Scalable Query and Transaction Processing over High-Performance Networks

Author(s):
Barthels, Claude

Publication Date:
2019

Permanent Link:

https://doi.org/10.3929/ethz-b-000343030 →

DISS. ETH NO. 25655

Scalable Query and Transaction Processing over High-Performance Networks

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zürich)

presented by

CLAUDE BARTHELS

Master of Science ETH in Computer Science, ETH Zürich

born on 29.01.1988

citizen of Luxembourg

accepted on the recommendation of

Prof. Dr. Gustavo Alonso (ETH Zürich), examiner
Prof. Dr. Torsten Hoefler (ETH Zürich), co-examiner
Prof. Dr. Spyros Blanas (The Ohio State University), co-examiner
Prof. Dr. Donald Kossmann (Microsoft Research), co-examiner

2019

Abstract

Distributed query and transaction processing has been an active field of research ever since the volume of data to be processed outgrew the storage and compute capacity of a single machine. For decades, distributed database systems have been designed and implemented under the assumption that the network is relatively slow compared to the local, in-memory processing speed. In recent years, high-performance networks have become a key element in database appliances and data processing systems to reduce the overhead of data movement. Many of these modern networks offer new communication primitives and use Remote Direct Memory Access (RDMA) – a hardware mechanism through which the network card can directly access parts of main memory without involving the processor – in order to achieve low latency and high throughput. However, these performance advantages can only be leveraged through thoughtful design of the distributed algorithms, in particular through careful management of the RDMA-enabled buffers used for transmitting and receiving data, and through interleaving computation and network communication.

In this dissertation, we analyze the impact of this new generation of networks on data management and processing in clusters of all sizes. For query processing, we focus on the implementation of hardware-conscious, distributed join algorithms, in particular the radix hash join and the sort-merge join. We evaluate these join algorithms on modern rack-scale systems with different interconnect technologies and, at large scale, on a supercomputer with hundreds of machines. Regarding transaction processing, we study the performance of lock-based concurrency control mechanisms and establish a new baseline for a conventional lock table manager running on thousands of processor cores.

Our findings show that the proposed algorithms can take advantage of modern communication primitives and are able to scale with increasing system resources. This dissertation is one of the first publications to combine traditional database algorithms with the technologies used in supercomputers and to evaluate these algorithms on thousands of cores, a scale usually reserved to large scientific computations. Furthermore, we provide detailed performance models for each of the proposed algorithms.

Using the insights gained from the implementation of the algorithms, this dissertation proposes several new communication primitives designed to overcome the limited programming interface of current RDMA-capable networks and to provide directions towards the development of novel communication abstractions for high-performance networks targeting data- and communication-intensive applications.

Zusammenfassung

Verteilte Abfrage- und Transaktionsverarbeitung ist ein aktives Forschungsgebiet seitdem das Volumen der zu verarbeitenden Daten die Speicher- und Verarbeitungskapazität einzelner Maschinen überschreitet. Seit Jahrzehnten wurden verteilte Datenbanksysteme in der Annahme entworfen und implementiert, dass das Netzwerk im Vergleich zu der lokalen Verarbeitungsgeschwindigkeit relativ langsam ist. In den letzten Jahren wurden Hochleistungsnetzwerke zu einem Schlüsselelement in Datenbankanwendungen und Datenverarbeitungssystemen um die Kosten des Datentransfers zu reduzieren. Viele dieser modernen Netzwerke bieten neue Kommunikationsmethoden an und verwenden nicht-lokalen direkten Speicherzugriff (Remote Direct Memory Access, RDMA) – ein Hardware-Mechanismus, über den die Netzwerkkarte direkt auf Teile des Hauptspeichers zugreifen kann, ohne den Prozessor miteinzubeziehen – um eine geringe Latenz und einen hohen Datendurchsatz zu erreichen. Diese Leistungsvorteile können jedoch nur durch ein gut durchdachtes Design der verteilten Algorithmen genutzt werden, insbesondere durch sorgfältige Verwaltung der für die Übertragung verwendeten Puffer und durch Überlappung von Berechnungen und Netzwerkkommunikation.

In dieser Dissertation analysieren wir die Auswirkungen dieser neuen Generation von Netzwerken auf Datenverwaltung und -verarbeitung in verteilten Rechenanlagen aller Grössen. In Bezug auf die Abfrageverarbeitung konzentrieren wir uns auf die Implementierung von hardware-optimierten, verteilten Verbundoperatoren, insbesondere auf hash- und sortierbasierte Lösungen. Wir evaluieren diese Algorithmen auf modernen Clustern mit verschiedenen Netzwerktechnologien und, in grossem Massstab, auf einem Supercomputer

mit Hunderten von Maschinen. Hinsichtlich Transaktionsverwaltung untersuchen wir die Leistung von sperr-basierten Mehrbenutzersynchronisationsmechanismen und etablieren einen neuen Referenzwert für die Ausführung eines konventionellen Transaktionsverwaltungssystems auf Tausenden von Prozessorkernen.

Unsere Ergebnisse zeigen, dass die vorgeschlagenen Algorithmen moderne Kommunikationsmechanismen effizient nutzen und mit den Systemressourcen skalieren. Diese Dissertation ist eine der ersten Arbeiten, die traditionelle Datenbankalgorithmen mit den in Supercomputern verwendeten Technologien verbindet und die vorgeschlagenen Algorithmen auf Tausenden von Prozessorkernen auswertet, eine Grössenordnung, die normalerweise nur grossen wissenschaftlichen Berechnungen vorbehalten ist. Darüber hinaus beinhaltet diese Dissertation analytische Modelle für jeden der Algorithmen.

Auf Basis der gewonnenen Erkenntnisse schlagen wir mehrere Kommunikationsmechanismen vor, welche die Einschränkungen jetziger RDMA-fähiger Netzwerke überwinden und die Programmierschnittstelle zukünftiger Hochleistungsnetzwerke mit neuen Kommunikationsabstraktionen für daten- und kommunikationsintensive Anwendungen erweitern.

Acknowledgments

During my time at ETH Zürich, I had the opportunity to work with many talented and dedicated researchers. First, I would like to express my gratitude to my adviser Gustavo Alonso for his invaluable support, advice, guidance, patience, and for helping me grow as a researcher. My gratitude extends to all the current and former professors of the Systems Group. In particular, I would like to thank Torsten Hoefler and Donald Kossmann for always giving me insightful feedback about my work. Many thanks to Spyros Blanas for taking part in my committee and for the interesting collaboration we had. I would also like to thank Eric Sedlar, Nipun Agarwal, Vikas Aggarwal, and Michael Duller for giving me the opportunity to work on Project Rapid. The internship at Oracle Labs during the early days of my doctoral studies has been a rewarding experience.

This dissertation was shaped through countless interactions and discussions with many exceptional colleagues and friends: Andrea, Anja, Besmira, Bojan, Cagri, Darko, David, Debopam, Feilong, Frank, Gerd, Ingo, Jana, Kaan, Konstantin, Lefteris, Lucas, Lukas, Maciej, Markus, Melissa, Merve, Moritz, Muhsen, Pratanu, Pravin, Renato, Reto, Roni, Sabir, Salvatore, Simon, Stefan, Tal, Timo, Vojislav, Yunyan, Zaheer, and Zsolt. I would like to take this opportunity to thank Eva, Jena, Karel, Nadia, and Simonetta for making the Systems Group a pleasant workplace. A special thanks goes to some of the wonderful people I have met in my life: Christopher, Laurent, Sophie, Tessy, Thierry, and Tom.

Finally, for everything I have achieved so far, I would like to thank my parents, Claudine and Nico. They always encouraged me to do what I enjoy and have been supportive like no one else. This dissertation is their merit as well.

Contents

1

Introduction

The majority of business activities such as sales, reporting, analytics, planning, and data archiving involves the use of a state-of-the-art computing infrastructure. Processing and extracting meaningful information from data requires a complex combination of data processing platforms, database systems, machine-learning applications, and sophisticated data analysis tools. At the same time, the volume of data that needs to be analyzed and managed by these information systems is increasing at an unprecedented rate. The ability to efficiently query vast volumes of data and execute a large number of transactions requires carefully tuned algorithms that take advantage of all the resources made available by the underlying hardware, including the network infrastructure.

Although the economics of main memory technologies have enabled a steady increase of the memory capacity of modern multi-socket servers, managing large amounts of data requires the use of a distributed computing infrastructure that enables users to scale out the memory and compute capacity simultaneously. In these systems, efficient inter-machine data movement is critical, forcing database algorithms to be aware of machine boundaries and to employ communication patterns suited for the underlying network technology. For a long time, rack-scale clusters have been a platform of choice for distributed data pro-

cessing. These systems are composed of several multi-core machines connected by a high-throughput, low-latency network. High-performance interconnects can significantly reduce the costs of small and large data transfers by offering high bandwidth and low latency. However, these performance advantages can only be leveraged through thoughtful design of the distributed algorithms and through the correct use of all available communication primitives and mechanisms.

Today, fast interconnects are no longer limited to rack-scale systems and are being introduced in many data centers and large compute clusters. Cloud computing providers are starting to equip their machines with high-speed interconnects. In the process, they expose new network interfaces to their customers. Thus, it is apparent that new techniques and algorithms need to be able to run on hundreds of machines and thousands of cores, using the processor, memory, and network resources efficiently. Therefore, it is a natural question to ask how to design scalable database systems for query and transaction processing that can run on large scale-out architectures in which the machines are connected by modern, high-speed networks.

1.1 Motivation and Challenges

The introduction of high-performance networks has led to new design possibilities and implementations for data management systems [JSL+11, DNCH14, KKA14, RMKN15, LPEK15, Röd16, BCG+16, LDSN16, BAH17, MGBA17, Mak17, BKG+18] and database algorithms, such as distributed joins [FGKT09, FGKT10, BLAK15, RIKN16, BMS+17]. Although many applications can benefit from the increased throughput and reduced latency of this new generation of network technologies, their full potential can only be harvested by employing the new communication primitives these networks provide. For example, several low-latency, high-throughput networks provide Remote Direct Memory Access (RDMA) as a light-weight communication mechanism to transfer data. RDMA is essential for high-performance applications because data is immediately written or read by the network card, thus eliminating the need to copy data across intermediate buffers inside

the operating system. Furthermore, one-sided operations make it possible to place data at specific locations in remote main memory. Applications that can use this Remote Memory Access (RMA) programming model have the potential to eliminate many copy operations within the application logic, thus further increasing performance. In order to implement efficient data transfer and coordination mechanisms, the design of the underlying database algorithms needs to address several crucial challenges.

***Connection and Buffer Management*:** Conventional distributed applications rely on the network stack within the operating system to manage buffers used for communication. For example, when executing a *send* call, data is copied from the application into the network stack. Within the stack, the content of the message is typically copied from one layer to the next until it has been divided into small packets. On the receiving side, these packets are being reassembled and, when a *receive* call is executed, copied into a user-level buffer. This whole process is controlled by the operating system and the application has no direct influence on these operations. On the other hand, RDMA-enabled network cards can read and write data directly from and to main memory without involving the network stack. In this model, the operating system is completely bypassed. That means that it is the responsibility of the application to explicitly manage the buffers involved in the communication. In most network implementations, the application needs to register a communication buffer with the network card before it is accessible over the network to RDMA operations. Once registered, the section of memory that can be used for RDMA transfers is of fixed size that often can only be changed by first de-registering the buffer and then registering it a second time with the new desired size. These buffer management operations usually incur a significant overhead [FA09, Fre10]. Therefore, applications need to be designed in such a way that most of their communication buffers can be allocated at system start-up time to avoid expensive registration calls at runtime. Furthermore, algorithms should re-use already registered RDMA-enabled buffers whenever possible, otherwise the overall application performance might degrade significantly. Besides registering communication buffers with the network card, many networks require an elaborate setup of connection-related objects such as queue pairs, completion queues, and receive queues. To amortize these costs, algorithms should setup all necessary connections ahead of time.

***Direct Data Placement*:** When using one-sided RMA programming, the initiator of a request can directly read from or write to specific locations of remote memory. Applications that have tight control of where data is located can potentially eliminate expensive application-level copy operations. On modern networks, remote data accesses do not involve the remote processor as RMA requests contain all necessary information for the network card to directly access and transmit (or overwrite) the desired content. From a technical perspective, the initiator of a request must be in possession of the address (or similar identifiers) of the remote buffer before initialing a request. The algorithm needs to be designed in such a way that this information is distributed to all the processes that require access to the RDMA-enabled buffer. On the application level, algorithms using one-sided RMA operations need to carefully lay out the data inside of these buffers. Management of the content of the buffers usually involves maintenance of auxiliary data structures, such as indexes or histograms, in order to determine the location of a specific piece of information within the buffer. Furthermore, the content of the communication buffers can be accessed and modified by any component in the system that has the necessary credentials. Similar to parallel applications running on large multi-core processors, access to these shared buffers needs to be synchronized and appropriate coordination mechanisms need to be in place.

***Interleaving of Computation and Communication*:** In contrast to traditional socket programming, many RDMA-capable networks are asynchronous and most of the network operations are non-blocking. Information related to a data transfer is wrapped into a *work request* object that is posted to the appropriate queue. These requests are taken from the queue and executed by the network card without any involvement of the processor. This means that the processor remains available for processing while a transfer is taking place in parallel. Applications built for traditional networks are often designed to wait for the transfer to complete before they continue processing. Although the time required to transmit a specific amount of data is shorter when using high-speed networks, the processor would still be idle for a significant period of time, leading to poor utilization of the resources. This problem is amplified for applications that have to transmit vast amounts of data during their processing phase. In order to avoid long processor idle times,

modern applications need to be designed such that they can interleave computation and communication. An application needs to be able to continue processing while a network operation is taking place in parallel. The interleaving of computation and communication not only leads to better resource utilization, but can also be used to hide parts (or all) of the communication latency. Given a specific network throughput, applications should be able to adjust the ratio of compute and communication tasks in order to achieve perfect interleaving, i.e., a compute task should take up the same amount of time than a network transfer. Applications that are designed to interleave both aspects can be accelerated significantly when using high-performance networks [FA09, BAH17].

***Network Scheduling*:** In order to achieve the highest performance possible, the algorithms do not only need to interleave communication with the computation, but also schedule the communication appropriately. Algorithms need to be aware of the distribution of the input data, the network topology, and the utilization of the network. However, the creation of an optimal plan for scheduling communication is not a straightforward task, especially when the cardinality of intermediate results is not known ahead of time [CKJE14, CKJE15, RCP17, LSBS18]. The problem is particularly difficult knowing that all data transfers are executed by the networking hardware without the involvement of the operating system. Furthermore, the amount of resources (e.g., queues and work requests) the hardware can hold at any given amount of time is limited. Therefore, the algorithms should be designed in such a way that they do not overwhelm the network card with requests at any given point in time. In addition, the communication patterns should be such that no contention is created within the network.

1.2 Focus

In this dissertation, we focus on distributed database algorithms for query and transaction processing. We will design and evaluate new variants of the radix hash join, the sort-merge join, as well as the Two-Phase Locking (2PL) and Two-Phase Commit (2PC) protocols. All algorithms are targeting the communication primitives offered by high-speed

interconnects, are designed to use one-sided RMA operations, take advantage of the low latency and high bandwidth offered by RDMA networks, and interleave computation and communication. Given the rapid adoption of high-performance networks, we evaluate the proposed algorithms not only at small scale but also on large supercomputers with several thousand processor cores. Furthermore, this dissertation provides analytical models that can be used to predict their performance on future interconnects.

***Query Processing*:** This dissertation focuses on designing and evaluating new distributed join algorithms, one of the most complex and communication-intensive operators in query processing. In relational database systems, join algorithms are considered important operators that need to exhibit the best performance possible. They appear frequently in many query workloads and often dominate the execution costs. Therefore, they have been the topic of several recent publications and many efficient implementations targeting modern, multi-core processors have been developed [KSC+09, BLP11, AKN12, BTAÖ13, BATÖ13, LLA+13, Bal14, BTAÖ15]. Joins are relevant not only in the context of database engines but also as a building block in many computational and machine learning algorithms [KNPZ16]. Moreover, there are multiple join strategies, each having different data processing and communication characteristics. Although having been an active topic of research for several years, there are opposing views on how to implement join algorithms on modern hardware. One of the controversies revolves around the discussion whether a sort- or a hash-based approach is the preferred option when using a large number of processor cores [KSC+09, BATÖ13, Bal14]. In this dissertation, we investigate both strategies and evaluate two join implementations: (i) the radix hash join, and (ii) the sort-merge join algorithm. We begin by designing the algorithms for rack-scale systems. In a second step, we combine the proposed algorithms with the technologies found in high-performance computing (HPC) systems in order to be able to scale them to hundreds of machines with several thousand cores. We will discuss the differences and similarities between database systems and HPC applications and explain how to combine the technologies found in both areas of computer science. By focusing on this important operator, this dissertation takes several important steps towards scaling out the query engines of distributed database systems. These findings can be used to accelerate a large variety of relational operators.

***Transaction Processing*:** Concurrency control is a cornerstone of distributed database engines and storage systems. Using an efficient coordination mechanism that supports a high throughput of transactions is a critical factor for parallel and distributed databases systems. Having always been a challenging problem, the increase in parallelism arising from multi-core systems and cloud platforms has motivated researchers and practitioners to explore alternative implementations and weaker forms of consistency. The vast majority of these efforts start from the assumption that Two-Phase Locking (2PL) and Two-Phase Commit (2PC) are not viable solutions due to their communication overhead and perceived lack of scalability. Many systems apply a wide range of optimizations that impose restrictions on the workloads the engine can support. For example, they give up serializability in favor of snapshot isolation [ZBKH17], impose restrictions on long-running transactions [KN11, TZK+13, DNN+15], assume partitioned workloads [KKN+08], or require to know the read and write set of transactions ahead of time [KKN+08, TDW+12]. Due to the very different assumptions made, and the wide range of performance levels achieved, these systems are difficult to compare to each other. In this dissertation, we develop a distributed lock table supporting all the standard locking modes used in database engines [BHG87, GR92]. We focus on strong consistency in the form of *strict serializability* implemented through strict 2PL but also explore other isolation level such as *read-committed*, a common isolation level used in many database systems. While the costs of synchronization and coordination might be significant on conventional networks, modern networks and communication mechanisms have significantly lowered them. We show that by using modern communication methods in combination with RDMA-enabled networks, 2PL and 2PC are a viable solution for large-scale transaction processing.

1.3 Contributions

In the context of large-scale query processing, we investigate distributed join algorithms in great detail. For workloads with a large number of transactions, we show that hardware-conscious implementations of a traditional locking system can be used to scale to a large number of cores. In summary, this dissertation makes the following contributions.

***Algorithmic Contributions and Implementations*:** Several main-memory hash and sort-merge join algorithms proposed in the literature are carefully analyzed. Many of these designs and implementations are targeting a single multi-core server. On top of this solid foundation, this dissertation introduces several new techniques and important optimizations leading to new, highly tuned, hardware-conscious, distributed join algorithms. In the context of the radix hash join, we interleave data partitioning and communication. For the sort-merge join, we develop a mechanism capable of interleaving the sort operation and the data exchange. Furthermore, we propose solutions to carefully lay out the data inside the buffers such that the content can directly be accessed using one-sided RMA operations. The optimizations reduce the amount of synchronization necessary during the data exchange, the most critical phase of the join algorithms. These techniques ensure that the proposed algorithms can scale to thousands of processor cores and hundreds of machines. For distributed concurrency control, we operate a conventional lock table and commit protocol. While these algorithms are already widely used in a lot of database engines, the implementation proposed in this dissertation is novel. We highlight how to structure the communication and manage the RDMA-enabled communication buffers. We make use of notified RMA operations as well as fast one-sided atomic operations to implement the concurrency control mechanism. The system supports all lock modes found in multi-level granularity locking, including intention locks. The proposed communication layer ensures that messages get delivered with very low latency. Because of the wide-spread adoption of lock-based concurrency control mechanisms, this work makes significant contributions towards scaling out existing, transaction-oriented database systems without compromising on performance nor isolation level guarantees.

***Combining Database Systems and HPC Applications*:** This is one of the first dissertations that discusses the use of technologies found in high-performance computing (HPC) systems in order to scale out database algorithms on modern hardware and networks. In the implementation of our algorithms, we use the Message Passing Interface (MPI), a de-facto standard communication layer used by many HPC applications, e.g., large scientific applications. In the context of HPC, the interface makes applications portable between different supercomputers that use different interconnect technologies.

Using MPI in the context of a database or data processing system is not a straight-forward task. Although the performance of both, database systems and scientific applications, depends significantly on the behavior of the network, both types of systems often differ significantly in their architecture and the communication patterns that they exhibit. As a result, many MPI implementations are optimized for specific workloads. We show that, despite these optimizations, the use of a high-level communication library brings many advantages for data processing applications. First, using a library makes the developed source code portable to new architectures and networks. Second, it offers a set of sophisticated communication operations that have proven to be useful when running a database system or a data processing application at large scale.

***Effects of Modern Networks on Data Processing Systems*:** We evaluate the algorithms presented in this dissertation on a variety of high-performance networks. We use several versions of InfiniBand, each having different bandwidth and latency characteristics. For the large-scale experiment, we conduct them on a high-end Cray supercomputer. This machine has a proprietary RDMA-capable network with a high bandwidth and a low latency. Many hardware parameters influence the performance of the proposed algorithms such as the size of the communication buffers, the number of threads involved in the communication, the choice of the communication library, and the types of operations used. The dissertation explores these parameters in a systematic way and studies their impact on network bandwidth, network latency, and overall application performance.

***Comparing Sort- and Hash-Joins on Thousands of Cores*:** With the advent of processors with many cores and large Single-Instruction-Multiple-Data (SIMD) vectors, it has been argued that a NUMA-aware (Non Uniform Memory Access) sort-merge join algorithm is becoming the preferable option compared to the radix hash join [KSC+09]. These predictions are mostly based on mathematical models. The dissertation analyzes these claims at large scale on several thousand processor cores in which memory exhibits strong NUMA effects, i.e., despite fast networks, remote RMA accesses still have a higher latency than accesses to local main memory. This dissertation contributes to this discussion and shows experimentally that, although the radix hash join has superior performance in small deployments, the sort-merge join is more scalable. This behavior is due to the

ratio of computation and communication and the ability of the sort-merge join to put a more predicable load on the network, making scheduling decisions easier.

***A new Baseline for Large-Scale Concurrency Control*:** Concurrency control has been a cornerstone of distributed database engines and storage systems. To achieve the highest level of isolation guarantees, many existing database engines use a lock-based concurrency control mechanism. However, distributed locking has not been perceived as a viable solution for distributed database systems. One of the main reasons for this is the latency overhead associated with accessing a remote lock when using a conventional network. In this dissertation, we show that, although remote accesses are more costly than modifying local memory, modern networks have reduced the latency to a point that a distributed lock-based concurrency control mechanism can support even the most demanding workloads. The result from these experiments is that, for TPC-C, 2PL and 2PC can be made to scale to thousands of cores and hundreds of machines, providing a throughput significantly higher than the fastest official TPC-C result published to date. Therefore, this work establishes a baseline for concurrency control mechanisms on thousands of cores.

***Performance Models*:** For all the algorithms presented in this dissertation, we provide analytical models. These models give us a lower bound of the execution time of the algorithms, respectively the maximum achievable throughput. We use these models to judge the efficiency of our implementations and make predictions on the performance of the algorithms as networks get faster. Using mathematical models is particular important for large-scale experiments in order to identify potential bottlenecks and analyze the effects arsing from large-scale distribution.

1.4 Outline

This dissertation is divided into seven chapters and is structured as follows:

***Chapter 2*:** This chapter provides the background material and gives an overview of the technology trends in the three areas of computer science research that are combined in this dissertation: (i) high-performance computing systems, (ii) high-performance networks, and

(iii) database and data processing systems. The overall goal of this chapter is to convey an understanding for each of these research topics.

***Chapter 3*:** In this chapter, we introduce the distributed radix hash join and sort-merge join. We describe how these algorithms organize their communication, i.e., connection and buffer management, RMA memory accesses, and the interleaving of computation and communication. We evaluate these algorithms on rack-scale computers with two generations of InfiniBand networks: (i) 4x QDR, that offers a throughput of 32 Gbit per second, and (ii) 4x FDR, that can transmit data at 56 Gbit per second. This allows us to study the performance of the algorithms as the network throughput increases.

***Chapter 4*:** Using the technologies found in HPC systems, we modify the rack-scale join algorithms to use MPI as their communication library. This enables the algorithms to run on high-end Cray supercomputers with thousands of processor cores. We analyze the behavior of the radix hash and sort-merge join algorithms on up to 4096 cores, compare both strategies, evaluate the costs of large-scale distribution, and discuss the importance of network scheduling.

***Chapter 5*:** In this part of the dissertation, we evaluate a lock-based concurrency control mechanism. We show that a conventional lock table and commit protocol, combined with a low-latency communication infrastructure, is a scalable solution for large-scale coordination and transaction management. Similar to the previous section, we explain how different system components use MPI to communicate and evaluate the algorithms on a Cray system with thousands of processor cores.

