	36	54,761	1	0

Table 2.3: Hardware events for the uniform dataset (millions).

radix partitioning. We summarize this notation in Table 2.2. Notice that the L2 numbers correspond to the best performing configuration settings in the experiment with the uniform dataset (see Figure 2.1).

We now use the hardware performance counters to understand the characteristics of these join algorithms. In the interest of space, we only present our findings from a single architecture: the Intel Nehalem. We first show the results from the uniform dataset in Table 2.3. Each row indicates one particular partitioning algorithm and join phase, and each column shows a different architectural event. First, notice the code path length. It takes, on average, about 55 billion instructions to complete the probe phase and an additional 50% to 65% of that for partitioning, depending on the algorithm of choice. The NO algorithm pays a high cost in terms of the L3 cache misses during the probe phase. The partitioning phase of the SN algorithm is fast but fails to contain the memory reference patterns that arise during the probe phase in the cache. The L2-S algorithm manages to minimize these memory references, but incurs a high L3 and TLB miss ratio during

		Cycles	L3 miss	Instruc--tions	TLB load miss	TLB store miss
NO	partition	0	0	0	0	0
	build	323	3	2,215	1	0
	probe	6,433	98	54,762	201	0
SN	partition	3,577	17	29,096	6	1
	build	329	8	2,064	0	0
	probe	13,241	61	54,761	80	0
L2-S	partition	36,631	79	34,941	67	106
	build	210	5	2,064	0	0
	probe	8,024	13	54,762	1	0
L2-R	partition	5,344	178	34,241	5	72
	build	209	4	2,064	0	0
	probe	8,052	13	54,761	1	0

Table 2.4: Hardware events for the high skew dataset (millions).

the partition phase compared to the NO and SN algorithms. The L2-R algorithm uses multiple passes to partition the input and carefully controls the L3 and TLB misses during these phases. Once the cache-sized partitions have been created, we see that both the L2-S and L2-R algorithms avoid incurring many L3 and TLB misses during the probe phase. In general, we see fewer cache and TLB misses across all algorithms when adding skew (in Table 2.4).

Unfortunately, interpreting performance counters is much more challenging with modern multi-core processors and will likely get worse. Processors have become a lot more complex over the last ten years, yet the events that counters capture have hardly changed. This trend causes a growing gap between the high-level algorithmic insights the user expects and the specific causes that trigger some processor state that the performance counters can capture. In a uniprocessor, for example, a cache miss is an indication that the working set exceeds the cache's capacity. The penalty is bringing the data from memory, an operation the costs hundreds

of cycles. However, in a multi-core processor, a memory load might miss in the cache because the operation touches memory that some other core has just modified. The penalty in this case is looking in some other cache for the data. Although a neighboring cache lookup can be ten or a hundred times faster than bringing the data from memory, both scenarios will simply increment the cache miss counter and not record the cause of this event.

To illustrate this point, let's turn our attention to a case in Table 2.3 where the performance counter results can be misleading: The probe phase of the SN algorithm has slightly fewer L3 and TLB misses than the probe phase of the NO algorithm and equal path length, so the probe phase of the SN algorithm should be comparable or faster than probe phase of the NO algorithm. However, the probe phase of the NO algorithm is almost 25% faster! Another issue is latch contention, which causes neither L3 cache misses nor TLB misses, and therefore is not reported in the performance counters. For example, when comparing the uniform and high skew numbers for the L2-S algorithm, the number of the L3 cache misses during the high skew experiment is 30% lower than the number of the cache misses observed during the uniform experiment. However, partitioning performance worsens by more than 3X when creating shared partitions under high skew!

The performance counters don't provide clean insights into why the non-partitioned algorithm exhibits similar or better performance than the other cache-efficient algorithms across all datasets. Although a cycle breakdown is still feasible at a macroscopic level where the assumption of no contention holds (for example as in Ailamaki et al. [4]), this experiment reveals that blindly assigning fixed cycle penalties to architectural events can lead to misleading conclusions.

2.5.6 Speedup from SMT

Modern processors improve the overall efficiency with hardware multithreading. Simultaneous multi-threading (SMT) permits multiple independent

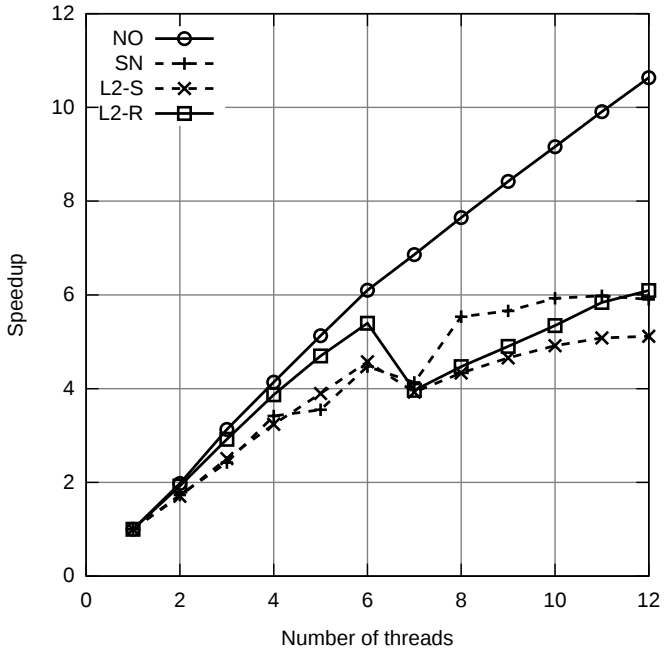


Figure 2.4: Speedup over single threaded execution, uniform dataset.

threads of execution to better utilize the resources provided by modern processor architectures. We now evaluate the impact of SMT on the hash join algorithms.

We first show a speedup experiment for the Intel Nehalem on the uniform dataset in Figure 2.4. We start by dedicating each thread to a core, and once we exceed the number of available physical cores (six for our Intel Nehalem), we then start assigning threads in a round-robin fashion to the available hardware contexts. We observe that the algorithms behave very differently when some cores are idle (fewer than six threads) versus in the SMT region (more than six threads). With fewer than six threads all the algorithms scale linearly, and the NO algorithm has optimal speedup. With more than six threads, the NO algorithm continues to scale, becoming almost 11X faster than the single-threaded version when using all available

	Uniform		
	6 threads	12 threads	Improvement
NO	28.23	16.15	1.75X
SN	34.04	25.62	1.33X
L2-S	19.27	18.13	1.06X
L2-R	14.46	12.71	1.14X
	High skew		
	6 threads	12 threads	Improvement
NO	9.34	6.76	1.38X
SN	19.50	17.15	1.14X
L2-S	38.37	44.87	0.86X
L2-R	15.04	13.61	1.11X

Table 2.5: Simultaneous multi-threading experiment on the Intel Nehalem, showing billions of cycles to join completion and relative improvement.

contexts. The partitioning-based algorithms SN, L2-S and L2-R, however, do not exhibit this behavior. The speedup curve for these three algorithms in the SMT region either flattens completely (SN algorithm), or increases at a reduced rate (L2-R algorithm) than the non-SMT region. In fact, performance drops for all partitioning algorithms for seven threads because of load imbalance: a single core has to do the work for two threads. (This imbalance can be ameliorated through load balancing, a technique that we explore in Section 2.5.7.)

We summarize the benefit of SMT in Table 2.5 for the Intel architecture, and in Table 2.6 for the Sun architecture. For the Intel Nehalem and the uniform dataset, the NO algorithm benefits significantly from SMT, becoming 1.75X faster. This algorithm is not optimized for cache performance, and as seen in Section 2.5.5, causes many cache misses. As a result, it provides more opportunities for SMT to efficiently overlap the memory accesses. On the other hand, the other three algorithms are optimized for cache performance to different degrees. Their computation is a large frac-

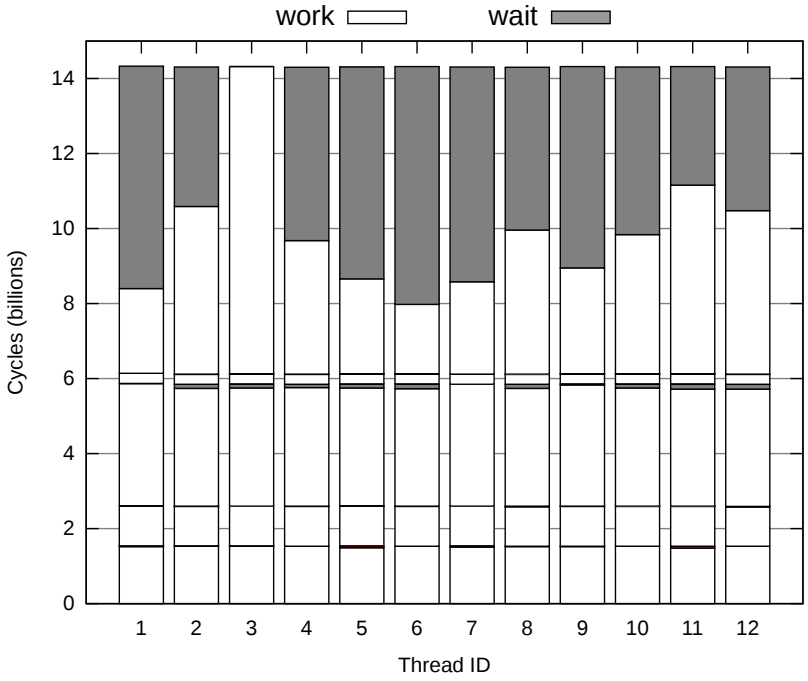
	Uniform		
	8 threads	64 threads	Improvement
NO	37.30	12.64	2.95X
SN	55.70	22.25	2.50X
L2-S	51.62	23.86	2.16X
L2-R	46.62	18.88	2.47X
	High skew		
	8 threads	64 threads	Improvement
NO	23.92	11.67	2.05X
SN	70.52	49.54	1.42X
L2-S	73.91	221.01	0.33X
L2-R	66.01	43.16	1.53X

Table 2.6: Simultaneous multi-threading experiment on the Sun UltraSPARC T2, showing billions of cycles to join completion and relative improvement.

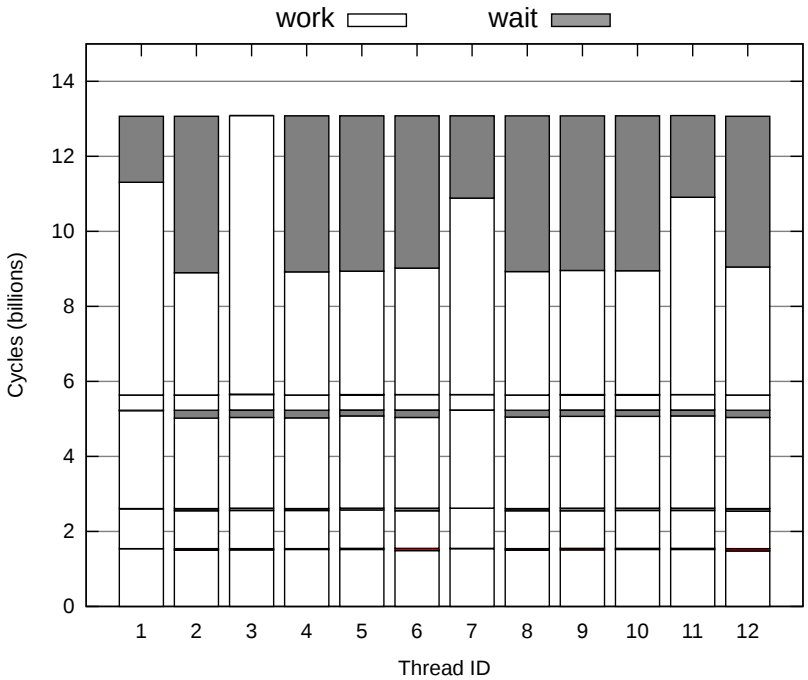
tion of the total execution time, therefore they do not benefit significantly from using SMT. In addition, we notice that the NO algorithm is around 2X slower than the L2-R algorithm without SMT, but its performance increases to almost match the L2-R algorithm performance with SMT.

For the Sun UltraSPARC T2, the NO algorithm also benefits the most from SMT. In this architecture the code path length (i.e. instructions executed) has a direct impact on the join completion time, and therefore the NO algorithm performs best both with and without SMT. As the Sun machine cannot exploit instruction parallelism at all, we see increased benefits from SMT compared to the Intel architecture.

When comparing the high skew dataset with the uniform dataset across both architectures, we see that the improvement of SMT is reduced. The skewed key distribution incurs fewer cache misses, therefore SMT loses opportunities to hide processor pipeline stalls.



(a) High skew dataset



(b) High skew dataset with work stealing

Figure 2.5: Time breakdown of the radix join.

2.5.7 Synchronization

Synchronization is used in multithreaded programs to guarantee the consistency of shared data structures. In our join implementations, we use barrier synchronization when all the threads wait for tasks to be completed before they can proceed to the next task. (For example, at the end of each pass of the radix partition phase, each thread has to wait until all other threads complete before proceeding.) In this section, we study the effect of barrier synchronization on the performance of the hash join algorithm. In the interest of space, we only present results for the Intel Nehalem machine. Since the radix partitioning algorithm wins over the other partitioning algorithms across all datasets, our discussion only focuses on results for the non-partitioned algorithm (NO) and the radix partitioning algorithm (L2-R).

Synchronization has little impact on the non-partitioned (NO) algorithm for both the uniform and the high skew datasets, regardless of the number of threads that are running. The reason for this behavior is the simplicity of the NO algorithm. First, there is no partition phase at all, and each thread can proceed independently in the probe phase. Therefore synchronization is only necessary during the build phase, a phase that takes less than 2% of the total time (see Figure 2.1). Second, by dispensing with partitioning, this algorithm ensures an even distribution of work across the threads, as all the threads are working concurrently on the single shared hash table.

We now turn our attention to the radix partitioning algorithm, and break down the time spent by each thread. Unlike the non-partitioned algorithm, the radix partitioning algorithm is significantly impacted by synchronization on both the uniform and the high skew datasets. Figure 2.5(a) shows the time breakdown for the L2-R algorithm when running 12 threads on the Intel Nehalem machine with the high skew dataset. Each histogram in this figure represents the execution flow of a thread. The vertical axis can be viewed as a time axis (in machine cycles). White rectangles in these

histograms represent tasks, the position of each rectangle indicates the beginning time of the task, and the height represents the completion time of this task for each thread. The gray rectangles represent the waiting time that is incurred by a thread that completes its task but needs to synchronize with the other threads before continuing. In the radix join algorithm, we can see five expensive operations that are synchronized through barriers: (1) computing the thread-private histogram, (2) computing the global histogram, (3) doing radix partitioning, (4) building a hash table for each partition of the relation R, and (5) probing each hash table with a partition from the relation S. The synchronization cost of the radix partitioning algorithm accounts for nearly half of the total join completion time for some threads.

The synchronization cost is so high under skew primarily because it is hard to statically divide work items into equally-sized subtasks. As a result, faster threads have to wait for slower threads. For example, if threads are statically assigned to work on partitions in the probe phase, the distribution of the work assigned to the threads will invariably also be skewed. Thus, the thread processing the partition with the most popular key becomes a bottleneck and the overall completion time is determined by the completion time of the partition with the most popular keys. In Figure 2.5(a), this is thread 3.

Load balancing

If static work allocation is the problem, then how would the radix join algorithm perform under a dynamic work allocation policy and highly skewed input? To answer this question, we tweaked the join algorithm to allow the faster threads that have completed their probe phase to steal work from other slower threads. In our implementation, the unit of work is a single partition. In doing so, we slightly increase the synchronization cost because work queues need to now be protected with latches, but we balance the load

Machine	NO	SN	L2-S	L2-R
Intel Nehalem	23%	4%	7%	10%
Sun UltraSPARC T2	29%	21%	20%	23%

Table 2.7: Overhead of materialization with respect to total cycles without materialization on the uniform dataset.

better.

In Figure 2.5(b) we plot the breakdown of the radix partitioning algorithm (L2-R) using this work stealing policy when running on the Intel Nehalem machine with the high skew dataset. Although the work is now balanced almost perfectly for the smaller partitions, the partitions with the most popular keys are still a bottleneck. In the high skew dataset, the most popular key appears 22% of the time, and thread 3 in this case has been assigned only a single partition which happened to correspond to the most popular key. In comparison, for this particular experiment, the NO algorithm can complete the join in under 7 billion cycles (Table 2.4), and hence is 1.9X faster. An interesting area for future work is load balancing techniques that permit work stealing at a finer granularity than an entire partition with a reasonable synchronization cost.

To summarize, under skew, a load balancing technique improves the performance of the probe phase but does not address the inherent inefficiency of all the partitioning-based algorithms. In essence, there is a coordination cost to be paid for load balancing, as thread synchronization is necessary. Skew in this case causes contention, stressing the cache coherence protocol and increasing memory traffic. On the other hand, the no partitioning algorithm does skewed memory loads of read-only data, which is handled very efficiently by modern CPUs through caching.

	Scale 0.5	Scale 1	Scale 2
NO	7.65 (0.47X)	16.15 (1.00X)	62.27 (3.86X)
SN	11.76 (0.46X)	25.62 (1.00X)	98.82 (3.86X)
L2-S	8.47 (0.47X)	18.13 (1.00X)	68.48 (3.78X)
L2-R	5.82 (0.46X)	12.71 (1.00X)	DNF

Table 2.8: Join sensitivity with varying input cardinalities for the uniform dataset on Intel Nehalem. The table shows the cycles for computing the join (in billions) and the relative difference to scale 1.

2.5.8 Effect of output materialization

Early work in main memory join processing [27] did not take into account the cost of materialization. This decision was justified by pointing out that materialization comes at a fixed price for all algorithms and, therefore, a join algorithm will be faster regardless of the output being materialized or discarded. Recent work by Cieslewicz et al. [22] highlighted the trade-offs involved when materializing the output.

In Table 2.7 we report the increase in the total join completion time when we materialize the output in memory for the uniform dataset and the partitioning strategies described in Table 2.2. If the join operator is part of a complex query plan, it is unlikely that the entire join output will ever need to be written in one big memory block, but, even in this extreme case, we see that no algorithm is being significantly impacted by materialization.

2.5.9 Cardinality experiments

We now explore how sensitive our findings are to variations in the cardinalities of the two input relations. Table 2.8 shows the results when running the join algorithms on the Intel Nehalem machine. The numbers obtained from the uniform dataset (described in detail in Section 2.5.1) are shown in the middle column. We first created one uniform dataset where both rela-

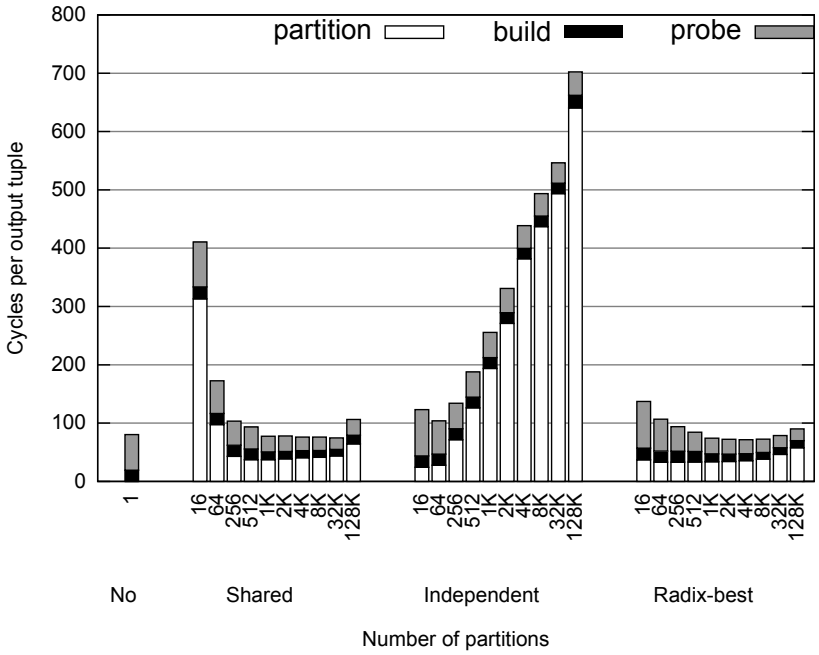


Figure 2.6: Performance on Intel Nehalem with uniform dataset and $|R|=|S|$.

tions are half the size (scale 0.5). This means the relation R has 8M tuples and the relation S has 128M tuples. We also created a uniform dataset where both relations are twice the size (scale 2), i.e. the relation R has 32M tuples and the relation S has 512M tuples. The scale 2 dataset occupies 9GB out of the 12GB of memory our system has (Table 2.1) and leaves little working memory, but the serial access pattern allows performance to degrade gracefully for all algorithms but the L2-R algorithm. The main memory optimizations of the L2-R algorithm cause many random accesses which hurt performance. We therefore mark the L2-R algorithm as not finished (DNF).

We now examine the impact of the relative size of the relations R and S. We fixed the cardinality of the relation S to be 16M tuples, making

$|R| = |S|$, and we plot the cycles per output tuple for the uniform dataset when running on the Intel Nehalem in Figure 2.6. First, the partitioning time increases proportionally to $|R| + |S|$. Second, the build phase becomes significant, taking at least 25% of the total join completion time. The probe phase, however, is at most 30% slower, and less affected by the cardinality of the relation R. Overall, all the algorithms are slower when $|R| = |S|$ because they have to process more data, but the no partitioning algorithm is slightly favored because it avoids partitioning both input relations.

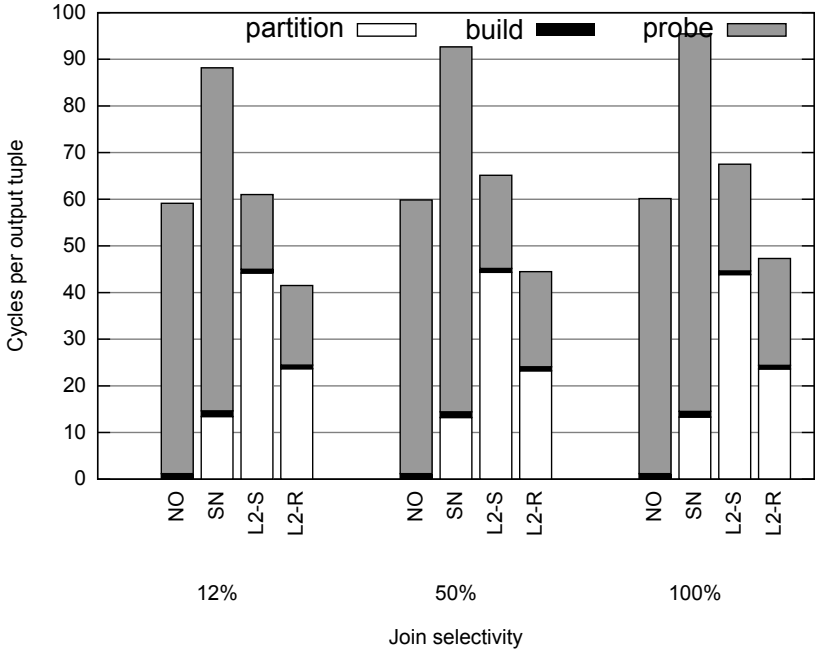
The results show that no join algorithm is particularly sensitive to our selection of input relation cardinalities, therefore our findings are expected to hold across a broader spectrum of cardinalities. The outcome of the experiments for the Sun UltraSPARC T2 is similar, and is omitted.

2.5.10 Selectivity experiment

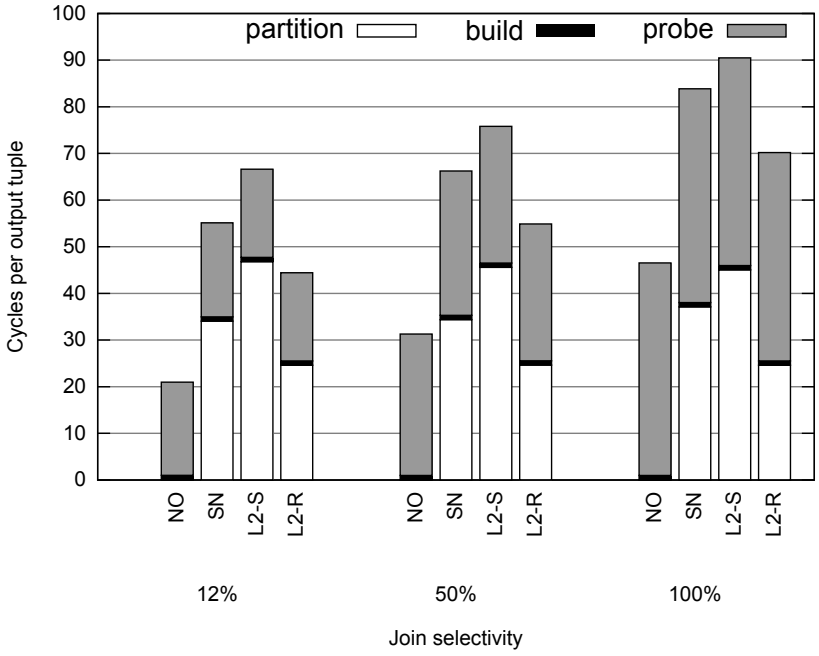
We now turn our attention to how join selectivity affects performance. As all our original datasets are examples of joins between primary and foreign keys, all the experiments that have been presented so far have a selectivity of 100%. For this experiment we created two different S relations that have the same cardinality but only 50% and 12.5% of the tuples join with a tuple in the relation R. The key distribution is uniform.

Results for the Intel Nehalem are shown Figure 2.7(a). Decreasing join selectivity has a marginal benefit on the probe phase, but the other two phases are unaffected. The outcome of the same experiment on Sun UltraSPARC T2 is shown in Figure 2.7(b). In this architecture, the benefit of a small join selectivity on the probe phase is significant.

Inspecting the performance counters in this experiment revealed additional insights. Across all the architectures, the code path length (i.e. instructions executed) increases as join selectivity increases. The Intel Nehalem is practically insensitive to different join selectivities, because its out-of-order execution manages to overlap the data transfer with the byte



(a) Intel Nehalem



(b) Sun UltraSPARC T2

Figure 2.7: Increasing join selectivity impacts the critical path for the Sun UltraSPARC T2, while the out-of-order execution on Intel Nehalem overlaps computation with data transfer.

shuffling that is required to assemble the output tuple. On the other hand, for the Sun UltraSPARC T2 machine, there is a strong linear correlation between the code path length and the cycles that are required for the probe phase to complete. The in-order Sun UltraSPARC T2 cannot automatically extract the instruction-level parallelism of the probe phase, unless the programmer explicitly expresses it by using multiple threads.

2.5.11 Implications

These results imply that DBMSs must reconsider their join algorithms for current and future multi-core processors. First, modern processors are very effective in hiding cache miss latencies through multi-threading (SMT), as it is shown in Tables 2.5 and 2.6. Second, optimizing for cache performance requires partitioning, and this has additional computation and synchronization overheads, and necessitates elaborate load balancing techniques to deal with skew. These costs of partitioning on a modern multi-core machine can be higher than the benefit of an increased cache hit rate, especially on skewed datasets (as shown in Figures 2.2 and 2.3.) To fully leverage the current and future CPUs, high performance main memory designs have to achieve good cache and TLB performance, while fully exploiting SMT, and minimizing synchronization costs.

2.6 Experimenting with a different implementation

Unfortunately, because of three differences in the experimental setup, a direct comparison between our results and [54] is impossible. First, the tuples we use are 16 bytes wide, but only 8 bytes in [54]. Second, most of the experiments show results where the size ratio between two the join inputs is $1/16$, to mimic a primary key–foreign key join in a decision support

environment where there is a small dimension table and larger fact table. In [54], the two inputs have equal size. Finally, although both studies use CPUs based on the Intel Nehalem microarchitecture, the hardware is not the same. Among other differences, the available memory bandwidth per core is the most striking: In the Xeon X5650 CPU used in our experiments, twelve hardware threads share the 6.4GT/s memory bus. In the i7-965 that is used in [54], only eight hardware threads share the memory bus of the same bandwidth. Although the exact effect of reduced memory bandwidth per core on parallel radix hash join performance is unknown, Intel reports that the i7-965 has 1.2X the effective bandwidth per core than the Xeon X5650 [48].

Although the implementation used in [54] was unavailable, the co-authors of the Kim et al. [54] paper at Oracle generously provided us with a different implementation of the parallel main-memory radix-partitioned join. We compared the two hash join implementations, and this report summarizes our findings. In Section 2.6.1, we compare the efficiency of the radix partitioning phase of the two implementations. In Section 2.6.2, we show where the time is spent in our radix-partitioned hash join implementation.

2.6.1 Radix partitioning efficiency

In this section we focus on the efficiency of the partitioning phase. We use the term "Wisconsin–original" to refer to the implementation described here, and the term "Oracle" to refer to the Oracle implementation.

By inspecting the Wisconsin and Oracle implementations, we noticed that the Oracle implementation differs in the handling of index metadata: it assumes that that the tuple is eight bytes, the key is four bytes, and that the key is the first element stored in the tuple (that is, the key is stored at offset 0 from the beginning of the tuple). Furthermore, hashing is always a binary AND on the four byte key, followed by a shift. In comparison, the Wisconsin implementation has separate classes to store index metadata,

Implementation	Partitions					
	64	256	1K	**4K**	16K	64K
Wisconsin–original	21.10	13.15	13.19	**14.10**	16.83	27.10
Wisconsin–hardcoded	7.59	6.80	7.77	**9.86**	13.29	25.49
Oracle	5.32	6.49	9.91	**10.68**	12.19	18.24

Table 2.9: Average time taken to partition a 256M tuple input table, in machine cycles per tuple, where each tuple is 8 bytes wide. We have configured all implementations to partition in one pass, and we highlight the column corresponding to the "radix-best" configuration.

and the partitioning code calls appropriate class methods for operations like attribute retrieval, value hashing and tuple copying. To facilitate direct comparisons, we modified the Wisconsin implementation to perform key retrieval, key hashing and tuple copying inline, using the same values appearing in the Oracle implementation. We use "Wisconsin–hardcoded" to refer to this modification.

For this experiment, we partition a single input table that contains 2^{28} (256M) tuples. Mimicking the experimental setup of [54], each tuple has a four byte key, followed by four bytes of data, for a total of eight bytes per tuple. We report averages over ten runs, where each run partitions random input that is read directly from `/dev/urandom`. We have configured all implementations to always partition in one pass after we experimentally verified that one-pass partitioning is the optimal setting for this range of output partitions, for all implementations.

Table 2.9 shows the average time to partition, in machine cycles per tuple, for each implementation, for a different number of partitions. We highlight the column that corresponds to the "radix-best" setting we used in Section 2.5 for a dataset of the same size.

From this experiment, we conclude that the Oracle and Wisconsin–hardcoded implementations are equally optimized, as they have comparable

performance between 256 and 16,384 partitions, after we account for experimental noise. Furthermore, we conclude that the partitioning performance numbers in Section 2.5 could be improved by an average of 4-5 cycles per tuple, if we hardcoded the data access operations for the particular dataset.

2.6.2 Overall join efficiency

In this section we turn our attention to the overall join efficiency of our implementation, when compared with the previously published numbers [54].

As the Intel implementation is unavailable, one question is how would the performance numbers published in 2009 look on newer hardware. (The CPU we use for evaluation in Section 2.5 became publicly available in 2010.) The main problem is that both CPUs have the same memory bus, but the Xeon CPU has 1.5X more cores. If the entire radix hash join algorithm was memory bound, the bottleneck would be on the memory bus, and we would expect to see no performance improvement from adding more cores. On the other hand, if the entire radix hash join algorithm was computation bound (and 1.5X more cores didn't saturate the memory bus), we would expect to see up to 1.5X improvement from adding 1.5X cores. Unfortunately, the radix hash join exhibits both behaviors, as some steps are memory bound and some others are computation bound [54].

Although predicting actual performance is impossible, we will now attempt estimate what the upper bound on performance would be. Let's disregard the different memory bandwidth available per core, and let's optimistically assume that performance scales linearly with the number of cores. We therefore assume that the (unavailable) Intel implementation would be 1.5X faster when upgrading from the 4-core i7 used in [54] to the 6-core Xeon used in Section 2.5. Looking at the figures of [54] closely, we see that for 8 byte tuples where the input tables have equal size the join finishes at 30-35 cycles per tuple. As the i7 CPU has four cores, this

becomes 120-140 cycles per tuple per core. We therefore assume that, if the Intel implementation ran on the Xeon X5650 CPU, the upper bound on performance would be 130 cycles per tuple per core.

We run an experiment where the ratio between the two join inputs is 1:1. Each input table has 2^{24} (16M) tuples, and all tuples are 8-byte wide. Results are shown in Table 2.10, expressed in cycles per tuple per core. The left column corresponds to the experiment shown in Fig. 2.6, if ran on a dataset that has 8-byte tuples. The right column discounts the overhead of staging the join input data contiguously in memory (a prerequisite of the radix partitioning implementation that no storage engine that we know of is capable of producing), as well as the overhead of supporting generic, user-defined data operations like typed value comparisons, user-defined key hashing, and arbitrary projection expressions. We believe that the numbers published in [54] do not account for these overheads.

As the results show, hardcoding the data operations can save more than 100 cycles per tuple per core. Similar optimizations could be applied to the hash table code, however we have not endeavored to tune it further, as this is an orthogonal optimization that will improve performance across all algorithms we compare against. Finally, we make no effort to exploit data-level parallelism through SIMD optimizations — these two factors could close the 75 cycle per tuple per core performance gap between the two implementations even further.

In conclusion, there exist implementation differences between the two code bases, but at the level of the common core components, the differences are small. The code that we described in Section 2.4 is part of a much larger data processing engine, so it is by necessity more generic (for example, we can handle different schemas, and we support data types other than integers), although it is by no means complete (for example, we are missing code to evaluate arbitrarily complex expressions at runtime). This report demonstrates that optimizations to selected portions of a hash join

Phase	Action	Cycles per tuple per core		
		As measured in Section 2.5	Hardcoding data operations	As reported in [54]
Partition	Data staging	32.72	—	90–95
	Core partitioning	131.75	97.43	
Build	Project on build tuple	3.87	—	35–40
	Hash table insert	64.26	64.26	
Probe	Hash table cursor read	47.10	47.10	
	Assemble output tuple	58.35	—	
	Total	338.05	208.79	$\sim$ 130

Table 2.10: Cycle breakdown during radix partitioned hash join.

algorithm, and a performance comparison based on these portions, can only shed light on a part of the picture. Per Amdahl's law, the results should be put in the context of the entire machinery that is needed in a real data processing engine to gauge the overall impact.

2.7 Concluding remarks

The rapidly evolving multi-core landscape requires that DBMSs carefully consider the interactions between query processing algorithms and the underlying hardware. In this chapter we examine these interactions when executing a hash join operation in a main memory DBMS. We implement a family of main memory hash join algorithms that vary in the way that they implement the partition, build, and probe phases of a canonical hash join algorithm.

We also evaluate our algorithms on two different multi-core processor architectures. Our results show that a simple hash join technique that does not do any partitioning of the input relations often outperforms the other more complex partitioning-based join alternatives. In addition, the relative performance of this simple hash join technique rapidly improves with increasing skew, and it outperforms every other algorithm in the presence of even small amounts of skew. Performance increases with data skew when the hash table is shared because skewed memory access patterns cause significantly fewer cache misses.

A promising area for future work is to investigate ways to lessen the impact of TLB misses on performance for workloads with large memory footprints. Most processors available at the time the experimental analysis was performed featured "large" page support where the size of each individual page was a few megabytes, and the total TLB-addressable virtual space size was still a tiny fraction of the hunderds of gigabytes of main memory found in a typical server system. Newer processors allow a dramatically bigger

fraction of memory to be directly translated by the TLB. For example, Intel has announced that the Xeon Sandy Bridge-EP generation of processors will feature four dedicated one gigabyte TLB entries, allowing for at least four GBs of memory to be directly translated by the TLB. Looking further ahead, techniques such as direct segment addressing have been proposed [11], which permit accesses to certain application-defined memory regions to completely bypass the TLB. Adopting such architectural advances will improve the performace of "big memory" workloads by significantly reducing the resources used to translate virtual to physical memory addresses.

To summarize, minimizing cache misses via partitioning requires additional computation, synchronization and load balancing to cope with skew. As our experiments show, these costs can be higher than the benefit of an increased cache hit rate. It is suprpising that the naive non-partitioned hash join that is oblivious to the cache size and requires no synchronization or explicit load balancing is often tied with or outperforms traditional highly-optimized cache-efficient hash join methods.

Chapter 3

Equi-join algorithms for memory-resident data

There is renewed interest in ad hoc equijoin algorithms for main memory databases. As with traditional disk-based databases, recent studies have focused on hash-based versus sort-based methods, and some appear to pronounce the death of hash-based join algorithms for main memory databases. This chapter systematically evaluates hash-based and sort-based methods in a multi-socket, multi-core main memory environment. Our results find that, far from being passé, the hash join performs very well in main memory environments, though sort-based algorithms also have a role to play. Main memory DBMSs that aim for high efficiency should implement both methods.

Our work also carefully considers the memory footprint that is needed to run each join method, a practical concern when building actual main memory DBMSs. Finally, we also consider the impact of various input physical properties and the physical properties of the output of each join method, as these are important aspects to consider when using join algorithms as building blocks in complex query processing pipelines.

3.1 Introduction

Main memory database management systems (DBMSs) are becoming a crucial component of the big data ecosystem. Their rise is driven largely by analytical applications that demand real-time results. To provide that level of performance, the DBMS can't afford to access data from a traditional I/O subsystem at query execution time. In parallel, DRAM prices have continued to drop rapidly, while memory densities have continued to increase rapidly following the beat set by Moore's Law. The end result is that it is now possible, and economical, to build systems that keep the entire database resident in main memory. In fact, many database vendors now offer such main memory database appliances, like SAP HANA [31] and Oracle Exalytics [65], or have incorporated main-memory technology, such as IBM Blink [10], to accelerate existing products.

While memory densities have been increasing, at the other end of the motherboard, another big change has been underfoot. Driven by power and heat dissipation considerations, about seven years ago, processors started moving from single core to multiple cores (i.e. multicore). The processing component has moved even further now into multi-sockets, and today multi-socket configurations are common in database servers (with each socket having multiple cores).

This chapter focuses on the workhorse operation of an analytical query processing engine, namely the equijoin operation. As is readily recognized, an equijoin operation is a common and expensive operation in analytics applications, and efficient join algorithms are critical in determining the overall performance of a main-memory DBMS.

Naturally, there has been a lot of recent interest in designing and evaluating main memory equijoin algorithms. The current literature, however, provides a confusing set of answers about the relative merits of different join algorithms. For example, we showed in Chapter 2 showed how the hash join can be adapted to single-socket multicore main memory config-

urations. Then, last year Albutiu et al. [5] showed that on multi-socket configurations, a new sort-merge join algorithm (called MPSM) far outperformed the hash join algorithm proposed in Chapter 2. All of these studies are set against the backdrop of an earlier study by Kim et al. [54] that described how sort-based algorithms are likely to outperform hash-based algorithms in future processors. At the same time, the study in [5] did not fully consider hash-based algorithms that are multi-socket friendly as they simply used the hash algorithm designed in Chapter 2, which was only designed for a single socket system. Thus, there is no clear answer regarding what equijoin algorithm works well, and under what circumstances, in main memory, multi-socket, multicore environments. We address this gap in the current research herein.

We systematically study hash-based and sort-based join algorithms in a multi-socket main memory setting. We assume the common Non-Uniform Memory Architecture (NUMA) model for memory accesses, i.e. memory is locally connected to each socket, but can be globally accessed from any socket, though "remote" accesses are more expensive than "local" accesses. We build on previous work, and adapt the hash join algorithm from [9] for a multi-socket setting. We also consider the recently proposed massively parallel sort-merge (MPSM) join algorithm [5], in addition to two more traditional sort merge join variants. We then identify two common physical properties of the input data, that greatly impact the overall performance of each algorithm. The two physical properties are (1) input being partitioned on the join key, and (2) input being pre-sorted on the join key. We also briefly consider how physical properties of the input affect the physical properties of the output of each join operator, as that potentially impacts which join algorithm to use when evaluating queries with multiple joins.

Finally, previous work in this space has largely ignored the issue of the working memory space that is needed to perform the join operation. With large inputs, and the need to keep all the data in main memory,

the memory footprint of the join algorithm becomes an important practical consideration. In this chapter we precisely characterize the memory demand of all the join algorithms that we study.

Our results highlight that it is too early to dismiss the hash join as a viable option for ad hoc equijoin processing in main memory DBMSs. In fact, in many cases the hash join algorithm far outperforms the sort-based methods. We also find that sort-based methods become competitive only when one of the join input is pre-sorted. In addition, the memory footprint of the hash join algorithm is generally smaller than that of the sort-based methods. Thus, when evaluating complex queries, optimizers may still favor the hash join algorithm in some cases. Our study concludes that main memory DBMSs should strongly consider implementing both hash-based and sort-based algorithms, and they must also pay attention to the memory footprint of the join algorithm. For more complex queries, it is necessary to take into account the physical properties input and the output of each join method to achieve the best performance.

The key contributions are as follows: First, we compare this hash join algorithm with three flavors of sort-based join algorithms, and conduct a comprehensive study of these join methods. Our results show that the hash join algorithm is competitive in many cases, and thus should not be immediately dismissed when building modern main memory DBMSs. Second, we carefully study the impact of the physical properties of the input and output of the join algorithms, and characterize each algorithm based on this dimension. This information is important when building query optimizers for main memory DBMSs. Finally, we also characterize the memory footprint that is required to run each join algorithm, as this is also an important practical consideration, which has been generally overlooked in previous work.

The remainder of this chapter is organized as follows. Section 3.2 describes related work in this area. We introduce the join algorithms in Sec-

tion 3.3, and provide implementation details in Section 3.4. We continue with the experimental evaluation in Section 3.5, and Section 3.6 contains concluding remarks.

3.2 Recent related work

There has been a lot of work on analyzing and comparing the performance of sort-based and hash-based join methods in the broader context of a relational query execution engine. An early performance model was described by DeWitt et al. [27], and Graefe introduced a widely-used pull-based model for encapsulating parallelism with the Volcano system [36]. Graefe [37] has also explored the strengths and weaknesses of sort-based or hash-based query execution plans in a disk-based relational database management system.

More recently, Harizopoulos et al. [43] explore how one can exploit sharing opportunities among multiple queries in the context of a disk-based system. Arumugam et al. [8] experiment with the DataPath engine, which is built around a push-based model. Giannikis et al. [34] built the SharedDB system and demonstrate that sharing opportunities can be exploited for performance in non-OLAP workloads that also do updates. Finally, Neumann [63] proposes leveraging the compiler to compile entire queries into a highly optimized operator to improve the performance of a main-memory engine.

3.3 Join algorithms

In this section, we describe key main memory join algorithms that have been recently proposed. We first briefly present each algorithm, and then describe how each algorithm fares with respect to the following three factors: (1) memory traffic and access pattern, (2) computation needed per tuple, and

(3) working memory size requirements. Paying attention to the first factor, memory accesses is important because the CPU may stall while waiting for a particular datum to arrive. Such stalls have been shown to have a significant impact on performance [4]. As modern CPUs pack more than ten cores per die, the available memory bandwidth per hardware thread has been shrinking, which means that access to memory is becoming relatively more expensive than in the past. The second factor, computation per tuple, needs to be low enough to ensure that the system processes data at a rate close to the memory transfer speed and does not become CPU-bound. The last factor, working memory size, is important because memory is a precious storage medium, and using it judiciously can allow a main memory DBMS to keep a bigger database in RAM.

We describe each algorithm assuming that the query engine is built using a parallel, operator-centric, pull-based model [36]. In this model, a query is a tree of operators, and T worker threads are available to execute the query. The leaves of the tree read data from the base tables or indexes directly. To execute a query, each operator recursively requests ("pulls") data from one or more children, processes them, and writes the results in an output buffer local to each worker thread. The output buffer is then returned to the parent operator. Multiple worker threads may concurrently execute (part of) the same operator.

When the algorithm operates on a single source, such as the partitioning and sorting algorithms we describe below, we use R to denote the entire input, and $|R|$ to denote the cardinality (number of tuples) of the input. When the algorithm needs two inputs, such as the join algorithms, we use R and S to refer to each input, and we assume that $|R| \leq |S|$. In the description of each join algorithm that follows, we focus on equi-joins: an output tuple is produced only if the join key in R is equal to the join key in S.

3.3.1 Partitioning

The partitioning operation takes as inputs a table R and a partitioning function $p(\cdot)$, such that $p(r)$ is an integer in $[1, P]$ for all tuples $r \in R$. R is partitioned on the join key if the partitioning function $p(\cdot)$ ensures that if two tuples $r_1, r_2 \in R$ have equal join keys, then $p(r_1) = p(r_2)$. The output of the partitioning operation is P partitions of R. We use R_i, where $i \in [1, P]$, to refer to the partition whose tuples satisfy the property $p(r) = i$, for $r \in R$.

Partitioning plays a key role in intra-operator parallelism in various DBMS settings, since once the input is partitioned, different worker threads can operate on different partitions [23, 28, 68] in parallel. The partitioning algorithm described here is the parallel radix partitioning described by Kim et al. [54], which is in turn based on the original radix partitioning algorithm by Boncz et al. [16]. The main idea behind this parallel partitioning algorithm is that one can eliminate expensive latch-based synchronization during repartitioning, if all the threads know how many tuples will be stored in each partition in advance.

This partitioning algorithm works as follows. Let T be the number of threads that participate in the repartitioning. The algorithm starts by having each thread i, where $i \in [1, T]$, construct an empty histogram H_i with P bins. We use $H_i(t)$ to refer to bin t of the histogram H_i, where $t \in [1, P]$. Every thread then reads a tuple r from the input table R, increments the count for the bin $H_i\big(p(r)\big)$, and stores the tuple r in a (NUMA) local buffer[1]. This process continues until all the tuples in R have been processed. Now the size of the output partition R_t, where $t \in [1, P]$, can be computed as $\sum_i H_i(t)$. In addition, thread i can write its output at location $L_i(t) = \sum_{k \in [1,i)} H_k(t)$ inside this partition R_t, without

[1] In general, it is faster to buffer the input locally than it is to re-evaluate the subquery that produces it. The buffering step can be eliminated in the special case where the input is in a materialized table.

interfering with any other thread. Finally, the actual repartitioning step takes place, and thread i reads each tuple r from its buffer, computes the output partition number $p(r)$, and writes r to the output location $L_i\big(p(r)\big)$ in the output partition $R_{p(r)}$. The local buffer can be deallocated once each thread has finished processing the R tuples.

When it comes to memory traffic, this partitioning algorithm first scans R twice, once from the input and once from each thread's local buffer, for a total of $2 \times |R|$ memory read operations and $|R|$ memory write operations. These reads and writes are sequential and to the local NUMA memory, making this phase extremely efficient. Producing the output partitions is a more costly operation as the writes may target remote NUMA memory. During repartitioning, the algorithm needs to keep track of P output locations, and as the number of output partitions P increases, one may observe a significant number of TLB misses on memory writes. The TLB pressure can be reduced by partitioning in multiple passes [16]. Overall, if partitioning is carried out in one pass, this partitioning algorithm incurs $2 \times |R|$ memory reads and $2 \times |R|$ memory writes.

Partitioning involves minimal computation, as the algorithm only needs to evaluate $p(\cdot)$ twice, once to create the histogram and once during repartitioning, for a total of $2 \times |R|$ evaluations. In general, evaluating the partitioning function takes only a few cycles, which is a small fraction of the typical memory access latency (hundreds of cycles). The evaluation therefore generally occurs in parallel with the data transfer.

The memory demand during the partitioning algorithm consists of $2 \times |R|$ tuples for the input and output buffers, and an array of $P \times T$ integers for the histogram, where P is the number of partitions and T is the number of threads. Once the repartitioning operation is complete, the input buffers and the histogram are deallocated, bringing the total space requirement down to $|R|$ tuples.

3.3.2 Hash join

The algorithm described here is the main-memory hash join proposed in Chapter 2, adapted to ensure that the hash table is striped across the local memory of all threads that participate in the join [9]. But, exactly as the original algorithm, it is otherwise oblivious to any NUMA effects.