***Chapter 6*:** Based on the experiences from the large-scale experiments, this chapter provides an overview of alternative designs for the proposed systems and algorithms. In this chapter, we pay particular attention to future networks. We consider not only faster interconnects with a higher throughput and a lower latency, but also networks that offer new communication primitives designed to support distributed data processing.

***Chapter 7*:** In the last chapter, we take a look at the contributions and results obtained in the dissertation. All our findings and conclusions are listed and summarized. The chapter discusses future research directions before presenting the final concluding remarks.

1.5 Publications

Part of the work presented in this dissertation has been published in leading database conferences and journals. This document includes results from collaborations with Simon Loesing, Ingo Müller, Timo Schneider, Feilong Liu, Hideaki Kimura, Garret Swart, Spyros Blanas, Donald Kossmann, Torsten Hoefler, and Gustavo Alonso.

The following papers and articles constitute a preliminary and condensed form of the material presented in this document:

- [BLAK15] Claude Barthels, Simon Loesing, Gustavo Alonso, Donald Kossmann. "Rack-Scale In-Memory Join Processing using RDMA". *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 1463-1475, June 2015.
- [BMS+17] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, Torsten Hoefler. "Distributed Join Algorithms on Thousands of Cores". *Proceedings of the VLDB Endowment*, vol. 10, no. 5, 517-528, January 2017
- [BAH17] Claude Barthels, Gustavo Alonso, Torsten Hoefler. "Designing Databases for Future High-Performance Networks". *IEEE Data Engineering Bulletin*, vol. 40, no. 1, 15-26, March 2017.
- [BMAH] Claude Barthels, Ingo Müller, Gustavo Alonso, Torsten Hoefler. "Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores". [Under submission]
- [LBB+] Feilong Liu, Claude Barthels, Spyros Blanas, Hideaki Kimura, Garret Swart. "Beyond RDMA: Towards a New Communication Abstraction for Data-Intensive Computing". [Under submission]

2

Background

This thesis combines algorithms, technologies, and methodologies from (i) high-performance computing, (ii) high-speed networks, and (iii) distributed data processing systems in novel ways. Therefore, this chapter provides the required background materials from these three areas of computer science.

Starting with high-performance computing (Hpc), we explain the importance of performance models and explore the architecture of large-scale clusters and supercomputers. We discuss the role of the Message Passing Interface (Mpi) for scaling applications to thousands of machines. Next, we analyze the features and communication primitives offered by modern networks, focusing on new hardware mechanisms such as Remote Direct Memory Access (Rdma) and programming models for these interconnects. We have a look at commonly used interfaces and the abstractions they provide. Last, but not least, we give an overview of recent developments in the area of distributed database and data processing systems. We pay special attention to analyzing the impact of emerging network technologies on the development of new database algorithms.

2.1 High-Performance Computing

As the name suggests, high-performance computing (Hpc) is a discipline that strives to achieve and deliver much higher performance than one could get out of a typical server or small cluster. Science and technology plays an important role in improving the quality of life. The development of many modern devices and important advances in medicine are only possible if researchers are able to solve numerous scientific and engineering challenges in an efficient way. Hpc plays an important role in solving such problems through computer modeling, simulation, and analysis. Instead of spending precious time conducting real-world experiments, researchers can describe their problems and solutions using mathematical models, translate these models to efficient algorithms, and launch their simulations on a supercomputer. The more efficient the machine operates, the faster the turn-around time is and the more insights can be gained in a given amount of time.

However, the computing power required to advance the state-of-the-art in science, technology, and business is growing at a significant rate. In order to meet these demands, an important aspect of Hpc is the aggregation of computing power, usually in the form of a high-end supercomputer. These machines are composed out of thousands of machines, referred to as *compute nodes*, that are connected through a high-performance interconnect. In order to build a supercomputer and run it efficiently, Hpc brings together several aspects of computer science and computer engineering such as micro-architecture, algorithms, applications, and system software.

The workloads targeted by Hpc systems are mostly composed of large scientific applications that are usually characterized by having many floating-point and memory operations. Typical users of Hpc systems are physics and biology research projects. For example, many supercomputers are used to run weather/climate predictions and protein folding simulations. These large scientific codes are optimized to run at extreme scale and often require significant investments at a national level.

2.1.1 Performance Modeling

Software development for HPC applications focuses on correctness, reliability, productivity, and scalability, but also places great emphasis on achieving the highest performance possible. In order to be able to determine how fast a computation can be, the HPC community relies on extensive modeling techniques [KAH+01, SCW+02, AV06, HGTT10, Hoe10, LMV15], including models for one-sided network programming abstractions [DLHV16].

The importance of modeling is also reflected throughout the whole development process and the tool-chains that are used. In addition to the traditional debuggers and profilers, advanced parallel performance analysis tools are often necessary to understand the behavior of applications on large HPC systems. NETGAUGE [HMLR07] is a high-precision network parameter measurement tool that can be used to analyze the performance of common communication patterns. LIBSCIBENCH [HSL10, HB15] is a framework that facilitates the adoption of statistically sound performance measurements for massively parallel HPC applications. HPCTOOLKIT [ABF+10] provides a measurement and analysis framework to track application performance, collect call stack profiles, and display space-time diagrams.

Abstract performance models are often used to determine lower bounds on the execution time and analyze the impact of large-scale distribution. While these models are generally less accurate in predicting absolute performance numbers, they are useful in predicting the behavior of the application on different or future systems. An alternative to creating an analytical model is to benchmark the code on each target architecture. While this leads to very accurate measurements for one particular type of machines, it is difficult to extend the results to include other systems and configurations. In order to accurately predict the execution time while keeping the complexity of the model at a minimum, a hybrid approach is often used. Empirical modeling is employed to create performance models that rely only in part on benchmarks of specific sections of the code. PEMOGEN [BH14b] is a compilation and modeling framework that automatically instruments applications to generate performance models during program execution. Instead of a human expert with advance knowledge of the application creating a model, some approaches suggest using events and performance counters (e.g., tracing memory accesses) to determine the

behavior of an application on a specific platform and draw conclusions about the expected performance on other systems.

2.1.2 High-Performance Computing Systems

HPC involves is the use of large supercomputers and parallel processing techniques for solving complex computational problems. These systems are scale-out architectures that can deliver exceptional performance through the concurrent use of a large number of computing resources. Usually the performance of a supercomputer is measured in the number of floating-point operations per second (FLOPS) that the machine is able to perform. The TOP500 project [Top18] ranks the most powerful supercomputers in the world. One common characteristic of the fastest and most efficient machines is the use of a high-bandwidth, low-latency network to connect the nodes, clusters, and specialized hardware devices.

Supercomputers have a large number of nodes and, in general, most of these nodes are configured identically. Access to the compute nodes is often restricted and programs are submitted in batches, called *jobs*, to a cluster management system that is responsible for assigning compute resources to a job, scheduling it, and starting the desired applications. The management system monitors the job and in case of errors (e.g., segmentation faults, node failures) aborts the execution. The input data is generally loaded through a shared network-attached file system to which every node has access. The result of the computation is stored in the same way.

Although the exact configuration of each supercomputer is unique, many have a hierarchical architecture. In the system that we are using in the experimental evaluation, each rack, also called a *cabinet*, can be fitted with several *chassis*. A chassis provides power to a dozen or more *compute blades*, which in turn are composed of the *compute nodes*. The compute nodes are connected through a high-throughput, low-latency network usually forming a Dragonfly [KDSA08] or Slimfly [BH14a] network topology. In the beginning of HPC, many supercomputers contained custom and highly specialized components. Today, despite the large amount of compute power packaged as one large installation, the individual compute nodes used in supercomputers resemble commodity hardware, i.e., regular X86 or ARM

processors with several gigabytes of main memory. The network features that have been available exclusively in HPC for a long time (e.g., RDMA, remote atomic operations) have found their way into many network technologies that are being offered commercially and that can be found in a lot of small-sized clusters and data centers.

2.1.3 Message Passing Interface

Given that the network plays an important role in many high-performance distributed applications, it is important to understand the origins of the Message Passing Interface (MPI), the de-facto standard for writing parallel HPC applications. MPI [Mes12] is the result of standardization efforts to make application code portable between systems with different interconnect technologies. The first version of the interface was released in 1994. Since then, it has been expanded twice, resulting in MPI-2 in 1997 and MPI-3 in 2012. Responsible for the development of the standard is the MPI Forum that is composed of computer scientists and engineers with various backgrounds, system vendors, supercomputer manufacturers, researchers, and representatives from large research laboratories.

The interface description places great emphasis on attributing precise semantics to the different methods it provides, and tries to expose a rich set of high-level operations to the application developer. Although it has been designed for large scale-out architectures, it can be used on a variety of different compute platforms, from laptops to high-end supercomputers. The reason why MPI can be found on many types of systems is that the standard consists just of the interface description. This allows the developers to create many different implementations, each optimized for a specific system. Many supercomputers ship with a highly optimized MPI implementation. There are also several general-purpose implementations that support a variety of commercially available networks, the two most popular implementations being OPENMPI [Ope18a] and MVAPICH [The18].

Most MPI implementations support applications written in the C programming language and in Fortran. Already the first release of the standard provided bindings for both languages. In addition to point-to-point message passing methods, the initial standard included basic collective operations such as reduce operations and type support to describe

the data layout in main memory. Later iterations of the standard extended the functionality offered by MPI beyond that of a pure message-based communication library. Most notably the current version of the interface, MPI-3, added support for new operations made possible by RDMA-capable networks such as one-sided Remote Memory Access (RMA) and remote atomic operations [GHTL14, HDT+15].

When writing an MPI application, the developer is shielded from the complexity of the distributed environment and is – to a large extent – unaware of the physical location of the different processes that make up his program. Many implementations have different communication mechanisms and it is the responsibility of the library to select the most appropriate method of communication for each message and pair of processes. For example, an MPI implementation would distinguish between small and large messages. For small messages, the *eager* protocol is used. Its goal is to provide low latency. The assumption is made by the sending process that the receiving process can store the message in a special buffer. This buffer is sometimes referred to as a *mailbox*. Therefore, as soon as the sender invokes the send operation, the content of the message is transferred to this pre-allocated intermediate buffer at the receiver side. This method can be used as long as the size of the messages does not exceed the capacity of the mailbox. The *rendez-vous* protocol is used for large messages. In this protocol, a synchronization phase is required in which the receiver provides an RDMA buffer large enough to receive the message in its entirety. The content of the message is then directly transmitted to that buffer using a zero-copy transfer. The rendez-vous protocol can therefore lower the costs of large data transfers as it avoids intermediate copies of the data at the expense of a higher latency introduced by the synchronization phase. In addition to different protocols for different message sizes, many implementations use shared memory for communication between processes on the same machine and network-based primitives for inter-node communication.

The fundamental unit of parallelism in MPI is the *process*. The degree of parallelism of an MPI application can be specified at start-up time (or the time of submission of a batch job) by specifying the desired number of processes. The MPI runtime system is in charge of instantiating the requested number of processes running the same code on all machines that have been assigned to the job. Each process is identified by a *rank*, i.e., a unique

integer number that identifies the process. When communicating with another process, the developer has to indicate the rank of the target process. Details how to design and develop programs with MPI and related networking interfaces are explained in Appendix B.

2.2 High-Performance Networks

Modern high-throughput, low-latency networks originate from advances made in high-performance computing (HPC) systems. Myrinet was one of the first high-speed networks used to interconnect machines in HPC clusters [BCF+95]. At the time of its release, Myrinet significantly lowered the processing overhead compared to other network technologies. It was the first network implementation that offered a mechanism to bypass the operating system. This bypass mechanism avoids that the content of messages is being copied across different buffers within the network stack and reduces the number of context switches. Quadrics was a supercomputer company that developed a proprietary network [PFH+02]. The Quadrics network introduced a novel mechanism to integrate the local virtual memory of each node into a global address space and included a programmable processor in the network interface that could be used to offload application-specific communication protocols to the network card. The Virtual Interface Architecture (VIA) is an abstract model of a user-level network [MIC97]. VIA introduced the concept of zero-copy messaging. The content of a zero-copy message is not stored in any temporary buffer. This approach is different from the operating system bypass mechanism introduced by Myrinet that does not exclude the use of intermediate staging buffers. VIA is the basis for InfiniBand, a widely used high-performance network [Inf07]. InfiniBand can be found in many high-end database appliances and clusters. It uses Remote Direct Memory Access (RDMA), a hardware mechanism through which the network card can directly access all or parts of the main memory of a remote node without involving the processor. RDMA over Converged Ethernet (RoCE) is a network protocol that brings RDMA functionality to conventional Ethernet networks [Inf10]. As the compute nodes in most modern data centers are connected through Ethernet, RoCE makes it possible to benefit from the advantages of RDMA while using existing networking infrastructure. The Aries interconnect

is a proprietary technology used by Cray supercomputers [AKR12]. The Aries device is a system-on-a-chip comprising several network cards and a router. Each network card is connected to one compute node and the router is connected to the chassis back plane and through it to the network fabric. Aries is optimized for large-scale computations, supporting a large number of compute nodes, a high bandwidth, and a high message rate.

2.2.1 Network Interfaces

Several interfaces have been proposed in combination with high-performance networks. The RDMA Protocol Verbs Specification (RDMA Verbs) is an abstract low-level interface description for RDMA-enabled network cards [HCPR12]. This interface is being used by many InfiniBand and RoCE vendors. The Distributed Memory Application (DMAPP) interface was developed for Cray systems to better support programs that use one-sided read and write communication primitives [tBR10]. The Portals Network Programming Interface provides triggered communication primitives [BLMR02]. In contrast to most other interfaces, the initiator does not specify a remote virtual address that will be accessed. Instead, the destination is determined by the initiator and the target. This is done by comparing the message header, set by the initiator, with the contents of list-like data structures at the destination, controlled by the target node. The result of this comparison determines the memory location that will be accessed. This flexibility enables the network card to have efficient implementations of both one-sided and two-sided communication protocols. The latest versions of Portals have been implemented in several proprietary HPC interconnects. Libfabric defines interfaces with the goal to reduce the gap between applications and underlying network primitives [GHS+15, Ope18b]. To that end, the interfaces have been co-designed with application developers and hardware providers, while being agnostic to the underlying networking protocols and the implementation of the networking devices. The Message Passing Interface (MPI) is the de-facto standard interface for writing parallel computations for high-performance computing (HPC) applications [Mes12]. Since its release in 1994, the interface has been extended to support not only message passing primitives, but also provides support for one-sided operations.

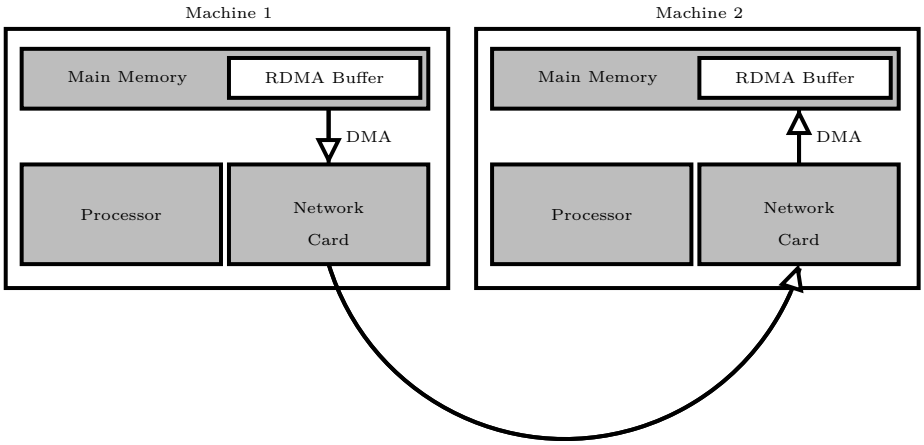


Figure 2.1: Data transfer using Remote Direct Memory Access (RDMA).

2.2.2 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a mechanism to directly access data in remote memory regions across an interconnection network. In most implementations, the network card contains a Direct Memory Access (DMA) engine to read from and write to parts of main memory. By using this DMA engine, the network card can access memory without having to interrupt the processor. As a consequence, the operating system is not aware of the data access happening through the network. Since the processor is not involved in the transfer, it remains available for processing, which allows the system to interleave computation and communication (see Figure 2.1). Furthermore, this mechanism makes it possible to place data directly at specific locations in main memory, thus eliminating the need for any intermediate buffers. RDMA implements zero-copy messaging, significantly reduces latency, and enables fast data transfers. However, in many implementations, buffers need to be registered with the network card before they are accessible over the interconnect. During the registration process, the memory is pinned such that it cannot be swapped out, and the necessary address translation information is installed on the card. These registration operations can occur a significant overhead [FA09, Fre10]. Although this process is needed for many high-speed networks, it is worth noting that some network implementations support registration-free memory access [PFH+02, CEH+11]. Sections of main memory that are accessible over the network are referred to as *memory regions*.

RDMA as a hardware mechanism does not specify the semantics of a data transfer. Most modern networks provide support for one-sided and two-sided memory accesses. Two-sided operations represent traditional message-passing semantics in which two processes are actively involved in the communication and need to be synchronized. One-sided operations on the other hand, represent memory access semantics in which only the source process is involved in the remote memory access. The processor on the target machine is not interrupted and is unaware of the memory access happening through the network card. In order to efficiently use remote one-sided memory operations, multiple programming models have been developed, the most popular of which are the Remote Memory Access (RMA) and the Partitioned Global Address Space (PGAS) concepts.

2.2.3 One-Sided Communication Primitives

Remote Memory Access (RMA) is a shared memory programming abstraction. RMA provides access to remote memory regions through explicit one-sided read and write operations. These operations move data from one buffer to another, i.e., a read operation fetches data from a remote machine and transfers it to a local buffer, while the write operation transmits the data in the opposite direction. Data located on a remote machine can therefore not be loaded immediately into a register, but needs to be first read into a local main memory buffer. Using the RMA memory abstractions is similar to programming non-cache-coherent machines in which data has to be explicitly loaded into the cache-coherency domain before it can be used and changes to the data have to be explicitly flushed back to the source in order for the modifications to be visible on the remote machine. The processes on the target machine are generally not notified about the RMA access, although many interfaces offer read and write calls with remote process notification. Apart from read and write operations, some RMA implementations provide support for additional functionality, most notably remote atomic operations. Examples of such atomic operations are remote fetch-and-add and compare-and-swap instructions.

RDMA-capable networks implement the functionality necessary for efficient low-latency, high-bandwidth one-sided memory accesses. Combining RDMA-enabled hardware with

RMA communication can be used to drastically reduce the overall costs of large data transfers. RDMA-capable networks enable zero-copy communication from the network perspective, while the ability of RMA to place data at specific addresses in remote memory can eliminate the need for expensive copy operations within the application. However, it is worth pointing out that RMA programming abstractions can also be used over networks which do not support RDMA, for example by implementing parts of the required operations in software [NMF10].

RMA has been designed to be a thin and portable layer compatible on top of many lower-level data movement interfaces. Therefore, RMA has been adopted by many libraries and interfaces as their one-sided communication and remote memory access abstraction. In this dissertation, we will have a more detailed look at RDMA Verbs [HCPR12], the interface used by InfiniBand and RoCE networks, and MPI-3 [Mes12], the version of MPI that introduced support for one-sided communication.

Alternatives to RMA Programming

It is worth noting that RMA is not the only programming model for using one-sided operations. Partitioned Global Address Space (PGAS) is a programming language concept for writing parallel applications for large distributed memory machines. PGAS assumes a single global memory address space that is partitioned among all the processes. The programming model distinguishes between local and remote memory. This can be specified by the developer through the use of special keywords and annotations. PGAS is therefore usually found in the form of a programming language extension and is one of the main concepts behind several languages, such as Co-Array Fortran or Unified Parallel C [CDM+05]. Local variables can only be accessed by local processes, while shared variables can be written or read over the network. In most PGAS languages, both types of variables can be accessed in the same way. It is the responsibility of the compiler to add the necessary code to implement a remote variable access. This means that from a programming perspective, a remote variable can directly be assigned to a local variable or a register and does not need to be explicitly loaded into main memory first as is the

case with RMA. When programming with a PGAS language, the developer needs to be aware of implicit data movement when accessing shared variable data, and careful non-uniform memory access (NUMA) optimizations are necessary for applications to achieve high performance [SWS+12, LLS+16].

2.2.4 Two-Sided Communication Primitives

When an application wants to use message-based communication, the receiver of a message first needs to register the designated receive buffer with the network card. Afterwards, a descriptor element for that buffer is inserted into a *receive queue*. When sending a message, in contrast to an RMA write operation, the initiator does not need to specify a target address when creating a two-sided data transfer request. Rather, the target network card takes the head element of the appropriate receive queue, verifies that the destination buffer is of sufficient size, and then transfers the data to that location.

Since the processor is completely bypassed, there is no automatic buffering within the network stack of the operating system and data is directly written to a user-level buffer by the DMA engine on the network card. In contrast to traditional socket programming, this mode of operation assumes that the application is able to manage its own communication buffers, in particular that it is able to allocate buffers of sufficient size or has advance knowledge of the sequence of incoming messages. An application that does not posses this information needs to allocate receive buffers of sufficient size (i.e., maximum possible size the process can ever receive) and quantity. It also has to monitor the state of the receive queue carefully in order to make sure that the queue does not get drained completely.

2.2.5 Network Programming with RDMA Verbs and MPI

In this dissertation, we are going to evaluate algorithms that are written against two communication interfaces: RDMA Verbs, a low-level API used by the InfiniBand [Inf07, HCPR12] network, and MPI-3 [Mes12], a high-level communication interface used by many HPC applications (see Section 2.1.3).

Terminology used in the RDMA Verbs Interface

The RDMA Protocol Verbs Specification [HCPR12] describes the interface between applications and RDMA-enabled network cards. It is the basis of the *ibVerbs* interface, the low-level interface of InfiniBand. Before an RDMA operation can take place, the application needs to establish connections and create the necessary queues and data structures.

- ***Device Context*:** The application can request a list of devices and can start using the device by opening one or more device-specific *context* objects. The context is the root object and all data structures will be created within a certain context.
- ***Protection Domain*:** The *protection domain* (PD) is a simple security mechanism in order to isolate different objects. Objects can only access and manipulate other objects within the same protection domain.
- ***Memory Region*:** A region of memory that is registered with the network card and is accessible through RDMA operations is referred to as a *memory region* (MR). After the registration, the region is identified by the local network card by its unique *steering key* (SKEY). When a remote entity creates an RMA operation targeting this region, the request needs to include the *remote key* (RKEY). Furthermore, a region has a starting address in the virtual address space of the application and a size, like any regular user-space buffer.
- ***Queue Pair*:** The specification proposes queue-based communication between the application and the network card. Within a certain protection domain, a *queue pair* (QP) can be created. As the name suggest, a queue pair contains two types of queues: an outgoing *send queue* (SQ) and an incoming *receive queue* (RQ). The send queue is used for outgoing send, write, and read operations, while the receive queue, holds descriptors of the buffers that can be consumed by inbound messages. Two queue pairs can be linked together such that the queue pair functions as the connection abstraction between two endpoints.
- ***Work Request*:** The content of the send and receive queues are descriptors that are referred to as *work requests* (WR). For outgoing operations, a work request

posted to the send queue describes the operation that needs to be executed as well as the buffers to operate on. For incoming messages, the work requests within the receive queue contain information about the buffers that can be used to receive data.

- ***Completion Queue*:** Special types of queues are used to notify the application about the completion of work requests. Once a request completed, the network card usually generates a corresponding *work completion* (WC) element and inserts it into the correct *completion queue* (WQ). Each of the two queues within a queue pair has a completion queue, that can also be shared.

- ***Shared Receive Queue*:** By default a queue pair has a dedicated send and receive queue. For incoming operations, some applications do not want to check each queue pair individually but rather have one common receive queue for all connections, i.e. a *shared receive queue* (SRQ). As the name suggest, this type of receive queue can be used by multiple queue pairs.

More details on the Verbs interface in terms of connection management, buffer registration, and the remote memory operations can be found in Appendix A.

Terminology used in the Message Passing Interface

MPI is a widely used interface in HPC applications. It provides similar functionality to the one present in RDMA Verbs interface. However, in addition to data transfer mechanisms, it also offers a rich high-level interface for many common communication patterns, such as data shuffling and reduce operations.

- ***MPI Process*:** The fundamental unit of parallelism within an MPI application is the MPI *process*. Many implementations map MPI processes to system-level processes. It is the responsibility of the library to always chose the most appropriate communication mechanism based on the relative distance of the processes, e.g. shared memory within a single node and RMA operations for processes on different machines.

With this model, the processes remain – to a large extent – unaware of the distribution, which is a key element when building large-scale distributed applications.

- ***Communication Group*:** Some operations, such as reduce operations, require the involvement of multiple or all processes. `MPI_COMM_WORLD` is the default communication group to which every process belongs. As applications start to get more complex, it becomes less feasible to always involve all processes. To that end, the application developer can create arbitrary groups of processes, often called *communication groups* or *communicators*.

- ***Rank*:** Within a communication group, each process is identified by a unique number, i.e. the *rank* of a process within the group. Mpi provides mechanisms to translate between the different ranks a process can have in different communication groups. System-wide, each process can be uniquely identified by the rank it has in the `MPI_COMM_WORLD` group.