The algorithm has a build phase and a probe phase. At the start of the build phase, all the threads allocate memory locally. The union of these memory fragments constitutes a single hash table that is shared by all the threads, where logically adjacent hash buckets are physically located in different NUMA nodes. (Tuples in a particular chain always reside on the same NUMA node.) Thread i then reads a tuple $r \in R$ and hashes on the join key of r using a pre-defined hash function $h(\cdot)$. It then acquires a latch on bucket $h(r)$, copies r in this bucket, and releases the latch. The memory writes required for this operation may either target local or remote NUMA memory. The build phase is completed when all the R tuples have been stored in the hash table.

During the probe phase, each thread reads a tuple $s \in S$, and hashes on the join key of s using the same function $h(\cdot)$. For each $r_{h(s)}$ tuple in the hash bucket $h(s)$, the join keys of $r_{h(s)}$ and s are compared and the tuples are joined together, if the join keys match. These memory reads may either target local or remote memory depending on the physical location of the hash bucket $h(s)$. Because the hash table is only read during the probe phase, there is no need to acquire latches. When all the S tuples have been processed, the probe phase is complete and the memory holding the hash table can be reclaimed. If the R tuples processed by a thread are pre-sorted or pre-partitioned on the join key, neither property will be reflected in the output. However, if the S tuples that are processed by a thread are pre-sorted or pre-partitioned on any S key, the tuples produced by this thread will also be sorted or partitioned on the same key.

The hash join algorithm exhibits a memory access pattern that is hard to

predict, and represents a demanding workload for the memory subsystem. Building a hash table requires writing $|R|$ tuples at the locations pointed to by the hash function. A good hash function would pick any bucket with the same probability for each distinct join key, causing these writes to be randomly scattered across the hash table buckets. This random access pattern leaves little room for performance improvements from the hardware, such as prefetching, and might cause expensive remote write traffic if the hash bucket of interest resides in a remote NUMA node. Similarly, probing the hash table causes $|S|$ reads, and these reads are randomly distributed across all the hash buckets, causing remote memory reads from other NUMA nodes. Each probe lookup reads the entire chain of tuples associated with a hash bucket, but, in the absence of skew, the chain length is low in correctly-sized hash tables [35]. Overall, the hash join algorithm performs $|R|$ writes and $|S|$ reads with a random memory access pattern, potentially to remote NUMA nodes.

The computational overhead of the hash join algorithm is mainly the cost of evaluating the hash function and comparing the join keys. The algorithm needs to compute the hash function for $|R|$ tuples in the build phase, and $|S|$ tuples in the probe phase, for a total of $|R| + |S|$ hash function evaluations. The number of cycles spent on computing the hash value per tuple depends on the chosen hash function and the size of the join key. In general, this number ranges from one or two cycles for hash functions based on bit masking and shifting, to several cycles per byte for hash functions that involve one multiplication per byte of input, like FNV-1a [32]. Finally, the join keys need to be compared to confirm whether the values are indeed equal or there has been a hash collision.

When it comes to memory footprint, the hash join algorithm needs to buffer all the tuples in the build table R. Additionally, each hash bucket needs to store metadata. For example, in our prototype implementation (which is described in detail in Section 3.4.2) each bucket needs to store

a latch for synchronization, a counter for how many bytes in this bucket are unused, and a pointer to another bucket in case of overflow. In order to exhibit good probe performance, the hash chains must be kept short. Therefore, hash tables are commonly sized such that the number of buckets is within the same order of magnitude as the cardinality of the build side R. As a consequence, the memory needed to store bucket metadata is significant, and can even be greater than the size of R if the size of each R tuple is small.

3.3.3 In-place sorting

Work on sorting algorithms can be traced back to the earliest days of computing, and textbooks on data structures, databases and algorithms commonly have a chapter dedicated to sorting. Graefe has written a survey [38] of sorting techniques used by commercial DBMSs, and fast parallel sort algorithms for main memory DBMSs is an active research area [54, 71].

We ultimately decided to use introspective sort [62], which is a popular in-place sorting algorithm, for three reasons. First, many highly-optimized sorting algorithms make strong assumptions about the width of the tuple and the physical placement of the data. In general, such algorithms use very small tuple sizes (for example, only 8 bytes in [54]), and expect tuples to be contiguously stored in memory, regardless of the size of the input. In the context of a query execution engine that stores tuples of a user-defined size and manages its own memory, the performance of these specialized algorithms, compared to their generic counterparts, remains an open question. The second reason for choosing introspection sort is that it is an established, well-studied algorithm, and is implemented in many widely-used and heavily optimized libraries, like the C++ standard template library. Our findings are therefore less likely to be affected by suboptimal configuration settings or poorly optimized code. Finally, introspection sort has been used in recently published results, for example it is the last-phase sorting algorithm

in Albutiu et al. [5]. Using the same algorithm promotes uniformity with prior work.

Introspective sort is a hybrid algorithm that starts with a sorting algorithm with a good average-case performance, quicksort, and then switches to an algorithm with better worst-case performance, heapsort, if it detects that quicksort is likely to exhibit worst-case performance when sorting a particular partition. Introspective sort bases this decision on the recursion depth, and the size of the data to be sorted.

We have also experimented with a sort-merge algorithm that implements a bitonic merge network using SIMD instructions to exploit data-level parallelism [54]. We show how query response time is affected by the input tuple size and the choice of the sorting algorithm in Appendix A.

Sorting is an expensive operation, with respect to memory traffic, as the introspective sorting algorithm reads and writes $\Theta(|R|log|R|)$ tuples, in the average case. The memory access pattern is random, but all operations are performed in a buffer that is local to each processing thread, so all memory operations target the local NUMA memory. The algorithm sorts in-place, therefore the memory space needed is $|R|$ tuples.

3.3.4 Streaming merge join (STRSM)

This algorithm is a parallel version of the standard merge join algorithm [27]. It is assumed that both R and S are already partitioned on the join key with some partitioning function that has produced as many partitions as the number of threads that participate in the join. If T is the number of threads, we use R_i to refer to the partition of R that will be processed by thread $i \in [1, T]$, and similarly we use S_i to denote the partition of S that will be processed by the same thread i. Furthermore, this algorithm requires that both R_i and S_i are already sorted on the join key.

Each thread i reads the first tuple r from R_i and the first tuple s from S_i. If the join keys match, then an output tuple is produced. If the r tuple

has a smaller join key value than the s tuple, then the r tuple is discarded and the next tuple from R_i is read. Otherwise, the s tuple is discarded and the next S_i tuple is read. This process is repeated until either R_i or S_i are depleted. The output of each thread is sorted and partitioned on the join key.

The streaming merge join algorithm is extremely efficient, if the input is already sorted and partitioned. When it comes to memory operations, the algorithm reads the input once and produces results on-the-fly. This is the minimum number of memory operations needed, assuming that the join algorithm needs to read the input to produce correct results. The read access pattern is sequential within each R_i and S_i input for each thread, a pattern which is easily optimized with data prefetching by the hardware. The computational overhead is minimal as it only entails join key comparisons. Finally, the memory requirements are also negligible: The memory manager needs to provide sufficient memory to each thread i to buffer the most commonly occurring join key in either R_i or S_i. Across all the T threads, this can be as little space to hold one tuple per thread, for a total of T tuples of buffer space, or at most $min(|R|, |S|)$ tuples of buffer space for the degenerate case where the join is equivalent to the cartesian product.

3.3.5 Range-partitioned MPSM merge join

This algorithm is the merge phase of the range-partitioned P-MPSM algorithm [5]. The algorithm requires R to be partitioned in T partitions on the join key. We keep the notation introduced in Section 3.3.4, and use R_i to refer to the partition of R that will be processed by thread i, where $i \in [1, T]$. The algorithm requires that each R_i partition has already been sorted on the join key.

Each thread i first allocates a private buffer B_i in NUMA-local memory, and starts copying tuples from S into B_i. (B_i is not a partition of S as two tuples with the same key might be processed by different threads, and

thus be stored in different buffers.) When S is depleted, each thread i sorts its B_i buffer in-place, using the algorithm described in Section 3.3.3. Let B_i^j be the region in the B_i buffer that corresponds to the tuples that join with tuples in partition R_j. Thread i then computes the offsets of every B_i^j region in the B_i buffer, for each $j \in [1, T]$. Because B_i is sorted on the join key, one can compute the partition boundary offsets efficiently using either interpolation or binary search. Once all the threads have computed these offsets, thread i proceeds to merge R_i and B_1^i using the streaming merge join algorithm described in Section 3.3.4. Thread i continues with merging R_i with B_2^i, then R_i with B_3^i, and so on, until R_i has been fully merged with B_T^i. The memory for the B_i buffer can only be reclaimed by the memory manager after all the threads have completed. The output of the MPSM join is partitioned on the join key with the same partitioning function as R per thread. If S is range-partitioned on the join key, even with a different partitioning function than that for R, each thread i can produce output that is sorted on the join key by processing each B_j^i fragment in order.

The MPSM algorithm can cause a lot of memory traffic. The number of tuples that MPSM reads grows linearly with the number of threads participating in the join; this algorithm reads a total of $|R| \times T + |S|$ tuples, where T is the number of threads. In contrast, both the hash join and streaming merge join algorithms, presented in Sections 3.3.2 and 3.3.4 respectively, perform a constant number of reads, regardless of the number of participating threads. On the positive side, the access pattern of MPSM is sequential and only two inputs are merged at any given time per thread. This memory access pattern is highly amenable to hardware prefetching. In addition, the algorithm is designed so that all the T scans of each R_i partition are local to thread i, which minimizes the traffic to different NUMA nodes. Because of these two properties, we have observed that this algorithm achieves very high read rates, as discussed further in Section 3.5.2.

The main computational overhead of the MPSM join is the extra $T - 1$

join key comparisons that are performed on the join key of each R_i tuple per thread, when scanning R_i multiple times. This comparison costs a few cycles per tuple, but it is negligible when compared to the transfer cost of reading R from local memory T times. In addition, the key comparison has little impact on response time as it is generally overlapped with the memory access, an operation that typically takes hundreds of cycles per tuple.

Regarding memory footprint, the MPSM join needs to store the entire S input into the B_i buffers, so the minimum memory capacity requirement is $|S|$ tuples. Although it is not necessary for correctness, we have observed that buffering R significantly improves performance: As the join is commonly an operator that is closer to the root than the leaves of a query tree, it is preferable to execute the subquery that produces R once, buffer the output and reuse this buffer $T-1$ times, than it is to execute the entire R subquery T times. Overall, the MPSM join needs $|R|+|S|$ tuples of space to perform well.

3.3.6 Parallel merge join (PARSM)

This algorithm is a variation of the MPSM join algorithm described above. Instead of scanning the R input T times, this algorithm scans R once and performs a merge of $T+1$ inputs on the fly. Compared to the original MPSM merge join algorithm, this variant reduces the volume of memory traffic and always produces output sorted on the join key, at the cost of a non-sequential access pattern when reading S tuples. As before, the algorithm assumes R has already been partitioned on the join key in T partitions, and that each partition R_i is sorted on the join key.

Each thread i that participates in the parallel merge join algorithm starts by reading and storing tuples from S in the private buffer B_i. Then, thread i sorts B_i on the join key, and the B_i^j regions are computed exactly as in the MPSM algorithm. The difference in the parallel merge join algorithm lies in how thread i merges all the B_j^i regions with R_i. The original MPSM

algorithm performs T passes, and then merges two inputs at a time. In the parallel merge join algorithm, thread i merges all B_j^i buffers from the S side with the R_i partition in one pass, where $j \in [1, T]$. This is a parallel merge between T inputs for the S side, and one input for the R side, which results in $T + 1$ input tuples being candidates for merging at any given time. The output of the parallel merge join is partitioned on the join key per thread, with the same partitioning function as R, and each thread's output is always sorted on the join key.

The parallel merge join causes significantly lower memory traffic compared to the MPSM join. The biggest gain comes from reading the inputs once, so the number of tuples that the parallel merge join reads remains constant regardless of the number of threads participating in the join. The memory access pattern, however, is very different. During the merge, each thread i needs to compare the join key of a single tuple from R_i with the join key of the first unprocessed tuple of each B_j^i region, for a total of T join candidate tuples from S. As we will show in Section 3.5.2, keeping track of $T + 1$ locations stresses the TLB, causing frequent page walks which waste many cycles. This access pattern also makes hardware prefetching less effective: Assume that the second tuple was automatically prefetched when reading from B_1^i. This second B_1^i tuple will be processed after the first tuple from B_T^i has been processed. The probability of having evicted the cache line holding the second B_1^i tuple increases as the number of threads T grows.

The computational overhead of the parallel merge join is minimal, as the algorithm only performs join key comparisons over the join keys of both the R and S inputs, which are scanned once. When it comes to memory space, unlike the MPSM algorithm, each R_i partition is scanned once by each thread i so there is no performance benefit in buffering the S input. The parallel merge join algorithm therefore only needs sufficient memory to buffer the entire S input.

3.4 Evaluation methodology

3.4.1 Hardware

The hardware we use for our evaluation is a Dell R810 server with four Intel Xeon E7-4850 CPUs clocked at 2GHz, running RedHat Enterprise Linux 6.0, with Linux kernel 2.6.32-279. This is a system with four NUMA nodes, one for each die, and 64 GB per NUMA node, or 256 GB for the entire system. Each E7-4850 CPU packs 10 cores (20 hyper-threads) that share a 24MB L3 cache, and have a private 256KB L2 cache and a private 32KB L1 data cache, and each cache line is 64 bytes wide. The L1 data TLB has 64 entries. Each socket is directly connected to every other socket via a point-to-point QPI link, and a single QPI link can transport up to 12.8 GB/sec in each direction. Finally, each socket has its own memory controller, which has 4 DDR3 channels to memory for a total theoretical bandwidth of about 33.3 GB/sec per socket.

We refer to a number of performance counters when presenting our experimental results in Section 3.5.2. We have written our own utility to tap into the Uncore counters, which are described in detail in [51]. When we refer to memory reads, we count the L3 cache lines filled as reported by the LLC_S_FILLS event. When we refer to memory writes, we count the L3 cache lines victimized in the M state, or the LLC_VICTIMS_M event. We obtain the QPI utilization by comparing the null idle flits sent across all QPI links (event NULL_IDLE) with the number of idle flits sent when the system is idle. We then report the utilization of the most heavily utilized QPI link (ie. the link most likely to be a bottleneck) averaged over the measurement interval. Finally, we obtain timings for the TLB page miss handler by measuring the DTLB_LOAD_MISSES.WALK_CYCLES event.

Physical property of S	Physical property of R	Label in graph	Query plan	Per-thread join output sorted?	Join output partitioned across threads?
random	random	HASH	Build hash table on R, Probe hash table with S, Sum	No	No
		STRSM	Partition R, Sort R, Partition S, Sort S, Streaming merge, Sum	Yes	Yes
		MPSM	Partition R, Sort R, Sort S, MPSM merge, Sum	No	Yes
		PARSM	Partition R, Sort R, Sort S, Parallel merge, Sum	Yes	Yes
	sorted	HASH	Build hash table on R, Probe hash table with S, Sum	No	No
		STRSM	Partition S, Sort S, Streaming merge, Sum	Yes	Yes
		MPSM	Sort S, MPSM merge, Sum	No	Yes
		PARSM	Sort S, Parallel merge, Sum	Yes	Yes
	hash	HASH	Probe hash table with S, Sum	No	No
		STRSM	Partition R, Sort R, Partition S, Sort S, Streaming merge, Sum	Yes	Yes
		MPSM	Partition R, Sort R, Sort S, MPSM merge, Sum	No	Yes
		PARSM	Partition R, Sort R, Sort S, Parallel merge, Sum	Yes	Yes
sorted	random	HASH	Build hash table on R, Probe hash table with S, Sum	Yes	No
		STRSM	Partition R, Sort R, Streaming merge, Sum	Yes	Yes
		MPSM	Partition R, Sort R, MPSM merge, Sum	Yes	Yes
		PARSM	Partition R, Sort R, Parallel merge, Sum	Yes	Yes
	sorted	HASH	Build hash table on R, Probe hash table with S, Sum	Yes	No
		STRSM	Streaming merge, Sum	Yes	Yes
		MPSM	MPSM merge, Sum	Yes	Yes
		PARSM	Parallel merge, Sum	Yes	Yes
	hash	HASH	Probe hash table with S, Sum	Yes	No
		STRSM	Partition R, Sort R, Streaming merge, Sum	Yes	Yes
		MPSM	Partition R, Sort R, MPSM merge, Sum	Yes	Yes
		PARSM	Partition R, Sort R, Parallel merge, Sum	Yes	Yes

Table 3.1: List of query plans that we evaluate: "random" means tuples are in random order and not partitioned; "sorted" means sorted on the join key and range-partitioned; "hash" means stored in a hash table and hashed on the join key.

3.4.2 Query engine prototype

We have implemented all algorithms described in Section 3.3 in a query engine prototype written in C++. The engine is parallel and NUMA-aware, and can execute simple queries on main memory data. The engine expects queries in the form of a physical plan, which is provided by the user in a text file. The plan carries all the information necessary to compute the query result. All operators in our query engine have a standard interface consisting of start(), stop() and next() calls. This is a pull-based model similar to the Volcano system [36]. We amortize the cost of function calls; each next() call returns a 64KB array of tuples.

Our query engine prototype implements all the algorithms described in Section 3.3. We base the hash join implementation on the code described in Chapter 2, which is publicly available. We have extended it to be NUMA-aware by allocating the entire hash bucket i from NUMA node $(i \, mod \, N)$, where N is the number of NUMA nodes in the system. This means that tuples in the same hash bucket always reside in the same NUMA node, and reading hash buckets sequentially accesses all NUMA nodes in a round-robin fashion. We size each hash bucket so that it fits two tuples before there is an overflow. In addition to the usable space, we allocate 16 bytes of metadata for each bucket, and these are used for: (1) a latch, (2) a counter holding the number of unused bytes in this hash bucket, and (3) an overflow pointer to a next bucket. The number of hash buckets is always a power of two. If the hash table has 2^b hash buckets, we use the hash function $h(x) = (x * 2654435761) \, mod \, 2^b$ to compute the hash value for integers. We then size the hash table such that the load factor is greater than one, but less than two. The original code preallocated the first hash bucket, and we do likewise.

We implemented the MPSM merge join algorithm from scratch based on the description of the P-MPSM variant in [5], as the original implementation is not available.

3.4.3 Metrics

The two metrics we report are response time and memory footprint. Short query response times are, obviously, important for a main memory DBMS. Memory footprint is perhaps a less obvious, but also an important metric. The first reason is that query plans that use less memory allow the DBMS to admit more queries, increasing throughput. Second, main memory DBMSs use RAM both as working memory for active queries, and as storage for user data. A main-memory DBMS that selects plans that use working memory frugally can keep a larger database in memory, improving the overall utility of the system.

3.5 Experimental results

3.5.1 Workload

In our experimental evaluation we simulate the expensive join operations that occur when executing ad-hoc queries against a decision support database. Such a database typically has many smaller dimension tables that contain categorical information, such as customer or product details, and a few large fact tables that capture events of business interest, like the sale of a product.

To speed up query processing, a table might have already been pre-processed in some form, for example, all tuples might have already been sorted on some particular key. We refer to such pre-processing as the *physical property* of a table. We focus on three such properties:

1. **Random**, which corresponds to the case where no pre-processing has been done and data is in random order. Random input is processed by worker threads in any order.
2. **Sorted**, which corresponds to the case where the input is already sorted

on the join key. Sorted input can easily be partitioned among worker threads in distinct ranges, as the range partition boundaries can be computed from existing statistics on the input.

3. **Hash**, which reflects the case where a table is stored in a hash table, indexed on a particular key. This type of pre-processing is common for dimension tables that are small and are updated less frequently. Storing the large fact table in a hash table may be prohibitive in terms of space overhead, and maintaining the hash table is costly in the presence of updates. We therefore only consider this physical property for dimension tables.

Query plans

A common pattern in decision support queries is the equijoin between a dimension table and a fact table, followed by an aggregation. Assuming that the dimension table R has two integer attributes `R.a` and `R.b`, and the fact table S has two integer attributes `S.c` and `S.d`, for all the experiments described here, we produce plans corresponding to the following SQL query:

```
SELECT SUM(R.b + S.d) FROM R, S WHERE R.a = S.c
```

Depending on the physical properties of R and S, this logical query may be translated into a number of physical execution plans. For our experimental evaluation, we consider candidate plans that are formed by combining the algorithms presented in Section 3.3. All query plans do a join between R and S, and materialize the output which is pipelined to the final operator, the aggregation. The aggregate operation takes a negligible amount of time (less than 1%), but this approach forces us to look at the holistic effect of a join algorithm in a complex pipeline, where results have to be piplelined to the next stage of the query processing pipeline. The exact query plans we execute, for different combinations physical properties of R

and S, are shown in Table 3.1. For all our experiments, we execute each query plan in isolation, and use 80 worker threads, which is the number of hardware contexts our system supports.

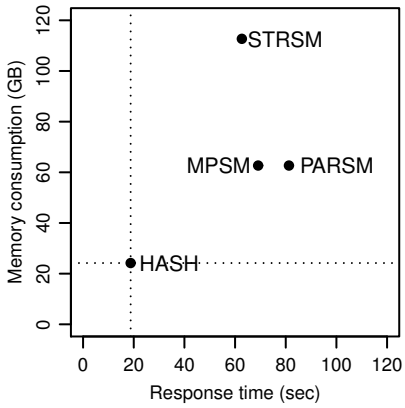
Datasets

We use two datasets for evaluation. Both datasets model an ad-hoc equijoin between the dimension table R, and the fact table S. The dimension table R contains the primary key and the fact table S contains the foreign key. R has 800×2^{20} tuples and each tuple is sixteen bytes wide. Each R tuple consists of an eight-byte unique primary key in the range of $\left[1, 800 \times 2^{20}\right]$, and an eight-byte random integer.

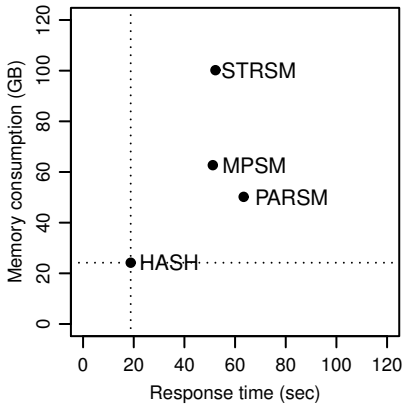
The fact table S has four times as many tuples as R, namely 3200×2^{20} tuples, and each tuple is also sixteen bytes wide. We picked the one-to-four ratio to match the cardinality ratios of a primary-key foreign-key join between tables `Orders` and `LineItem`, or `Part` and `PartSupp` of the TPC-H decision support benchmark. The first eight-byte attribute of an S tuple is the foreign key of R, and the second eight-byte attribute is a random integer. The cardinality of the output of the primary-key foreign-key join is the cardinality of S, or 3200×2^{20} tuples.

Uniform dataset: This dataset does not have any skew. Every primary key in R occurs in S exactly four times.

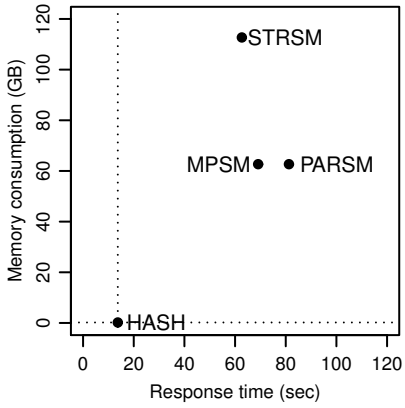
Skewed dataset: For this dataset, the foreign keys in S are generated using the Zipf distribution with parameter $s = 1.05$. With this setting, the top 1000 keys occur nearly half the time (48%). We chose this parameter to match the "low skew" dataset that was used in Section 2.5.1.



(a) R is in random order



(b) R is sorted in ascending join key order



(c) R is in hash table on join key

Figure 3.1: Response time (in seconds) and memory consumption (in GB) when S is in random order, uniform dataset

In Figure	3.1(a)	3.1(b)	3.1(c)	3.2(a)	3.2(b)	3.2(c)
HASH	S in random order			S in join key order		
	R rnd	R ord	R ht	R rnd	R ord	R ht
Build HT on R	4.2	5.0	—	5.1	5.6	—
Probe HT with S	13.5	13.7	13.6	6.0	5.3	6.0
Query	18.8	18.8	13.9	11.1	11.1	6.1
STRSM	S in random order			S in join key order		
	R rnd	R ord	R ht	R rnd	R ord	R ht
Partition R	1.5	—	1.5	2.8	—	2.8
Sort R	7.2	—	7.2	7.3	—	7.3
Partition S	6.4	8.8	6.4	—	—	—
Sort S	36.3	36.7	36.3	—	—	—
Stream merge	5.9	2.0	5.9	1.8	1.9	1.8
Query	62.7	52.3	62.7	11.9	2.0	12.0
MPSM	S in random order			S in join key order		
	R rnd	R ord	R ht	R rnd	R ord	R ht
Partition R	3.4	—	3.4	2.7	—	2.7
Sort R	5.9	—	5.9	6.7	—	6.7
Sort S	29.2	29.1	29.2	—	—	—
MPSM merge	24.5	16.2	24.5	1.7	1.3	1.7
Query	69.1	51.2	69.1	16.2	7.5	16.2
PARSM	S in random order			S in join key order		
	R rnd	R ord	R ht	R rnd	R ord	R ht
Partition R	3.1	—	3.1	2.7	—	2.7
Sort R	5.8	—	5.8	6.7	—	6.7
Sort S	28.9	29.0	28.9	—	—	—
Parallel merge	35.6	28.2	35.6	4.8	4.4	4.8
Query	81.3	63.4	81.3	19.5	10.6	19.5

Table 3.2: Time spent in each operator for the uniform dataset. "R rnd" means that R is in random order, "R ord" means that R is sorted in join key order, and "R ht" means that R is in a hash table on the join key. "Query" reflects the query response time as shown in Figures 3.1 and 3.2, and is not the sum due to synchronization and buffering overheads that are not reflected in the breakdown above.

3.5.2 Results from the uniform dataset

As described in Section 3.4.3, we measure cost both in terms of response time, and memory space. In all figures in this chapter, we plot the response time in the x-axis in seconds, and the memory consumption in gigabytes (2^{30} bytes) on the y-axis. We use "HASH" for the hash-based plan, and the "SM" suffix for the three sort-based query plans: "STRSM" for the plan with the streaming merge join operator, "MPSM" for the plan containing the MPSM merge join operator, and "PARSM" for the plan with the parallel merge join operator. Table 3.1 shows the actual operators for each plan, and Table 3.2 breaks down time per operator.

S in random order

We start with the case where the S table contains tuples with keys in random join key order. This requires each algorithm to pay the full cost of transforming the input S to have the physical property necessary to compute the join, like being sorted or partitioned on the join key. In Figure 3.1 we plot the response time and the memory consumption of each plan, for all three physical properties of R we explore: random, sorted, and hash. We describe each in turn.

R in random order

We start with the general case where both the R and S tables contain tuples with join keys in random order. The response time and memory demand for all the four query plans is shown in Figure 3.1(a), and a breakdown per operator is shown in Table 3.2, column 3.1(a).

The hash join query plan computes the query result in 18.8 seconds, the fastest among the all four query plans. The hash join query plan also uses the least space, needing 24.2 GB of scratch memory space, nearly all of which (99%) is used to store the hash table. The majority of the time is

spent in the probe phase, which does about 1.90 memory reads per S tuple. Because the S tuples are processed in random join key order, these reads are randomly scattered across the shared hash table. This random memory access pattern causes a significant number of TLB misses, with 10% of the probe time, or 1.31 seconds, being spent on handling page misses on memory loads. As the hash table is striped across all the NUMA nodes, the cross-socket traffic is the highest among all the query plans: we have observed an average QPI utilization of 34%, which means that remote memory access requests might get queued at the QPI links during bursts of higher memory activity.

The streaming sort-merge plan (STRSM) needs 62.7 seconds and 113 GB of memory to produce the output, and the majority of the memory space (100 GB) is needed to partition the S table. The majority of the time is spent in sorting the larger S table. Repartitioning and sorting both R and S causes significant memory traffic: 4.92 memory reads and 3.72 memory writes are performed, on average, per processed tuple.

The MPSM sort-merge join query plan takes 69.1 seconds and 62.7 GB of space to run. Compared to the streaming sort-merge query plan, MPSM is only 10% slower, and, because the MPSM merge-join algorithm avoids repartitioning the larger S table, it needs about half the space. The MPSM merge join algorithm performs T scans of R during the merge phase (see Section 3.3.5), which means reading an additional 0.96 TB, for this particular dataset. This causes 4.32 memory reads per S tuple, during the merge phase, which accounts for 35% of the total time.

The parallel sort-merge query plan (PARSM) is identical to the MPSM sort-merge join query plan, described above, for the partitioning and sort phases. During the merge phase, however, the parallel merge join algorithm does not partition S and does not scan R multiple times. This cuts the memory traffic significantly, resulting in only 0.38 memory reads per S tuple for the merge phase. Merging from 81 $(T + 1)$ locations in parallel,

however, overflows the 64-entry TLB (see Section 3.3.6 for a description of the algorithm). We observed that page miss handling on memory load accounts for at least 37% of the merge time (13.2 seconds). The high TLB miss rate causes the merge phase to take 11 more seconds than the MPSM merge phase, which brings the end-to-end response time to a total of 81.3 seconds, making the parallel merge-join plan the slowest of the four alternatives.

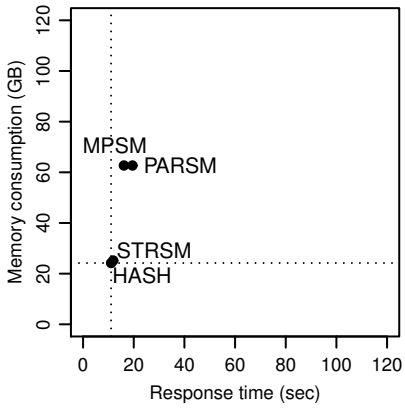
Overall, when R and S are processed in random order, the hash join query plan produces results 3.34× faster and uses 2.59× less memory than the best sort-merge based query plan, despite the many random accesses to remote NUMA nodes that result in increased QPI traffic.

R in ascending join key order

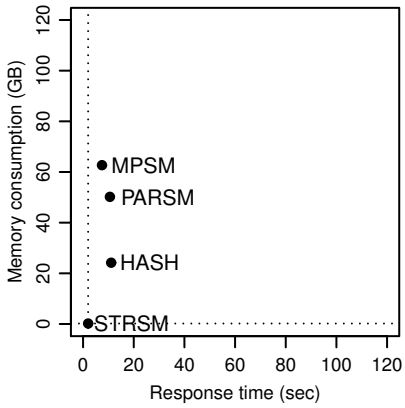
We now consider the case where the smaller table R is sorted, and the larger table S is in random order. We plot the response time and memory demand for the four query plans in Figure 3.1(b), and a breakdown per operator is shown in Table 3.2, column 3.1(b).

The response time and memory consumption of the hash-join query plan are 18.8 seconds and 24.2 GB respectively. Comparing with the case where R is in random order, Figure 3.1(a), we see that performance is not affected by whether the build side R is sorted or not. Also, the number of memory operations and the QPI link utilization are the same as when processing R with tuples in random physical order. This happens because the hash function we use, described in Section 3.4.2, scatters the sorted R input among distant buckets. This results in memory accesses that are not amenable to hardware caching or prefetching, exactly as when processing R in random order.

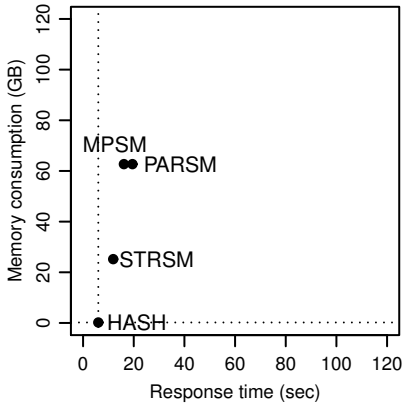
Moving to the stream-merge join query plan (STRSM), we find that it completes the join in 52.3 seconds and has a memory footprint of about



(a) R is in random order



(b) R is sorted in ascending join key order



(c) R is in hash table on join key

Figure 3.2: Response time (in seconds) and memory consumption (in GB) when S is sorted in ascending join key order, uniform dataset

100 GB, with nearly all of this space being used for repartitioning S. There is no need to repartition and sort R, as it is already in join key order. Repartitioning and sorting S causes 5.25 memory reads and 3.82 memory writes, on average, per S tuple.

The MPSM merge join query plan has a response time of 51.2 seconds and needs 62.7 GB of memory. Sorting S takes the majority of the time, and then the merge phase performs T scans of the pre-sorted, pre-partitioned side R to produce the result.

The parallel merge join query (PARSM) takes 63.4 seconds to produce the join output and needs 50.2 GB of memory space, nearly all of which is needed to buffer and sort S. The merge phase, however, is takes 75% more time than the MPSM merge phase, primarily due to the page miss handling overhead associated with keeping track of $T + 1$ locations in parallel.

In summary, when R is sorted on the join key and the tuples of S are in random physical order, the hash join query plan has 2.72× lower response time and 2.07× smaller memory footprint compared to all other sort-merge based queries.

R in hash table, on join key

We now move to the case where R is stored in a hash table created on the join key, and the S tuples are in random order. We plot the results in Figure 3.1(c). We observe that the three sort-based queries have similar response time and memory consumption as when the R tuples are in random order (cf. Figure 3.1(a)). Due to the pre-allocated space in the hashtable buckets, no pointer chasing is needed to read the first bucket of each hash chain, allowing for sequential reading across all NUMA nodes. R is partitioned in buckets based on hash value, so all sort-based plans first repartition R to create range partitions. These partitions subsequently need to be sorted, before being processed by each merge-join algorithm. Overall,

these steps result in the same data movement operations as when R is in random order.

The hash join plan can take advantage of the fact that R is already in a hash table on the join key, as one can now skip the build phase. This reduces the response time of the query to 13.9 seconds, which is 4.51× faster than the fastest sort-based query. As there is no hash table to allocate, populate and discard, the hash join algorithm now becomes a streaming algorithm, needing a fixed amount of memory regardless of the input size. In our prototype, the total space needed was 0.01 GB for holding output buffers, metadata and input state.

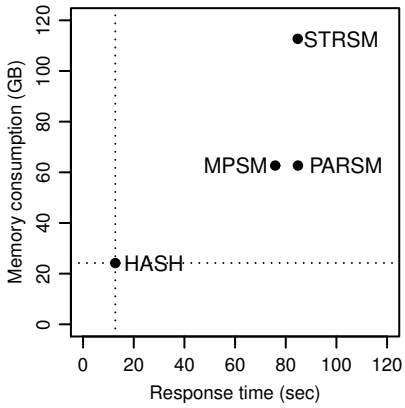
S in ascending join key order

We now consider the case when the S table is sorted in ascending join key order. In this case, we also assume that S is partitioned, as each thread i can discover the boundaries of the S_i partition by performing interpolation or binary search in the sorted table S. In the results that follow, we have discounted the cost of computing the partition boundaries in this fashion. We plot the response time and the memory consumption of each query, for all three physical properties of R we explore. These results are shown in Figure 3.2.

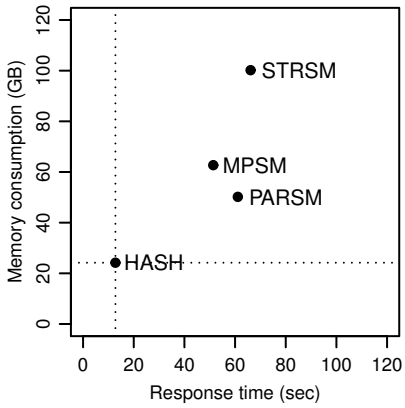
R in random order

First we consider the case when the larger table S is sorted on the join key, but the smaller table R is not. We plot the results in Figure 3.2(a), and a breakdown per operator is shown in Table 3.2, column 3.2(a).

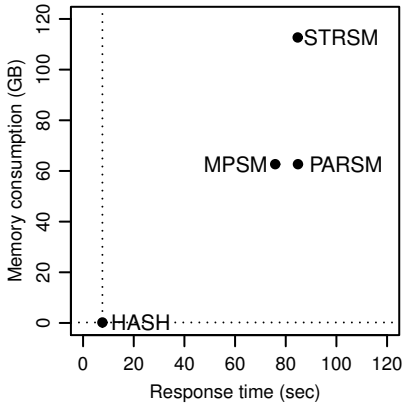
The hash join query computes the result in 11.1 seconds and needs 24.2 GB of memory space. If we compare with the case where S is in random order, in Figure 3.1(a), we see a 1.70× improvement in response time, if S is sorted on the join key. This happens because now that S is sorted, the



(a) R is in random order



(b) R is sorted in ascending join key order



(c) R is in hash table on join key

Figure 3.3: Response time (in seconds) and memory consumption (in GB) when S is in random order, skewed dataset

four S tuples that match a given R tuple occur in sequence. This allows the R tuple to be read only once and then stay in the local cache, reducing the total number of memory operations. Looking at the performance counters, we find that 0.36 memory reads occur per S tuple, which is 5.31× less than the number of memory operations that happen when S is randomly ordered. Probing the hash table with a sorted S input also results in fewer cycles spent on TLB miss handling (0.6 seconds, or 10% of the probe time, a 2.22× improvement), as well as lower QPI link utilization during the probe phase (14.6% utilization on average, a 2.35× improvement) when compared with processing a randomly ordered S table.

The streaming sort-merge join plan (STRSM) has a response time of 11.9 seconds and a memory footprint of 25.2 GB, and sorting R takes the majority of the time. Repartitioning and sorting R is an expensive operation in terms of memory traffic: it takes 5.02 memory reads and 3.31 memory writes per R tuple.

In this experiment, both the MPSM and the PARSM query plans represent degenerate cases where each thread i pays the full overhead of tracking and merging T partitions of S with its R_i partition. However, because S is prepartitioned, $T - 1$ partitions are empty and the merge is a two-input streaming merge, as described in the previous paragaph. We include the data points here for completeness, and to demonstrate what the response time and memory consumption would be in this setting. The MPSM plan finishes in 16.2 seconds, which is 36% slower than the STRSM plan. The MPSM plan uses 62.7 GB of memory, 50 GB of which is used for buffering S. The PARSM plan is similar in all phases except the final merge join phase, which it completes in 4.77 seconds, bringing the total time to 19.5 seconds.

To summarize, when looking at response time, we find that both the hash join and streaming merge join plans perform comparably as they return results in 11-12 seconds, and both need about 25 GB of memory; both

outperform the other two plans.

R in ascending join key order

This is the case where both R and S are presorted on the join key. The results for this experiment are shown in Figure 3.2(b), and a breakdown per operator is shown in Table 3.2, column 3.2(b). The hash join plan has a response time of 11.1 seconds, and uses 24.2 GB to store the hash table for R. The hash join plan cannot take advantage of the sorted R side and needs to construct a hash table on R. Join keys appear sequentially in S, and this translates into less memory traffic when probing the hash table due to caching. The streaming merge join plan (STRSM) is the fastest, as it only needs to scan the pre-sorted and pre-partitioned inputs in parallel to produce the join output. The response time of the STRSM plan is 1.98 seconds, and because of its streaming nature the memory size is fixed, regardless of the input size. We measured memory consumption to be 0.01 GB, primarily for output buffer space. As before, the MPSM and the PARSM plans are degenerate two-way stream merge join queries.

Overall, when both R and S are sorted on the join key, the preferred join plan is the streaming merge join which needs minimal memory space and is 5.60× faster than the hash join.

R in hash table, on join key

Finally, we consider the case where R is stored in a hash table, on the join key, and S is sorted in ascending join key order. We plot the response time and memory demand of each query plan in Figure 3.2(c), and a breakdown per operator is shown in Table 3.2, column 3.2(c).

The hash join plan omits the build phase, as R is already in the hash table, and proceeds directly to the probe phase. The hash join plan completes the query in 6.1 seconds, and needs only 0.01 GB of space for storing metadata, the input state and the output buffers of each thread. None of the

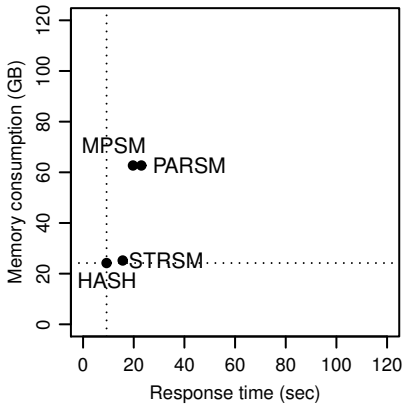
sort-based plans can take advantage of the hash-partitioned R. As a consequence, the first step in all plans is to repartition R and produce range partitions. The streaming merge join plan (STRSM) takes 12.0 seconds, and needs 25 GB for the repartitioning, and the remaining two sort-based plans are degenerate cases of the streaming merge join query.

To summarize, if R is already stored in a hash table, and S is sorted on the join key, the preferred join strategy is the hash join, because it can take advantage of the physical property of R without additional operations. The hash join query in this case is nearly 2× faster in response time and only uses minimal memory.

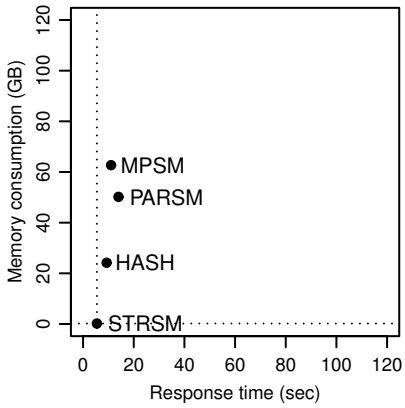
3.5.3 Results from the skewed dataset

We now discuss how the results are affected by the presence of data skew. For each physical property of R of interest, we plot the response time and memory consumption of all queries in Figure 3.3, when S is in random order, and in Figure 3.4 when S is presorted on the join key. The aggregate memory demand for each query does not change with skew, as the two datasets are of equal size, however the memory needs of individual threads are different, depending on the size of the partitions they are assigned to.

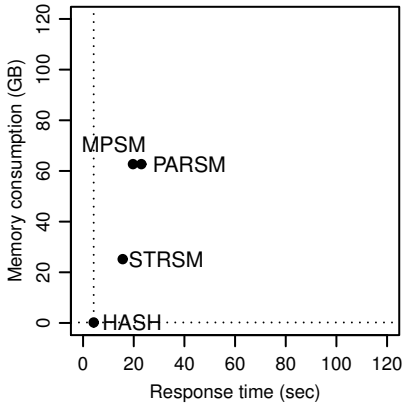
Starting from the hash join algorithm, we see that the response time of the hash join query plans actually improves with skew, similar to what we observed in Section 2.5 for the single-socket case. Comparing with the uniform dataset, we find that response time has dropped to 12.7 seconds from 18.8 seconds, a 1.48× improvement, for Figures 3.3(a) and 3.3(b). We see a smaller 1.19× benefit, to 9.28 seconds from 11.1 seconds for Figures 3.4(a) and 3.4(b). When R is already in a hash table, the hash join query plan has a response time of 7.63 seconds when S is in random order (Figure 3.3(c), a 1.80× improvement), and a response time of 4.19 seconds when S is sorted (Figure 3.4(c), a 1.43× improvement). Breaking this down further, we see



(a) R is in random order



(b) R is sorted in ascending join key order



(c) R is in hash table on join key

Figure 3.4: Response time (in seconds) and memory consumption (in GB) when S is sorted in ascending join key order, skewed dataset

that the build phase is nearly unchanged, and all the response time gains come from the probe phase. This happens because of caching, as now the most popular items, if accessed frequently enough, will remain cached. This significantly reduces the number of memory operations needed to complete the hash join. For example, for the case where both R and S tuples are in random physical order (Figure 3.3(a)), the number of memory reads per S tuple has improved 2.79× and now is 0.68, down from 1.90 for the uniform dataset (Figure 3.1(a)).

Turning our attention to the sort-merge based query plans, we find that they are negatively impacted by data skew. The MPSM and parallel merge plans (PARSM) are more resilient, but the streaming merge plan (STRSM) is affected to a larger degree. For instance, when both the R and S are in random physical order (Figure 3.3(a)), response time for the STRSM plan is 84.7 seconds, or 1.35× higher than the 62.7 seconds with the uniform dataset (Figure 3.1(a)).

The culprit here is the partitioning step that all sort-merge based query plans rely on. Data skew in this case results in skewed partition sizes, and as partitions are assigned to specific threads, this results in a skewed work distribution. This effect can be ameliorated by smart partition boundary selection [5, 28] and work stealing. However, it cannot be eliminated, as when a popular data item is in a single partition by itself, it cannot be further subdivided into smaller partitions. This is an important limitation if the number of partitions is associated with the maximum degree of parallelism one can achieve: In our 80-thread system, the partitions will be imbalanced if any key occurs more frequently than $\frac{1}{80} = 1.25\%$ of the time. As the number of threads in a system increases, the probability of having an imbalanced work distribution for a given skewed dataset increases. In comparison, the hash join algorithm accesses a read-only shared hash table in the probe phase. Any thread that participates in the probe phase can process any input tuple, resulting in a uniform work distribution regardless

Attributes per tuple (N)	R tuple size (bytes)	$\|R\|$	S tuple size (bytes)	$\|S\|$
1	8	1280×2^{20}	8	5120×2^{20}
2	16	640×2^{20}	16	2560×2^{20}
4	32	320×2^{20}	32	1280×2^{20}
8	64	160×2^{20}	64	640×2^{20}
16	128	80×2^{20}	128	320×2^{20}

Table 3.3: Properties of the five different datasets we use to explore the impact of tuple size on query response time. $|R|$ denotes the cardinality of the dimension table R and $|S|$ denotes the cardinality of the fact table S. Four S tuples match exactly one tuple in R.

of the dataset and the number of available threads.

To summarize, the hash join is the preferred algorithm when there is data skew in a primary key-foreign key join. There are two reasons for this. First, regardless of the dataset and the number of threads, the hash join algorithm creates a balanced work distribution for all threads during the probe phase. Partitioning-based algorithms, in comparison, cannot guarantee a uniform work distribution, and their response time increases with higher skew, if a few threads must handle disproportionate amount of work. Second, frequent accesses to the hottest items effectively "pin" them in the cache, significantly reducing memory traffic. This causes the response time of the hash join algorithm to improve as skew increases, similar to what has already been observed for single-socket systems in Section 2.5.

3.5.4 Impact of wider tuples

The datasets we have experimented with so far have tuples that are only sixteen bytes wide. While such small tuple sizes are common in data warehousing environments that store data in a column-oriented fashion, tuples may be substantially larger in other settings.

We generate five different datasets to explore how different tuple sizes affect query response time, as shown in Table 3.3. Across all five datasets, we keep the total amount of processed data constant, and we vary the tuple size. The dimension table R has N integer attributes r1, r2, ... , rN, and the fact table S also has N integer attributes s1, s2, ... , sN. The join key is always fixed to eight bytes, and we experiment with tuples as thin as eight bytes (N=1) and as wide as 128 bytes (N=16). Larger tuple sizes, such as 128 bytes, capture the case when tuples are stored in a row-oriented fashion, as is the case when data is retrieved from a transaction processing engine. Smaller tuple sizes, such as eight bytes, enable additional performance optimizations by cleverly using SIMD registers to form a bitonic merge network for sort-merge join [54].

All datasets model an ad-hoc equijoin between a dimension table R, and a fact table S. The dimension table R contains the primary key and the fact table S contains the foreign key. Each R tuple consists of an eight-byte unique primary key, and the remaining attributes are random integers. The fact table S has four times as many tuples as R, exactly as in the datasets described in Section 3.5.1. The first eight-byte attribute of an S tuple is the foreign key of R, and the remaining attributes are random integers. The cardinality of the output of the primary-key foreign-key join is the cardinality of S.