- ***Collective Operation*:** An operation that requires the involvement of multiple processes is called a *collective operation*. A collective call operates on a communication group and every process has to participate in the operation, even if it has nothing to contribute directly. Some collective calls are blocking and represent an implicit point of synchronization within the execution of the program. Examples of collective operations are reduce operations, but also many memory management operations, such as window allocation, are implemented as collective primitives.

More details on the Mpi interface in terms of connection management, buffer registration, and the remote memory operations can be found in Appendix B.

2.3 Database Systems and Data Processing

In this section, we provide an overview of relational database systems, their workloads, and related data processing frameworks. Furthermore, we highlight recent trends in non-relational database systems, such as key-value stores and graph database systems.

2.3.1 Traditional Relational Database Systems

A relational database management system (RDBMS) is a system for storing, organizing, and querying data. In the relational model, a data entry is represented as *tuple*. Tuples are grouped into *relations*. The relational model was first described by Edgar F. Codd in 1969. The interface used to communicate with most relational database systems is the Structured Query Language (SQL). SQL provides methods to define a schema of the data (domain definition language), insert, update, and delete data (data manipulation language), and query said data (query language). The RDBMS has a cost-based optimizer that, given a SQL statement (e.g., a query), can determine the optimal way to execute the query. For example, if data from two tables needs to be combined, i.e., joined, the optimizer determines the most efficient join order and decides which algorithms to use. Because the schema is well-defined, a relational database system can not only optimize the incoming queries but also the storage layout. The logical view of the data, i.e., relations and tuples, is decoupled from the way data is physically stored. This flexibility allows system designers to optimize the RDBMS for specific workloads.

Processing large amounts of insert, update, and delete operations is referred to as *online transaction processing* (OLTP), while processing vast amounts of queries is called *online analytical processing* (OLAP). Hybrid workloads containing both a large number of transactions and queries are called mixed workloads. Closely coupling transactions and analysis is referred to as *hybrid transactional-analytical processing* (HTAP). As of to date, most database systems are either optimized for OLTP or for OLAP. This is often reflected in the storage layout. OLTP transactions manipulate individual records. In order to benefit from locality, attributes belonging to the same record should be stored consecutively. This storage layout is called a *row store*. In contrast, OLAP workloads often require aggregations on columns. In order to benefit from Single-Instruction-Multiple-Data (SIMD) vector instructions offered by modern processors, it is beneficial to store data in the form of columns, i.e., in *column store* format. Specialized in-memory database engines and system extensions using one or multiple of these data layouts have been developed [GKP+10, DFI+13, LCC+15] Recently, there has been a growing focus on building

new architectures that reduce workload interference and are able to provide and maintain good performance for hybrid and mixed workloads [MGBA17, Mak17, LMK+17].

In order to compare the performance of database systems, several benchmarks have been developed, each testing different elements of the system. Some widely-used benchmark suites are created by the Transaction Processing Performance Council (TPC). The TPC-C [Tra10] benchmark simulates a complete order-entry environment in which users execute transactions (e.g., new order entry, warehouse management, shipment tracking) against the database, while the TPC-H [Tra17] and TPC-DS [Tra18a] benchmarks mostly consist of a suite of business-oriented ad-hoc queries that test the query execution engine.

A relational database system will provide the application with guarantees when it comes to the atomicity, consistency, isolation, and durability (ACID) of transactions. The same is true for queries that execute while concurrent data modifications are taking place. A database system can offer many different levels of transaction isolation guarantees and, if desired by the user, can relax some assumptions in order to achieve higher performance. Nevertheless, providing the highest levels of transaction isolation usually requires significant amounts of communication and coordination, especially when the execution of a query or transaction involves multiple machines.

Given the increase of data volume and the resulting need to scale beyond a single machine, a platform of choice for data management are rack-scale clusters composed of several multi-core machines connected by a high-throughput, low-latency network. The adoption of rack-scale architectures has been further accelerated through the recent introduction of several commercial database appliances. Despite the use of high-performance interconnects, there has been little work on how to fundamentally re-architect database systems and algorithms for these types of networks. Recent efforts use remote memory to expand the main-memory storage capacity of a single machine [LDSN16]. SAP HANA [FML+12] is a widely-used database system that requires cache-coherent shared memory to operate. Their rack-scale solution involves the use of SGI computers. Unique to SGI is that the cluster nodes can be configured to provide the illusion of a single machine with shared memory of several terabytes. High-performance networks are used to efficiently run the

cache-coherency protocol. Oracle Exadata [Ora12] is a commercial, rack-scale solution that uses InfiniBand to connect compute and storage servers. It integrates hardware and software optimized for running the Oracle Database. Oracle RAPID [BKG+18] is a research project targeting large-scale data management and analysis. RAPID is designed from the ground up with hardware-software co-design in mind. Its goal is to provide high performance while consuming less power in comparison to the modern database appliances. Its processing engines have been designed around a new custom processor called the Data Processing Unit (DPU) and a novel Data Movement System (DMS) that in part relies on high-performance networks.

2.3.2 Non-Relational Database Systems

In order to allow systems to dynamically scale to a large number of machines (e.g., in elastic cloud environments), provide high-availability, support new types of workloads, and keep the design complexity of the system at a minimum, many data management systems have been created that do not use the relational model. As an alternative, data is stored in the form of key-value pairs, wide columns, complex documents, multi-dimensional cubes, graphs, or other application-specific data structures.

Key-value stores use the associative array as their fundamental data model. The array is often referred to as a map or dictionary. Some key-value stores have been optimized to run on high-performance networks and use RDMA to accelerate the communication [JSL+11, DNCH14, KKA14]. A wide column store is a two-dimensional key-value store. Wide column stores must not be confused with the column-oriented storage layout described in the previous section. Document-oriented database systems are designed to work with semi-structured data. Documents encapsulate and encode information in a standard format or encoding, such as XML or JSON. Document databases usually do not enforce that a specific schema is maintained. Graph database engines focus on the rapid traversal of the relationships (i.e., the edges) between objects (i.e., the vertexes). A triple store is a special kind of graph database optimized to process subject-predicate-object triples. All these systems are commonly referred to as Not-only-SQL (NOSQL) database systems.

NoSql systems often give up strong consistency guarantees in favor of scalability and elasticity. They offer a concept of *eventual consistency* in which changes are propagated to all compute nodes only at a later point in time, allowing for some period during which queries do not operate on a consistent view of the data. These systems converge to a stable and consistent state in the future, which is different from the strict Acid guarantees of a traditional Rdbms that do not allow any intermediate inconsistent state to be visible.

RocksDB [Fac18] is a high-performance, embedded database system for key-value data. It is used as a storage engine in multiple data management services at various web-scale enterprises. FaRM [DNCH14, DNN+15] and Herd [KKA14] are experimental key-value store systems that use Rdma to speed up access to the data on high-speed interconnects. Google BigTable [CDG+06], Apache HBase [Apa18d], and Apache Cassandra [Apa18a] are examples of wide column store implementations focusing on performance, scalability, elasticity, and fault-tolerance. Apache CouchDB [Apa18b] and MongoDB [Mon18] store collections of independent documents. These systems provide support for meta-data and indexing structures for managing and retrieving stored documents. They also ensure limited transactional guarantees within the scope of a single document. Neo4j [Neo18] is a native graph database. A core concept in Neo4j is index-free adjacency by which neighbors of a vertex can directly be referenced without the need for an index lookup. Other graph engines such as Microsoft Trinity [SWL13] are based on a distributed key-value store system. Cray offers the Cray Graph Engine [RHMM18], a triple-based graph processing system that can be used on supercomputers.

2.3.3 Modern Data Processing Frameworks

MapReduce [DG04] is a programming model for analyzing large sets of data. It is designed with massive parallelism in mind. The processing can be divided into three stages. In the *Map*-phase, each worker is assigned to a particular section of the data to which it applies the *map function* that transforms the input and generates intermediate data. This output it redistributed in the *Shuffle*-phase. Finally, in the third stage, the intermediate output of the first phase is grouped together by the *reduce function* to produce the final result. For

processing sets of data that can be partitioned, MapReduce frameworks and implementations make it possible to run at very large scale and process vasts amount of input data. The most popular open-source implementation is Apache Hadoop [Apa18e]. Optimization strategies targeting RDMA-capable networks have been proposed to accelerate MapReduce implementations [WIL+13].

Dryad [IBY+07, YIF+08] is an infrastructure platform allowing the programmer to write several sequential programs and connect these programs using one-way channels, thus structuring the computation as a directed graph. Naiad [MMI+13] is the successor of Dryad. It is a distributed system for executing parallel and cyclic dataflow programs. The computational model that is the core of these systems is the *dataflow* model. Stateful vertices – representing parts of the computation – send and receive messages with a logical time-stamp along directed edges. These time-stamped messages are used by the system to track the progress and schedule parts of the computation. *Timely* is also the name of an open-source implementation of this model that is currently being extended to make use of advanced network features such as RDMA.

Performing computations over streams of data and reacting to specific events with low latency has become increasingly important for many business-critical applications. Several systems and data processing frameworks have been proposed to perform these types of computation, some of which operate on batches of data while others – from the perspective of the programmer – process each record individually. Apache Spark [Apa18f] is an example of a cluster-computing framework that can be used for micro-batch processing, while Apache Flink [Apa18c] is a record-at-a-time, high-throughput, low-latency stream processing engine that can execute dataflow programs on streams. These systems have also been tested and evaluated on RDMA-enabled network technologies [LWI+14, LCX16]. Other projects are a combination of a stream processing framework and a database system. By using a shared-scan approach and a delta-main data-structure Analytics-in-Motion (AIM) [BEG+15] proposes a system that can store and run analytics on data coming from streaming systems.

2.3.4 Modern Distributed Relational Database Systems

Modern relational database systems seek to provide the same performance as NoSql systems when it comes to Oltp workloads and the same scalability as modern data processing frameworks when it comes to Olap queries. At the same time, these systems maintain the same Acid guarantees as traditional database systems. Database systems designed from the ground up to achieve these goals are called NewSql systems. Although the internal architecture of NewSql systems varies significantly from one system to the next, all of them are primarily based on the relational model. Many systems target specific workloads and can therefore incorporate workload-specific optimizations.

H-Store [KKN+08] is an example of a database system that was developed as a parallel, row-storage Rdbms that is designed to operate in a distributed cluster of shared-nothing nodes. The data is partitioned into disjoint subsets. Each subset is assigned to a single-threaded execution engine. Because of this design, transactions touching a specific tuple are serialized by the execution engine. VoltDB [SW13] follows a similar concept. It is a scale-out database that relies on horizontal partitioning down to the individual hardware context (i.e., processor thread). Data is partitioned and managed on a per-core level. It uses a combination of snapshots and command logging to ensure durability. Calvin [TDW+12] is a partitioned database system optimized for Oltp workloads. In order to achieve high scalability, Calvin uses a transaction scheduling mechanism that provides deterministic ordering guarantees and reduces contention costs associated with distributed transactions. Google Spanner [CDE+12] is a large-scale distributed database system that focuses on geographic distribution. The system not only uses a lock-based concurrency control mechanism, but also relies on the Global Positioning System (Gps) and atomic clocks to serialize transactions at a global scale.

ScyPer [MRR+13] extends the HyPer [KN11] database to provide scalable analytics on remote replicas by propagating updates either using a logical or physical redo log. The systems has also been extended to run on high-performance networks [RMKN15, Röd16]. BatchDB [MGBA17, Mak17] is an in-memory database engine designed for hybrid Oltp and Olap workloads. It achieves good performance, provides a high level of data fresh-

ness, and minimizes workload interaction by using specialized replicas. Each replica is optimized for a specific workload. A high-performance update propagation mechanism and scheduling system ensures that queries get routed to the appropriated replica and can operate on the latest version of the data. The MemSQL [Sha14] database is a distributed database system that uses a two-level hierarchy composed of leaf and aggregation nodes. An aggregator is responsible for distributing the queries across leaf nodes and aggregating results. MemSQL places great emphasis on using lock-free data structures to support parallel and concurrent execution of queries and transactions. Other systems like NAM-DB [BCG+16, SBK+17] and Tell [LPEK15, PBB+17] propose new architectures that target RDMA-capable, high-throughput, low-latency interconnects. These new designs have been evaluated on modern InfiniBand networks and exhibit good performance and scalability.

2.3.5 Concurrency Control in Relational Database Systems

There are several concurrency control mechanisms that are being used in database systems, e.g., Two-Phase Locking (2PL), optimistic concurrency control (OCC), multi-version concurrency control (MVCC), and timestamp ordering (TS). These mechanisms have been evaluated and compared against each other in recent publications [YBP+14, HAPS17]. Furthermore, there has been a significant focus on building reliable, fair, starvation-free locking mechanisms for HPC systems as well as cloud environments [Bur06]. The design of these systems focuses on achieving a high throughput for a small number of highly-contented locks and often expects coarse-grained locks to be taken. Many recent RMA locking mechanisms offer support for reader/writer (shared/exculusive) locks, but are difficult to extend to more sophisticated locking schemes (e.g., intention locks) given the current network technology [SBH16, YCM18].

A traditional lock table of a database system offers a large number of locks, most of which are not contended. Typically, a relational database system does not make assumptions about the granularity of the locks. Therefore, it offers several lock modes, including intention locks. The dominant locking method used in database management systems is multi-level granularity locking [BHG87, GR92]. It solves the problem that different transactions

Table 2.1: Multi-level granularity locking.

(a) Request mode compatibility matrix.

	NL	**IS**	**IX**	**S**	**SIX**	**X**
NL	✓	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	✓	
IX	✓	✓	✓			
S	✓	✓		✓		
SIX	✓	✓				
X	✓					

(b) Lock mode computation.

Granted Modes	**Lock Mode**
{NL}	NL
{IS, {IS}}	IS
{IX, {IX} {IS}}	IX
{S, {S} {IS}}	S
{SIX, {IS}}	SIX
{X}	X

need to lock and modify resources with a different granularity, e.g., one transaction might only be interested in modifying a single record, while others need to access and modify entire tables or ranges. If locks are too coarse-grained, concurrent processing of transactions targeting different tuples might not be possible, resulting in reduced throughput. On the other hand, fine-grained locks add a significant overhead when processing a transaction that is forced to acquire many locks. Multi-level granularity locking makes use of the hierarchical structure of the data in a database, e.g., a schema contains tables, which in turn contain ranges of tuples. Locks can be acquired at any level in the hierarchy.

Before a lock can be acquired on a certain object, all its parent elements (i.e., the elements that contain the object) need to be locked as well. To that end, the locking scheme does not only provide shared (S) and exclusive (X) locks, but also intention locks. The intention shared (IS) and intention exclusive (IX) locks are used to signal that the transaction intents to lock elements further down in the hierarchy in either shared respectively exclusive mode. The shared and intention exclusive mode (SIX) is a combination of the S and IX modes, locking an element in shared mode while stating that one or more child elements will be locked in exclusive mode. Finally, the no lock (NL) mode is used to indicate that the lock is not taken. The overall mode of the lock is dependent on the types of locks that have been granted and have not yet been released. The compatibility matrix for each

combination of lock modes as well as the resulting lock mode are shown in Tables 2.1a and Table 2.1b respectively.

In order to guarantee serializability, many database systems use strict Two-Phase Locking (2PL). In addition, in a distributed system, the Two-Phase Commit (2PC) protocol ensures that data modified by a transaction on different nodes is in a consistent state before the transaction is allowed to commit.

3

Rack-Scale Join Processing

Recent advances in processor architecture caused by multi-socket and multi-core systems have triggered a re-design and re-evaluation of database algorithms, in particular join processing [KSC+09, BLP11, BTAÖ13, BATÖ13, Bal14, BTAÖ15]. The ability to efficiently process complex queries over large sets of data is a fundamental requirement for database systems and data processing frameworks. Joins appear frequently in query workloads and are commonly accepted to be compute- and communication-intensive. Therefore, they usually dominate the query execution costs in OLAP workloads. As a result, the relational join operator is considered one of the most important database operators. Some join implementations are carefully tuned to the underlying hardware in order to provide the best performance possible, i.e., *hardware-conscious algorithms* [AKN12, BTAÖ13, BATÖ13, Bal14], while other approaches argue that modern hardware is good enough at hiding most cache and TLB misses such that careful tailoring of the algorithm to fit the hardware is no longer needed, i.e., *hardware-oblivious algorithms* [KSC+09, BLP11]. In addition, there are two major algorithmic approaches for implementing joins, namely *hash*-based and *sort*-based algorithms. The former approach creates and probes hash tables, while the latter relies on sorting both input relations.

Given the increase in data volume, rack-scale clusters composed of several multi-core machines connected by a high-throughput, low-latency network have become increasingly popular for data management and analysis. Processing vasts amounts of relational data involves complex, large join operations. Thus, these systems would benefit from having efficient distributed join algorithms that are aware of machine boundaries and employ communication mechanisms suited for the underlying network technology. In this chapter, we investigate hardware-conscious hash and sort-merge join algorithms that are optimized to run on RDMA-capable interconnects. We explain the necessary modifications to both algorithms in order to run on a rack-scale cluster, evaluate and compare both approaches on two generations of InfiniBand networks, and propose detailed performance models for each of the two join algorithms.

3.1 Problem Statement and Novelty

Several low-latency networks provide Remote Direct Memory Access (RDMA) as a light-weight communication mechanism to transfer data. RDMA is essential for high-performance applications because the data is immediately written or read by the network card, thus eliminating the need to copy the data across intermediate buffers inside the operating system (see Section 2.2.2). This in turns reduces the overall costs of large data transfers. However, these performance advantages can only be leveraged through thoughtful management of the RDMA-enabled buffers, the correct and careful use of one-sided RMA operations, and the ability of the algorithm to interleave processing and communication [FA09, Fre10, BLAK15, Röd16, BCG+16, BAH17, BMS+17, LYB17].

In this chapter, we design, model, and evaluate novel join algorithms optimized for this new generation of networks. Building upon recent work on main-memory multi-core join algorithms [BTAÖ13, BATÖ13, BTAÖ15], this dissertation is one of the first to analyze how join algorithms need to be adapted in order to run on a modern rack-scale database cluster. In the description of the algorithm we place special emphasis on the registration, de-registration, and management of RDMA-enabled buffers as these are critical compo-

nents in the data exchange phase. To generalize our findings, we develop a novel theoretical model allowing us to predict the performance of the algorithms based on the system configuration and input data size. Last but not least, we evaluate our prototype implementation on two database clusters. The experimental results validate the accuracy of the analytical model and provide new insights on the importance of interleaving computation and communication, the role of the network bandwidth, the effects of skew, and the impact of different relation sizes.

3.2 Distributed Join Algorithms using RDMA

In this section, we explain implementation details of the radix hash and sort-merge join algorithms. The focus is on the implementation of the network-centric phases of both join algorithms, namely the network-partitioning and the network-sorting phases, respectively.

3.2.1 Radix Hash Join Algorithm

The radix hash join is a hardware-conscious, main-memory hash join algorithm that operates in two stages. First, both input relations R and S are divided into disjoint partitions according to the join attributes. The goal of the partitioning stage is to ensure that the resulting partitions fit into the private cache of each processor core. Next, a hash table is built over each partition of the inner relation and is probed using the data of the corresponding partition of the outer relation. Producing partitions and hash tables that fit into the cache has a major impact on performance compared to accessing large hash tables, which would result in a higher cache miss rate [MBK02]. Figure 3.1 illustrates the execution of the radix hash join on two machines. In this example, data is first divided into four partitions. Once the data has been exchanged, a second partitioning pass further subdivides the data. In the illustration, the fan-out of the partitioning passes is set to four. Given the number of cache lines and the number of TLB entries, modern processors support a larger fan-out of 512 to 2048 partitions without any significant loss in performance [BTAÖ13, Bal14, BTAÖ15].

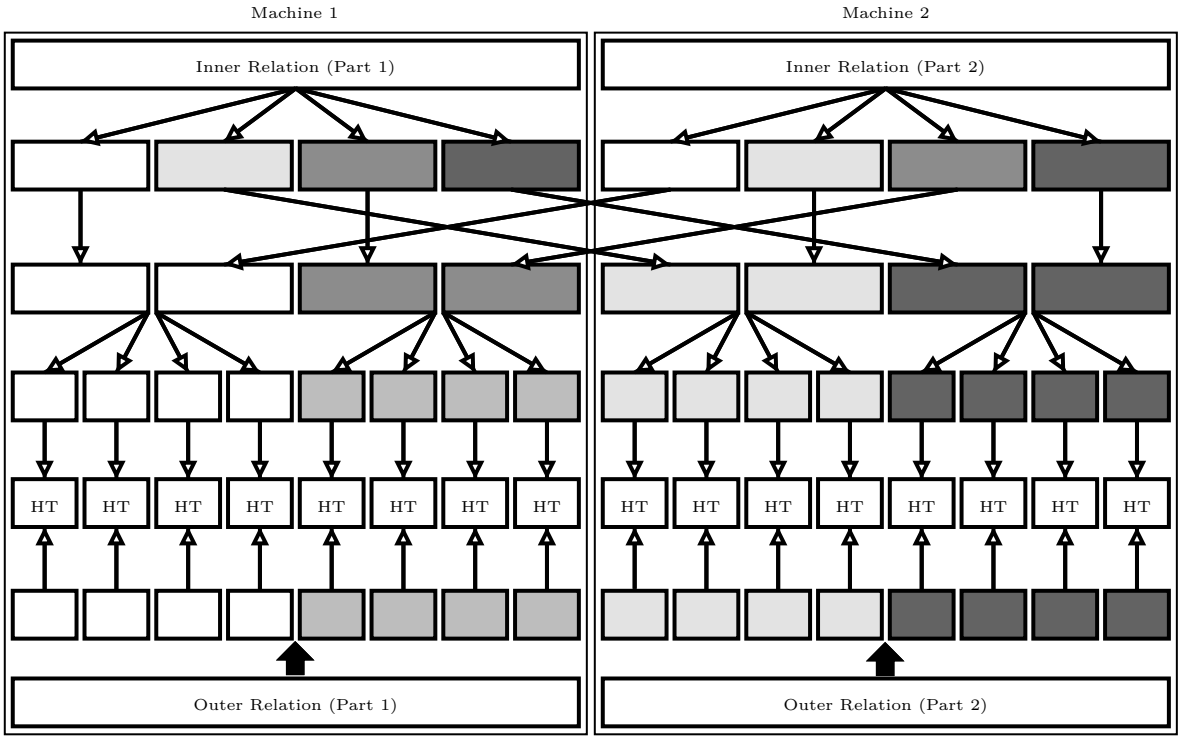


Figure 3.1: Execution of the radix hash join on two machines.

Histogram Computation

As a first step in the algorithm, all threads compute a histogram over the input data. By assigning the threads to non-overlapping sections of the input relations of equal size, we can ensure an even load distribution among the worker threads. The histogram contains information about the number of tuples in each partition that is about to be created. All the threads within the same machine exchange their histograms and combine them into one machine-level histogram providing an overview of the data residing on a particular machine. Computing the machine-level histograms is identical to the histogram computation of the join algorithm described by Balkesen at al. [BTAÖ13].

The machine-level histograms are then exchanged over the network. They can either be sent to a predesignated coordinator or distributed among all the nodes. The machine-level histograms are in turn combined into a global histogram providing a global overview of the partition sizes and the necessary size of the buffers which need to be allocated to store

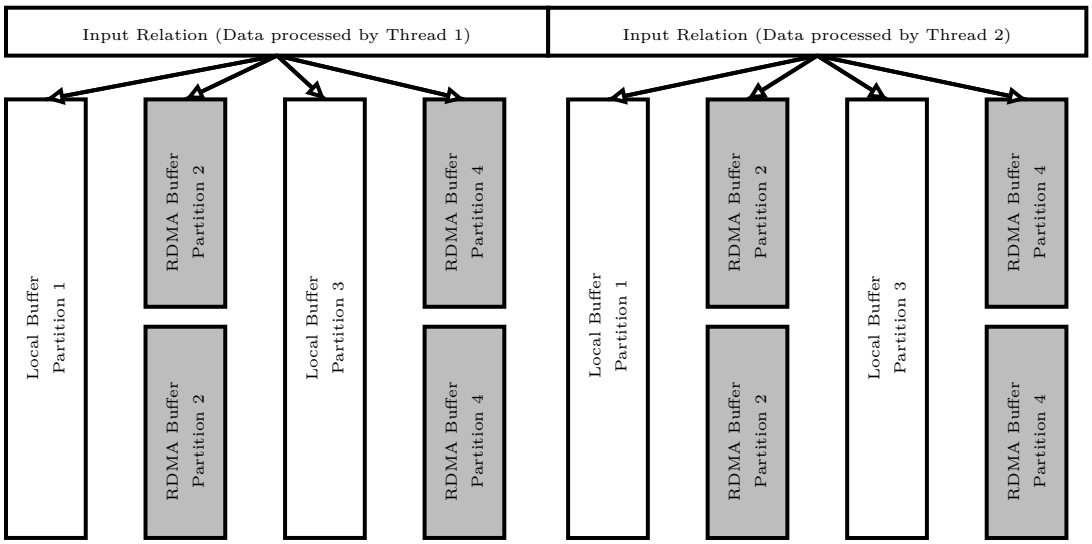


Figure 3.2: Buffer management for outgoing, partitioned data.

the data received over the network. From the machine-level and global histograms the join algorithm computes a machine-partition assignment for every node in the cluster. This assignment can be dynamic or static. The algorithm computing the machine-partition assignment is independent of the rest of the join algorithm and several approaches have been proposed to distribute the data, for example, taking data skew into account [RIKN16].

Partitioning Phase

The purpose of the partitioning phase of the radix hash join is to ensure that the partitions and hash tables fit into the processor cache. For the distributed radix join, we additionally want to ensure maximum resource utilization. In particular, we need to be able to assign at least one partition to each processor core. Therefore, the number of partitions needs to be at least equal to the total number of cores in order to prevent cores from becoming idle. In the multi-pass partitioning phase of the algorithm we distinguish between two different types of partitioning passes: (i) a network-partitioning pass that interleaves the computation of the partitions with the network transfer, and (ii) local partitioning passes that partition the data locally in order to ensure that the partitions fit into the processor cache. The latter does not involve any network transfer.

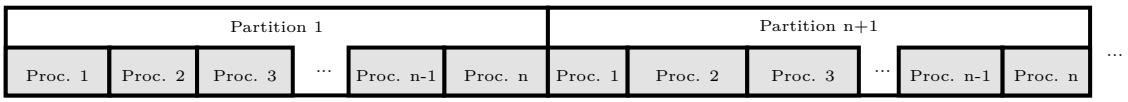


Figure 3.3: Buffer management for incoming, partitioned data.

To efficiently use the asynchronous nature of RDMA, the data needs be transmitted over the network in parallel with the computation. When designing the algorithm, we need to avoid having a separate network transmission phase during which the processor cores are idle. To achieve these goals, we introduce the concept of a network-partitioning pass in which the data is partitioned and distributed in parallel. Crucial for high performance processing is the management of the partitioning buffers, in particular the ability to reuse existing RDMA-enabled buffers [FA09]. For each partition which will be processed locally, a thread receives a local buffer for writing the output. Based on the histogram computation, the required size of the local buffers can be determined such that local buffers do not overflow. Remote partitions need to be transmitted over the network. For processing remote partitions, a thread receives multiple fixed-sized RDMA-enabled buffers. Data belonging to a remote partition is partitioned directly into these buffers. When a remote buffer is full, it will be transmitted over the network to the target machine.

In order to continue processing while a network operation is taking place, at least two RDMA-enabled buffers are assigned to each thread for a given partition. The buffers assigned to one partition can be used in turn and reused once the preceding network operation completed. Figure 3.2 shows the assignment of buffers to threads and partitions. To hide the buffer registration costs, the RDMA-enabled buffers are drawn from a pool containing pre-allocated and pre-registered memory. All buffers, both local and RDMA-enabled buffers, are private to each thread, such that no synchronization is required while partitioning the input relations.

On the target machine, the incoming data needs to be written to the correct address within main memory. In order to use one-sided operation and avoid any intermediate data copies, the buffer used for receiving data is structured as follows: Based on the global histogram, the size of each partition is known. The partitions that have been assigned to a particular

node are stored consecutive in memory. Within a partition, machine-level histograms provide the necessary information to determine exclusive locations where the individual processes can write their part of the data. This is done by computing a prefix-sum over all machine-level histograms. Figure 3.3 shows that the resulting data layout ensures that tuples belonging to the same partition are written consecutive in main memory.

The goal of the partitioning phase is to speed up the build-probe phase by creating cache-sized partitions. To ensure that the resulting partitions fit into the private processor caches, subsequent partitioning passes not involving network operations might be required depending on the input data size.

Build and Probe Phase

In the build-probe phase, a hash table is built over the data of each partition of the inner relation. Data from the corresponding partition of the outer relation is used to probe the hash table. Because there is no data dependency between two partitions, this phase can be processed in parallel. The result containing the matching tuples can either be output to a local buffer or written to RDMA-enabled buffers, depending on the location where the result will be processed further. Similar to the partitioning phase, we transmit a RDMA-enabled buffer over the network once it is full. To be able to continue processing, each thread receives multiple output buffers for transmitting data. The buffers can be reused once the proceeding network operation completed.

When operating on a skewed data set, the computation of the build-probe phase of a partition can be shared among multiple threads. If the partition of the outer relation contains more tuples than a predefined threshold, it is split into distinct ranges. Multiple threads can then be used to probe the hash table, each operating on its range of the outer relation. No synchronization between the threads is needed as the accesses to the common hash table are read-only. Skew on the inner relation can cause that the hash tables do not fit into the processor cache. This can be compensated by splitting the large hash table into a set of smaller hash tables. In this case, the tuples of the outer relation need to be used to probe multiple tables, however, this probing can also be executed in parallel.

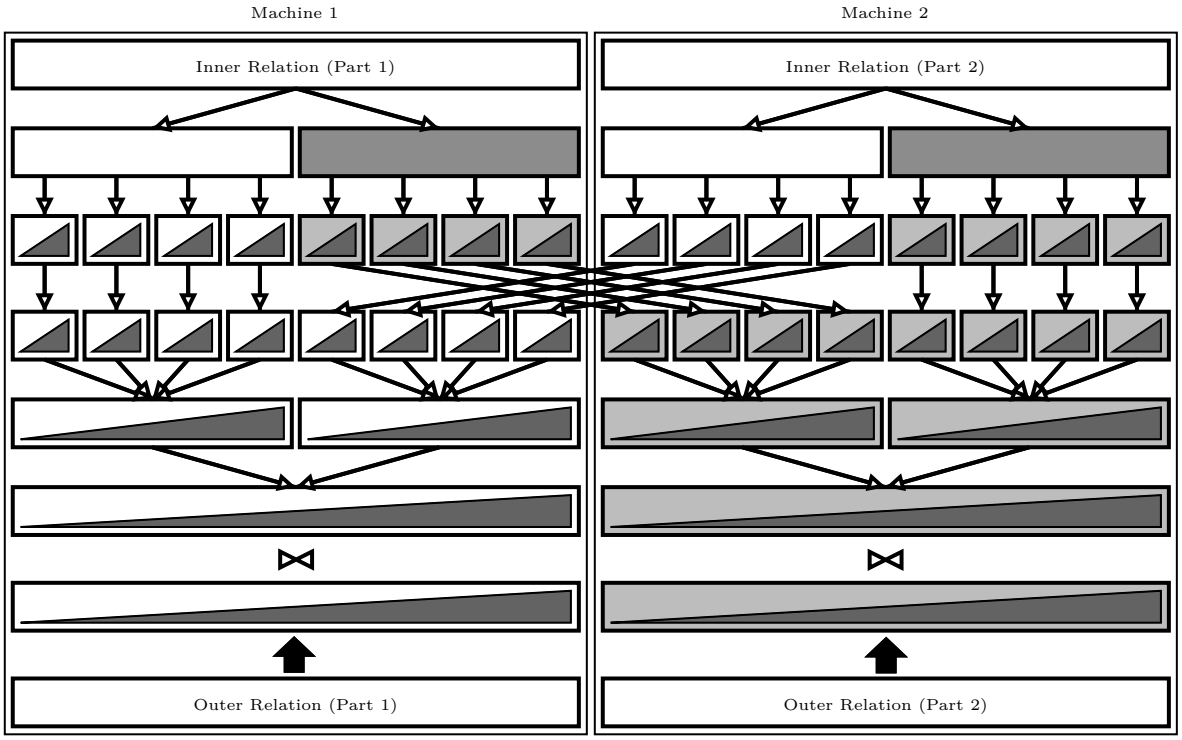


Figure 3.4: Execution of the sort-merge join on two machines.

3.2.2 Sort-Merge Join Algorithm

The sort-merge join presented in this section uses modern hardware features such as large SIMD vectorization units to speed up the sorting operation [BATÖ13]. It is composed out of two main steps. First, the input is being sorted using merge-sort. While data is being sorted, it is also redistributed among the nodes through an interleaved sorting and exchange strategy. After both input relations R and S have been sorted, they are scanned sequentially to find matching tuples. This part of the algorithm is commonly referred to as the merge phase and should not be confused with the merge operations that are performed as part of the sorting phase. To avoid confusion, we will refer to the last phase as the *matching* phase. Figure 3.4 illustrates the execution of the sort-merge algorithm with two machines. Data is first partitioned among the nodes and small sorted runs are created during the exchange phase. These runs are merged using a multi-way merge tree to produce a sorted output that can be easily scanned to find matching tuples.

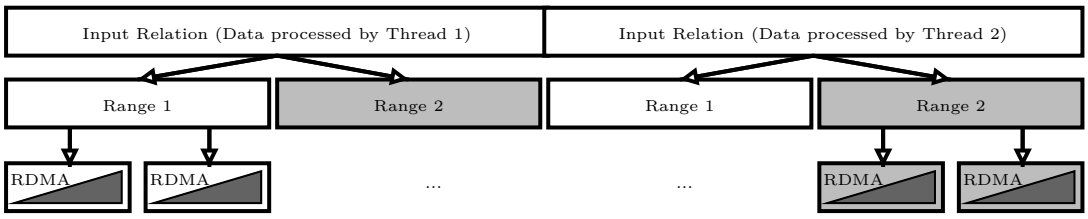


Figure 3.5: Buffer management for outgoing, sorted data.

Sorting Phase

As a first step in the distributed sorting algorithm, each thread partitions the input data. We use range-partitioning to ensure that matching elements in both relations will be assigned to the same machine for processing. Because we use a continuous key space, we can split the input relations into ranges of identical size. For relations where this is not the case, the algorithms would need to be augmented with a splitter-based approach to find the optimal splitter values to sub-divide the relation [FM70, HY83, DNS91, KK93, SK10]. To prevent cores from becoming idle, we create as many partitions as there are cores.

Afterwards, each thread creates runs of fixed size, which are sorted locally. For sorting, we use an in-cache sorting algorithm with vector instructions based on sorting networks [BATÖ13, Bal14]. The sorted output is written into an RDMA-enabled buffer. When a run has been sorted, it is immediately transmitted asynchronously to the target machine. While the network transfer is taking place, the process can continue sorting the next run of input data, thus interleaving processing and communication. Figure 3.5 illustrates this process with two threads partitioning and sorting the input data. To avoid contention on the receiving node, not every process starts sorting the first partition. Instead, process i starts processing partition $i+1$. Individual runs are appended one after the other. Because the amount of data in a partition is not necessarily a multiple of the run size, the last run might contain fewer elements.

On the target machine, each remote process has an exclusive range into which the process can write data (see Figure 3.6). These ranges are sized according to the information contained in a histogram generated during the partitioning phase. Next, the algorithm

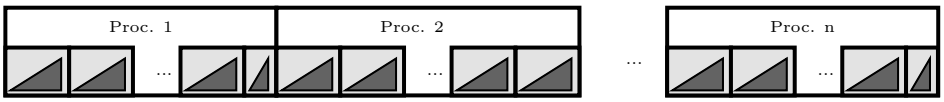


Figure 3.6: Buffer management for incoming, sorted data.

merges the sorted runs into one single relation. Multiple runs are merged simultaneously using an in-cache merge tree. The merge process is accelerated through the use of large SIMD vector instructions. To balance computation and memory bandwidth requirements within the nodes, multi-way merging is used to reduce the amount of round-trips to and from main memory [BATÖ13, Bal14].

Matching Phase

After the data has been sorted, the relations are partitioned into p ranges – where p is the number of processor cores – and all elements within a range have been sorted. The partitioning step of the sorting phase ensures that matching elements from both relations have been assigned to the same process. At this stage, every thread can start joining its part of the data. No further communication or synchronization between processes is necessary. Scanning both relations is a linear-time operation, and modern hardware is optimized for very fast sequential accesses, making the matching phase highly efficient. Two head pointers keeps track of the current position in their respective relation. The join condition is evaluated on the head elements and, if it holds, an output tuple is generated.

3.3 Performance Models

In Section 2.1.1, we explained the importance of performance models in order to understand the behavior of an HPC application. In this section, we provide analytical models of the proposed algorithms that will be compared against the results of the experimental evaluation. The goal of these models is to provide a lower bound for the execution of the join algorithms and be able to judge the efficiency of our implementations with respect to

this bound. Furthermore, having a performance model is useful to predict the influence future hardware (e.g., faster networks) will have on the execution time of the algorithms.

3.3.1 Radix Hash Join Algorithm

The distributed radix hash join starts by computing a global histogram in order to determine the size of the communication buffers and memory windows. The time required to compute the histograms $T_{\text{histogram}}$ depends on the size of both input relations (R and S) and the rate P_{scan} at which each thread can scan over the data. The total number of threads in the system depends on the number of machines N_{machines} and the number of threads per machine $N_{\text{threads/machine}}$.

$$T_{\text{histogram}} = \frac{|R| + |S|}{N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{scan}}} \tag{3.1}$$

The partitioning speed of the network-partitioning pass is composed of two parts: (i) the speed at which tuples are written locally to the respective buffers $P_{\text{partition}}$, and (ii) the speed at which tuples belonging to remote partitions can be transmitted over the network to a remote machine P_{network}. The network bandwidth bw is shared equally among all the threads running on the same machine.

$$P_{\text{network}} = \frac{bw}{N_{\text{threads/machine}}} \tag{3.2}$$

Assuming uniform distribution of the data over all machines in the system (i.e., N_{machines}), we can estimate that $(|R|+|S|) \cdot \frac{1}{N_{\text{machines}}}$ tuples belong to local partitions. The rest is send to remote machines. At this point, the system can either be limited by the partitioning speed of the threads (compute-bound) or by the available network bandwidth on each host (network-bound). A system is network-bound if the tuples belonging to remote partitions are output at a faster rate than the network is able to transmit.

$$\frac{N_{\text{machines}} - 1}{N_{\text{machines}}} \cdot P_{\text{partition}} > P_{\text{network}} \tag{3.3}$$

In systems that are compute-bound, the overall processing rate is fully determined by the partitioning speed of each thread $P_{\text{partition}}$. The entire system is composed of N_{machines} machines, each of which contains $N_{\text{threads/machine}}$ processor cores.

$$P_{\text{partition\_network}} = N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{partition}} \tag{3.4}$$

On the other hand, if the system is network-bound, meaning the partitioning speed exceeds the maximum network processing speed, threads have to wait for network operations to complete before they are able to reuse RDMA-enabled buffers. The observed partitioning speed of each thread is a combination of $P_{\text{partition}}$ and P_{network}.

$$\begin{aligned} P_{\text{partition\_net\_bound}} &= \frac{1}{\frac{1/N_{\text{machines}}}{P_{\text{partition}}} + \frac{(N_{\text{machines}}-1)/N_{\text{machines}}}{P_{\text{network}}}} \\ &= \frac{N_{\text{machines}} \cdot P_{\text{partition}} \cdot P_{\text{network}}}{(N_{\text{machines}} - 1) \cdot P_{\text{partition}} + P_{\text{network}}} \end{aligned} \tag{3.5}$$

From the above, we can determine the partitioning rate of the network pass in systems that are limited by the performance of the network.

$$P_{\text{partition\_network}} = N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{partition\_net\_bound}} \tag{3.6}$$

Local partitioning passes do not involve any network transfer and all threads in the system partition the data at their maximum partitioning rate. Therefore, the global processing speed of this phase increases with the total number of available processor cores.

$$P_{\text{partition\_local}} = N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{partition}} \tag{3.7}$$

The partitioning phase is composed of d passes, one of them involving the transfer of the data over the network, the other $d-1$ passes operate on local data only and do not involve the network. The partitioning passes operate at a rate of $P_{\text{partition\_network}}$ and $P_{\text{partition\_local}}$,

respectively. Therefore, we can derive an expression for the total time required to partition both input relations of size $|R|$ and $|S|$.

$$T_{\text{partition}} = \left(\frac{1}{P_{\text{partition\_network}}} + \frac{d-1}{P_{\text{partition\_local}}} \right) \cdot (|R| + |S|) \tag{3.8}$$

In the build phase, a hash table is created for each partition R_i of the inner relation. Because the hash table fits into the processor cache, the build operation can be performed at a high rate P_{build}. The number of generated partitions depends on the partitioning fan-out of each pass $N_{\text{fan-out}}$ and the number of partitioning passes d. Creating the hash tables requires one pass over every element of the inner relation R.

$$\begin{aligned} T_{\text{build}} &= (N_{\text{fan-out}})^d \cdot \frac{|R_i|}{N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{build}}} \\ &= \frac{|R|}{N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{build}}} \end{aligned} \tag{3.9}$$

Data from the corresponding partition S_i of the outer relation is used to probe the hash table. Probing the in-cache hash tables requires a single pass over the outer relation S.

$$\begin{aligned} T_{\text{probe}} &= (N_{\text{fan-out}})^d \cdot \frac{|S_i|}{N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{probe}}} \\ &= \frac{|S|}{N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{probe}}} \end{aligned} \tag{3.10}$$

Equation 3.10 does not include the time required to materialize the output of the join. The cost of fetching additional payload data over the network depends on the selectivity of the join and the size of the payload fields.

The hash join executes the histogram computation, partition, build, and probe phases sequentially. Assuming no interference between the phases and an ideal synchronization

of all threads, we can determine a lower bound for the execution time of the radix hash join algorithm T_{rhj}.

$$T_{\text{rhj}} = T_{\text{histogram}} + T_{\text{partition}} + T_{\text{build}} + T_{\text{probe}} \tag{3.11}$$

3.3.2 Sort-Merge Join Algorithm

Both algorithms share a lot of commonalities. For example, the sort-merge join starts by creating histograms and by range-partitioning the data, similar to the radix hash join. Therefore, the time required to compute the histograms of the sort-merge join $T_{\text{histogram}}$ is identical to the one described by Equation 3.1. The partitioning phase does not involve the network and is purely local. Therefore, its time can be determined through Equation 3.7, keeping in mind that the exact value of the partitioning rate between both algorithms is subject to change due to the different partitioning fan-out and function used to compute the assignment of tuples to partitions.

Once the data has been partitioned, individual runs of fixed size l are created. The total number of runs depends on the size of the two relations and the size of each run l.

$$N_{\text{R}} = \frac{|R|}{l} \quad \text{and} \quad N_{\text{S}} = \frac{|S|}{l} \tag{3.12}$$

A run is sorted and then transmitted asynchronously to the target node. While the network transfer is taking place, the process can continue sorting the next run of input data, thus interleaving processing and communication. The performance of the algorithm can either be limited by the rate $P_{\text{sort\_run}}$ at which a run can be sorted (compute-bound) or by the available network bandwidth bw shared by all $N_{\text{threads/machine}}$ threads on the same machine (network-bound).

For CPU-bound systems, the total rate at which all $N_{\text{R}} + N_{\text{S}}$ runs can be sorted and transmitted is equal to the local sorting rate $P_{\text{sort\_run}}$. This sorting rate is dependent on

the length l of the sorted runs, the number of threads per machine, and the total number of machines in the system.

$$P_{\text{sort}} = N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{sort\_run}}(l) \tag{3.13}$$

For systems that are network-bound, the sorting rate is a combination of the local sorting rate $P_{\text{sort\_run}}$ and the rate at which sorted runs can be transmitted over the network P_{network}. In this case, P_{network} is identical to the one described by Equation 3.2. Assuming uniform distribution, $(|R| + |S|) \cdot \frac{1}{N_{\text{machines}}}$ tuples remain local.

$$\begin{aligned} P_{\text{sort\_net\_bound}} &= \frac{1}{\frac{1/N_{\text{machines}}}{P_{\text{sort\_run}}(l)} + \frac{(N_{\text{machines}}-1)/N_{\text{machines}}}{P_{\text{network}}}} \\ &= \frac{N_{\text{machines}} \cdot P_{\text{sort\_run}}(l) \cdot P_{\text{network}}}{(N_{\text{machines}} - 1) \cdot P_{\text{sort\_run}}(l) + P_{\text{network}}} \end{aligned} \tag{3.14}$$

Therefore, we can determine the rate at which the threads can sort and transmit all runs in a network-bound system.

$$P_{\text{sort}} = N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{sort\_net\_bound}} \tag{3.15}$$

The total time required to sort the input tuples into small sorted runs depends primarily on the input size.

$$T_{\text{sort}} = (N_{\text{R}} + N_{\text{S}}) \cdot \frac{l}{P_{\text{sort}}} = \frac{|R| + |S|}{P_{\text{sort}}} \tag{3.16}$$

After a thread has sorted its input data, it waits until it has received all the sorted runs of its range from the other nodes. Once all the data has been received, the algorithm starts merging the sorted runs using m-way merging, which combines multiple input runs into one sorted output. Several iterations over the data might be required until both relations

are fully sorted. The number of iterations $d_{\{R,S\}}$ needed to merge the data depends on the number of runs $N_{\{R,S\}}$ and the merge fan-in $N_{\text{fan-in}}$.

$$\begin{aligned} d_R &= \lceil \log_{N_{\text{fan-in}}}(N_R/(N_{\text{machines}} \cdot N_{\text{threads/machine}})) \rceil \\ d_S &= \lceil \log_{N_{\text{fan-in}}}(N_S/(N_{\text{machines}} \cdot N_{\text{threads/machine}})) \rceil \end{aligned} \tag{3.17}$$

From the depth of both merge trees and the rate P_{merge} at which each thread can perform the merge operation, we can determine the time required to merge the runs of both relations in order to create two globally sorted relations.

$$\begin{aligned} T_{\text{merge}} = d_R \cdot \frac{|R|}{N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{merge}}} \\ + d_S \cdot \frac{|S|}{N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{merge}}} \end{aligned} \tag{3.18}$$

After the sorting phase, both relations are partitioned among all the nodes. Within each partition, the elements are fully sorted. To compute the join result, each thread scans the inner relation with the corresponding partition of the outer relation at the rate P_{scan}.

$$T_{\text{match}} = \frac{|R| + |S|}{N_{\text{machines}} \cdot N_{\text{threads/machine}} \cdot P_{\text{scan}}} \tag{3.19}$$

Similar to the radix hash join, Equation 3.19 does not include the time required to materialize the output of the join as this depends vastly on the selectivity of the join.

Since all the phases described by the formulas above, i.e., histogram computation, local partitioning, sorting, merging, and matching, are executed in sequence with no overlap, the total execution time of the sort-merge join T_{smj} is equal to the sum of the execution times of each of those phases.

$$T_{\text{smj}} = T_{\text{histogram}} + T_{\text{partition\_local}} + T_{\text{sort}} + T_{\text{merge}} + T_{\text{match}} \tag{3.20}$$

3.4 Experimental Evaluation

We evaluated our implementation of the distributed join algorithms on a cluster of ten machines connected by 4x QDR and 4x FDR InfiniBand. All algorithms use RDMA Verbs as their communication interface. The goal of this evaluation is to understand how to use RDMA in the context of distributed rack-scale database systems. We also compare the distributed algorithms to highly optimized, single-machine implementations.

3.4.1 Workload and Setup

To facilitate comparisons with existing results, we use similar workloads to the ones employed by previous work on join processing [KSC+09, AKN12, BTAÖ13, Bal14, BTAÖ15, BLAK15, BMS+17]. These studies assume a column-oriented storage model in which join algorithms are evaluated on narrow 16-byte tuples, containing an 8-byte key and an 8-byte record id (RID). The record identifiers are range partitioned among the compute nodes. By default, the key values follow a uniform distribution and can occur in arbitrary order. Similar to previous work, we focus on highly distinct value joins. For each tuple in the inner relation, there is at least one matching tuple in the outer relation. The ratio of the inner and outer relation sizes which are used throughout the experiments are either 1-to-1, 1-to-2, 1-to-4 or 1-to-8. To analyze the impact of data skew, we generated two skewed datasets, with different values of the Zipf distribution: a low-skew dataset with a value of 1.05 and high-skew dataset with a skew factor of 1.20. To assign partitions to nodes, we implemented a static round-robin assignment and, for skewed workloads, a dynamic algorithm which first sorts the partitions based on their element count before assigning them evenly over all machines.