We only consider the case were both the R and S inputs are in random order. For the dataset where the tuple width is 8 bytes, all sort-based query plans use an implementation of the SIMD-optimized bitonic sort-merge join algorithm that Kim et al. [54] describe in Section 5.2 for sorting. For the experiments in this section, we produce plans corresponding to the following SQL query:

```
SELECT SUM(R.r1 + R.r2 + ··· + R.rN
         + S.s1 + S.s2 + ··· + S.sN)
FROM R, S
```

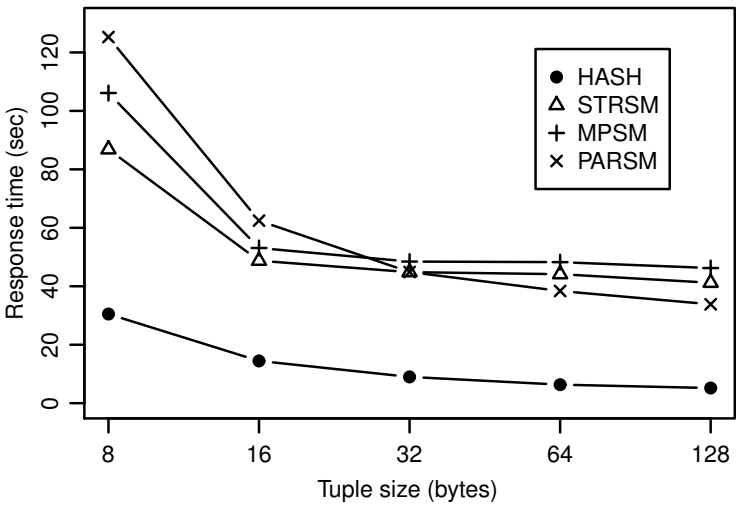


Figure 3.5: Response time (in seconds) as dataset size remains fixed and tuple size varies between 8 and 128 bytes. S and R are in random order. When the tuple size is 8 bytes, all sort-based plans ("SM" suffix) use the SIMD-optimized bitonic sort-merge join algorithm that is described in Section 5.2 by Kim et al. [23] for sorting.

```
WHERE R.r1 = S.s1
```

The query plans that are produced for this query consist of the same operators as the query plans discussed in Section 3.5.2 and described in Table 3.1. Furthermore, because the total size of each dataset is the same, the memory footprint for a given query plan remains nearly unchanged as the tuple size varies. In the interest of space, we omit the discussion of the memory footprint of each query plan, and refer the reader to the proportional differences in memory footprint shown in Figure 3.1(a).

We plot the results in Figure 3.5.4. The horizontal axis corresponds to different datasets of varying tuple width, which are described in Table 3.3. The vertical axis shows the response time, in seconds, of each join query. As in Section 3.5.2, we use "HASH" for the hash-based plan, and the "SM"

suffix for the three sort-based query plans: "STRSM" for the plan with the streaming merge join operator, "MPSM" for the plan containing the MPSM merge join operator, and "PARSM" for the plan with the parallel merge join operator.

Starting from the hash-based query plan, we find that the total response time when processing eight-byte tuples is 29.8 seconds. As tuples get wider, response time drops down to 5.2 seconds for 128-byte wide tuples. All sort-based plans have higher response times, starting at 86.9 seconds for the streaming merge ("STRSM") join plan with the bitonic sort-merge sorting for eight-byte tuples. Response time drops to 33.9 seconds for the parallel merge join operator ("PARSM") when tuples are 128 bytes wide. Although the response time for all sort-based plans is similar, the streaming merge join plan needs twice as much memory space as the parallel merge join and MPSM join plans due to buffering (see Figure 3.1(a) and Section 3.5.2 for details.)

The response time drops as tuples get wider primarily due to reduced memory activity. This is especially pronounced for the hash join plan: 16.0 billion memory operations take place when the dataset consists of eight-byte tuples, compared with only 2.9 billion memory operations when the same hash join query plan is executed on 128-byte wide tuples. Performance improves with wider tuples because of better spatial locality. Intuitively, as tuples get wider, the hash join query plan based processes more bytes per hash table lookup, and the sort-based plans process more bytes per tuple copy (as they sort in-place).

To summarize, regardless of the size of the input tuples, when R and S are in random join key order, the hash join plan has the shortest response time. The sort-based query plans cannot close the performance gap even when using a SIMD-optimized bitonic sort-merge sorting algorithm [54] that is optimized for eight-byte tuples.

3.5.5 Summary of our findings

The hash join is the best join strategy when tuples in the fact table S are randomly ordered. The hash join outperforms all other algorithms both in terms of response time, and memory consumption: For the uniform dataset, the hash-based join plans have from 2.72× to 4.57× lower response time compared to their fastest sort-based counterpart, while using at least 2.07× less memory. The response time margin widens even further, in favor of the hash join, when the larger table S has data skew, as the popular items get cached and memory traffic is reduced. MPSM is the preferred sort-based join plan when S is in random order, because it uses half the memory than the streaming merge join plan (STRSM) and has comparable response time.

The sort-based algorithms only have competitive response times when the larger table S is presorted on the join key. This improves the performance of the hash join as well, albeit to a smaller degree. When both R and S are pre-sorted on the join key, the streaming merge join plan (STRSM) has the lowest response time and the smallest memory footprint.

3.6 Concluding remarks

In this chapter we have shown that it is too early to write off the hash join algorithm when evaluating equijoins in modern main memory DBMSs. We have carefully characterized the impact of various input physical properties on the simple hash-based join and three flavors of sort-based equijoin algorithms, and shown that the hash-based join algorithm often outperforms the current state-of-the-art sort-based methods. We also characterize the memory footprint that is required to run each join method, as that is an important practical consideration when building an actual main memory DBMS. Overall, our results find that there is a place for both hash-based and sort-based methods in a high performance main memory DBMS.

There are many directions for future work, including expanding this

study to more complex queries with multiple joins, considering query optimization techniques for complex queries in main memory DBMSs, considering non-equijoins, and further investigating methods to improve the basic join algorithms.

Chapter 4

Concurrency control for main memory databases

A database system optimized for in-memory storage can support much higher transaction rates than current systems. However, standard concurrency control methods used today do not scale to the high transaction rates achievable by such systems. In this chapter we introduce two efficient concurrency control methods specifically designed for main-memory databases. Both use multiversioning to isolate read-only transactions from updates but differ in how atomicity is ensured: one is optimistic and one is pessimistic. To avoid expensive context switching, transactions never block during normal processing but they may have to wait before commit to ensure correct serialization ordering. We also implemented a main-memory optimized version of single-version locking. Experimental results show that while single-version locking works well when transactions are short and contention is low performance degrades under more demanding conditions. The multiversion schemes have higher overhead but are much less sensitive to hotspots and the presence of long-running transactions.

4.1 Introduction

Current database management systems were designed assuming that data would reside on disk. However, memory prices continue to decline; over the last 30 years they have been dropping by a factor of 10 every 5 years. The latest Oracle Exadata X2-8 system ships with 2TB of main memory and it is likely that we will see commodity servers with multiple terabytes of main memory within a few years. On such systems the majority of OLTP databases will fit entirely in memory, and even the largest OLTP databases will keep the active working set in memory, leaving only cold, infrequently accessed data on external storage.

A DBMS optimized for in-memory storage and running on a many-core processor can support very high transaction rates. Efficiently ensuring isolation between concurrently executing transactions becomes challenging in such an environment. Current DBMSs typically rely on locking but in a traditional implementation with a separate lock manager the lock manager becomes a bottleneck at high transaction rates as shown in experiments by Johnson et al. [52]. Long read-only transactions are also problematic as readers may block writers.

In this chapter, we investigate what are the appropriate high-performance concurrency control mechanisms for memory-resident OLTP workloads. We found that traditional single-version locking is "fragile": It works well when all transactions are short and there are no hotspots but performance degrades rapidly under high contention or when the workload includes even a single long transaction.

Decades of research has shown that multiversion concurrency control (MVCC) methods are more robust and perform well for a broad range of workloads. This led us to investigate how to construct MVCC mechanisms optimized for main memory settings. We designed two MVCC mechanisms: the first is optimistic and relies on validation, while the second one is pessimistic and relies on locking. The two schemes are mutually compatible

in the sense that optimistic and pessimistic transactions can be mixed and access the same database concurrently. We systematically explored and evaluated these methods, providing an extensive experimental evaluation of the pros and cons of each approach. The experiments confirmed that MVCC methods are indeed more robust than single-version locking.

This chapter makes three contributions. First, we propose an optimistic MVCC method designed specifically for memory resident data. Second, we redesign two locking-based concurrency control methods, one single-version and one multiversion, to fully exploit a main-memory setting. Third, we evaluate the effectiveness of these three different concurrency control methods for different workloads. The insights from this study are directly applicable to high-performance main memory databases: single-version locking performs well only when transactions are short and contention is low; higher contention or workloads including some long transactions favor the multiversion methods; and the optimistic method performs better than the pessimistic method.

The rest of this chapter is organized as follows. Section 4.2 gives a brief overview of related work. Section 4.3 covers preliminaries of multiversioning and describes how version visibility and updatability are determined based on version timestamps. The optimistic scheme and the pessimistic scheme are described in Sections 4.4 and 4.5, respectively. Section 4.6 reports performance results, and Section 4.7 offers concluding remarks.

4.2 Related work

Concurrency control has a long and rich history going back to the beginning of database systems. Several excellent surveys and books on concurrency control are available [13, 40, 55, 77].

Multiversion concurrency control methods also have a long history. Chapter 5 in [13] describes three multiversioning methods: multiversion times-

tamp ordering (MVTO), two-version two-phase locking (2V2PL), and a multiversion mixed method. 2V2PL uses at most two versions: last committed and updated uncommitted. They also sketch a generalization that allows multiple uncommitted versions and readers are allowed to read uncommitted versions. The mixed method uses MVTO for read-only transactions and Strict 2PL for update transactions.

The optimistic approach to concurrency control originated with Kung and Robinson [56], but they only considered single-version databases. Many multiversion concurrency control schemes have been proposed [2, 14, 15, 18, 19, 41, 57, 67], but we are aware of only two that take an optimistic approach: Multiversion Serial Validation (MVSV) by Carey [20, 21] and Multiversion Parallel Validation (MVPV) by Agrawal et al. [1]. While the two schemes are optimistic and multiversion, they differ significant from our scheme. Their isolation level is repeatable read; other isolation levels are not discussed. MVSV does validation serially so validation quickly becomes a bottleneck. MVPV does validation in parallel but installing updates after validation is done serially. In comparison, the only critical section in our method is acquiring timestamps; everything else is done in parallel. Acquiring a timestamp is a single instruction (an atomic increment) so the critical section is extremely short.

Snapshot isolation (SI) [12] is a multiversioning scheme used by many database systems. Several database management systems support snapshot isolation to isolate read-only transactions from updaters: Oracle, PostgreSQL and SQL Server [61] and possibly others. However, SI is not serializable and many papers have considered under what circumstances SI is serializable or how to make it serializable. Cahill et al. [19] published a complete and practical solution in 2009. Their technique requires that transactions check for read-write dependencies. Their implementation uses a standard lock manager and transactions acquire "locks" and check for read-write dependencies on every read and write. The "locks" are non-

blocking and used only to detect read-write dependencies. Whether their approach can be implemented efficiently for a main-memory DBMS is an open question. Techniques such as validating by checking repeatability of reads and predicates have already been used in the past [17].

Oracle TimesTen [64], IBM's solidDB [47] and SAP HANA [31] are three commercially available main-memory DBMSs. TimesTen uses single-version locking with multiple lock types (shared, exclusive, update) and multiple granularities (row, table, database). For main-memory tables, solidDB also uses single-version locking with multiple lock types (shared, exclusive, update) and two granularities (row, table). For disk-based tables, solidDB supports both optimistic and pessimistic concurrency control. HANA has a transaction manager that supports ACID transactions and is based on multi-version concurrency control. We are not aware of any published information on how this multi-version concurrency control method is implemented, and whether it is based on locking or validation.

Main-memory concurrency control kernels frequently implement a superset of the functionality offered by software transactional memory [73] implementations, as they add support for durability and application-specific isolation requirements for each transaction. Hardware transactional memory [45] has been proposed for better performance, and Tran et al. [78] have explored how to use hardware transactional memory to execute entire transactions. The emergance of hardware transactional memory support in mainstream Intel and IBM processors allows designers to simplify the logic and the underlying data structures in light of concurrent modifications, and is an interesting area for future work.

4.3 Multi-version storage engine

A transaction is by definition serializable if its reads and writes logically occur as of the same time. The simplest and most widely used MVCC

method is snapshot isolation (SI). Snapshot isolation does not guarantee serializability because reads and writes logically occur at different times: reads occur at the beginning of the transaction and writes at the end. However, a transaction is serializable if we can guarantee that it would see exactly the same data if all its reads were repeated at the end of the transaction.

To ensure that a transaction T is serializable we must guarantee that the following two properties hold:

Read stability: If T reads some version V1 of a record during its processing, we must guarantee that V1 is still the version visible to T as of the end of the transaction, that is, V1 has not been replaced by another committed version V2. This can be implemented either by read locking V1 to prevent updates or by validating that V1 has not been updated before commit. This ensures that nothing has disappeared from the view.

Phantom avoidance: We must also guarantee that the transaction's scans would not return additional new versions. This can be implemented in two ways: by locking the scanned part of an index/table or by rescanning to check for new versions before commit. This ensures that nothing has been added to the view.

Lower isolation levels are easier to support:

1. For repeatable read, we only need to guarantee read stability.

2. For read committed, no locking or validation is required; always read the latest committed version.

3. For snapshot isolation, no locking or validation is required; always read as of the beginning of the transaction.

We have implemented a prototype main-memory storage engine. We begin with a high-level overview of how data is stored, how reads and updates are handled, and how it is determined what versions are visible to a reader, and that a version can be updated.

4.3.1 Version format

In a multi-version scheme, a record version is visible only to transactions whose logical read time is within a particular timestamp interval. Storing the timestamp range of this interval is sufficient when there is no update activity in the system. However, when a transaction is performing an update, the precise visibility interval is not known. In those cases, we need to transiently store a unique transaction identifier that corresponds to the transaction that modified the visibility interval.

In our prototype, we keep track of two additional fields in each database record to determine whether a version is visible or not to transactions. In addition to the user-defined columns, each record contains a `Begin` and an `End` field:

Begin A field that stores the earliest logical read time this version is visible to transactions. This is a 64-bit field and consists of:

1. A flag (1 bit) indicating whether a transaction is currently changing the beginning of the visibility interval.
2. If the 1-bit flag is set, the remaining 63 bits contain a unique *transaction identifier* that corresponds to the transaction changing the visibility of this version. If the 1-bit flag is unset, the remaining 63 bits store a *timestamp*, which indicates the earliest logical read time this version is visible to transactions.

End A field that stores the latest logical read time this version is visible to transactions. This is also a 64-bit field and consists of:

1. A flag (1 bit) indicating whether a transaction is currently changing the end of the visibility interval.
2. If the 1-bit flag is set, the remaining 63 bits contain a unique *transaction identifier* that corresponds to the transaction changing the visibility of this version. If the 1-bit flag is unset, the remaining 63 bits store a *timestamp*, which indicates the latest logical read time this version is visible to transactions.

Both fields are 64 bits wide so as to be changed atomically with the atomic compare-and-swap instruction provided by the underlying hardware. This eliminates the need to acquire latches to read or change the visibility of a version and greatly improves concurrency.

4.3.2 Storage and indexing

Our prototype currently supports only hash indexes which are implemented using lock-free hash tables. A table can have many indexes, and records are always accessed via an index lookup; there is no direct access to a record without going through an index. (To scan a table, one simply scans all buckets of any index on the table.) The techniques presented here are general and can be applied to ordered indexes implemented by trees or skip lists. Figure 4.1 shows a simple bank account table containing six versions. Ignore the numbers (100) in red for now. The table has two (user) columns: Name and Amount. Each version has a valid time defined by timestamps stored in the `Begin` and `End` fields. A table can have several indexes, each one with a separate (hash) pointer field. The example table has one index on the `Name` column. For simplicity we assume that the hash function just returns the first letter of the name. Versions that hash to the same bucket are linked together using the `HashPtr` field.

Hash bucket J contains four records: three versions for John and one version for Jane. The order in which the records appear is immaterial.

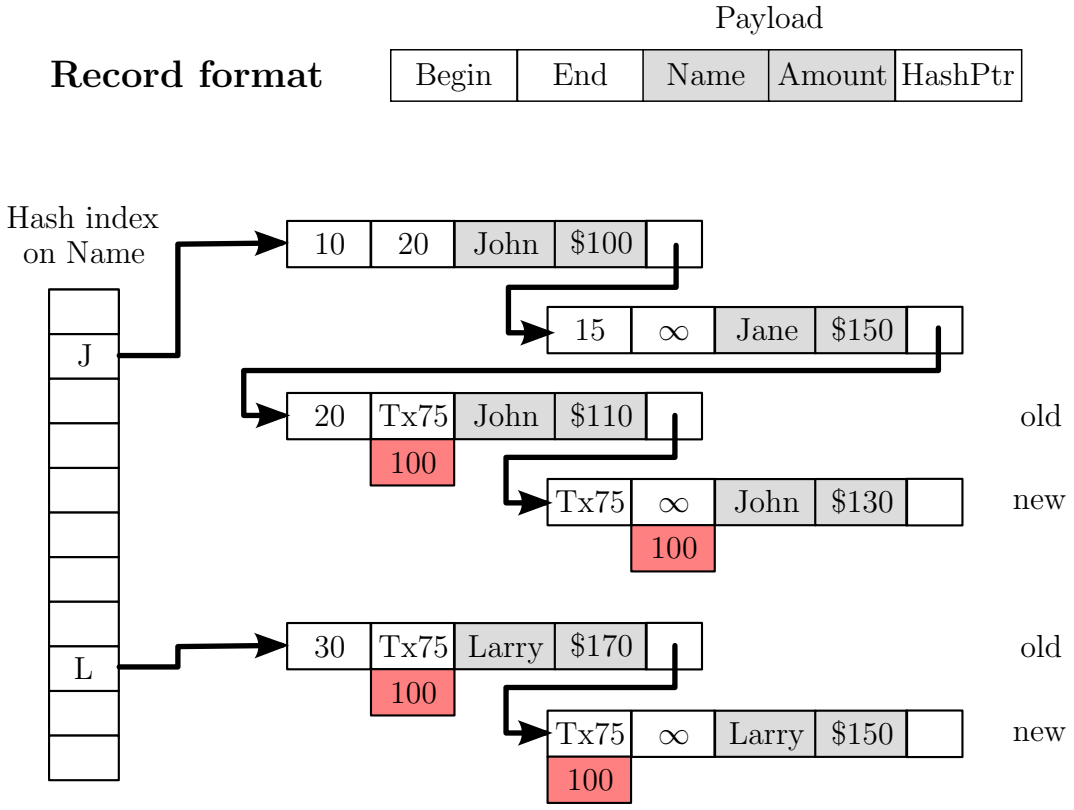


Figure 4.1: Example account table with one hash index. Transaction 75 has transferred $20 from Larry's account to John's account but has not yet committed.

Jane's single version (Jane, 150) has a valid time from 15 to infinity meaning that it was created by a transaction that committed at time 15 and it is still valid. John's oldest version (John, 100) was valid from time 10 to time 20 when it was updated. The update created a new version (John, 110) that initially had a valid time of 20 to infinity. We will discuss John's last version (John, 130) in a moment.

4.3.3 Reads

Every read specifies a logical (as-of) read time and only versions whose valid time overlaps the read time are visible to the read; all other versions are ignored. Different versions of the same record have non-overlapping valid times so at most one version of a record is visible to a read. A lookup for John, for example, would be handled by a scan of bucket J that checks every version in the bucket but returns only the one whose valid time overlaps the read time.

4.3.4 Updates

Bucket L contains two records which both belong to Larry. Transaction 75 is in the process of transferring $20 from Larry's account to John's account. It has created the new versions for Larry (Larry, 150) and for John (John, 130) and inserted them into the appropriate buckets in the index.

Note that transaction 75 has stored its transaction identifier in the `Begin` and `End` fields of the new and old versions, respectively. A transaction identifier stored in the `End` field serves as a write lock and prevents other transactions from updating the same version and it identifies which transaction has updated it. A transaction identifier stored in the `Begin` field informs readers that the version may not yet be committed and it identifies which transaction owns the version.

Now suppose transaction 75 commits with validation timestamp 100. It

then returns to the old and new versions and sets the `Begin` and `End` fields, respectively, to 100. The final values are shown in red below the old and new versions. The old version (John, 110) now has the valid time 20 to 100 and the new version (John, 130) has a valid time from 100 to infinity.

Every update creates a new version so we need to discard old versions that are no longer needed to avoid filling up memory. A version can be discarded when it is no longer visible to any transaction. In our prototype, once a version has been identified as garbage, collection is handled cooperatively by all threads. Although garbage collection is efficient and fully parallelizable, keeping track of the old versions does consume some processor resources.

4.3.5 Transaction phases

A transaction can be in one of four states: Active, Preparing, Committed, or Aborted. Figure 4.2 shows the possible transitions between these states.

A transaction has two unique timestamps: (1) a *start timestamp* reflects the logical time the transaction started executing, and (2) a *validation timestamp* reflects the final serialization order of this transaction with respect to all other transactions in the system. The timestamp is acquired by atomically reading and incrementing a single, global, and monotonically increasing counter.

Property 4.1. *All timestamps are assigned by atomically reading and incrementing a global counter.*

Property 4.2. *Every transaction has a unique start timestamp ST and a unique validating timestamp VT, with* $ST < VT$.

A transaction goes through three different phases. We outline the processing in each phase only briefly here; it is fleshed out in more detail in connection with each concurrency control method in Sections 4.4 and 4.5.

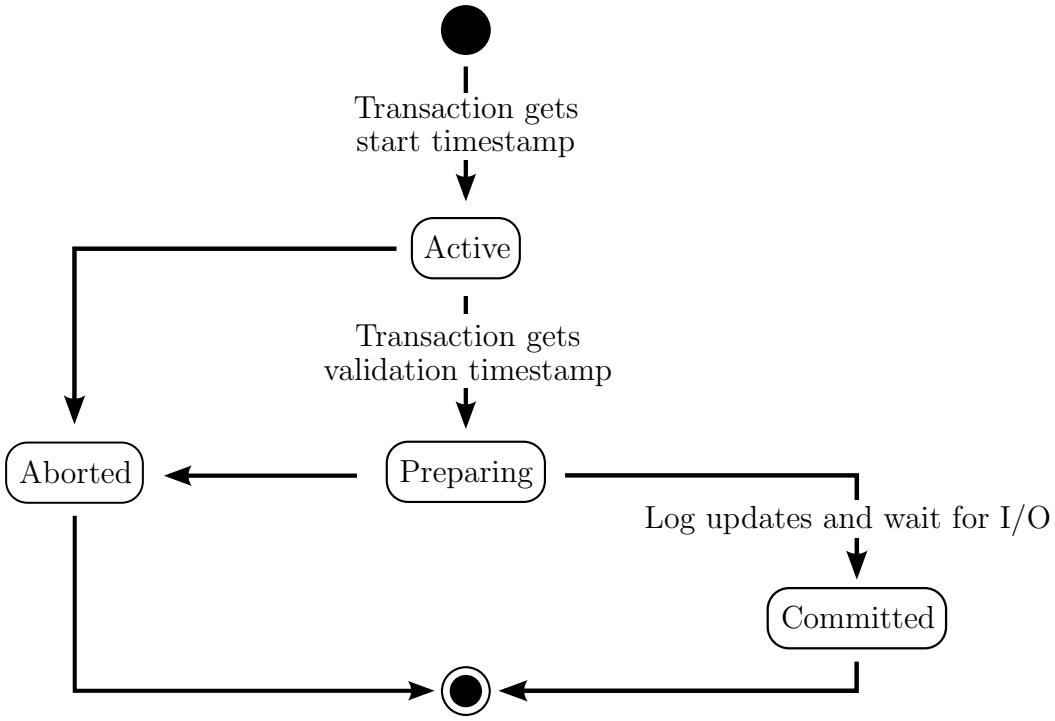


Figure 4.2: State transitions for each transaction.

1. The transaction is created; it acquires a unique *start timestamp* and sets its state to Active.

2. **Normal processing phase.** The transaction does all its normal processing during this phase. A transaction never blocks during this phase. For update operations, the transaction copies its transaction identifier into the `Begin` field of the new versions and into the `End` field of the old or deleted versions. If it aborts, it changes its state to Aborted and skips directly to step 4. When the transaction has completed its normal processing and requests to commit, it acquires a unique *validation timestamp* and switches to the Preparing state.