During the partitioning phase, the 16-byte `<key, RID>` tuples are compressed into 8-byte values using prefix compression. Radix partitioning groups keys with $\log(N_{\text{fan-out}})$ identical bits. The partitioning bits can be removed from the key once the tuple has been assigned to a partition. A similar operation can be applied to the common bits within a range of the sort-merge join. If an input relation contains less than 274 billion tuples (4 Tbytes

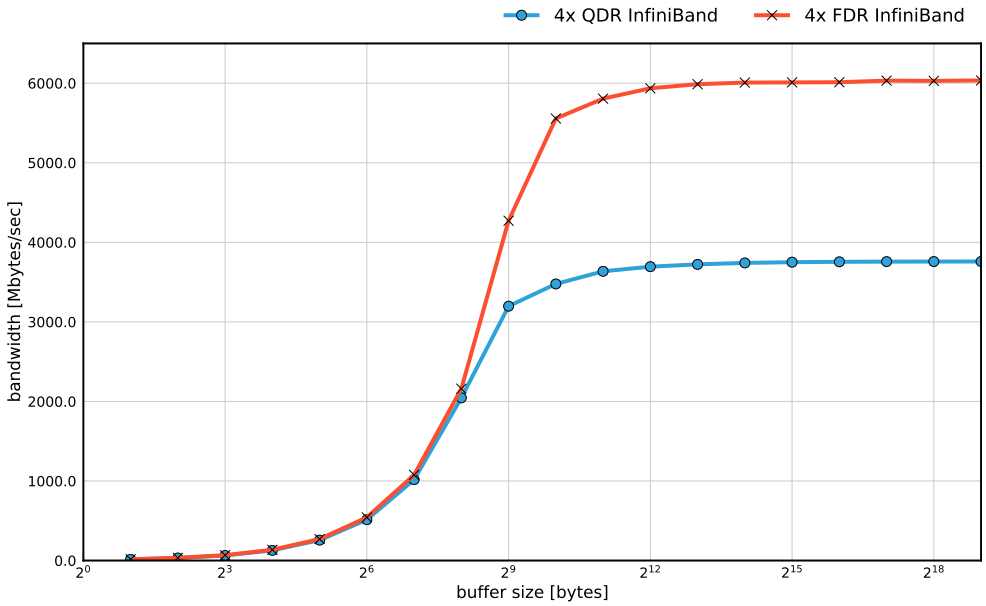


Figure 3.7: Performance of the InfiniBand network for different message sizes.

per relation), the key and the record id can be represented with 38 bits each. When the system has 4096 threads, the minimum fan-out that is needed in order to assign at least one partition to every thread is 2^{12}. Hence, a tuple can be compressed into $2 \cdot 38 - 12 = 64$ bits. This compression algorithm reduces the total amount of data that needs to be transmitted by a factor of two.

We evaluated our implementation of the distributed join on two clusters of machines connected by a QDR and FDR InfiniBand network. The machines are connected through a single InfiniBand switch. Each machine has a multi-core processor and several gigabytes of main memory. The network cards are of type Mellanox ConnectX3. Networks can either be bound by the maximum package rate which can be processed by the network card or by the available network bandwidth. Figure 3.7 shows the observed bandwidth on both the QDR and FDR network between two machines for message sizes ranging from 2 bytes to 512 Kbytes. One can observe that both systems can reach and maintain full bandwidth for buffers larger than 8 Kbytes. This means that, unless otherwise stated, the size of the RDMA-enabled buffers used in the communication phase of the algorithms is fixed to 64 Kbytes. We also measured the maximum throughput that can be achieved

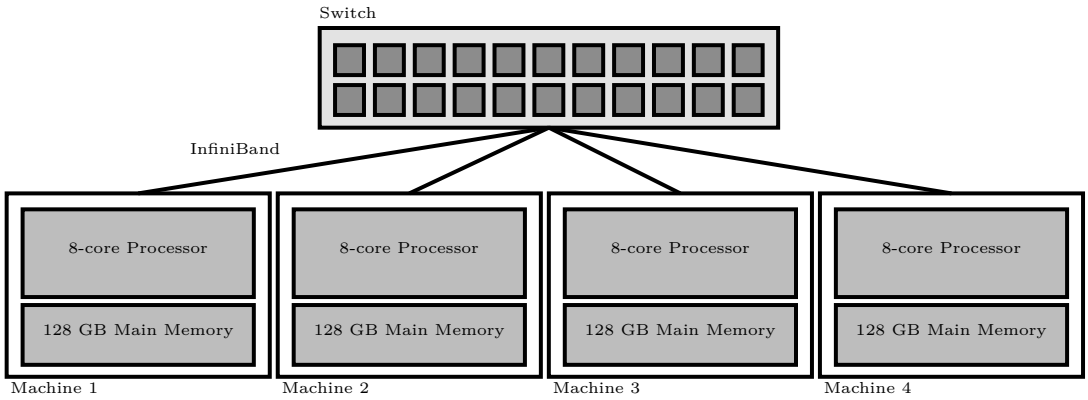
with the IPoIB compatibility layer. This layer allows conventional Tcp/Ip applications to run on InfiniBand networks. We measured a throughput of 1.4 Gbytes per second, which is significantly lower than the native InfiniBand performance. An illustration of the distributed InfiniBand setup can be found in Figure 3.8a.

In order to make our results comparable to a public baseline, we compare against two highly optimized, single-machine algorithms [BTAÖ13, BATÖ13]. Previous work noticed that the algorithm by Balkesen et al. [BTAÖ13] did not run beyond certain amounts of data [LLA+13]. We have extended the algorithm such that it can process larger data sizes. In order to have a more realistic baseline, we have also modified the algorithm to make it more Numa-aware. In particular, we created multiple task queues, one for each Numa region. If a buffer is located in region i, it is added to the i-th queue. A thread first checks the task queue belonging to the local Numa region and only when there is no local work to be done, will it check other queues. With these modifications, the single-machine algorithm reaches a throughput of 1.6 billion join argument tuples-per-second for 8-byte tuples on 32 cores. An illustration of the large multi-core server for running the single-node experiments can be found in Figure 3.8b.

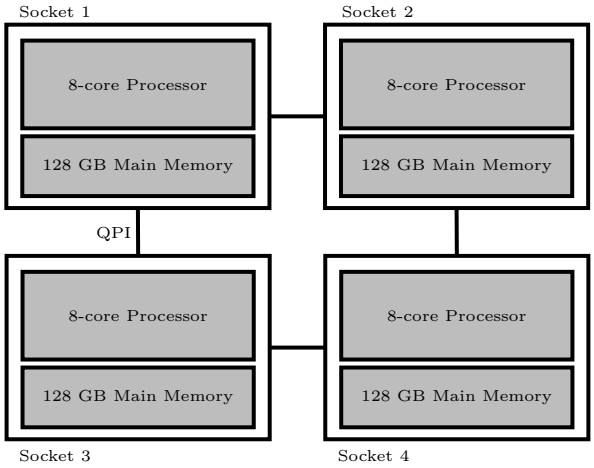
3.4.2 Comparison with Centralized Algorithms

One of the first questions to ask is how the join algorithms behaves on the different hardware configurations described in Figure 3.8. In order to be able to compare the distributed join with the implementations from Balkesen et al. [BATÖ13], we selected a high-end multi-processor server containing four sockets using eight out of the ten cores on each socket and compared it against four nodes from the Fdr and Qdr clusters. On each of the cluster machines we used eight cores. Thus, the total number of processor cores for each of the hardware configurations is 32 physical cores.

Inside the high-end server the processors are connected via QuickPath (Qpi). Each processor is attached to two neighbors. Using the Stream benchmark [McC95], we measured the bandwidth with which one core can write to a remote Numa region. The total bandwidth offered by Qpi is not fully available to a single core. On different hardware configurations,

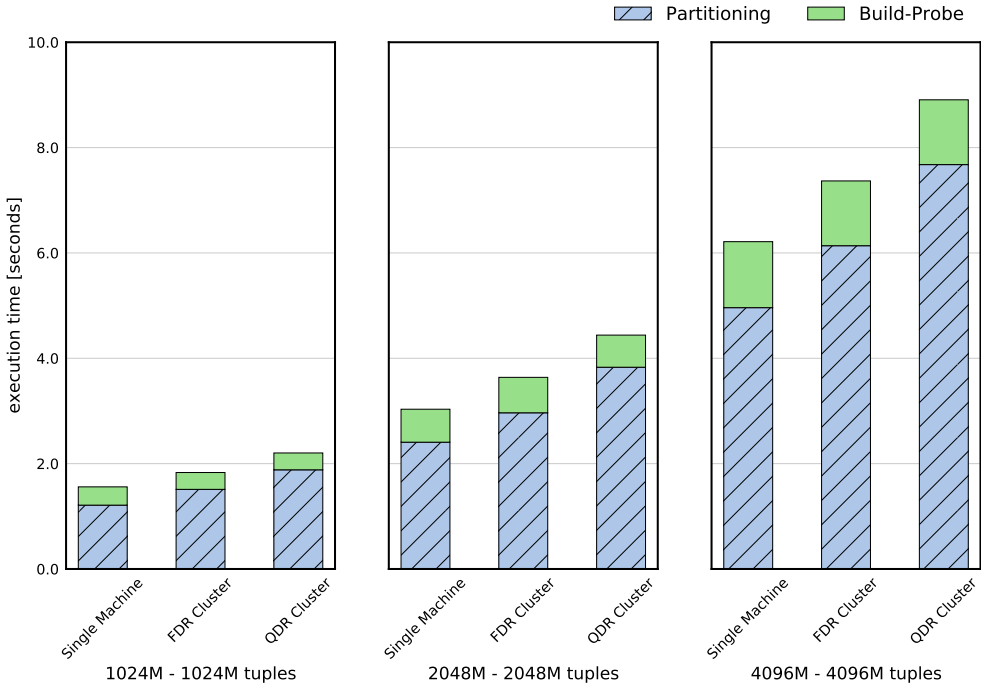


(a) Topology of the InfiniBand network.

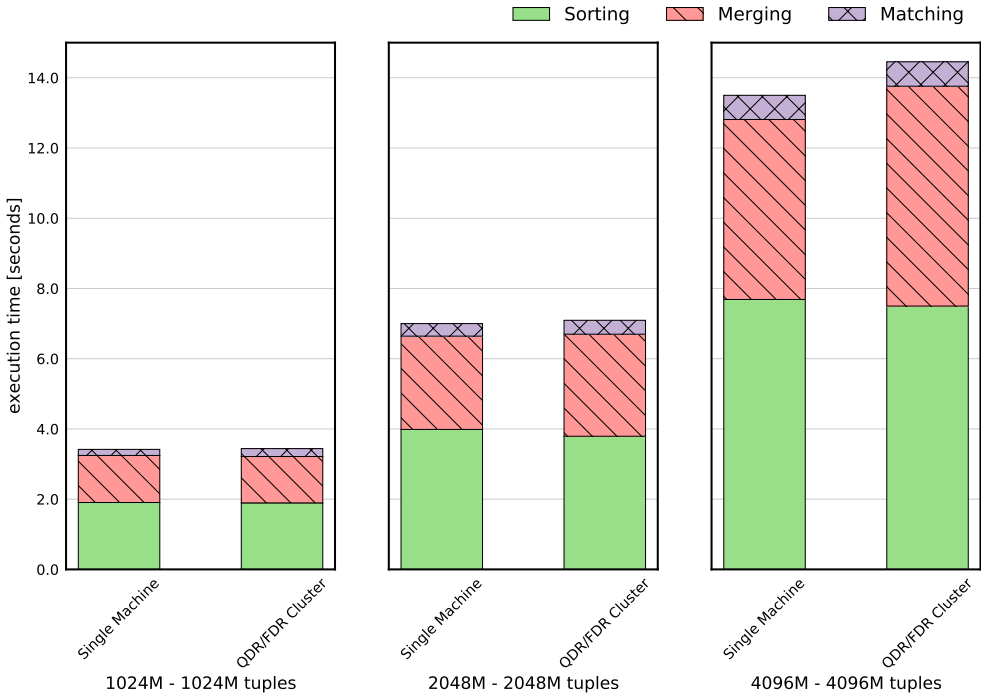


(b) Socket topology of a high-end server machine.

Figure 3.8: Experimental setup composed of a high-end server machine and a large InfiniBand cluster.



(a) Comparison of the distributed and centralized radix hash join for different data input sizes and network speeds.



(b) Comparison of the distributed and centralized sort-merge join for different data input sizes and network speeds.

Figure 3.9: Comparison of distributed and centralized join algorithms.

we measured different values for the per-core write bandwidth, even within the same processor family. In this dissertation, we show the results for the configuration which offered us the highest inter-socket bandwidth, which peaked at 8.4 Gbytes per second. For the distributed system, the measured bandwidth on the QDR network is around 3.6 Gbytes per second. The FDR network offers a higher bandwidth with a peak performance close to 6.0 Gbytes per second. Both configurations have the same amount of main memory and number of cores. The architecture of both systems is illustrated in Figure 3.8.

In the first experiment, we used three different workloads consisting of 1024 million, 2048 million and 4096 million tuples per relation. Because the baseline algorithms do not use compression, we use 8-byte tuples for the centralized algorithms and 16-byte tuples for the distributed versions. Although that means that the input size is twice as much, the latter will be compressed to 8-byte elements early in the execution as explained in Section 3.2. The results of the experiments are shown in Figure 3.9. For the hash join (see Figure 3.9a), the centralized algorithm outperforms the distributed version for all data sizes. This is expected because the algorithm has a lower coordination overhead and the bandwidth between cores is slightly higher than the inter-machine bandwidth. For large data sizes, the distribution overhead is amortized. The execution time for 4096 million tuples per relation shows an increase of less than 20%. The sort-merge join (see Figure 3.9b) has a different ratio of compute and communication. The higher compute costs of the sorting operation puts significantly less load on the network. The performance of the sort-merge join on the FDR and QDR cluster is very similar. Therefore, we will only show the numbers gathered on the FDR network in this dissertation. Both the centralized and distributed algorithms exhibit identical performance. The execution of the distributed algorithm for 4096 million tuples per relation shows an increase of less than 10% over the centralized sort-merge join algorithm.

Both results clearly show that modern network technologies have fundamentally reduced the costs of communication compared to traditional networks. The gap in performance between the internal processor interconnect and the external network is becoming narrower. Distributed database algorithms are competitive and can reach a performance comparable to that of single-machine algorithms.

3.4.3 Scale-Out Experiments

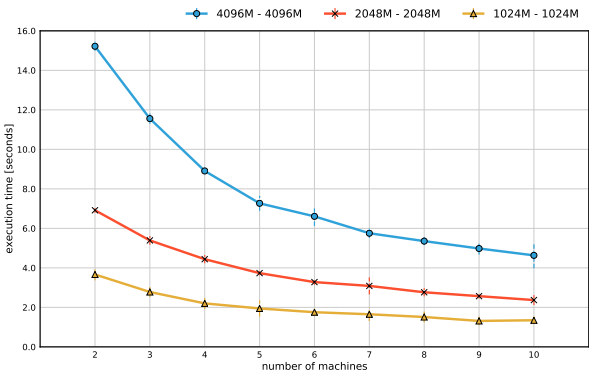
In this section, we study the behavior of the proposed algorithms as we increase the number of machines in the cluster.

Large–to–Large Joins

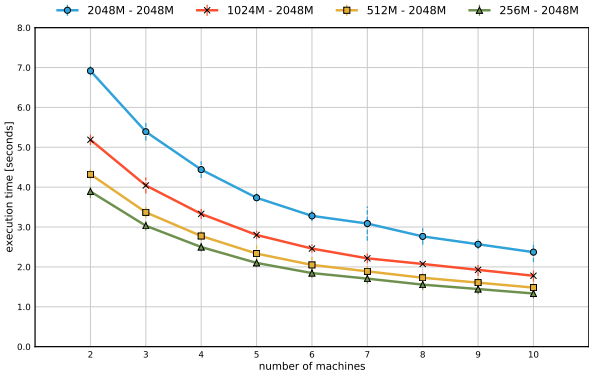
To study the impact of the input relation sizes on the performance of the distributed join algorithms, we varied the input relation sizes and the number of machines. In large-to-large table joins, both input relations are of the same size and each element of the inner relation is matched with exactly one element of the outer relation. In this experiment, we use relations ranging from 1024 million to 4096 million tuples per relation, and we increase the number of machines from two to ten machines.

Figure 3.10a, Figure 3.10c, and Figure 3.10e present the average execution time of (i) the radix hash join on the QDR cluster, (ii) the radix hash join on the FDR cluster, and (iii) the sort-merge join on the QDR and FDR cluster (identical performance due to the low network bandwidth requirements of the sort-merge join) for each of the three workloads using different numbers of machines. We can observe that the execution time doubles when doubling the amount of input data. The relative difference in execution time between the first two workloads is on average a factor of 1.89, 1.89, and 2.07, for each join algorithm respectively. The difference between the second and third workload is a factor of 2.00, 1.98, and 2.02, for each algorithm respectively. The error bars shown in Figure 3.10 represent the 95% confidence intervals.

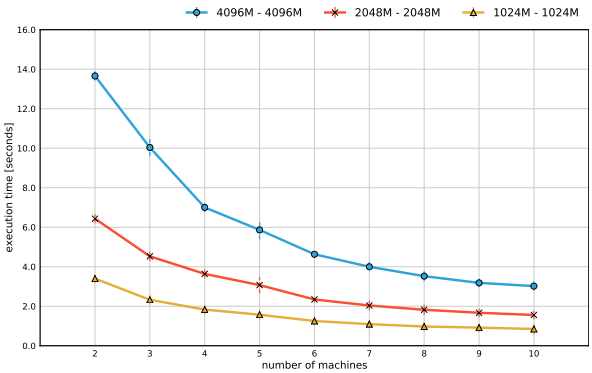
The experiment shows that the execution time for a large-to-large join increases linearly with the size of both input relations: doubling the relation sizes results in a doubling of the total execution time of the join algorithm. The execution time for all three workloads reduces as we increase the number of machines. However, we can also observe a sub-linear speed-up when comparing the configuration with two and ten nodes of the radix hash join on the QDR cluster. Assuming an optimal speed-up, this setup should lead to a five times improvement in the execution time, which cannot be observed in this experiment.



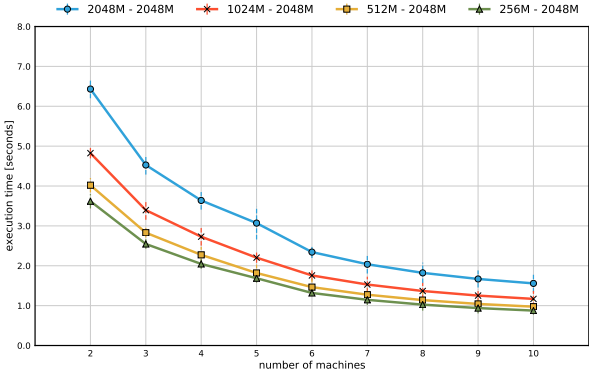
(a) Execution time of the radix hash join on the QDR cluster for large-to-large joins.



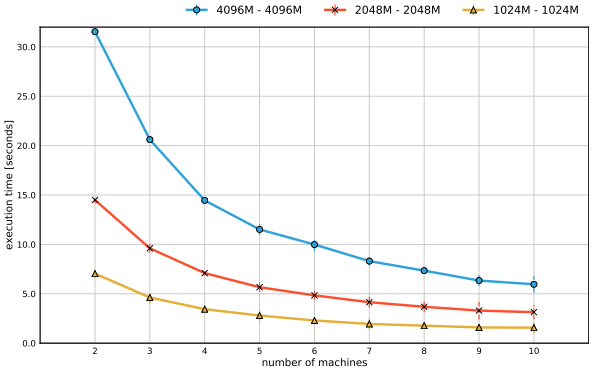
(b) Execution time of the radix hash join on the QDR cluster for small-to-large joins.



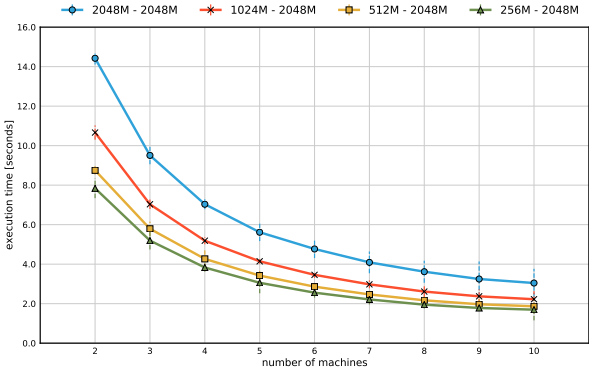
(c) Execution time of the radix hash join on the FDR cluster for large-to-large joins.



(d) Execution time of the radix hash join on the FDR cluster for small-to-large joins.



(e) Execution time of the sort-merge join on the FDR cluster for large-to-large joins.



(f) Execution time of the sort-merge join on the FDR cluster for small-to-large joins.

Figure 3.10: Execution time of the radix hash and sort-merge join algorithms for large-to-large and small-to-large joins.

Small–to–Large Joins

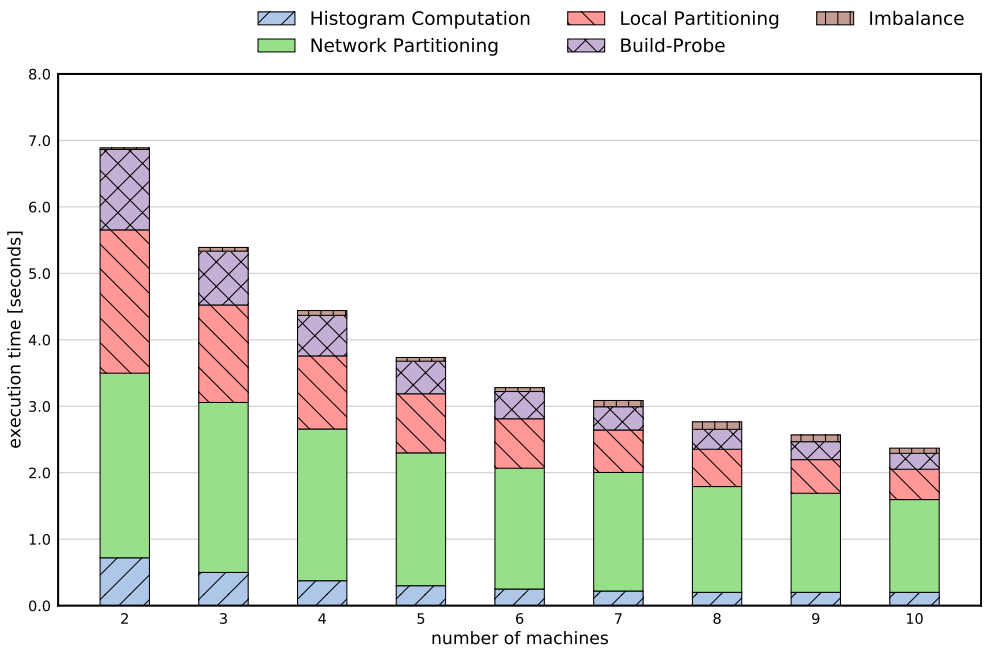
To explore the impact of the relative sizes of the inner and outer relations, we measured the performance of the distributed join using an outer relation of fixed size, composed of 2048 million tuples, and a variable number of tuples for the inner relation. As described in Section 3.4.1, the size of the inner relation ranges from 2048 million tuples (1-to-1 workload) to 256 million tuples (1-to-8 workload).

In Figures 3.10b, 3.10d, and 3.10f, we can observe that the execution time decreases when reducing the size of the inner relation. The execution of the radix hash join is dominated by the time needed to partition the data and the execution of the sort-merge join is dominated by the time required to sort the input. These costs decrease with the size of the relations. Therefore, when keeping the size of the outer relation fixed at 2048 million tuples and decreasing the number of tuples in the inner relation, we can see a reduction in the execution time by almost half when comparing the 1-to-1 to the 1-to-8 workload.

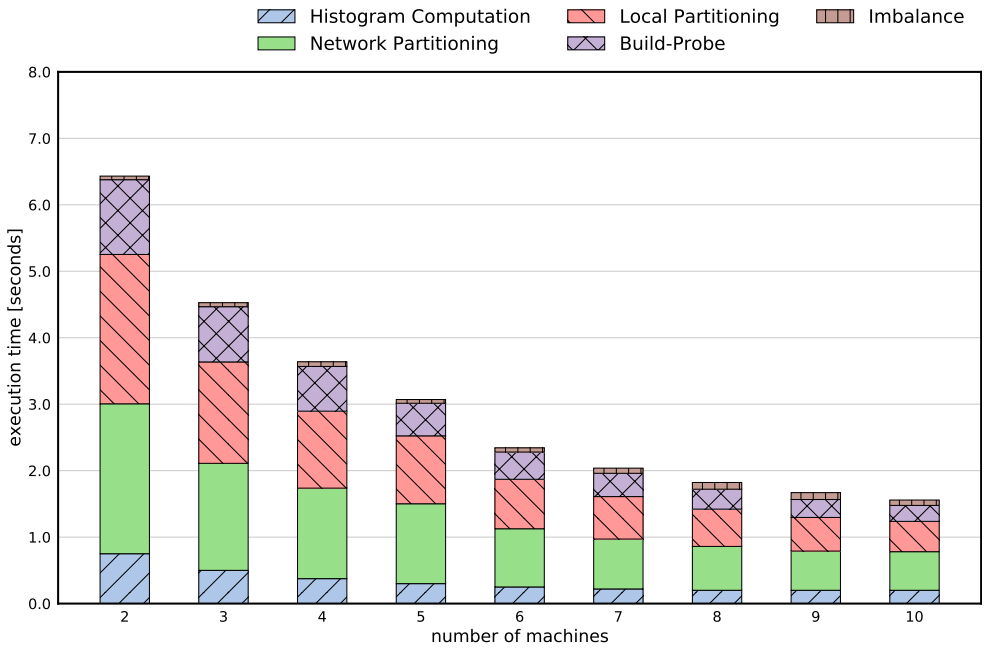
Analysis of the Radix Hash Join

In previous experiments, we observed a sub-linear reduction in the execution of the radix hash join for all relation sizes when increasing the number of machines. To understand the cause of this behavior, we take a closer look at the 2048 million $\bowtie$ 2048 million tuple join on the QDR and FDR clusters. Figure 3.11 visualizes the execution time of the different phases of the join and illustrates the effects of scale-out in more detail. Since we only consider a join operation complete once the last thread finishes, we include the difference between the sum of the averaged phase-wise execution times and the maximum execution time as the *load imbalance*.