3. **Preparation phase.** During this phase the transaction determines whether it can commit or is forced to abort. If it has to abort, it switches its state to Aborted and continues to the next phase. If it is ready to commit, it writes information about all its new versions and deleted versions to a redo log record and waits for the log record to reach non-volatile storage. The transaction then switches its state to Committed.

4. **Postprocessing phase.** If the transaction has committed, it proceeds to replace its transaction identifier with its validation timestamp from both the `Begin` field of the new versions and from the `End` field of the old or deleted versions. If the transaction has aborted, it marks all its new versions as garbage and discards them immediately.

5. The transaction is terminated. The old versions are assigned to the garbage collector, which is responsible for discarding them when they are no longer needed.

4.3.6 Version visibility

A read must specify a logical read time under multiversioning. Only versions whose valid time overlaps the logical read time are visible to the read. The read time can be any value between the transaction's start timestamp and the current time. Which read time is chosen depends on the concurrency control method used and the transaction's isolation level; more about this in Sections 4.4 and 4.5.

While determining the visibility of a version is straightforward in principle, it is more complicated in practice as we do not want a transaction to block (wait) at any time during normal processing. Recall that a version's `Begin` or `End` fields can temporarily store a transaction identifier, if the version is being updated. If a reader encounters such a version, determining visibility without blocking requires checking another transaction's state and validation timestamp and potentially even restricting the serialization or-

der of transactions. (We discuss this mechanism in detail in Section 4.3.8.) Table 4.1 summarizes the four possible cases.

We now examine each case in turn, beginning from the easiest and most common case where both fields contain a timestamp. We then discuss the cases where the `Begin` or `End` fields contain transaction identifiers.

Both `Begin` and `End` fields contain timestamps

In the absence of concurrent updates, both the `Begin` and `End` fields of a version contain timestamps. In this common case, a simple comparison suffices to determine whether the version is visible. Let RT denote the logical read time being used by transaction T. To determine whether version V is visible to T, we check V's `Begin` and `End` fields. If both fields contain timestamps, V is visible to T if and only if RT falls between the two timestamps.

`Begin` field contains a transaction identifier

When reading a version shortly after its creation, its `Begin` field may transiently contain a transaction identifier and not a timestamp. Suppose transaction T reads version V and finds that V's `Begin` field contains the identifier of a transaction TB. Whether version V is visible depends on transaction TB's state and TB's validation timestamp. For example, TB may have committed already but not yet finalized the `Begin` fields of its new versions. If so, V is a committed version with a well-defined `Begin` timestamp. Table 4.2 summarizes the cases that may occur and the action to take depending on the state of the transaction TB.

If transaction TB is still in the Active state, the version is uncommitted and thus is not visible to any other transaction. (If T = TB, T can still see its own updates.) If TB has updated a record multiple times, only the latest version is visible to it.

V's `Begin` field is	V's `End` field is	Outcome
Timestamp	Timestamp	V is visible if and only if `Begin` $<$ RT $<$ `End`.
Transaction identifier	Timestamp	Visibility depends on the state of the transaction whose identifier is stored in the `Begin` field. (See Table 4.2 for the analysis.) If needed, proceed to check whether RT $<$ `End` to determine visibility.
Timestamp	Transaction identifier	If RT $<$ `Begin`, V is not visible. Otherwise, visibility depends on the state of the transaction whose identifier is stored in the `End` field. (See Table 4.3 for the analysis.)
Transaction identifier	Transaction identifier	Visibility depends on the state of the transactions whose identifiers are stored in the `Begin` and `End` fields. First, follow the case analysis shown in Table 4.2. Continue, if needed, with the analysis shown in Table 4.3 to determine visibility.

Table 4.1: Case analysis of action to take when transaction T checks visibility of version V as of logical read time RT.

If TB's state is	Action to take when transaction T checks visibility of version V.
Active	If T = TB, treat V's `Begin` field as if it contains a timestamp of zero when testing for visibility. Otherwise, V is not visible.
Preparing	T *speculatively reads* V using TB's validation timestamp as V's `Begin` field when testing for visibility.
Committed	Use TB's validation timestamp as V's `Begin` field to test visibility.
Aborted	Ignore V.
Terminated or TB not found	Check visibility of V again.

Table 4.2: Case analysis of action to take when transaction T checks visibility of version V, and the `Begin` field of version V contains the identifier of transaction TB.

If TB is in the Preparing state, is is unknown whether TB will eventually commit or abort. A safe approach in this situation would be to have transaction T wait until transaction TB either commits or aborts. However, we want to avoid all blocking during normal processing, so instead T *speculates* that TB will commit. If TB commits, V's `Begin` timestamp field will contain TB's validation timestamp. T can therefore speculatively use TB's validation timestamp as V's `Begin` timestamp and continue with the visibility test. To guarantee serializability, transaction T acquires a *commit dependency* on TB, restricting the serialization order of the two transactions. (That is, once T takes a commit dependency on TB, T is allowed to commit only if TB commits.) Commit dependencies are discussed in more detail in Section 4.3.8.

If TB is in the Committed state, V's `Begin` timestamp field will soon

be changed to TB's validation timestamp. T can therefore proceed and use TB's validation timestamp as V's `Begin` timestamp to check for visibility. There is no need to constrain the serialization order by acquiring a commit dependency in this case, as it is certain that TB has already committed.

If TB is Aborted, this means that transaction T read a version V that was created by TB, but will soon be discarded as part of TB's post-processing. T ignores V in this case.

Finally, there is the rare case where TB's state is Terminated or TB can not be found in the system. This occurs only when TB completes all post-processing in the very short time interval between when T checked V's `Begin` field and when T checked TB's state. (Although this is a very short window, the two checks are not atomic.) In this rare case, T simply re-reads version V's `Begin` field and repeats the visibility check.

Property 4.3. *A transaction T with a validation timestamp VT has created a new version V. The new version V is invisible to transactions reading earlier than VT.*

`End` field contains a transaction identifier

Once it has been determined that version V's valid time begins before transaction T's read time RT, we proceed to check V's `End` field. If it contains a timestamp, determining visibility is straightforward: V is visible to T if and only if RT is less than V's `End` timestamp. However, if version V is concurrently being deleted or updated by another transaction TE, the `End` field contains the identifier of transaction TE. In this case we have to check the state and validation timestamp of TE to determine whether V is visible. Table 4.3 summarizes the various cases and the actions to take, assuming that we have already determined that V's begin timestamp is, or will be, less than RT.

If transaction TE is Active, the version V is uncommitted and not visible to any other transaction but TE.

If TE's state is	Action to take when transaction T checks visibility of a version V.
Active	V is visible only if T $\neq$ TE.
Preparing	Let RT be transaction T's logical read time, and TS be transaction TE's validating timestamp. If RT $<$ TS, V is visible. Otherwise, T *speculatively ignores* V.
Committed	Use TE's validation timestamp as V's `End` field when testing for visibility.
Aborted	V is visible.
Terminated or TE not found	Reread V's `End` field.

Table 4.3: Case analysis of action to take when transaction T checks visibility of version V, and the `End` field of version V contains the identifier of transaction TE.

If TE's state is Preparing, it has a validation timestamp TS that will become the `End` timestamp of V if TE commits. If the logical read time RT is before TE's validation timestamp TS, V will be visible regardless whether TE commits or aborts: If TE commits, it will store its validation timestamp TS in the `End` field of V during post-processing. If TE aborts, V will still be visible, because any transaction that updates V after TE has aborted will obtain a validation timestamp greater than TS. If the logical read time RT is after TE's validation timestamp TS, we have a more complicated situation. If TE commits, V will not be visible to T; but if TE aborts, V will be visible. We could handle this by forcing T to wait until TE commits or aborts but we want to avoid all blocking during normal processing. Instead we allow T to *speculatively ignore* V and proceed with its processing: Transaction T acquires a commit dependency on TE, that is, T is allowed to commit only if TE commits. (See Section 4.3.8 for details on the commit dependency

mechanism.)

The case when TE's state is Committed is obvious, but the Aborted case warrants some explanation. If TE has aborted, some other transaction TO may have sneaked in after T read V's `End` field, discovered that TE has aborted and updated V. However, TO must have updated V's `End` field after T read V, and TO must have been in the Active state. TO's validation timestamp would be assigned when switching to the Preparing state. Thus, TO's validation timestamp must be later than T's logical read time. It follows that it doesn't matter if a transaction TO "sneaked in" and updated the version; if TE is in the Aborted state, V is always visible to T.

If TE has terminated or is not found, TE must have finalized V's validation timestamp since we read the field. So we read the `End` field again and try again.

Property 4.4. *An uncommitted version is never visible.*

4.3.7 Updating a version

Suppose transaction T wants to update a version V. The version V is updatable only if it is the latest version, that is, it has an `End` timestamp field equal to infinity, or its `End` field contains the identifier of a transaction TE that is in the Aborted state. If the state of the transaction TE is Active, V is the latest committed version but there is a later uncommitted version. This is a write-write conflict. We follow the first-writer-wins rule and force transaction T to abort.

Property 4.5. *An update or delete to a version is always preceded by a read to the same version. If transaction T creates new version VN, T has first read old version V.*

Suppose transaction T finds that version V is updatable. It will create a new version and proceeds to install it in the database. The first step is

to atomically compare-and-swap T's transaction identifier in V's `End` field to prevent other transactions from updating V. If the swap fails because the `End` field has changed, T must abort because some other transaction has sneaked in and updated V before T managed to install its update. If the swap succeeds, T then connects the new version into all indexes it participates in. T also saves pointers to the old and new versions; they will be needed during post-processing.

Property 4.6. *A transaction will abort if it attempts to update or delete a historic version of a record. A historic version has an* ***`End`*** *field that is either a timestamp not equal to infinity, or it contains a transaction identifier and the referenced transaction has committed.*

4.3.8 Commit dependencies

When a transaction T1 speculates on the outcome of another transaction T2, it is necessary to restrict the serialization order to guarantee correctness. We achieve this by keeping track of the commit dependencies between transactions.

A transaction T1 has a commit dependency on another transaction T2, if T1 is allowed to commit only if T2 commits. If T2 aborts, T1 must also abort, so cascading aborts are possible. T1 acquires a commit dependency either by speculatively reading or speculatively ignoring a version, instead of waiting for T2 to commit. We implement commit dependencies by a register-and-report approach: T1 registers its dependency with T2 and T2 informs T1 when it has committed or aborted. We track commit dependencies with three additional variables for each transaction:

1. A counter, `CommitDepCounter`, that counts how many unresolved commit dependencies a transaction still has. A transaction cannot commit until this counter is zero.

2. A boolean variable, `AbortNow`, that other transactions can set to force this transaction to abort.

3. Each transaction T also has a set, `CommitDepSet`, that stores the identifiers of the transactions that depend on T.

To take a commit dependency on a transaction T2, T1 increments its `CommitDepCounter` and adds its transaction identifier to T2's `CommitDepSet`. If T2 commits, it locates each transaction in its `CommitDepSet` and decrements their `CommitDepCounter`. If T2 aborts, it signals the dependent transactions to also abort by setting their `AbortNow` flags. T2 takes no action if a dependent transaction is not found, as this implies that the dependent has already aborted.

Note that transaction T1 with a commit dependency on T2 may not have to wait at all — T2 may have committed and disappeared before T1 is ready to commit. In essence, commit dependencies consolidate all waits into a single wait and postpone the wait to just before commit.

With the commit dependency mechanism, some transactions may have to wait until the `CommitDepCounter` is zero before they commit. Waiting raises a concern of deadlocks. However, deadlocks cannot occur because a commit dependency is registered only from an Active transaction to a Preparing transaction. Therefore, it follows that an older transaction (lower validation timestamp) will never wait on a younger transaction (higher validation timestamp). If one were to construct a wait-for graph, the direction of edges would always be from a younger transaction to an older transaction, so cycles are impossible.

4.4 Optimistic transactions

This section describes in more detail the processing performed in the different phases for optimistic transactions. We first consider serializable trans-

actions and then discuss lower isolation levels. We then prove that the concurrency control scheme described here admits only 1SR multi-version histories.

The original paper by Kung and Robinson [56] introduced two validation methods: backward validation and forward validation. We use backward validation but optimize it for in-memory storage. Instead of validating a read set against the write sets of all other transactions, we simply check whether a version that was read is still visible as of the end of the transaction. A separate write phase is not needed; a transaction's updates become visible to other transactions when the transaction changes its state to Committed.

A serializable optimistic transaction keeps track of its reads, scans and writes. To this end, a transaction object contains three sets:

1. The **ReadSet** contains pointers to every version read.

2. The **WriteSet** contains pointers to versions updated (old and new), versions deleted (old) and versions inserted (new).

3. The **ScanSet** stores information needed to repeat every scan.

4.4.1 Normal Processing Phase

Normal processing consists of scanning indexes (see Section 4.3.2) to locate versions to read, update, or delete. Insertion of an entirely new record or updating an existing record creates a new version that is added to all indexes for records of that type.

To do an index scan, a transaction T specifies an index, a predicate P, and a logical read time RT. The predicate is a conjunction $P = P_S \wedge P_R$ where P_S is a search predicate that determines what part of the index to scan and P_R is an optional residual predicate. For a hash index, P_S is an

equality predicate on columns of the hash key. For an ordered index, P_S is a range predicate on the ordering key of the index.

We now outline the processing during a scan when transaction T runs at the serializable isolation level. All reads specify the start timestamp of T as the logical read time.

Start scan When a scan starts, it is registered in T's ScanSet so T can check for phantoms during validation. Sufficient information must be recorded so that the scan can be repeated. In our implementation, we record the index the scan operates on, and the predicate P.

Check visibility Next we check whether version V is visible to transaction T as of time RT (see Section 4.3.6). The result of the test may be conditional on another transaction T2 committing. If so, T registers a commit dependency with T2 (see Section 4.3.8). If the visibility test returns false, the scan proceeds to the next version.

Check predicate If a version V doesn't satisfy P, it is ignored and the scan proceeds. If the scan is a range scan and the index key exceeds the upper bound of the range, the scan is terminated. If the scan is an equality predicate for a hash index, the scan is terminated when the end of the hash chain is encountered.

Read version Transaction T records the read by adding a pointer to version V to its ReadSet. The pointer will be used during validation. T can now read V without any further checks.

Check updatability If transaction T intends to update or delete V, we must check whether the version is updatable. A visible version is updatable if its `End` field equals infinity or it contains a transaction identifier and the referenced transaction has aborted. Note that speculative updates are allowed, that is, an uncommitted version can be updated but the transaction that created it must have completed normal processing.

Update version To update version V, transaction T first creates a new version VN and then atomically sets V's `End` field to T's transaction identifier. This fails if some other transaction T2 has already set V's `End` field. This is a write-write conflict and T must abort.

In the most likely outcome, T succeeds in setting V's `End` field. This serves as an exclusive write lock on V because it prevents further updates of V. Transaction T records the update by adding two pointers to its WriteSet: a pointer to V (old version) and a pointer to VN (new version). These pointers are used later for three purposes: (1) for logging new versions during commit, (2) for post-processing after commit or abort, and (3) for locating old versions when they are no longer needed and can be garbage collected.

The new version VN is not visible to any other transaction until T switches to the Preparing state, therefore T can proceed to include VN in all indexes that the table participates in.

Delete version A delete is an update of V that doesn't create a new version. The `End` timestamp of V is first checked and then set in the same way as for updates. If this succeeds, a pointer to V (old version) is added to the write set and the delete is complete.

When transaction T reaches the end of normal processing, it *pre-commits* and enters its preparation phase. Pre-commit simply consists of acquiring the transaction's unique validation timestamp and setting the transaction state to Preparing.

4.4.2 Preparation phase

The preparation phase of an optimistic transaction consists of three steps: read validation, waiting for commit dependencies, and logging. We discuss each step in turn.

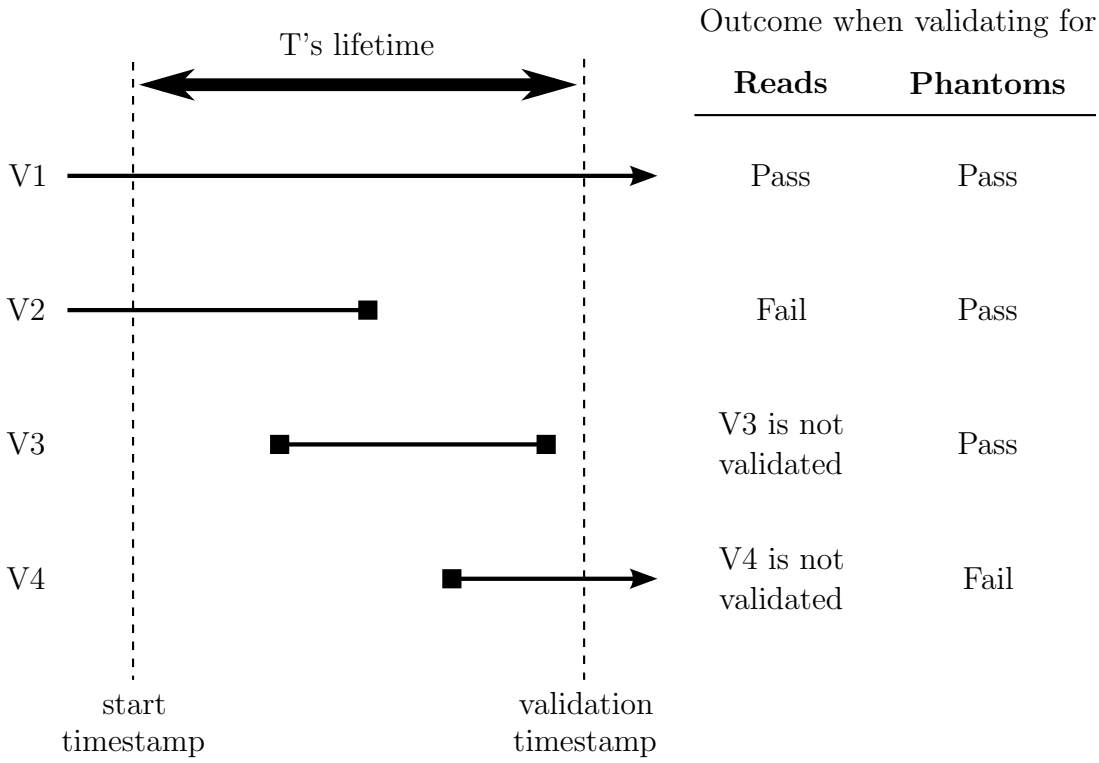


Figure 4.3: Possible transaction validation outcomes.

Validation consists of two steps: checking visibility of the versions read and checking for phantoms. To check visibility, transaction T scans its ReadSet. For each version read, T checks whether the version is still visible as of the end of the transaction. To check for phantoms, T walks its ScanSet and repeats each scan looking for versions that came into existence during T's lifetime and are visible as of the end of the transaction. (T may acquire additional commit dependencies during validation but only if it speculatively ignores a version.)

Figure 4.3 illustrates the different cases that can occur. It shows the lifetime of a transaction T, the valid times of versions V1 to V4 of four

different records, and the expected outcome of read validation and phantom detection. We assume that all four versions satisfy the search predicate used by T and that they were all created and terminated by transactions other than T.

Version V1 is visible to T both at its start and validation timestamp. If V1 is included in T's ReadSet, it passes read validation and also phantom detection.

Version V2 is visible to T as of its start timestamp but not at the end of the transaction (that is, as of the validation timestamp). If V2 is included in T's ReadSet, it fails read validation. V2 is not a phantom.

Version V3 both began and ended during T's lifetime, so it is not visible to T at the start nor at the end of the transaction. It is not included in T's ReadSet so it won't be subject to read validation. V3 is not visible at the end of T, so V3 is not a phantom.

Version V4 was created during T's lifetime and is visible at the end of T, so V4 is a phantom. It is not included in T's ReadSet because it was not visible as of T's start time.

If T fails validation, it is not serializable and must abort. If T passes validation, it must wait for outstanding commit dependencies to be resolved, if it has any. More specifically, T can proceed to the post-processing phase if either its CommitDepCounter is zero or its AbortNow flag is set.

Property 4.7. *Let transaction T have a start timestamp ST and a validating timestamp VT. Every read that transaction T performed during the Active phase is associated with some logical read time RT, such that* $ST < RT < VT$*. During the Preperation phase, every read will be repeated as of logical time VT. Transaction T aborts if the read as of logical time RT and the read as of logical time VT return different versions.*

To complete the commit, T scans its WriteSet and writes the new versions it created to a persistent log. Commit ordering is determined by transaction validation timestamps, which are included in the log records

so that multiple log streams on different devices can be used to ameliorate bottlenecks due to logging [53]. Deletes are logged by writing a unique key or, in the worst case, all columns of the deleted version. After the log writes have been completed and the log record is stored in non-volatile storage, T sets its transaction state to Committed, thereby signaling the end of this phase.

4.4.3 Postprocessing

During this phase a committed transaction TC propagates its validating timestamp to the `Begin` and `End` fields of new and old versions, respectively, listed in its WriteSet. An aborted transaction TA can discard all the new versions it created immediately. In addition, transaction TA attempts to reset the `End` fields of any old versions to infinity. However, another transaction may already have detected that TA aborted, created another new version and reset the `End` field of the old version. If so, TA leaves the `End` field unchanged.

The transaction then processes all outgoing commit dependencies listed in its `CommitDepSet`. If it aborted, it forces the dependent transactions to also abort by setting their `AbortNow` flags. If it committed, it decrements the target transaction's `CommitDepCounter` and wakes up the dependent transaction if the count becomes zero.

Once post-processing is done, other transactions no longer need to refer to the transaction object. It can be removed from the transaction table but it will not be discarded entirely; the pointers to old versions in its WriteSet are needed for garbage collection.

4.4.4 Lower isolation levels

Enforcing lower isolation levels requires less bookkeeping and fewer instructions. A transaction requiring a higher isolation level bears the full cost

of enforcing stronger isolation and does not burden transactions running in lower isolation levels.

Repeatable read Repeatable read is required to enforce read stability but not to prevent phantoms. We implement repeatable read simply by validating a transaction's ReadSet before commit. As phantom detection is not required, a transaction's scans are not recorded. The transaction's start timestamp is used as the logical read time.

Read committed Read committed guarantees that only committed versions are read. We implement read committed by always using the current time as the logical read time. No validation is required, as uncommitted versions will never be visible (see Section 4.3.6). A transaction's reads and scans are not recorded at this isolation level.

Snapshot isolation Implementing snapshot isolation with a multi-version storage engine is straightforward: always read as of the start timestamp of the transaction. No validation is needed, so scans and reads are not tracked.

Read-only transactions If a transaction is known to be read-only, the best performance is obtained by running it under snapshot isolation or read committed depending on whether it needs a transaction-consistent view or not.

4.4.5 Correctness proof

We now prove that the multi-version optimistic scheduler only admits 1SR multi-version histories. We use the notation from Section 5.2 of [13].

The multi-vesrion serialization graph $MVSG(H, \ll)$ is a graph defined on a multi-version history H and a version total order $\ll$. Each node in the $MVSG$ is a committed transaction in H. By definition, the $MVSG$ has an edge from transaction T_i to transaction T_j ($i \neq j \neq k$) if and only if:

1. T_i writes version V_i and T_j reads version V_i, or

2. T_i writes version V_i and T_k reads version V_j, where $V_i \ll V_j$, or

3. T_i reads vesrion V_k and T_j writes version V_j, where $V_k \ll V_j$.

We will now prove that there are no cycles in $MVSG$ because every edge in $MVSG$ is ordered with respect to the validating timestamp order of the transactions involved. Let $E(T)$ denote the validating timestamp of transaction T. We prove that an edge $T_i \rightarrow T_j$ exists in $MVSG$ if and only if $E(T_i) < E(T_j)$.

1. We start with the first case, where an edge is in $MVSG$ because T_i writes version V_i and T_j reads version V_i. Suppose T_j read V_i at timestamp $R(T_j)$. From Property 4.7, it follows that $R(T_j) < E(T_j)$. If $R(T_j) < E(T_i)$, it would have been impossible for T_j to read V_i, as it is uncommitted (Properties 4.3 and 4.4). Therefore $E(T_i) \leq R(T_j)$, so $E(T_i) < E(T_j)$.