During the first partitioning pass the data is distributed over the network. This phase is completed once all the data has been sent out and acknowledged by the receiving hosts. When increasing the number of machines from two to ten machines, we expect – in an ideal scenario – a speed-up factor of 5. However, when examining the execution time of the individual phases, one can observe a near-linear speed-up of the second partitioning



(a) Radix hash join on the QDR cluster.



(b) Radix hash join on the FDR cluster

Figure 3.11: Breakdown of the execution time of the radix hash join for 2048 million tuples per relation.

pass (speed-up by 4.79) and of the build-probe phase (speed-up by 4.57). On the QDR network, the speed-up of the first partitioning pass on the other hand is limited because the network transmission speed of 3.6 Gbytes per second is significantly lower than the partitioning speed of a multi-core machine. As a consequence, the network presents a major performance bottleneck and limits the speed-up.

On the FDR cluster, the higher bandwidth mitigates this problem for small deployments. However, with an increasing number of machines, a larger percentage of the input data needs to be transmitted over the network, which puts additional pressure on the network component and does not allow us to fully leverage the performance gains of the increased parallelism. Furthermore, adding machines to the network is likely to increase overall network congestion during the network-partitioning pass if communication is not scheduled carefully. The overall speed-up when scaling from two to ten machines is 2.92 on the QDR cluster and 4.13 on the FDR network.

Analysis of the Sort-Merge Join

The sort-merge join interleaves sorting and data transfer. Because sorting tuples is a more complex operation than looking at the radix bits of the join key, the sort-merge join has a higher compute-to-communication ratio, which has a significant effect on the scalability of the algorithm. Figure 3.12 provides a breakdown of the execution time for a 2048 million $\bowtie$ 2048 million tuple join on the FDR cluster. Because of the low bandwidth requirements during the sorting phase, the numbers gathered on the QDR cluster are identical.

We observe that the absolute execution time of the sort-merge join is significantly higher than that of the radix hash join. Despite this fact, the time required to complete all phases of the sort-merge decreases as we add more machines. This is expected in a compute-bound system. By adding additional compute nodes, the total amount of processors increases, meaning that the amount of data each processor has to sort and join is decreasing linearly. Although the compute part of the sorting phase is accelerated by a factor of 4.22, having an increased number of parallel threads and processes increases the likelihood of stragglers. In our implementation of the algorithm, a process waits for all incoming data before starting

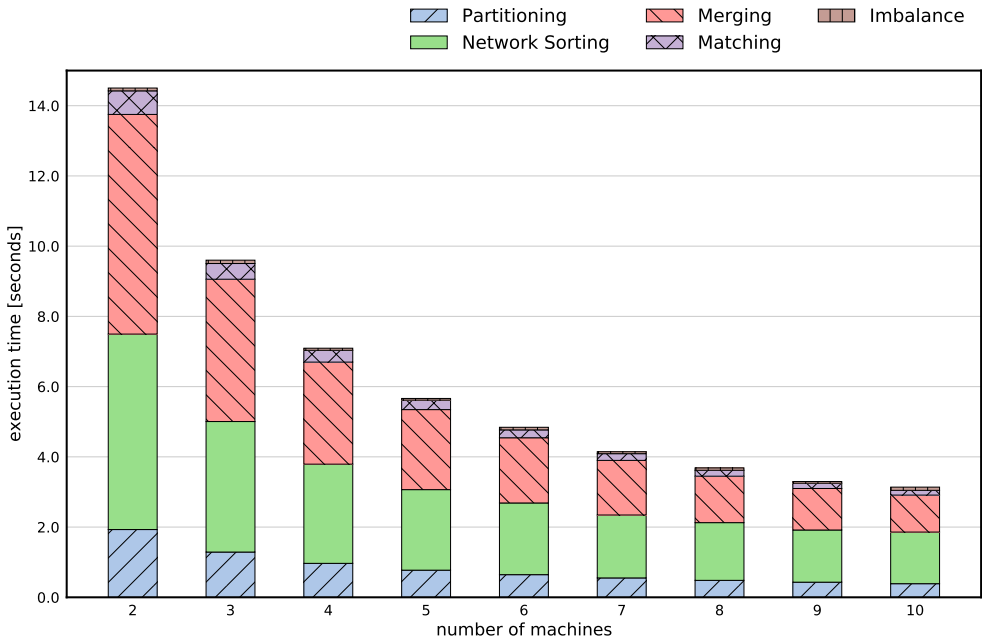


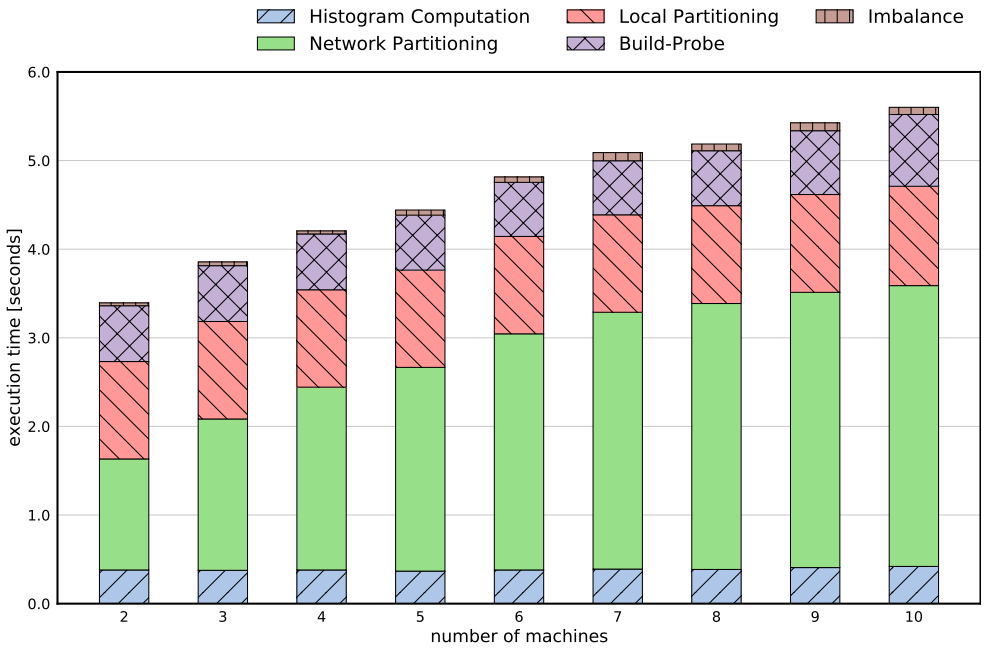
Figure 3.12: Breakdown of the execution time of the sort-merge join for 2048 million tuples per relation.

the merge operation. Therefore, the overall execution time of the sort phase that includes both processing outgoing elements and waiting for incoming tuples is only reduced by a factor of 3.79. Nevertheless, the sort-merge join is able to scale to ten nodes with an overall reduction of the execution time by a factor of 4.62, comparable to that of the radix hash join on the FDR network.

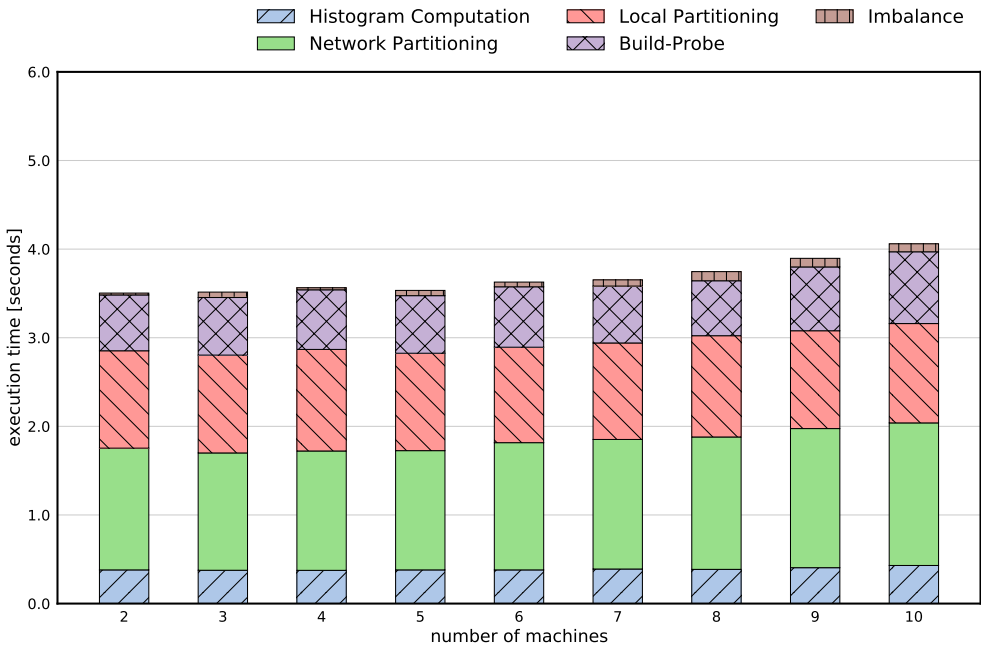
3.4.4 Scale-Out Experiments with Increasing Workload

In order to deal with ever-increasing workload sizes, a common approach is to add more resources to an existing system to maintain a constant execution time despite the increase in data volumes. In the experiment, we vary the workload size from 2×1024 million (≈30 GB) to 2×5120 million (≈150 GB) tuples. For each increase in the data size by 512 million tuples per relation, we add another machine to the system.

Figure 3.13 shows the execution time of each phase. One can observe that the algorithm maintains a constant performance for the second partitioning pass as well as the build-



(a) Breakdown of the execution time of radix hash join on the QDR cluster.



(b) Breakdown of the execution time of radix hash join on the FDR cluster.

Figure 3.13: Breakdown of the execution time of radix hash join for an increasing number of tuples and machines.

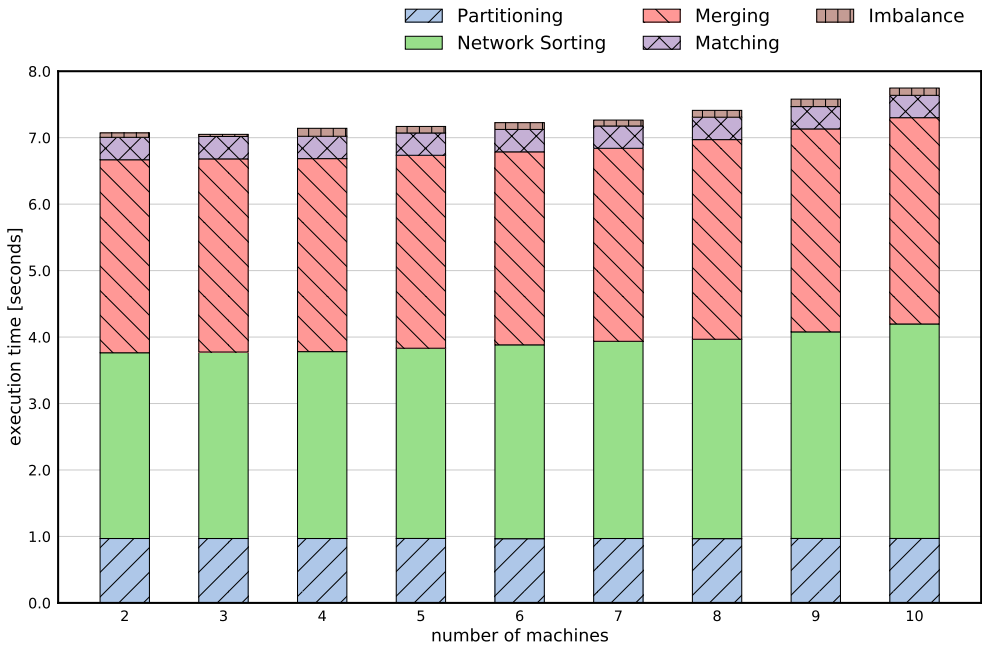


Figure 3.14: Breakdown of the execution time of sort-merge join for an increasing number of tuples and machines.

probe phase. When increasing the input sizes along with the number of machines, the amount of data which needs to be processed per machine remains identical. Thus, all local partitioning passes and the build-probe phase show constant execution time. However, increasing the number of machines, leads to a higher percentage of the data that needs to be exchanged over the network. Because the QDR network bandwidth is significantly lower than the combined partitioning speed of all threads, the network will become a significant performance bottleneck, thus leading to an increase in the execution time of the network-partitioning phase. For the FDR network, the hash join becomes bandwidth-bound only from seven machines onward, resulting in constant execution time for small, and in a slightly higher execution time for large deployments.

The sort-merge join algorithm (see Figure 3.14) is compute-bound and can take full advantage of the added compute resources, resulting in a constant execution time. As we increase the number of machines, small load imbalances cause a minor increase of the sorting phase as threads not only have to process their input but also have to wait for incoming data before being able to proceed.

3.4.5 Impact of Data Skew

Similar to the authors of previous work [BLP11, Bal14], we populate the foreign key column of the outer relation with two data sets. The first one with a low data skew which follows a Zipf distribution law with a skew factor of 1.05 and a highly skewed data set with a factor of 1.20. The relation sizes are 128 million tuples for the inner relation and 2048 million tuples for the outer relation.

In order to ensure that two skewed partitions are not assigned to the same machine, we use a dynamic partition-machine assignment. In this dynamic assignment the partitions are first sorted in decreasing order according to their element count. Next, they are distributed among the nodes in a round-robin manner, thus preventing that the largest partitions are assigned to the same machine.

With this workload, we see an increase in the execution time. This is true for all phases of the proposed algorithms. The network phases are dominated by the time it takes to send all the data to the machine responsible for processing the largest partition. Similarly, the execution times of the local processing phases are also dominated by that same machine. This effect is more pronounced for higher skew factors and a larger number of machines, i.e., systems that offer more parallelism that is not used by the join algorithms because, in the current implementation, each partition is processed by a single thread. For a setup with four machines, we observed an increase in the execution time by up to 1.20× in the presence of light skew, and 2.03× for the high-skew case. On eight machines, these numbers increase up to 1.77× and 3.29× respectively. This result highlights the need to share tasks between machines. Although heavily skewed partitions can be split and distributed among threads in order to allow for a higher degree of parallel processing, the current implementation only allows work sharing among threads within the same machine and not across multiple machines, thus not fully exploiting the parallelism of the entire system. Nevertheless, we are confident that this issue can be addressed by extending the algorithm to allow work sharing between machines. Several recent publications address this issue in the context of join operators [PSR14, RIKN16]. We include a discussion of these mitigation strategies in Section 3.6.

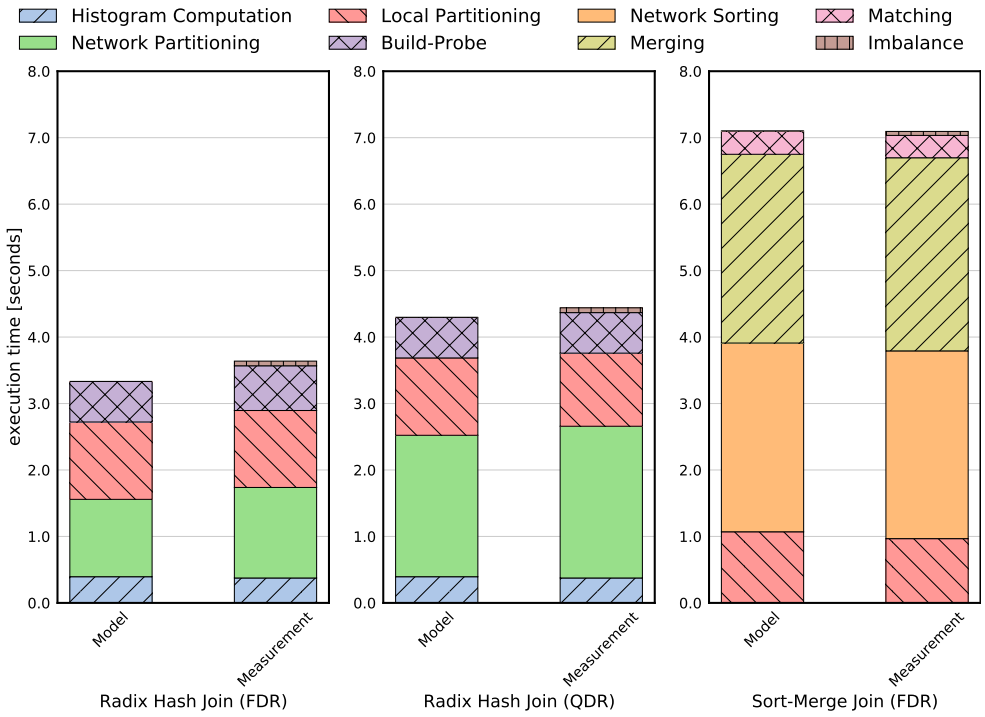
3.5 Evaluation of the Performance Models

In this section we validate the accuracy of the analytical model described in Section 3.3 by comparing its predictions to the experimental results gathered on both clusters. The measured network throughput between two machines is 6.0 Gbytes per second on the FDR network, respectively 3.6 Gbytes per second on the QDR network. In addition, we observed a small performance degradation of the useful throughput when increasing the number of machines on both InfiniBand clusters. This decrease is due to the fact that adding machines increases the overall network congestion. For all the experiments conducted in Section 3.4 we used eight cores on each machine. In this configuration, each thread is able to reach a local partitioning speed of 110 million tuples per second.

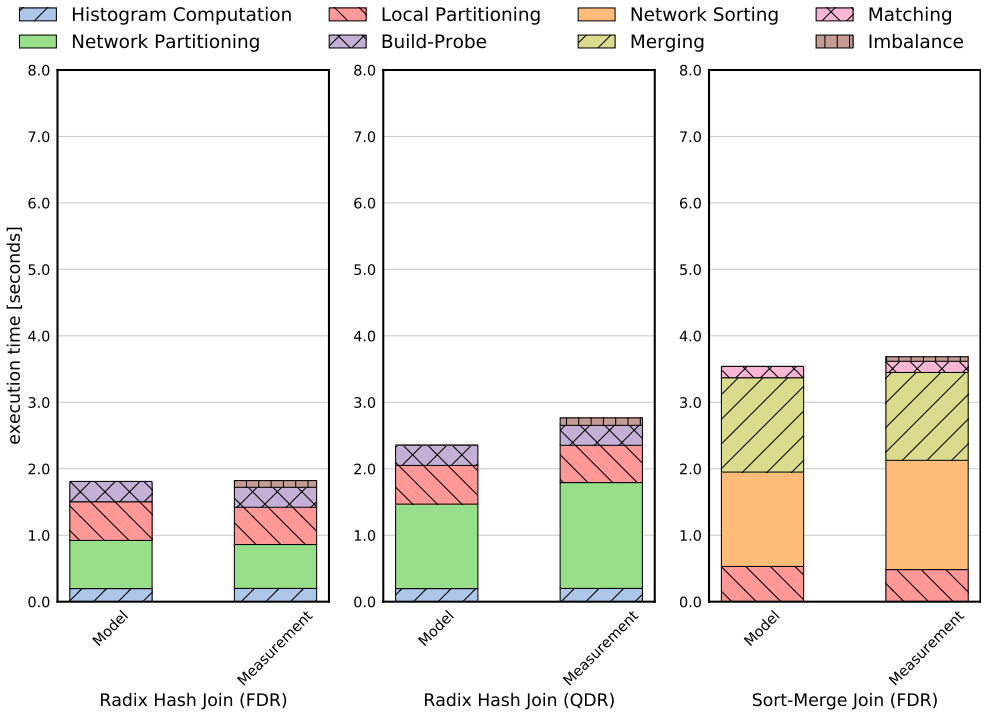
Using Equation 3.2, we know that the join is compute-bound on the FDR network for small deployments (i.e., six machines or fewer). Thus, the model predicts that all threads partition the data at their full processing capacity (compute-bound). In all the other case (i.e., on the QDR network and for large deployments of the FDR network) the radix hash join is network-bound. Using Equation 3.2, we can compute the partitioning speed of a thread for the network-partitioning pass. The second local partitioning pass is always executed at the local partitioning rate. For the sort-merge join, we know that all phases are bound by the speed of the processor and little load is put on the network. Figure 3.15 shows the predicted and measured performance of a 2048 million $\bowtie$ 2048 million tuple join for four and eight machines. One can clearly see that the predictions provide a lower bound on the execution time and closely match the experimental results.

Optimal Number of Threads for Network-Bound Systems

The analytical model described in Section 3.3 allows us to find the optimal number of threads for a given hardware specification. Given Equation 3.2, we know that in order to achieve maximum utilization of the network and processing resources, the number of partitioning threads should be such that it can saturate the network without being fully bound by the network bandwidth.



(a) Evaluation of the performance model with four machines



(b) Evaluation of the performance model with eight machines

Figure 3.15: Evaluation of the performance models of the radix hash and the sort-merge join algorithms on four and eight machines.

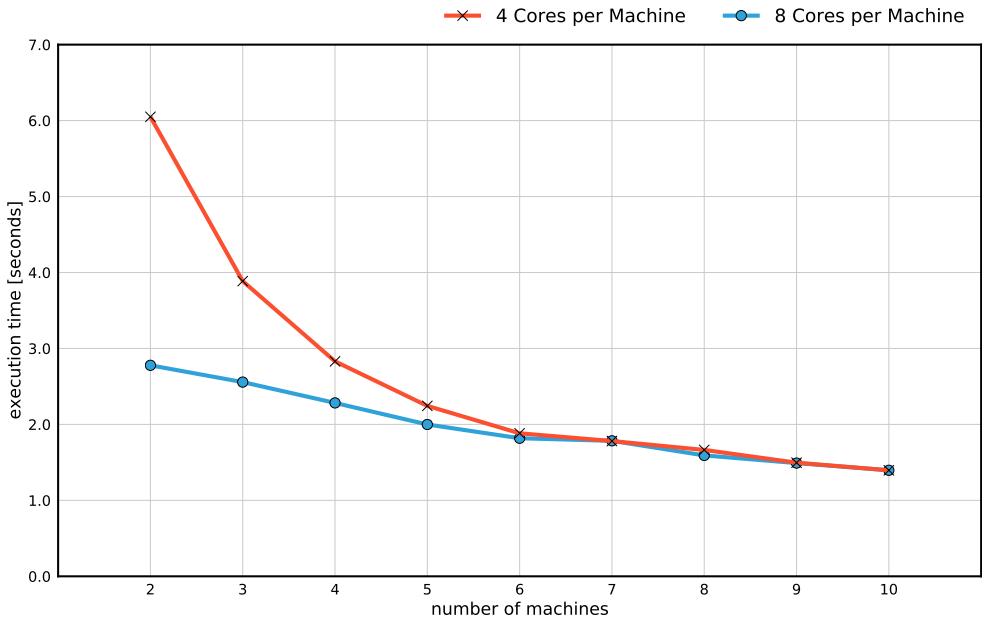


Figure 3.16: Execution time of the network-partitioning phase with four and eight threads per machine on the QDR network.

Given the network speed and partitioning rate of the radix hash join, we can determine the required number of processor cores for each algorithm and network. For the sort-merge join this amounts to nine cores on the QDR and seventeen cores on the FDR network. For the radix hash join on the other hand, this number is four cores per machine on the QDR and seven cores per machine on the FDR cluster (assuming one thread per core). This means that the configuration we used in the experimental evaluation for the radix hash join on leads to a network-bound system on the QDR cluster. To verify this result, we conducted two runs of experiments: the first run was performed with four and the second run with eight threads. In Figure 3.16 we compare the execution times of the network-partitioning pass on the QDR network. When increasing the number of machines, the percentage of data which needs to be exchanged over the network increases. We can observe that from five machines on-wards, four partitioning threads are sufficient to fully saturate the QDR network. Adding additional cores (i.e., eight threads) will not speed up the execution as the threads need to wait for network operations to complete before being able to reuse the RDMA-enabled buffers.

3.6 Discussion

***Data Processing over High-Performance Networks*:** In this chapter, we have developed distributed versions of the most common join algorithms using RDMA. However, the ideas described in this work, i.e. memory layout, reuse of buffers, and interleaving computation and communication are general techniques and can be used to create distributed versions of many database operators for use in combination with high-performance networks. In the experimental evaluation, we investigated a configuration that is network-bound (i.e., radix hash join on the QDR network), configurations that use an optimal number of processor cores compared to the network bandwidth (i.e., radix hash join on the FDR cluster and the sort-merge join on the QDR cluster), as well as a configuration that is compute-bound (i.e., sort-merge join on the FDR cluster). We observed that when a faster network is used, the radix hash join can out-perform the sort-merge approach by a factor of two. These findings are in line with related work on centralized algorithms [SD89, BATÖ13, Bal14]. Using the RMA network primitives in combination with a RDMA-capable network, no core needs to be dedicated to receive incoming data transmissions. Instead, data is immediately written to the correct location by the network card. RMA operations move data from one buffer to another, i.e., a read operation fetches data from a remote machine and transfers it to a local buffer, while the write operation transmits the data in the opposite direction. Data located on a remote machine can therefore not be loaded immediately into a register, but needs to be first read into a local main memory buffer. This approach makes the distributed implementations very similar to their single-machine counterparts, with the exception that data has to be explicitly flushed (i.e., RDMA write operation), similar to programming non-cache-coherent machines in which data has to be explicitly loaded into the cache-coherency domain before it can be used and changes to the data have to be explicitly flushed back to the source in order for the modifications to be visible on the remote side.