2. The second case occurs when T_i writes version V_i and T_k reads version V_j, where $V_i \ll V_j$. Suppose T_k read V_j at timestamp $R(T_k)$. As T_k reads V_j and uncommitted versions are invisible (Property 4.4), it follows that some transaction T_j created V_j and committed successfully. From Property 4.7, $E(T_j) \leq R(T_k)$, otherwise V_j would have not been visible to T_k as of time $R(T_k)$. Furthermore, from Property 4.7, it follows that $R(T_j) < E(T_j)$. There exists no version V_k such that $V_i \ll V_k \ll V_j$, otherwise T_j would have aborted during the update (Property 4.6). Since T_j wrote V_j, that implies that T_j read V_i (Property 4.5), hence $E(T_i) \leq R(T_j)$. Therefore, $E(T_i) < E(T_j)$.

3. The third case occurs when T_i reads vesrion V_k and T_j writes version V_j, where $V_k \ll V_j$. Suppose T_i read V_k at $R(T_i)$. From Property 4.7, $R(T_i) < E(T_i)$. Suppose that $E(T_j) < E(T_i)$. This ordering is impossible

if T_j and T_i are committed transactions from Property 4.7: Transaction T_i would validate whether V_k is still visible as of $E(T_i)$. T_j has committed and $V_k \ll V_j$, therefore V_k is no longer visible as of $E(T_i)$ and validation would fail. T_i would abort and would never be a node in $MVSG$. Therefore, $E(T_i) < E(T_j)$.

We have shown that an edge $T_i \rightarrow T_j$ exists in $MVSG$ if and only if $E(T_i) < E(T_j)$, where $E(T)$ is the validating timestamp of committed transaction T. As validating timestamps are integers that are assigned from a monotonically increasing counter (Property 4.1), edges in $MSVG$ cannot be involved in a cycle. Hence, the multi-version histories accepted by our multi-version optimistic concurrency control scheduler are 1SR.

4.5 Pessimistic transactions

An optimistic transaction running at a higher isolation level may find its reads being invalidated repeatedly from short transactions. This can cause large read-mostly transactions to starve. Instead of going through a validation phase at the end, a transaction can choose to be *pessimistic* and acquire read locks to prevent its transactional reads from being invalidated. This section describes our design for multiversion locking optimized for main-memory databases.

We first describe the additional data structures and lock types needed to efficiently implement multiversion locking. We then summarize the processing phases a pessimistic transaction goes through, and highlight the differences compared to the optimistic scheme.

A serializable pessimistic transaction must keep track of which versions it read, which hash buckets it scanned, and its new and old versions. To this end, the transaction maintains three sets:

1. The **ReadSet** contains pointers to versions read-locked by the transac-

tion.

2. The **BucketLockSet** contains pointers to hash buckets visited and locked by the transaction.

3. The **WriteSet** contains references to versions updated (old and new), versions deleted (old) and versions inserted (new).

4.5.1 Lock types

We use two types of locks: record locks and bucket locks. Record locks are placed on versions to ensure read stability. Bucket locks are placed on (hash) buckets to prevent phantoms. The name reflects their use for hash indexes in our prototype but range locks for ordered indexes can be implemented in the same way.

Record locks

Updates or deletes can only be applied to the latest version of a record; older versions cannot be further updated. Thus, locks are required only for the latest version of a record; never for older versions. So what's needed is an efficient many-readers-single-writer lock for this case.

We do not want to store record locks in a separate table — it's too slow. Instead we embed record locks in the `End` field of versions so no extra space is required. In our prototype, the `End` field of a version is 64 bits. As described earlier, this field stores either a timestamp or a transaction identifier with one bit indicating what the field contains. We change how we use this field to make room for a record lock.

1. ContentType (1 bit): indicates the content type of the remaining 63 bits.

2. Timestamp (63 bits): when ContentType is zero.

3. RecordLock (63 bits): when ContentType is one.

3.1. NoMoreReadLocks (1 bit): a flag set when no further read locks are accepted. Used to prevent starvation.

3.2. ReadLockCount (8 bits): number of read locks.

3.3. WriteLock (54 bits): identifier of the transaction holding a write lock on this version or infinity (max value).

We do not explicitly keep track of which transactions have a version read locked. Each transaction records its ReadSet so we can find out by checking the ReadSets of all current transactions. This is only needed for deadlock detection which occurs infrequently.

A transaction acquires a read lock on a version V by atomically incrementing V's ReadLockCount. No further read locks can be acquired if the counter has reached its max value (255) or the NoMoreReadlocks flag is set. If so, the transaction aborts.

A transaction write locks a version V by atomically copying its transaction identifier into the WriteLock field. This action both write locks the version and identifies who holds the write lock.

Bucket Locks (Range Locks)

Bucket locks are used only by serializable transactions to prevent phantoms. When a transaction TS begins a scan of a hash bucket, it locks the bucket. Multiple transactions can have a bucket locked. A bucket lock consists of the following two fields:

1. LockCount: number of locks on this bucket.

2. LockList: list of (serializable) transactions holding a lock on this bucket.

The current implementation stores the LockCount in the hash bucket to be able to check quickly whether the bucket is locked. LockLists are implemented as arrays stored in a separate hash table with the bucket address as the key.

To acquire a lock on a bucket B, a transaction TS increments B's LockCount, locates B's LockList, and adds its transaction Id to the list. To release the lock it deletes its transaction identifier from B's LockList and decrements the LockCount.

Range locks in an ordered index can be implemented in the same way. If the index is implemented by a tree structure, a lock on a node locks the subtree rooted at that node. If the index is implemented by skip lists, locking a tower locks the range from that tower to the next tower of the same height.

4.5.2 Eager Updates, Wait-For Dependencies

In a traditional implementation of multiversion locking, an update transaction TU would block if it attempts to update or delete a read locked version or attempts to insert or update a version in a locked bucket. This may lead to frequent blocking and thread switching. A thread switch is expensive, costing several thousand instructions. In a main-memory system, just a few thread switches can add significantly to the cost of executing a transaction.

To avoid blocking we allow a transaction TU to eagerly update or delete a read locked version V but, to ensure correction serialization order, TU cannot precommit until all read locks on V have been released. Similarly, a transaction TR can acquire a read lock on a version that is already write locked by another transaction TU. If so, TU cannot precommit until TR has released its lock.

Note that an eager update or delete is not speculative because it doesn't matter whether TR commits or aborts; it just has to complete and release its read lock. The same applies to locked buckets. Suppose a bucket B is locked by two (serializable) transactions TS1 and TS2. An update transaction TU is allowed to insert a new version into B but it is not allowed to precommit before TS1 and TS2 have completed and released their bucket locks.

We enforce correct serialization order by wait-for dependencies. A wait-

for dependency forces an update transaction TU to wait before it can acquire a validation timestamp and begin commit processing. There are two flavors of wait-for dependencies, read lock dependencies and bucket lock dependencies that differ in what event they wait on.

A transaction T needs to keep track of both incoming and outgoing wait-for dependencies. T has an incoming dependency if it waits on some other transaction and an outgoing dependency if some other transaction waits on it. To track wait-for dependencies, the following fields are included in each transaction object.

1. WaitForCounter: indicates how many incoming dependencies the transaction is waiting for.

2. NoMoreWaitFors: when set the transaction does not allow additional incoming dependencies. Used to prevent starvation by incoming dependencies continuously being added.

3. WaitingTxnList: Tracks outgoing wait-for dependencies by storing the identifiers of transactions waiting on this transaction to complete.

Read lock dependencies

A transaction TU that updated or deleted a version V has a wait-for dependency on V as long as V is read locked. TU is not allowed to acquire a validation timestamp and begin commit processing unless V's ReadLockCount is zero.

When a transaction TU updates or deletes a version V, it acquires a write lock on V by copying its transaction identifier into the WriteLock field. If V's ReadLockCount is greater than zero, TU takes a wait-for dependency on V simply by incrementing its WaitForCounter.

TU may also acquire a wait-for dependency on V by another transaction TR taking a read lock on V. A transaction TR that wants to read a version V

must first acquire a read lock on V by incrementing V's ReadLockCount. If V's NoMoreReadLocks flag is set or ReadLockCount is at max already, lock acquisition fails and TR aborts. Otherwise, if V is not write locked or V's ReadLockCount is greater than zero, TR increments V's ReadLockCount and proceeds. However, if V is write locked by a transaction TU and this is the first read lock on V (V's ReadLockCount is zero), TR must force TU to wait on V. TR checks TU's NoMoreWaitFors flag. If it is set, TU cannot install the wait-for dependency and aborts. Otherwise everything is in order and TR acquires the read lock by incrementing Vs' ReadLockCounter and installs the wait-for dependency by incrementing TU's WaitForCounter.

When a transaction TR releases a read lock on a version V, it may also need to release a wait-for dependency. If V is not write locked, TR simply decrements V's ReadLockCounter and proceeds. The same applies if V is write locked and V's ReadLockCounter is greater than one. However, if V is write locked by a transaction TU and V's ReadLockCounter is one, TR is about to release the last read lock on V and therefore must also release TU's wait-for dependency on V. TR atomically sets V's ReadLockCounter to zero and V's NoMoreReadLocks to true. If this succeeds, TR locates TU and decrements TU's WaitForCounter.

Setting the NoMoreReadLocks flag before releasing the wait-for dependency is necessary because this may be TU's last wait-for dependency. If so, TU is free to acquire a validation timestamp and being its commit processing. In that case, TU's commit cannot be further postponed by taking out a read lock on V. In other words, further read locks on V would have no effect.

Bucket lock dependencies

A serializable transaction TS acquires a lock on a bucket B by incrementing B's LockCounter and adding its transaction identifier to B's LockList. The purpose of TR's bucket lock is not to disallow new versions from being

added to B but to prevent them from becoming visible to TR. That is, another transaction TU can add a version to B but, if it does, then TU cannot precommit until TS has completed its processing and released its lock on B. This is enforced by TU obtaining a wait-for dependency on TS.

TU can acquire this type of dependency either by acquiring one itself or by having one imposed by TS. We discuss each case.

Suppose that, as a result of an update or insert, TU is about to add a new version V to a bucket B. TU checks whether B has any bucket locks. If it does, TU takes out a wait-for dependency on every transaction TS in B's LockList by adding its own transaction identifier to TS's WaitForList and incrementing its own WaitForCounter. If TU's NoMoreWaitFors flag is set, TU can't take out the dependency and aborts.

Suppose a serializable transaction TS is scanning a bucket B and encounters a version V that satisfies TS's search predicate but the version is not visible to TS, that is, V is write locked by a transaction TU that is still active. If TU commits before TS, V becomes a phantom to TS. To prevent this from happening, TS registers a wait-for dependency on TU's behalf by adding TU's transaction identifier to its own WaitingTxnList and incrementing TU's WaitForCounter. If TU's NoMoreWaitFors flag is set, TS can't impose the dependency and aborts.

When a serializable transaction TS has precommitted and acquired its validation timestamp, it releases its outgoing wait-for dependencies. It scans its WatingTxnList and, for each transaction T found, decrements T's WaitForCounter.

4.5.3 Processing phases

This section describes how locking affects the processing done in the different phases of a transaction.

Normal processing phase

Recall that normal processing consists of scanning indexes to select record versions to read, update, or delete. An insertion or update creates a new version that has to be added to all indexes for records of that type.

We now outline what a pessimistic transaction T does differently than an optimistic transaction during a scan and how this depends on T's isolation level. For snapshot isolation, the logical read time is always the transaction start timestamp. For all other isolation levels, it is the current time which has the effect that the read sees the latest version of a record.

Start scan If T is a serializable transaction, it takes out a bucket lock on B to prevent phantoms and records the lock in its BucketLockSet. Other isolation levels do not take out a bucket lock.

Check predicate This phase is the same as for optimistic transactions. (Refer to Section 4.4.1 for details.)

Check visibility This is done in the same way as for optimistic transaction, including taking out commit dependencies as needed. If a version V is not visible, it is ignored and the scan continues for all isolations levels except serializable. If T is serializable and V is write locked by a transaction TU that is still active, V is a potential phantom so T forces TU to delay its precommit by imposing a wait-for dependency on TU.

Read version If T runs under serializable or repeatable read and V is a latest version, T attempts to acquire a read lock on V. If T can't acquire the read lock, it aborts. If T runs under a lower isolation level or V is not a latest version, no read lock is required.

Check updatability This phase is the same as for optimistic transactions. (Refer to Section 4.4.1 for details.)

Update version As for optimistic transactions, T creates a new version N, sets V's WriteLock and, if V was read locked, takes out a wait-for dependency on V by incrementing its own WaitForCounter. T then proceeds to add N to all indexes it participates in. If T adds N to a locked index bucket B, it takes out wait-for dependencies on all (serializable) transactions holding locks on B.

Delete version A delete is essentially an update of V that doesn't create a new version. T sets V's WriteLock and if V was read locked, takes out a wait-for dependency on V by incrementing its own WaitForCounter.

When transaction T reaches the end of normal processing, it releases its read locks and its bucket locks, if any. If it has outstanding wait-for dependencies (its WaitForCounter is greater than zero), it waits. Once its WaitForCounter is zero, T precommits, that is, acquires a validation timestamp and sets its state to Validating.

Preparation phase

Pessimistic transactions require no validation that's taken care of by locks. However, a pessimistic transaction T may still have outstanding commit dependencies when reaching this point. If so, T waits until they have been resolved and then proceeds to write to the log and commit. If a commit dependency fails, T aborts.

Postprocessing phase

Postprocessing is the same as for optimistic transactions. Note that there is no need to explicitly release write locks; this is automatically done when the transaction updates `Begin` and `End` fields.

4.5.4 Deadlock detection

Commit dependencies are only taken on transactions that have already precommitted and are completing validation. As discussed earlier Section 4.3.8 commit dependencies cannot cause or be involved in a deadlock.

Wait-for dependencies, however, can cause deadlocks. To detect deadlocks we build a standard wait-for graph by analyzing the wait-for dependencies of all transactions that are currently blocked. Once the wait-for graph has been built, any algorithm for finding cycles in graphs can be used. Our prototype uses Tarjan's algorithm [75] for finding strongly connected components.

A wait-for graph is a directed graph with transactions as nodes and waits-for relationships as edges. There is an edge from transaction T2 to transaction T1 if T2 is waiting for T1 to complete. The graph is constructed in three steps.

Create nodes Scan the transaction table and for each transaction T found, create a node N(T) if T has completed its normal processing and is blocked because of wait-for dependencies.

Create egdes from explicit dependencies Wait-for dependencies caused by bucket locks are represented explicitly in WaitingTxnLists. For each transaction T1 in the graph and each transaction T2 in T1's WaitingTxnList, add an edge from T2 to T1.

Create edges from implicit dependencies A wait-for dependency on a read-locked version V is an implicit representation of wait-for dependencies on all transaction holding read locks on V. We can find out which transactions hold read locks on V by checking transaction read sets. Edges from implicit dependencies can be constructed as follows. For each transaction T1 in the graph and each version V in T1's ReadLockSet: if V is write locked by a transaction T2 and T2 is in the graph, add an edge from T2 to T1.

While the graph is being constructed normal processing continues so wait-for dependencies may be created and resolved and transactions may become blocked or unblocked. Hence, the final graph obtained may be imprecise, that is, it may differ from the graph that would be obtained if normal processing stopped. But this doesn't matter because if there truly is a deadlock neither the nodes nor the edges involved in the deadlock can disappear while the graph is being constructed. There is a small chance of detecting a false deadlock but this is handled by verifying that the transactions participating in the deadlock are still blocked and the edges are still covered by unresolved wait-for dependencies.

4.5.5 Correctness proof

The multi-version locking scheme is a MV2PL scheduler. Section 5.5.2 of [79] describes MV2PL in detail and proves it only admits 1SR multi-version histories. In our implementation, having the NoMoreReadLocks bit set on the latest version of a record (see Section 4.5.2) is equivalent to holding a certify (commit) lock on a data item in MV2PL.

4.5.6 Peaceful coexistence

An interesting property of our design is that optimistic and pessimistic transactions can be mixed. The change required to allow optimistic transactions to coexist with pessimistic transactions is straightforward: optimistic update transactions must honor read locks and bucket locks. Making an optimistic transaction T honor read locks and bucket locks requires the following changes:

1. When T write locks a version V, it uses only a 54-bit transaction identifier and doesn't overwrite read locks.

2. When T updates or deletes a version V that is read locked, it takes a wait-for dependency on V.

3. When T inserts a new version into a bucket B, it checks for bucket locks and takes out wait-for dependencies as needed.

4.6 Experimental results

Our prototype storage engine implements both the optimistic and the pessimistic scheme. We also implemented a single-version storage engine with locking for concurrency control. The implementation is optimized for main-memory databases and does not use a central lock manager, as this can become a bottleneck [52]. Instead, we embed a lock table in every index and assign each hash key to a lock in this partitioned lock table. A lock covers all records with the same hash key which automatically protects against phantoms. We use timeouts to detect and break deadlocks.

The experiments were run on a two-socket Intel Xeon X5650 @ 2.67 GHz (Nehalem) that has six cores per socket. HyperThreading was enabled. The system has 48 GB of memory, 12 MB L3 cache per socket, 256 KB L2 cache per core, and two separate 32 KB L1-I and L1-D caches per core. This is a NUMA system and memory latency is asymmetric: accessing memory on the remote socket is approximately 30% slower than accessing local memory. We size hash tables appropriately so there are no collisions. The operating system is Windows Server 2008 R2.

The I/O bandwidth required for logging is moderate: Each update produces a log record that stores the difference between the old and new versions, plus 8 bytes of metadata. Even with millions of updates per second, the I/O bandwidth required is within what even a single desktop hard drive can deliver. However, logging each transaction individually involves a very high number of I/O operations. For our experiments, each transaction generates log records but these are asynchronously written to durable storage;

transactions do not wait for log I/O to complete. This choice allows us to focus on the effect of concurrency control. Asynchronous logging allows us to submit log records in batches (group commit), greatly reducing the number of I/O operations for our experiments. The emergence of fast non-volatile storage that can sustain a very high number of I/O operations promises to ameliorate this problem in the future.

Traditional disk-based transaction processing systems require hundreds of concurrently active transactions to achieve maximum throughput. This is to give the system useful work to do while waiting for disk I/O. Our main-memory engine does not wait for disk I/O, so there is no need to over-provision threads. We observed that the CPU is fully utilized when the multi-programming level equals the number of hardware threads; allowing more concurrent transactions causes throughput to drop. We therefore limited the number of concurrently active transactions to be at most 24, which matches the number of threads our machine supports.

We experiment with three CC schemes: the single-version locking engine (labeled "1V" the multi-version engine when transactions run optimistically ("MV/O") and the multi-version engine where transactions run pessimistically ("MV/L").

4.6.1 Homogeneous workload

We first show results from a parameterized artificial workload. By varying the workload parameters we can systematically explore how sensitive the different schemes are to workload characteristics. We focus on short update transactions which are common for OLTP workloads. The workload consists of a single transaction type that performs R reads and W writes against a table of N records with a unique key. Each row is 24 bytes, and reads and writes are uniformly and randomly scattered over the N records.

The memory footprint of the database differs for each concurrency control scheme. In 1V, each row consumes 24 bytes (payload) plus 8 bytes for

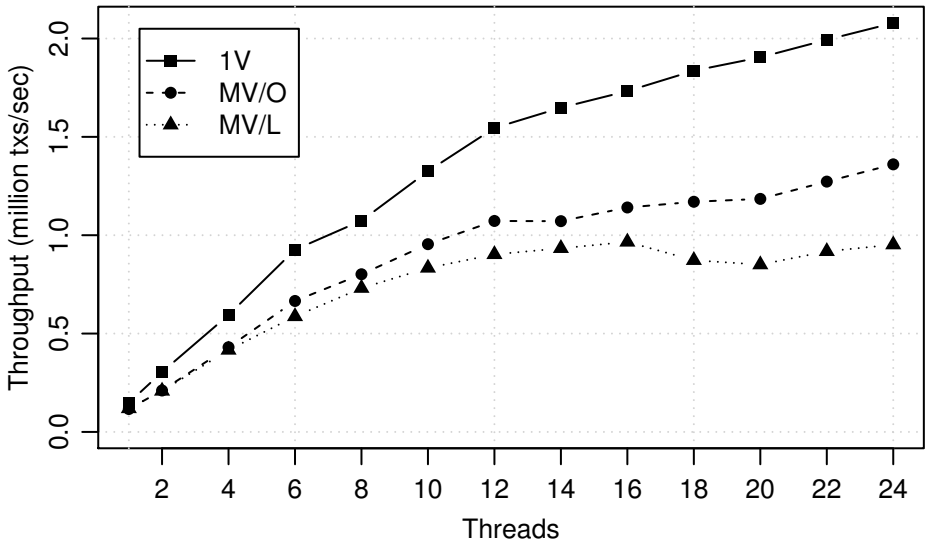


Figure 4.4: Scalability under low contention.

the "next" pointer of the hash table. Both MV schemes additionally use 16 bytes to store the `Begin` and `End` fields (cf. Figure 1), but the total consumption depends on the average number of versions required by the workload. Comparing the two MV schemes, MV/L has the biggest memory footprint, due to the additional overhead of maintaining a bucket lock table.

Scalability (Read Committed)

We first show how transaction throughput scales with increasing multiprogramming level. For this experiment each transaction performs 10 reads and 2 writes (R=10 and W=2) against a table with N=10,000,000 rows. The isolation level is Read Committed.

Figure 4.4 plots transaction throughput (y-axis) as the multiprogramming level increases (x-axis). Under low contention, throughput for all three schemes scales linearly up to six threads. After six threads, we see the effect

of the higher access latency as the data is spread among two NUMA nodes, and beyond twelve threads we see the effect of HyperThreading.

For the 1V scheme, HyperThreading causes the speed-up rate to drop but the system still continues to scale linearly, reaching a maximum of over 2M transactions/sec. The multiversion schemes have lower throughput because of the overhead of version management and garbage collection. Creating a new version for every update and cleaning out old versions that are no longer needed is obviously more expensive than updating in place.

Comparing the two multiversion schemes, MV/L has 30% lower performance than MV/O. This is caused by extra writes for tracking dependencies and locks, which cause increased memory traffic. It takes MV/L 20% more cycles to execute the same number of instructions and the additional control logic translates into 10% more instructions per transaction.

Scaling under contention (Read Committed)

Records that are updated very frequently (hotspots) pose a problem for all CC schemes. In locking schemes, high contention causes transactions to wait because of lock conflicts and deadlocks. In optimistic schemes, hotspots result in validation failures and write-write conflicts, causing high abort rates and wasted work. At the hardware level, some data items are accessed so frequently that they practically reside in the private L1 or L2 caches of each core. This stresses the hardware to the limits, as it triggers very high core-to-core traffic to keep the caches coherent.

We simulate a hotspot by running the same R=10 and W=2 transaction workload from Section 5.1.1 on a very small table with just N=1,000 rows. Transactions run under Read Committed. Figure 4.5 shows the throughput under high contention. Even in this admittedly extreme scenario, all schemes achieve a throughput of over a million transactions per second, with MV/O slightly ahead of both locking schemes. 1V achieves its highest throughput at six threads, then drops and stays flat after 8 threads.

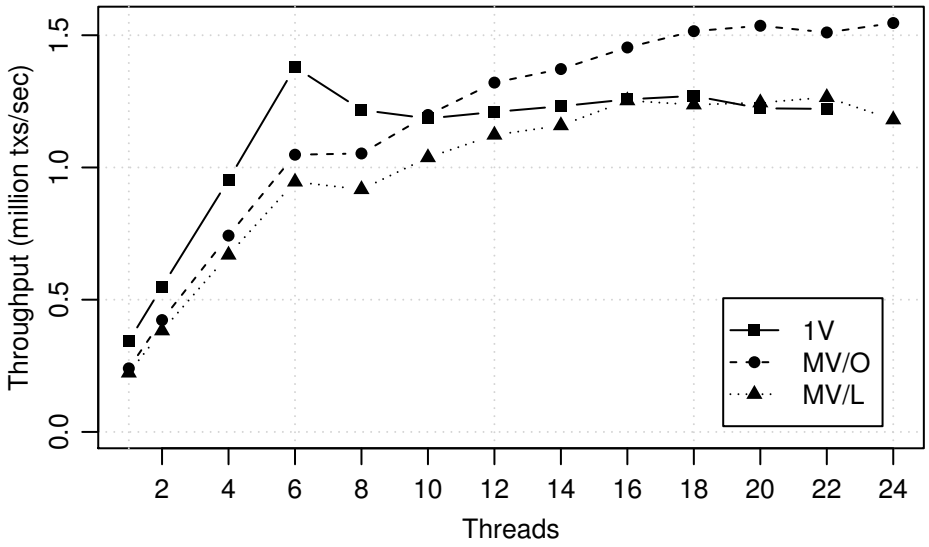


Figure 4.5: Scalability under high contention.