***Distributed v.s. Centralized Join Algorithms*:** The experiments clearly show that distributed joins are at a similar level of performance than single-machine, parallel join algorithms. In fact, our results indicate that modern multi-core hardware should be

treated more and more as a distributed system as it has been suggested for operating systems [BBD+09]. Our findings suggest that the answer to the question whether join performance can be improved by scaling up or scaling out is dependent on the bandwidth provided by the NUMA interconnect and the network. For instance, faster processor interconnects and a higher number of cores per processor favor vertical scale-up, whereas a higher inter-machine bandwidth would favor horizontal scale-out. In the experimental evaluation, we could show that our implementation of a distributed join exhibits good performance, despite the network being a bottleneck as we increase the number of cores and the number of machines. Technical road-maps project that upcoming generations of high-speed networks will be able to offer a significantly higher bandwidth (see Chapter 6), which suggests that the impact of this bottleneck will be reduced and the performance of the proposed algorithms will increase further when using many cores per machine.

***Data Skew*:** In the experimental evaluation, we use uniform data, that is distributed evenly among all the processor cores. The goal of this study is to investigate the maximum achievable performance of the most popular join algorithms on large scale-out architectures. To be able to process skewed data, good load-balancing needs to be achieved. Several techniques have been introduced for hash and sort-merge algorithms. These techniques are orthogonal to our evaluation and both join implementations could be enhanced to effectively mitigate workload imbalances caused by data skew. Rödiger et al. [RIKN16, Röd16] propose to detect skewed elements in the input with approximate histograms. The performance impact of heavy hitters is reduced through redistribution and replication of the skewed elements. The authors show that their join implementation achieves good performance and is able to scale well on a rack-scale system. This process can be integrated into the histogram computation and the network-partitioning pass of our radix hash join. In HPC applications, sorting is a commonly used operation. By default, sorting algorithms can work with skewed data. Most distributed sorting algorithms can be put in one of two categories: merge-based and splitter-based approaches. Merge-based sorting algorithms combine data from two or more processes [Bat68]. Splitter-based approaches try to subdivide the input into chunks of roughly equal size [FM70, HY83, DNS91, KK93, SK10]. The latter category utilize minimal data movement because the data only moves during

the split operation. In our implementation of the sort-merge join, we use a splitter-based approach. When processing skewed data, techniques for finding the optimal pivot elements can be used [SK10]. We expect that a histogram-based technique for finding the optimal splitter values can be integrated into the partitioning phase of our sort-merge join implementation.

***Result Materialization*:** In this work, we treated the join operation as part of an operator pipeline in which the result of the join is materialized at a later point in the query execution. It is worth pointing out that distributed result materialization involves moving large amounts of data over the network and will therefore be an expensive operation. Algorithms for slower networks, such as the TrackJoin [PSR14], compute an optimal assignment of tuples to machines in order to minimize data movement. Although, high-performance networks offer a significantly higher bandwidth, efficient materialization techniques remain a crucial component of any query pipeline.

3.7 Related Work

***Parallel Join Algorithms*:** In the Gamma database machine [DGG+86, DGS+90] tuples are routed to processing nodes using hash-based split tables. Identical split tables are applied to both input relations, thus sending matching tuples to the same processing node. This method reduces a join of two large relations to a set of separate joins which can be executed in parallel. Schneider et al. [SD89] compared hash and sort-merge joins on the Gamma database machine. They conclude that with a sufficient amount of main-memory, hash-based join algorithms have superior performance to sort-merge joins. Most modern hash join algorithms build upon the idea of the Grace hash join [KTM83], where both input relations are first scanned and partitioned according to the join attribute before a hash table is created for each partition of the inner relation and probed with the tuples from the corresponding partition of the outer relation. The findings of Shatdal et al. [SKN94] and Manegold et al. [MBK02, MBN04] showed that a Grace hash join which partitions the data such that the resulting hash tables fit into the processor cache can

deliver higher performance because it reduces the number of cache misses while probing the hash tables. To avoid excessive TLB misses during the partitioning phase caused by many random memory access to a large number of partitions, Manegold et al. [MBK02] proposed a partitioning strategy based on radix-clustering. In cases where the number of partitions exceeds the number of TLB entries or cache lines, the partitioning is performed in multiple passes, each with a limited fan-out.

***Join Algorithms on Modern Hardware*:** Kim et al. [KSC+09] have compared hash and sort-merge joins to determine which type of algorithm is better suited to run on modern multi-core machines. In addition to their experiments, the authors also developed a model in order to predict the performance of the algorithms on future hardware. Although modern hardware currently favors hash join algorithms, they estimated that future hardware with wider single instruction over multiple data (SIMD) vectors would significantly speed up sort-merge joins. Blanas et al. [BLP11] re-examined several hash join variants, namely the no partitioning join, the shared partitioning join, the independent partitioning join, and the radix hash join. The authors argue that the no partitioning join, which skips the partitioning stage, can still outperform other join algorithms because modern machines are very good at hiding latency caused by cache and TLB misses. Their results indicate that the additional cost of partitioning can be higher than the benefit of having a reduced number of cache and TLB misses, thus favoring the no partitioning join. Albutiu et al. [AKN12] looked at parallel sort-merge join algorithms. The authors report that their implementation of the massively parallel sort-merge (MPSM) join is significantly faster than hash joins, even without SIMD instructions. Balkesen et al. [BTAÖ13] implemented efficient versions of two hash join algorithms – the no partitioning join and the radix join – in order to compare their implementations with previous studies and report a maximum throughput of 750 million tuples (16-byte tuples) per second on 64 cores. They show that a carefully tuned hardware-conscious radix join algorithm outperforms a no partitioning join. Furthermore, the authors argue that the number of hardware-dependent parameters is low enough, such that hardware-conscious join algorithms are as portable as their hardware-oblivious counterparts. In a follow-up paper [BATÖ13], the authors further show that the radix hash join is still superior to sort-merge approaches

for the current width of vector instructions. Lang et al. [LLA+13] show the importance of NUMA-awareness for hash join algorithms on multi-cores. Their implementation of a NUMA-aware join claims an improvement over previous work by a factor of more than two.

***Distributed Join Algorithms*:** The work of Goncalves et al. [GK10, Gon13] and Frey et al. [FGKT09, FGKT10] has resulted in a novel join algorithm, called cyclo-join, optimized for ring-shaped network topologies. In the setup phase of the cyclo-join, both relations are fragmented and distributed over all machines. During the execution, data belonging to one relation is kept stationary while elements of the second relation are passed on from one machine to the next. Similar to our approach, the idea is that the data is too large to fit in one machine, but can fit in the distributed memory of the machines connected on the ring. The cyclo-join uses RDMA as a transport mechanism. The cyclo-join differs from our work in that it runs on an experimental system that explores how to use the network as a form of storage. The hot set of the data is kept rotating in the ring and several mechanisms have been proposed to identify which data should be put on the storage ring [GK10]. In DaCyDB, the authors use RDMA to connect several instances of MonetDB in a ring architecture [Gon13]. FlowJoin is a distributed hash join algorithm developed by Rödiger et al. [RIKN16] that can mitigate negative effects on performance caused by data skew. Through the use of histograms, frequent elements can be detected and redistributed in specific ways. This approach is complementary to the algorithms that are studied in this dissertation. Rödiger et al. [RMU+14] propose locality-sensitive data shuffling, a set of techniques, which includes optimal assignment of partitions, network communication scheduling, adaptive radix partitioning, and selective broadcast intended to reduce the amount of communication of distributed operators. Liu et al. [LYB17] design and evaluate RDMA-aware data shuffling operators and compare different strategies and implementations for exchanging vast amounts of data over high-performance networks. Polychroniou at al. [PSR14] propose three variants of a distributed join algorithm which minimize the communication costs. The authors tested their implementation of the proposed join algorithms on a Gigabit Ethernet network. They show that the 3-phase and 4-phase track join algorithms can significantly reduce the overall network traffic. Recent work around distributed joins [AU11, OR11] in map-reduce environments focuses on care-

fully mapping the join operator to the relevant data in order to minimize network traffic. These contributions show that the network is the main bottleneck for join processing, in particular on conventional networks.

3.8 Summary

In this chapter, we presented two distributed join algorithms (radix hash and sort-merge join) that make use of one-sided RMA operations as a light-weight communication mechanism. We evaluated both algorithms on RDMA-capable networks. We described how RDMA-enabled buffers can be used to partition, sort, and distribute data efficiently. We were able to show that the performance of the distributed join algorithms is highly dependent on the right combination of processing power and network bandwidth. In addition to the prototype implementations, we presented models of both algorithms and were able to show that these models can be used to predict the performance of the algorithm with very high accuracy. We performed an experimental evaluation of the algorithm on multiple hardware platforms using two different low-latency networks and a high-end server machine. Our results show that both algorithms, the radix hash and the sort-merge join, are able to scale well in rack-scale clusters and that the performance of the hash join is superior to the sort-based approaches in such systems.

4

Large-Scale Join Processing

The ability to efficiently query large sets of data is crucial for a variety of applications, including traditional data warehouse workloads and modern machine learning applications [KNPZ16]. As seen in the previous chapter, our carefully tuned, distributed join implementations for multi-core machines and rack-scale data processing systems exhibit good performance. However, all these algorithms have been designed for and evaluated on rack-scale systems with a limited number of processor cores.

This chapter not only addresses the challenges of running state-of-the-art, distributed radix hash and sort-merge join algorithms on high-speed, RDMA-capable networks, but also investigates their behavior at scales usually reserved to massively parallel scientific applications or large map-reduce batch jobs. Operating at large scale requires careful process orchestration and efficient communication. For example, a join operator needs to keep track of data movement between the compute nodes in order to ensure that every tuple is transmitted to the correct destination node for processing. Computation and communication need to be interleaved in order to achieve maximum performance. These problems become more challenging as we add machines and compute resources. In this part of the dissertation, we explore how modern join implementations behave on a large

number of cores when specialized communication libraries, such as MPI (see Section 2.1.3), replace hand-tuned code. We show that at large scale, the performance of the algorithm is dependent on having a good communication infrastructure that automatically selects the most appropriate method of communication between two processes.

4.1 Problem Statement and Novelty

We implemented state-of-the-art, distributed radix hash and sort-merge join algorithms on top of MPI, a standard library interface used in high-performance computing (HPC) applications, and evaluated the join implementations on two large-scale systems with a high number of cores connected through a high-throughput, low-latency network fabric. All algorithms are hardware-conscious, make use of vector instructions to speed up the processing, access remote data through fast one-sided memory operations, and use remote direct memory access (RDMA) to speed up the data transfer.

This is one of the first projects to bridge the gap between database systems and high-performance computing. In the experimental evaluation, we provide a performance analysis of the distributed joins running on 4096 processor cores with up to 4.8 Tbytes of input data. Novel insights from this work include: (i) Although both join algorithms scale well to thousands of cores, communication inefficiencies have a significant impact on performance. (ii) Hash and sort-merge join algorithms have different communication patterns that incur different communication costs, making the scheduling of the communication between the compute nodes a crucial component. (iii) Our performance models indicate that the sort-merge join implementation achieves its maximum performance. The radix hash join on the other hand is far from its theoretical maximum, but is still able to slightly outperform the sort-merge join. However, in contrast to our findings in the previous chapter, we will observe that due to communication inefficiencies in the network-partitioning phase, the difference in performance between both approaches is no longer a factor of two, making the sort-merge join a competitive approach on large-scale systems such as supercomputers or cloud environments.

4.2 Distributed Join Algorithms using MPI

The algorithms presented in this part of the dissertation follow the same mode of operation as described in Section 3.2.1 and Section 3.2.2 for the radix hash join and the sort-merge join respectively. It is worth pointing out, that the implementation of the algorithms changes significantly when using Mpi. In the following section, we describe the elements of the algorithm that had to be adapted to be able to run on a high-end supercomputer. Hpc applications are structured differently from system software. In particular, Mpi applications use a process-centric model and both join algorithms have been modified to use multiple processes with one thread each, instead of instantiating a single process per machine that uses many threads. Several of the hand-written sections of the code to manage meta-data, such as histograms, have been replaced by high-level reduce operations.

4.2.1 Radix Hash Join Algorithm

Using Mpi requires some fundamental changes to all communication-intensive phases of the algorithm. For the radix hash join, these phases include the histogram computation and the network-partitioning phase.

Histogram Computation

Mpi provides many high-level communication primitives such as *reduce* operations that are useful for implementing a scalable histogram computation algorithm. Just like the rack-scale version of the radix hash join algorithm, each process scans its part of the input data and computes two process-level histograms – one for each input relation. In the Mpi version of the algorithm, these local histograms are combined into a global histogram through an `MPI_Allreduce` call. We use the `MPI_SUM` operator as an argument to the call. This operation combines the values from all processes – in our case it computes the element-wise sum – and distributes the result back, such that each process receives a copy of the global histogram.

The join supports arbitrary partition-process assignments. Just like the algorithm described in the previous chapter, in this implementation, we use a round-robin scheme to assign partitions to Mpi processes. To compute the window size, each process masks the assignment vector with its process number such that the entries of the assigned partitions are one, and zero otherwise. This mask is applied to the global histogram. The sum of all remaining entries is equal to the required window size.

Computing the private offsets for each process and each partition is performed in three steps. First, the base offsets of each partition are computed. The base offsets are the starting offsets of each partition in relation to the starting address of the window. Second, the relative offsets within a partition need to be computed from the local histograms using a prefix sum computation. To perform this prefix computation across all processes, Mpi provides an `MPI_Scan` functionality. This function returns for the i-th process the reduction (calculated according to a user-defined function) of the input values of processes 0 to i. In our case, the prefix sum is implemented by combining the `MPI_Scan` function with the `MPI_SUM` operator. Third, the private offsets of a process within a window can be determined by adding the starting offset of a partition and the relative private offset. At the end of this computation, each process is aware of (i) the assignment of partitions to processes, (ii) the amount of incoming data, and (iii) the exact location to which the process has exclusive access when partitioning its input.

Partitioning Phase

From the histogram computation stage, we know the exact incoming data size for each process and input relation. Mpi represents registered memory that can be accessed through Rma operations in the form of a *window* (see Appendix B.2). Two windows will be allocated: one for the inner and one for the outer relation. Because `MPI_Win_create` is a collective routine, this phase requires global synchronization. After the window allocation phase, each process acquires an `MPI_LOCK_SHARED` lock on all the windows. We allow concurrent accesses because the histogram computation provides us with the necessary information to determine ranges of exclusive access for each partition and process. Next,

similar to the rack-scale version, each process allocates a set of communication buffers for each partition into which the process will partition the data.

After the setup phase, the algorithm starts with the actual partitioning and redistribution of the input. Data is partitioned using the same vector instructions as the rack-scale implementation. When an output buffer is full, the process will issue an `MPI_Put` into its private offset in the target window. Interleaving computation and communication is essential to reach good performance. Therefore, we allocate multiple (at least two) output buffers for each remote partition. When all the buffers of a specific partition have been used once, the process needs to ensure that it can safely reuse them. This is achieved by executing an `MPI_Win_flush`. This operation ensures completion of all pending RMA requests, independent of whether these requests operate on the same partition or not. Therefore, this call is over-conservative. The alternative is to use request-based flushing operations. However, the latter are not supported by all MPI RMA implementations.

After having partitioned the data, the shared window lock is released which ensures successful completion of all outgoing RMA operations. After the call returns, the process can release all its partitioning buffers. However, it needs to wait for the other processes to finish writing to its window. This synchronization is realized through the use of an `MPI_Barrier` call at the end of the partitioning phase.

Local Processing

The local processing phase includes subsequent partitioning passes as well as the build-probe phase. This part of the computation does not require network communication. Therefore, no modifications needed to be made to this part of the algorithm.

4.2.2 Sort-Merge Join Algorithm

Similar to the previous algorithms, significant changes to the implementation were required to run on the supercomputer. The partitioning and sorting phases of the sort-merge join algorithm have been modified to use MPI as part of their communication.

Local Partitioning

During the local partitioning operation, every process tracks how many elements are assigned to each partition, thus creating a histogram. To compute the window size, a process must know how much data has been assigned to it. The histogram from the partitioning phase, together with the `MPI_SUM` operator, is given as an input to the `MPI_Reduce_scatter_block` call. This call performs an element-wise reduction – in this case it computes a sum – of all the histograms and scatters the result to the nodes. This means that node i will receive the sum of the i-th element of the histograms. The result of the reduction is equal to the required window size.

Sorting Phase

The window size is passed as an argument to the `MPI_Win_create` call. To determine the private offsets into which processes can write, the join algorithm uses the `MPI_Scan` function with the histogram data and the `MPI_SUM` operator as input in order to perform a distributed element-wise prefix sum computation, which provides the private offsets in the memory windows into which a process can write.

Because of the variable size of the last run, the receiving process needs to be aware of the amount of incoming data from every process. Otherwise, the algorithm cannot determine where the last sorted run of process i ends and the first run for process $i + 1$ starts. To that end, `MPI_Alltoall` is called on the histogram data, which sends the j-th element of the histogram from process i to process j, which in turn receives it in the i-th place of the result vector. From this information, the algorithm can determine the start and end offset of every run.

The sorting process proceeds by transmitting sorted chunks of data to the target window by issuing `MPI_Put` calls. At the end of these operations, a `MPI_Win_flush` call ensures that the data has been transmitted to the remote processes before releasing the `MPI_LOCK_SHARED` locks that have been taken on the window buffers. The processes then wait at an `MPI_Barrier` in order to ensure that all incoming data has been received.

Local Processing

The subsequent merge passes on the chunks of sorted data, as well as the scan through the sorted data to find matching tuples, does not require any network communication or coordination and thus did not require any modifications to run on the supercomputer.

4.3 Experimental Evaluation

In this section, we evaluate the MPI-based implementations on two high-end Cray supercomputers with thousands of cores and a high-speed Aries interconnection network.

4.3.1 Workload and Setup

In order to make our results comparable to previous work on join algorithms [KSC+09, BLP11, BTAÖ13, BATÖ13, BTAÖ15, BLAK15, BMS+17], we use the same workloads as discussed in the previous chapter. The experiments focus on large-to-large joins with highly distinct key values. The data is composed of narrow 16-byte tuples, containing an 8-byte key and an 8-byte record id (RID). The record identifiers are range partitioned among the compute nodes. The key values can occur in arbitrary order. Each core is assigned to the same amount of input data. In our experiments, one process serves up to 40 million tuples per relation, which results in a total of 4.8 Tbytes of input data on 4096 cores. The relative size of the inner and outer relation varies between 1-to-1 and 1-to-8. The impact of different selectivities is also studied.

The Cray XC30 [Cra18] used in the experimental evaluation has 28 compute cabinets implementing a hierarchical architecture: each cabinet can be fitted with up to three chassis. A chassis can hold up to sixteen compute blades, which in turn are composed of four compute nodes. The overall system can offer up to 5272 usable compute nodes [Swi18]. Compute nodes contain a single-socket 8-core processor (Intel Xeon E5-2670) and 32 GB of main memory. They are connected through an Aries routing and communications ASIC

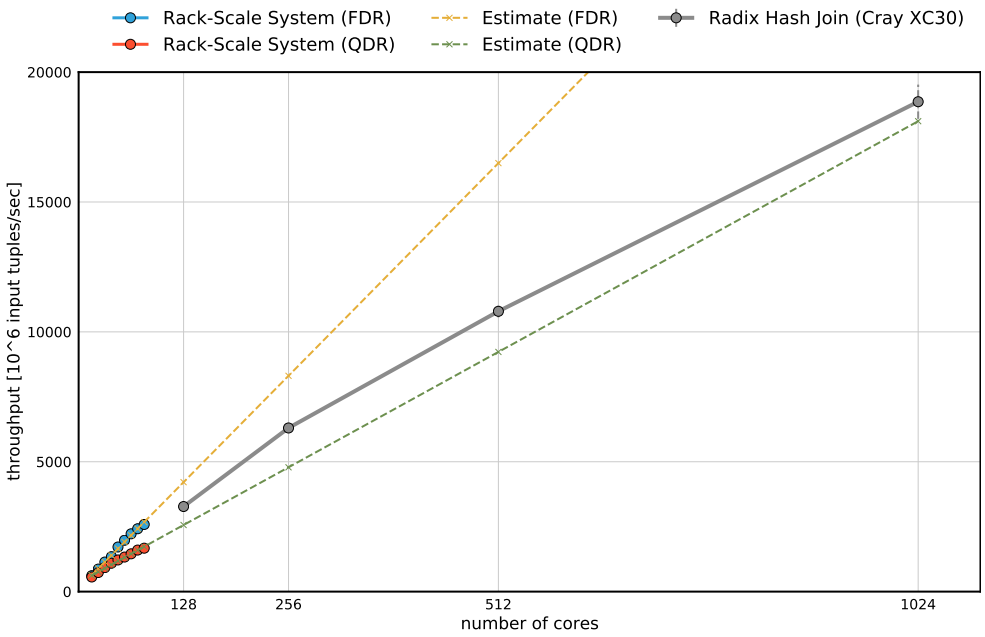
and a Dragonfly network topology [KDSA08] with a peak network bisection bandwidth of 33 Tbytes per second. The Aries Asic is a system-on-a-chip device comprising four NICs and an Aries router. The NICs provide network connectivity to all four nodes of the same blade. Each NIC is connected to the compute node by a 16x PCI Express 3 interface. The router is connected to the chassis back plane and through it to the network fabric. The second machine used for the experiments is a Cray XC40 machine. It has the same architecture as the XC30 but differs in the node design: each compute node has two 18-core processors (Intel Xeon E5-2695 v4) and 64 GB of main memory per node.

The algorithms use foMPI [GBH13], a scalable Mpi Rma library that, for intra-node communication, uses Xpmem, a Linux kernel module that enables mapping memory of one process into the virtual address space of another, and, for inter-node communication, uses Dmapp [tBR10], a low-level networking interface of the Aries network.

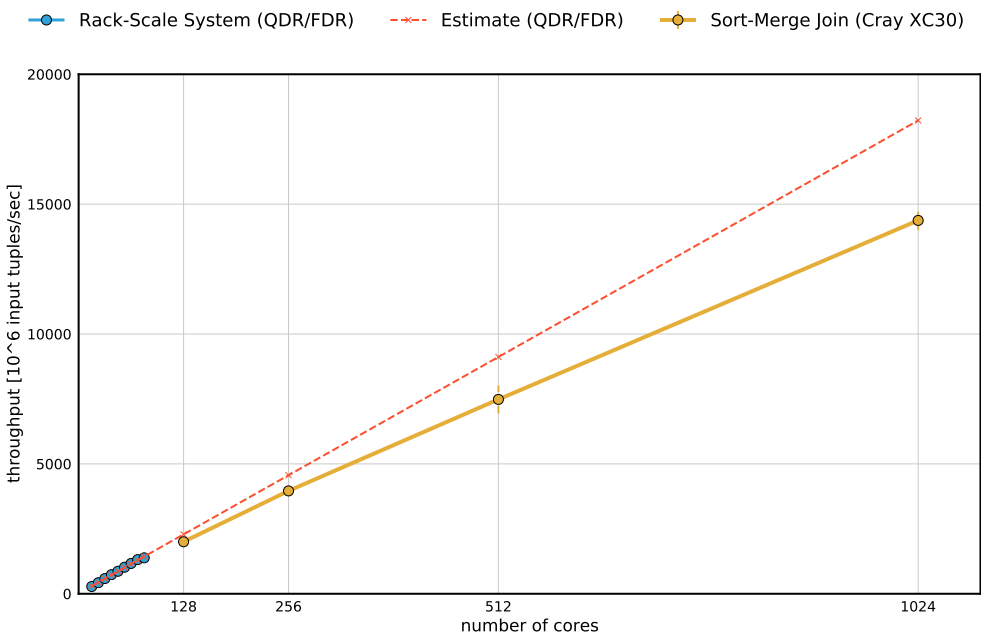
4.3.2 Comparison with Rack-Scale Joins

The experiments in the previous chapter have been conducted on two generations of InfiniBand networks. Because the nodes of the supercomputer are also composed of multi-core Intel Xeon CPUs and are connected through a low-latency network, we use the performance results gathered on rack-scale systems as a baseline. We extrapolate the performance of both algorithms on a larger number of cores using linear regression and compare this estimate with the measured performance on the Cray XC30 system.

The comparison between the estimated and measured performance is shown in Figure 4.1 along with the 95% confidence intervals. For the sort-merge join, we can observe that the measured performance follows the extrapolated line very closely. This is expected as the algorithms puts little load on the network. For the radix hash join, we observe that the performance is significantly below the expected performance on the Fdr cluster and is more in line with the results from the Qdr network. This behavior is not expected, given that the bandwidth offered by the Cray Aries network is significantly higher than the one offered by Qdr InfiniBand. In Section 4.3.3, we will show that the communication pattern of the radix hash join incurs some significant performance costs that prevent



(a) Radix Hash Join



(b) Sort-Merge Join

Figure 4.1: Comparison of the throughput of join algorithms on rack-scale systems and the Cray supercomputer.