Higher isolation levels

The experiments in the previous section ran under Read Committed isolation level, which is the default isolation level in many commercial database engines [46, 61], as it prevents dirty reads and offers high concurrency. Higher isolation levels prevent more anomalies but reduce throughput. In this experiment, we use the same workload from Section 4.6.1, we fix the multiprogramming level to 24 and we change the isolation level.

In Table 4.4, we report the transaction throughput from each scheme and isolation level. We also report the throughput drop as a percentage of the throughput when running under the Read Committed isolation level.

The overhead for Repeatable Read for both locking schemes is very small, less than 2%. MV/O needs to repeat the reads at the end of the transaction, and this causes an 8% drop in throughput. For Serializable, the 1V scheme protects the hash key with a lock, and this guarantees phan-

	Read Committed	Repeatable Read		Serializable	
	Throughput (tx/sec)	Throughput (tx/sec)	% drop vs RC	Throughput (tx/sec)	% drop vs RC
1V	2,080,492	2,042,540	1.8%	2,042,571	1.8%
MV/L	974,512	963,042	1.2%	877,338	10.0%
MV/O	1,387,140	1,272,289	8.3%	1,120,722	19.2%

Table 4.4: Throughput at higher isolation levels, and percentage drop compared to Read Committed (RC).

tom protection with very low overhead (2%). Both MV schemes achieve serializability at a cost of 10%- 19% lower throughput: MV/L acquires read locks and bucket locks, while MV/O has to repeat each scan during validation. Under MV/O, however, a transaction requesting a higher isolation level bears the full cost of enforcing the higher isolation. This is not the case for locking schemes.

4.6.2 Heterogeneous workload

The workload used in the previous section represents an extreme update-heavy scenario. In this section we fix the multiprogramming level to 24 and we explore the impact of adding read-only transactions in the workload mix.

Impact of short read transactions

In this experiment we change the ratio of read and update transactions. There are two transaction types running under Read Committed isolation: the update transaction performs 10 reads and 2 writes (R=10 and W=2), while the read-only transaction performs 10 reads (R=10 and W=0).

Figure 4.6 shows throughput (y-axis) as the ratio of read-only transactions varies in the workload (x-axis) in a table with 10,000,000 rows. The

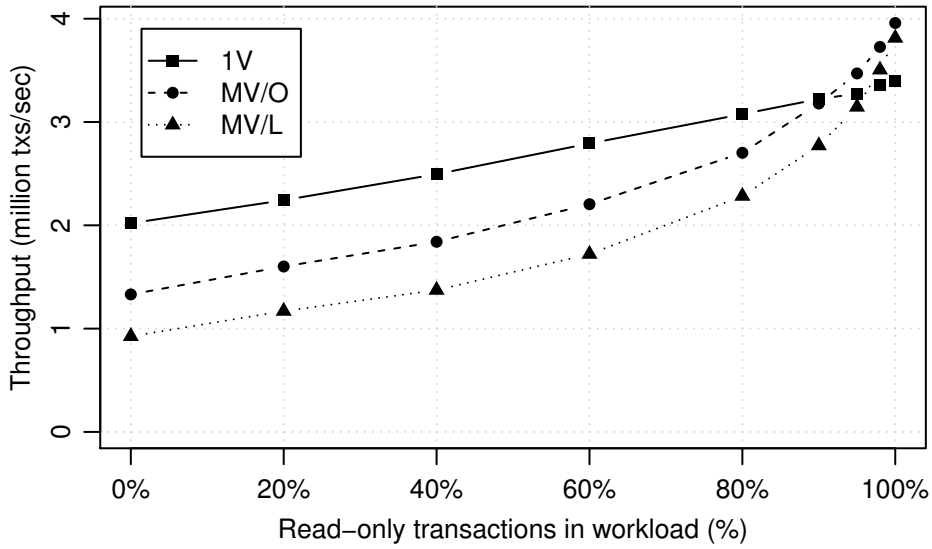


Figure 4.6: Impact of read-only transactions on throughput (low contention).

leftmost point (x=0%) reflects the performance of the update-only workload of Section 4.6.1 at 24 threads. As we add read-only transactions to the mix, the performance gap between all schemes closes. This is primarily because the update activity is reduced, reducing the overhead of garbage collection.

The MV schemes outperform 1V when most transactions are read-only. When a transaction is read-only, the two MV schemes behave identically: transactions read a consistent snapshot and do not need to do any locking or validation. In comparison, 1V has to acquire and release short read locks for cursor stability even for read-only transactions which impacts performance.

In Figure 4.7 we repeat the same experiment but simulate a hotspot by using a table of 1,000 rows. The leftmost point (x=0%) again reflects the performance of the update-only workload of Section 5.1.12 under high contention at 24 threads. The MVCC schemes have a clear advantage at

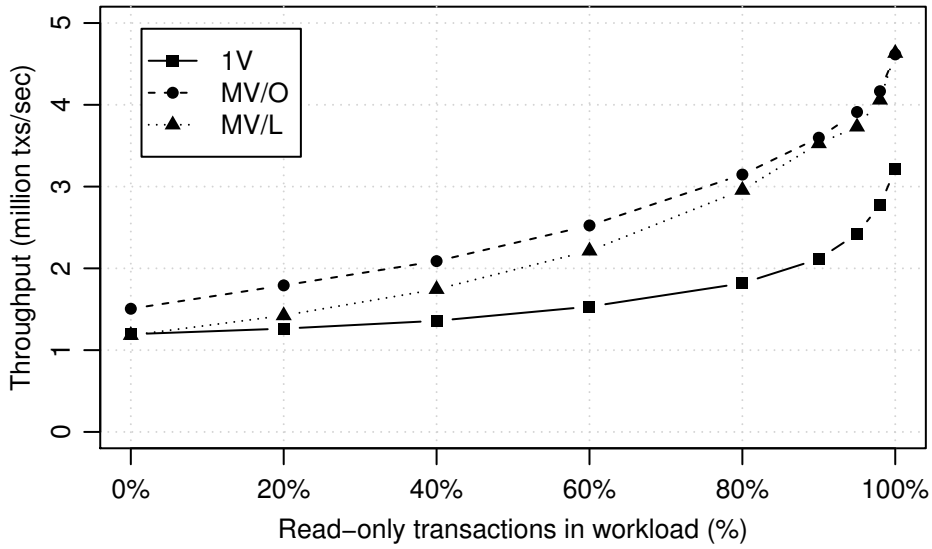


Figure 4.7: Impact of read-only transactions on throughput (high contention).

high contention, as snapshot isolation prevents read-only transactions from interfering with writers. When 80% of the transactions are read-only, the MVCC schemes achieve 63% and 73% higher throughput than 1V.

Impact of long read transactions

Not all transactions are short in OLTP systems. Users often need to run operational reporting queries on the live system. These are long read-only transactions that may touch a substantial part of the database. The presence of a few long-running queries should not severely affect the throughput of "normal" OLTP transactions.

This experiment investigates how the three concurrency control methods perform in this scenario. We use a table with 10,000,000 rows and fix the number of concurrently active transactions to be 24. The workload consists of two transaction types: (a) Long, transactionally consistent (Serializable),

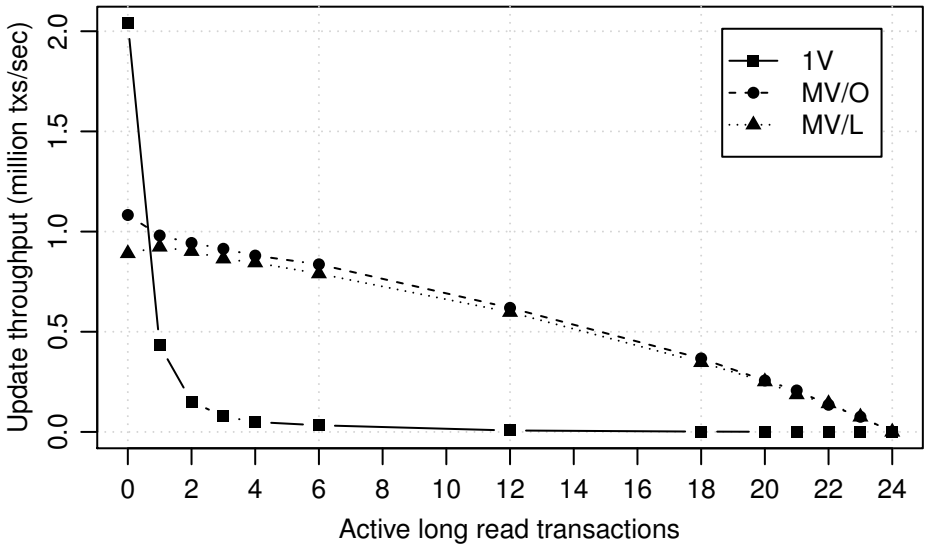


Figure 4.8: Update throughput with long read transactions. Each update transaction performs ten reads and two updates.

read-only queries that touch 10% of the table (R=1,000,000 and W=0) and (b) Short update transactions with R=10 and W=2.

Figures 4.8 and 4.9 show update and read throughput (y-axis) as we vary the number of concurrent long read-only transactions in the system (x-axis). At the leftmost point (x=0) all transactions are short update transactions, while at the rightmost point (x=24) all transactions are read-only. At x=6, for example, there are 6 readonly and 24-6=18 short update transactions running concurrently.

Looking at the update throughput in Figure 7, we can see that 1V is twice as fast as the MV schemes when all transactions are short update transactions, at x=0. (This is consistent with our findings from the experiments with the homogeneous workload in Section 4.6.1.) However the picture changes completely once a single long read-only transaction is present in the system. At x=1, update throughput drops 75% for the single

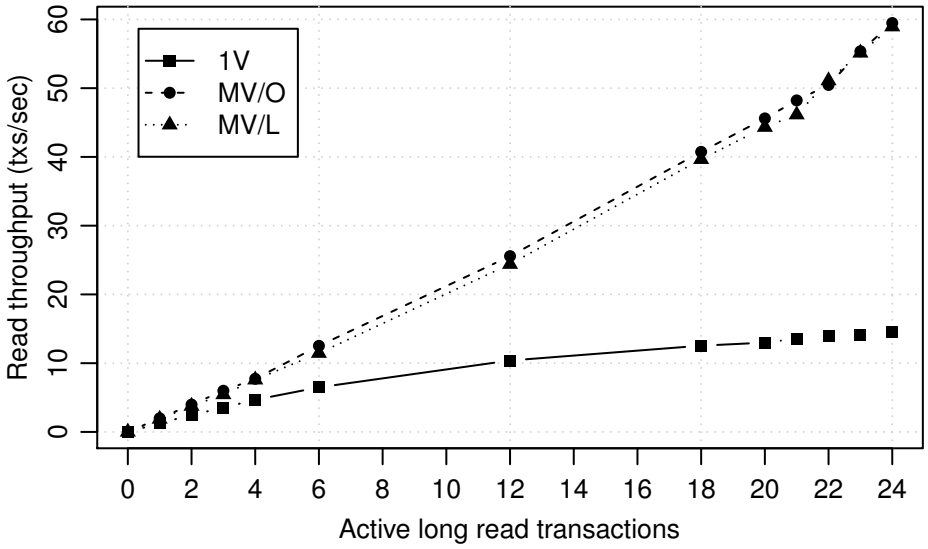


Figure 4.9: Read throughput with long read transactions. Each long read transaction performs one million reads (10% of the database).

version engine. In contrast, update throughput drops only 5% for the MV schemes, making MV twice as fast as 1V. The performance gap grows as we allow more read-only transactions. When 50% of the active transactions are long readers, at x=12, MV has 80X higher update throughput than 1V. In terms of read throughput (Figure 4.9), both MV schemes consistently outperform 1V.

4.6.3 TATP results

The workloads used in the previous sections allowed us to highlight fundamental differences between the three concurrency control mechanisms. However, real applications have higher demands than randomly reading and updating values in a single index. We conclude our experimental evaluation by running a benchmark that models a simple but realistic OLTP

	Transactions per second
1V	4,220,119
MV/L	3,129,816
MV/O	3,121,494

Table 4.5: TATP results.

application.

The TATP benchmark [76] simulates a telecommunications application. The database consists of four tables with two indexes on each table to speed up lookups. The benchmark runs a random mix of seven short transactions; each transaction performs less than 5 operations on average. 80% of the transactions executed only query the database, while 16% update, 2% insert, and 2% delete items. We sized the database for 20 million subscribers and generated keys using the non-uniform distribution that is specified in the benchmark. All transactions run under Read Committed.

Table 4.5 shows the number of committed transactions per second for each scheme. Our concurrency control mechanisms can sustain a throughput of several millions of transaction per second on a low-end server machine. This is an order of magnitude higher than previously published TATP numbers for disk-based systems [52] or main memory systems [50].

4.7 Concluding remarks

In this chapter we investigated concurrency control mechanisms optimized for main memory databases. The known shortcomings of traditional locking led us to consider solutions based on multiversioning. We designed and implemented two multi-version concurrency control methods, one optimistic using validation and one pessimistic using locking. For comparison purposes

we also implemented a variant of single-version locking optimized for main memory databases. We then experimentally evaluated the performance of the three methods. Several conclusions can be drawn from the experimental results.

- Single-version locking can be implemented efficiently and without lock acquisition becoming a bottleneck.
- Single-version locking is fragile; it performs well when transactions are short and contention is low but suffers under more demanding conditions.
- The multi-version concurrency control schemes have higher overhead but are more resilient, retaining good throughput even in the presence of hotspots and long read-only transactions.
- The optimistic multi-versioning concurrency control scheme consistently achieves higher throughput than the pessimistic scheme.

Chapter 5

Conclusions and future work

This dissertation makes three contributions to high-performance main memory database management systems. First, in Chapter 2 we introduced a simple non-partitioned hash join variant that has comparable performance with much more sophisticated hash join methods. A feature of particular significance is that the performance of the non-partitioned hash join improves with higher skew. This simple algorithm performs so well primarily because of the introduction of hardware multi-threading in new CPUs, which in turn is a product of the fundamental shift towards more power-efficient processors.

Second, in Chapter 3 we demonstrated that the hash join is commonly advantageous over sort-merge join plans, even in the presence of NUMA effects. In cases where the query response time is comparable, the hash join query plan frequently consumes less memory because it needs to buffer the smaller side only, whereas the sort phase of the sort-merge algorithm would need enough space to buffer both inputs.

Third, in Chapter 4 we showed how to architect a transactional storage engine for memory-resident data. In particular, we designed and implemented two concurrency control schemes that are optimized for main-memory storage: one is optimistic and relies on read validation, and the

other is pessimistic and relies on locking. All concurrency control schemes achieve a throughput of millions of transactions per second without compromising atomicity, isolation or durability.

Future directions

One promising avenue for future work is investigating how other components of the data processing stack can take advantage of innovations in the underlying hardware. This section points to a number of open problems in this space.

There are many open questions in the area of complex query evaluation on a memory-resident dataset, especially in query execution, cost modelling and query optimization. Modern computer systems have much higher bandwidth to the last-level cache than to the main memory. This highlights the need to revisit the role of compression for memory-resident data, and find suitable algorithms that have latencies that are measured in machine cycles, and not microseconds, as for disk-based systems. In addition, as the vast majority of database servers are shared-memory systems, an open question is whether to represent a tuple by copying the data or referencing a particular memory location, and how to most efficiently propagate data from one operator to the next in a complex query pipeline. There is also potential in exploring how to synergistically execute queries between the main CPU and a hardware accelerator, such as a programmable vector processor, either on the same die or in a separate chip. Such a synergy might be desirable for achieving higher performance, or for guaranteing data confidentiality [6, 7].

There are promising avenues for future research in the transaction processing space as well. It remains an open question how to best integrate and utilize fast non-volatile storage. Today fast storage comes in the form of flash memory on the motherboard-level interconnect and is block addressable through the file system stack of the operating system. In the future, non-volatile memory will likely be directly attached to the memory con-

troller and will be byte-addressable by the database management system. Another interesting topic is how can one leverage hardware transactional memory to speed up transaction execution [78]. With the fourth generation of Intel Core CPUs implementing a restricted form of transactional memory, research in this area can have a big impact on the design of consistent and highly concurrent data stores [49].

Finally, research in this area presents many collaborative opportunities with other research fields. Data-intensive applications frequently encounter bottlenecks both inside the operating system and in the underlying hardware. While some bottlenecks are fundamental in nature, many represent work that the system undertakes for buggy or legacy applications. While good design principles mandate the existence of such protective mechanisms, there can be different interfaces for high-performance applications that choose to opt-out of such behavior. For example, hardware designers invest significant effort to ensure a consistent and coherent view of the entire memory, even for large multi-socket NUMA systems. A vertically integrated system could offer higher performance by requiring the data management application to designate which memory areas need not be kept coherent, reducing the cross-NUMA traffic, and to permit weaker memory consistency for this particular application, reducing the memory access latency. Identifying how to implement, expose and virtualize such features requires a confluence of database, operating systems and computer architecture research.

Bibliography

[1] Divyakant Agrawal, Arthur J. Bernstein, Pankaj Gupta, and Soumitra Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1):45–59, 1987.

[2] Divyakant Agrawal and Soumitra Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. In *SIGMOD Conference*, pages 408–417, 1989.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[5] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[6] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. Engineering security and performance with cipherbase. *IEEE Data Eng. Bull.*, 35(4):65–72, 2012.

[7] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramaratnam Venkatesan. Orthogonal Security with Cipherbase. In *CIDR*, 2013.

[8] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Leopoldo Perez. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD*, pages 519–530, 2010.

[9] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, 2013.

[10] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. Business analytics in (a) blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.

[11] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *ISCA*, pages 237–248, 2013.

[12] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD Conference*, pages 1–10, 1995.

[13] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

[14] Paul M. Bober and Michael J. Carey. Multiversion query locking. In *VLDB*, pages 497–510, 1992.

[15] Paul M. Bober and Michael J. Carey. On mixing queries and transactions via multiversion locking. In *ICDE*, pages 535–545, 1992.

[16] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[17] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, and Alan Fekete. One-copy serializability with snapshot isolation under the hood. In *ICDE*, pages 625–636, 2011.

[18] Albert Burger, Vijay Kumar, and Mary Lou Hines. Performance of multiversion and distributed two-phase locking concurrency control mechanisms in distributed databases. *Information Sciences*, 96(1&2):129–152, 1997.

[19] Michael J. Cahill, Uwe Röhm, and Alan David Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems*, 34(4), 2009.

[20] Michael J. Carey. Multiple versions and the performance of optimistic concurrency control. Technical report, #517, Computer Sciences Department, University of Wisconsin–Madison, October 1983.

[21] Michael J. Carey and Waleed A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems*, 4(4):338–378, 1986.

[22] John Cieslewicz, William Mee, and Kenneth A. Ross. Cache-conscious buffering for database operators with state. In *DaMoN*, pages 43–51, 2009.

[23] John Cieslewicz and Kenneth A. Ross. Data partitioning on chip multiprocessors. In *DaMoN*, pages 25–34, 2008.

[24] John Cieslewicz, Kenneth A. Ross, and Ioannis Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, 2007.

[25] David J. DeWitt and Jim Gray. Parallel database systems: The future of database processing or a passing fad? *SIGMOD Record*, 19(4):104–112, 1990.

[26] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6), 1992.

[27] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *SIGMOD Conference*, pages 1–8, 1984.

[28] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *PDIS*, pages 280–291, 1991.

[29] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–40, 1992.

[30] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching nd smarter hashing. In *Proceedings of the 10th USENIX NSDI*, Lombard, IL, April 2013.

[31] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database - Data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[32] Glenn Fowler, Landon Curt Noll, and Phong Vo. FNV hash. `http://www.isthe.com/chongo/tech/comp/fnv/`. [Online reference; accessed June 11, 2013].

[33] Philip Garcia and Henry F. Korth. Database hash-join algorithms on multithreaded computer architectures. In *Conf. Computing Frontiers*, pages 241–252, 2006.

[34] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. SharedDB: Killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.

[35] Gaston H. Gonnet. Expected length of the longest probe sequence in hash code searching. *J. ACM*, 28:289–304, April 1981.

[36] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[37] Goetz Graefe. Sort-merge-join: An idea whose time Has(h) passed? In *ICDE*, pages 406–417, 1994.

[38] Goetz Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3), 2006.

[39] Jim Gray. A "measure of transaction processing" 20 years later. *IEEE Data Engineering Bulletin*, 28(2):3–4, 2005.

[40] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN 1-55860-190-2.

[41] Theo Härder and Erwin Petry. Evaluation of a multiple version cheme for concurrency control. *Information Systems*, 12(1):83–98, 1987.

[42] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[43] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

[44] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*, chapter 1, pages 3, 24. Morgan Kaufmann, 5th edition, 2012. ISBN 978-0-12-383872-8.

[45] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[46] *IBM DB2 10.1 for Linux, UNIX, and Windows: Troubleshooting and Tuning Database Performance*, chapter 3, page 153. January 2013. Reference number: SC27-3889-01.

[47] *IBM solidDB: In-Memory Database User Guide*, March 2013. Reference number: SC27-3845-04.

[48] "Compare Intel(r) Products" website. `http://ark.intel.com/compare/47922,37149`. [Online reference; accessed June 11, 2013].

[49] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 1, chapter 14. Reference number: 253665-047US.

[50] *Intel and IBM Collaborate to Double In-Memory Database Performance*. Whitepaper reference number: IMW14204-USEN-00.

[51] *Intel Xeon Processor 7500 Series Uncore Programming Guide*, March 2010. Reference number: 323535-001.

[52] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Improving OLTP scalability using speculative lock inheritance. *PVLDB*, 2(1):479–489, 2009.

[53] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: A scalable approach to logging. *PVLDB*, 3(1):681–692, 2010.

[54] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[55] Vijay Kumar, editor. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice-Hall, 1996. ISBN 0-13-065442-6.

[56] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[57] Sanjay Kumar Madria, Mohammed Baseer, Vijay Kumar, and Sourav S. Bhowmick. A transaction model and multiversion concurrency control for mobile database systems. *Distributed and Parallel Databases*, 22(2-3):165–196, 2007.

[58] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *VLDB*, pages 339–350, 2000.

[59] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[60] John C. McCallum. Memory prices (1957-2013). `http://www.jcmit.com/memoryprice.htm`. [Online reference; accessed June 11, 2013].

[61] *Microsoft SQL Server 2008 R2 Books Online: Isolation Levels in the Database Engine.* Available at `http://msdn.microsoft.com/en-us/library/ms189122.aspx` [Online reference; accessed June 11, 2013].

[62] David R. Musser. Introspective sorting and selection algorithms. *Softw., Pract. Exper.*, 27(8):983–993, 1997.

[63] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[64] *Extreme Performance Using Oracle TimesTen In-Memory Database*, July 2009. An Oracle technical whitepaper.

[65] *Oracle Exalytics In-Memory Machine: A Brief Introduction*, October 2011.

[66] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[67] Christos H. Papadimitriou and Paris C. Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems*, 9(1):89–99, 1984.

[68] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[69] Jun Rao and Kenneth A. Ross. Making B^+-trees cache conscious in main memory. In *SIGMOD Conference*, pages 475–486, 2000.

[70] Kenneth A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301, 2007.

[71] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, 2010.

[72] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, pages 510–521, 1994.

[73] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[74] Michael Stonebraker. The case for shared nothing. In *HPTS*, 1985.

[75] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[76] *Telecommunication Application Transaction Processing (TATP) Benchmark Description.* Available at `http://tatpbenchmark.sourceforge.net` [Online reference; accessed June 11, 2013].

[77] Alexander Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.

[78] Khai Q. Tran, Spyros Blanas, and Jeffrey F. Naughton. On Hardware Transactional Memory, spinlocks, and database transactions. In *ADMS*, 2010.

[79] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002. ISBN 1-55860-508-8.

[80] Marcin Zukowski and Peter A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin*, 35(1):21–27, 2012.