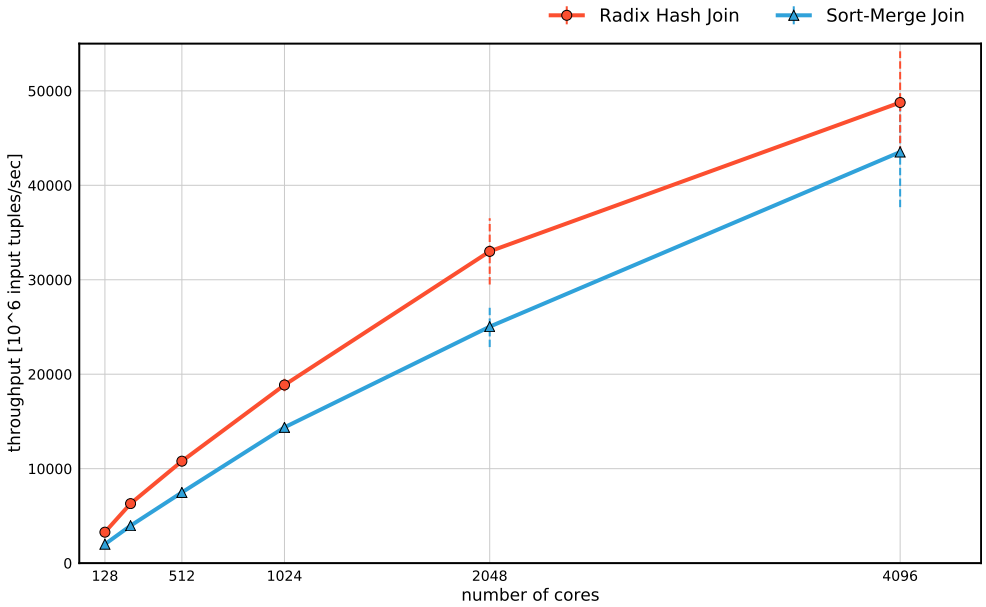


Figure 4.2: Scale-out experiments of the radix hash join and sort-merge join algorithms on the Cray supercomputer.

it from reaching its maximum performance and that cannot be observed in small-scale experiments. In general, we can conclude that the algorithms proposed in this section achieve similar performance than the rack-scale baseline presented in Chapter 3.

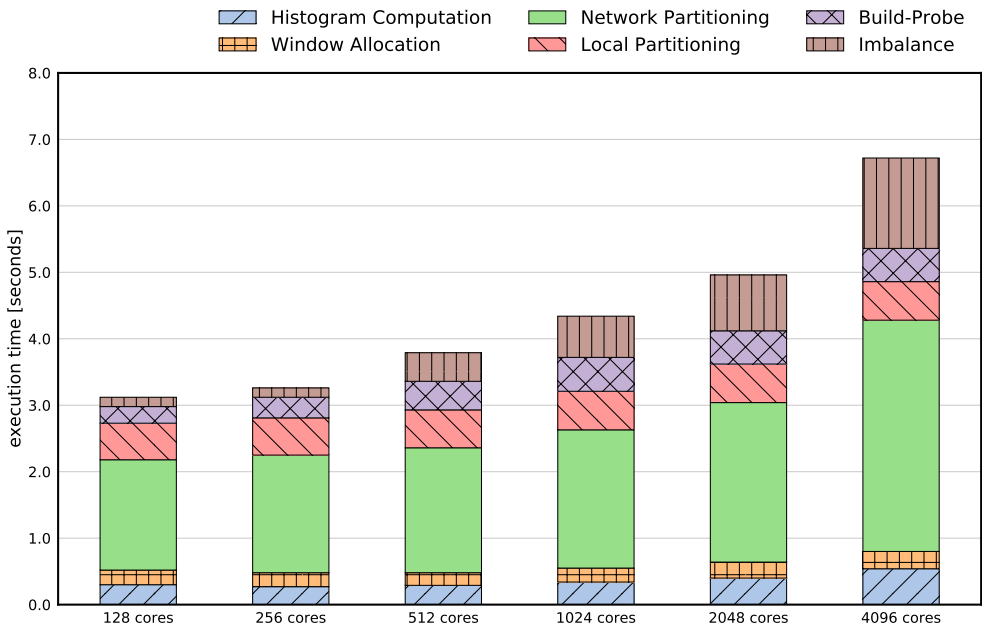
4.3.3 Scale-Out Experiments

Figure 4.2 shows the overall throughput of the radix hash join and the sort-merge join along with the 95% confidence intervals as an error metric. We assign 40 million tuples to each relation and core. Every tuple of the inner relation matches with exactly one element of the outer relation. The results show that both algorithms are able to increase their respective throughput as more cores are added to the system. At its peak, the radix hash join can process 48.7 billion tuples per second. The sort-merge join reaches a maximum throughput of 43.5 billion tuples per second on 4096 cores. The scale-out behavior of both algorithms is sub-linear. When using 4096 cores, hashing outperforms the sort-merge approach by 12%, which is significantly less than the performance difference observed on rack-scale systems in Chapter 3.4.

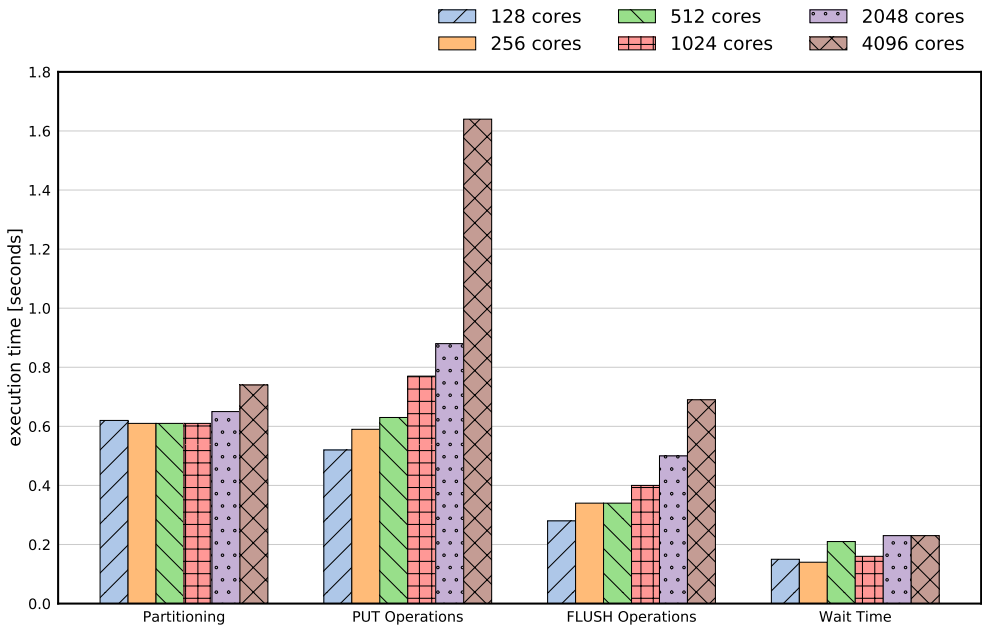
Analysis of the Radix Hash Join

Figure 4.3a shows the execution time of the different phases of the radix hash join and illustrates the effects of scale-out in more detail. We break down the execution of the join as follows: (i) the histogram computation, which involves computing the local histogram, the exchange of the histograms over the network, and the computation of the partition offsets, (ii) the time required to allocate the RMA windows, (iii) the network-partitioning phase, which includes the partitioning of the data, the asynchronous transfer to the target process, and the flushing of buffers, (iv) the local partitioning pass, which ensures that the partitions fit into the cache, and (v) the build and probe phase, in which a hash table is created over each partition of the inner relation and probed using the data from the corresponding partition of the outer relation. All times are averaged over all the processes. Because we consider the join only to be finished when the last process terminates, we report the difference between the maximum execution time and the sum of the averaged execution times as the *load imbalance*. This value gives an indication of how evenly the computation has been balanced across all cores and whether there are stragglers or not.

Given that we scale out the system resources and the input size simultaneously, one would expect constant execution time of all phases. However, we observe an increase in execution time as we add more cores, which explains the sub-linear increase in throughput shown in Figure 4.2. We observe that the execution time of the histogram computation and the window allocation phase remains constant. The network-partitioning phase on the other hand increases significantly. Figure 4.3b shows a detailed breakdown of this phase. One can observe that the time required to partition the data remains constant up to 1024 cores. Starting from 1024 cores, the partitioning fan-out has to be increased beyond its optimal setting, which incurs a minor performance penalty. Most of the additional time is spent in the `MPI_Put` and `MPI_Flush` operations which generate the requests to transmit the data, respectively ensure that the data transfers have completed. This increase is caused by the additional overhead of managing a larger number of buffers and the lack of any network scheduling. More details on the costs of communication at large scale are provided later in this section. The local partitioning phase exhibits constant execution



(a) Total join execution



(b) Network-partitioning pass

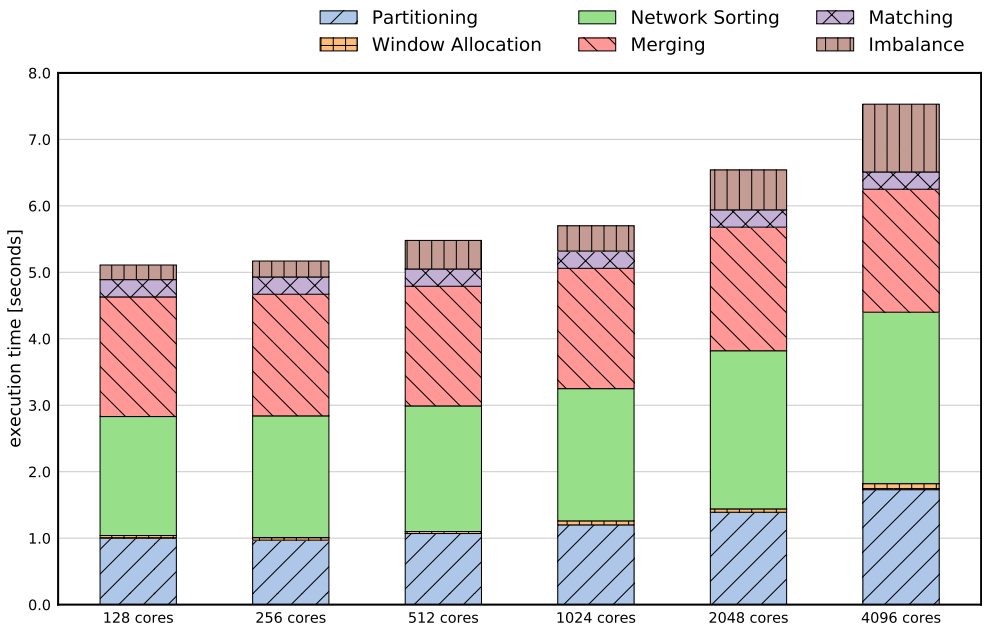
Figure 4.3: Breakdown of the execution time of the radix hash join for 40 million tuples per relation per core.

time because the per-core amount of data is kept constant throughout the experiment. The build-probe operation on the other hand shows a minor increase in execution time because the generated partitions get larger as we add more cores and process more data overall. For the compute imbalance, i.e., the time difference between the average and maximum execution time, we observe a clear increase as we add cores to the system. This is expected as the supercomputer is shared by multiple organizations and complete performance isolation cannot be guaranteed for large deployments. Furthermore, the nodes involved in a large experiment cannot always be physically co-located, resulting in a higher remote memory access latency for some nodes. We observe that the performance of the hash join is influenced by a small number of stragglers.

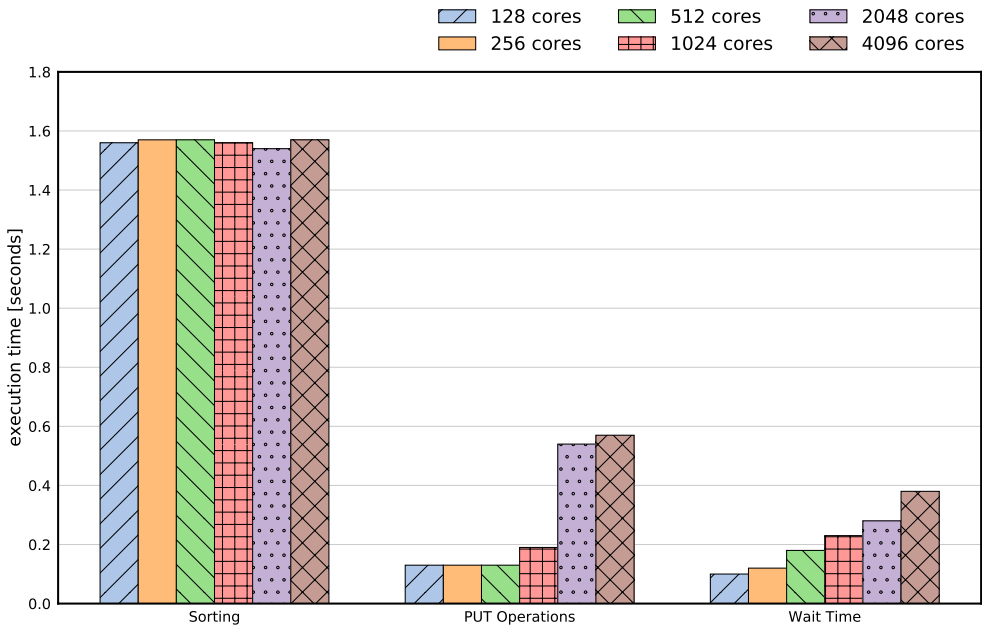
Analysis of the Sort-Merge Join

Figure 4.4a shows the breakdown of the execution time of the sort-merge join. The individual phases are (i) the range partitioning phase, which includes the histogram and offset computation, (ii) the window allocation time, (iii) the time needed to sort and transmit the tuples, (iv) the time required to merge the sorted runs, and (v) the time required to join both relations. Similar to the hash join, the execution times shown in Figure 4.3a are averaged over all processes and the difference between the average and total execution time is reported as the *load imbalance*. For the sort-merge join, we can observe an increase in the time required to partition and sort the data. For 2048 and 4096 cores, the partitioning fan-out has to be pushed beyond its optimal configuration, which leads to a small increase in execution time. The sort-merge join uses one single partitioning pass over the data. However, given that the performance loss is small, a second partitioning pass does not pay off at these scales.

In Figure 4.4b, we see that the sorting phase is dominated by the time required to sort the tuples. The `MPI_PUT` operation time remains constant up to 1024 cores, followed by a sudden increase in its execution time. This effect can be explained by the fact that sorting is more compute intensive than hashing, which allows for better interleaving of computation and communication. Furthermore, the communication pattern of the sort-



(a) Total join execution



(b) Network sorting phase

Figure 4.4: Breakdown of the execution time of the sort-merge join for 40 million tuples per relation per core.

merge join is better suited for the underlying network hardware. A detailed discussion is provided later in this section. Given that the per-core data size remains constant, the time required to merge and match the data does not change.

Analysis of the Network Communication

A key performance factor for both algorithms is the cost of communication. In previous paragraphs we made the following observations: (i) The time required to execute all `MPI_Put` calls is significantly higher for the hash join than for the sort-merge join. (ii) The cost of enqueuing an `MPI_Put` request steadily increases for the hash join as the number of cores is increased. (iii) The `MPI_Put` cost remains constant for the sort-merge join up to 1024 cores, followed by a sudden increase in execution time.

These observations can be explained by the fact that the two algorithms have different communication patterns. The hash join interleaves the partitioning and the network communication. To that end, it allocates a temporary buffer space into which data is written. Once a buffer is full, an `MPI_Put` request is generated and a new buffer is used to continue processing. Because the amount of buffer space is the same for every partition and uniform data is used, the partition buffers will be scheduled for transmission at similar points in time, causing temporal hotspots on the network. This is aggravated by having more processes per machine. Because the hardware has a limited request queue, the processes will be blocked while trying to enqueue their request, causing a significant increase in the time spend in the `MPI_Put` call. This problem is further compounded as the partitioning fan-out increases. During the network-partitioning phase, every process communicates with every other process in the system simultaneously. Having more active communication channels incurs a significant overhead.

The sort-merge join partitions the data into individual ranges before it interleaves the sorting operation and the network transfer. A process only sorts one run at a time. After the run is sorted, it is immediately enqueued for transfer. Alternating between sorting and executing an `MPI_Put` calls creates an even transmission rate on the sender side. To avoid over-saturation at the receiver, each thread starts processing a different range, i.e., the

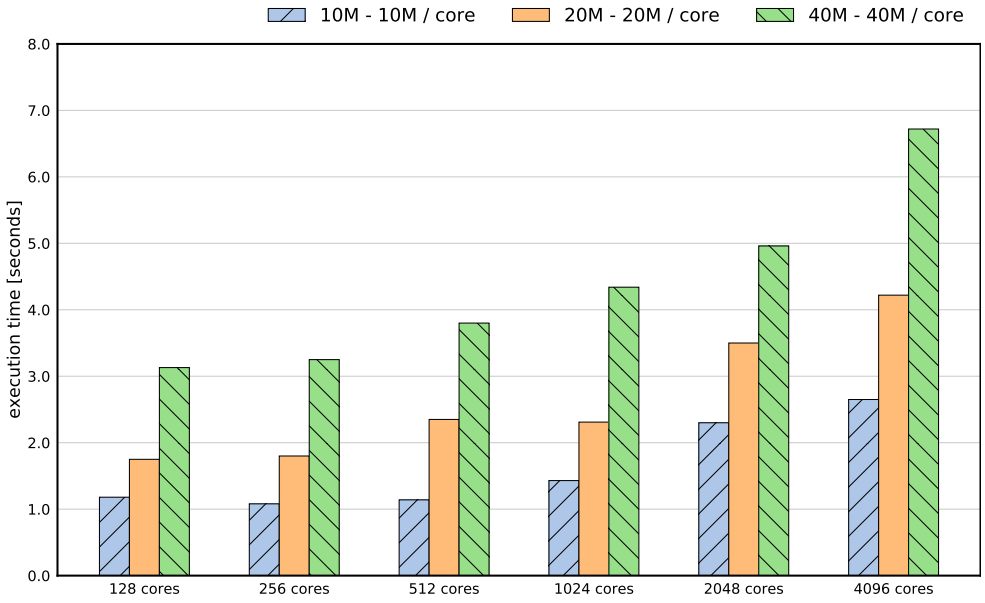
Table 4.1: Execution time for different workloads with variable relation sizes and selectivities for 1024 processes.

Workload		**Radix Hash Join**		**Sort-Merge join**	
Input	Output	Time	95% CI	Time	95% CI
40M/40M	40M	4.34s	±0.15s	5.70s	±0.14s
20M/40M	40M	3.45s	±0.15s	4.67s	±0.23s
10M/40M	40M	2.88s	±0.29s	3.83s	±0.27s
10M/40M	20M	2.92s	±0.10s	3.75s	±0.25s
10M/40M	10M	2.91s	±0.18s	3.87s	±0.41s

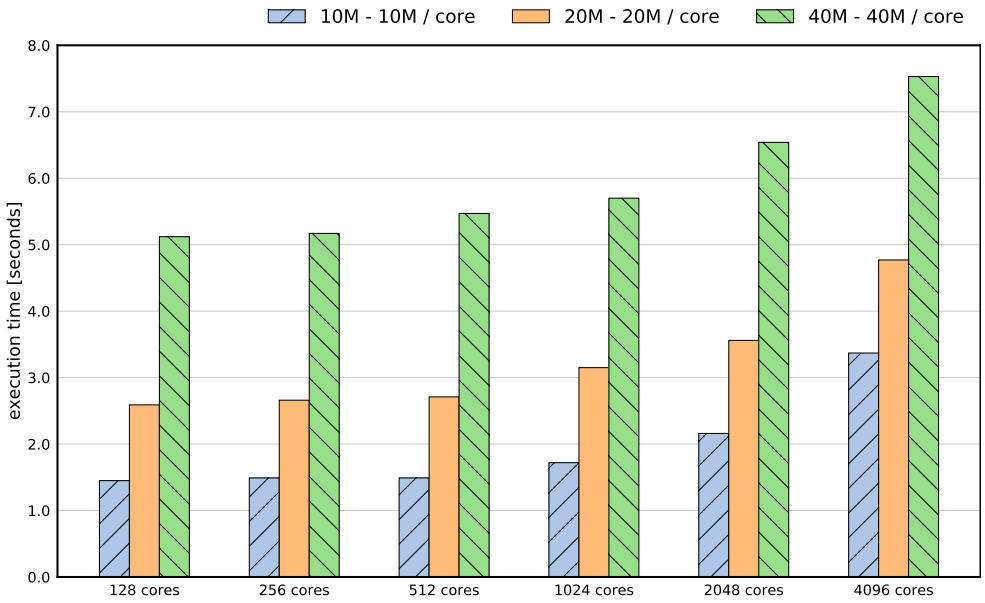
i-th process starts sorting range $i + 1$. Since the data is distributed uniformly and the processes are synchronized at the start of the sorting phase, for small deployments, they remain synchronized throughout the entire phase. During any point in time, a process i is transmitting data to exactly one process j, which in turn receives data only from the i-th process. Without synchronization, this pair-wise communication pattern can only be maintained for small deployments. In large deployments, nodes cannot be guaranteed to be physically co-located and variable network latencies disrupt this pattern, causing the increase in `MPI_Put` costs for 2048 and 4096 cores.

Effect of Input Size

To study the effect of different input data sizes and the ratio of the inner and outer relation, we use several workloads: (i) a 1-to-1 input where each tuple of the inner relation matches with exactly one element in the outer relation. We use 10, 20, and 40 million tuples per relation and core; (ii) 1-to-N workloads, where each element in the inner relation finds exactly N matches in the outer relation. In Figure 4.5a, we see the performance of the hash join for different input sizes. We observe that a reduction of the input size by half does not lead to a 2× reduction in execution time. The execution time of both partitioning passes as well as the build-probe phase is directly proportional to the input size. However, the histogram computation, window allocation, and the compute imbalance are not solely



(a) Radix hash join



(b) Sort-merge join

Figure 4.5: Execution time of the radix hash join and the sort-merge join algorithms for different input sizes.

dependent on the input but have additional fixed costs. For the sort-merge join (see Figure 4.5b), the time for sorting, merging, and matching the tuples is reduced by almost half. Window allocation and compute imbalance are not directly affected by the input size, resulting in a sub-linear speed-up. Table 4.1 (lines 1-3) shows the execution time of both algorithms on 1024 cores for different relation sizes. One can observe that the execution time depends primarily on the input size and is therefore dominated by the larger relation.

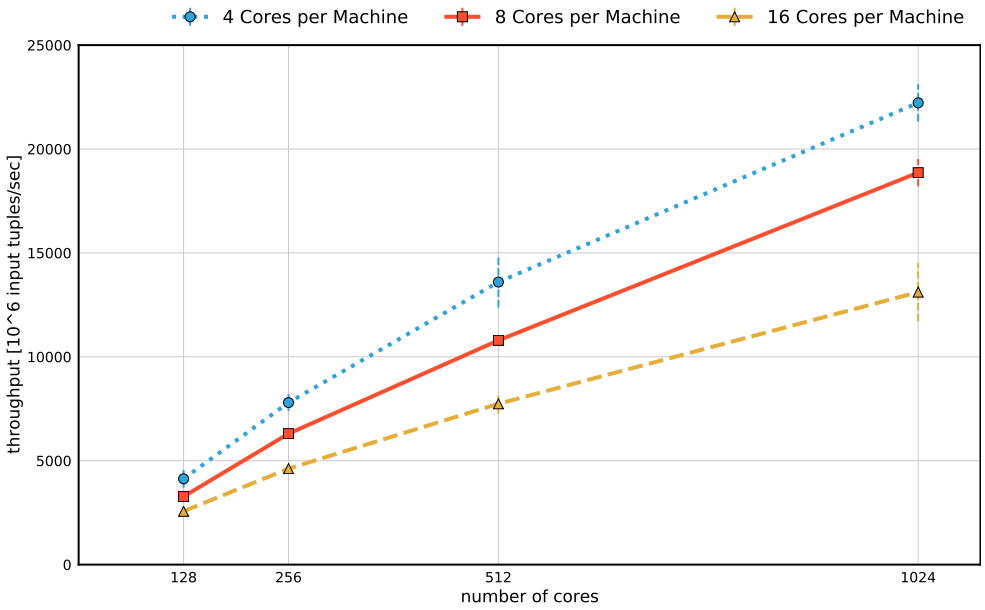
Effect of Input Selectivity

To study the impact of selectivity on the join algorithms, we use 1-to-4 workloads with 10 million and 40 million tuples per core. For each of the workloads, a different number of output tuples is produced. In Table 4.1 (lines 3-5), we show that the performance of the join remains constant for all three workloads. This is due to the fact that the execution time of the join is determined by the size of the input, not its selectivity. The actual matching of tuples only accounts for a small percentage of the execution time. Similar to previous work [KSC+09, BLP11, BTAÖ13, BATÖ13, BTAÖ15, BLAK15, BMS+17], we investigate the join operation in isolation and do not materialize the output, i.e., we do not fetch additional data over the network after the join result has been computed.

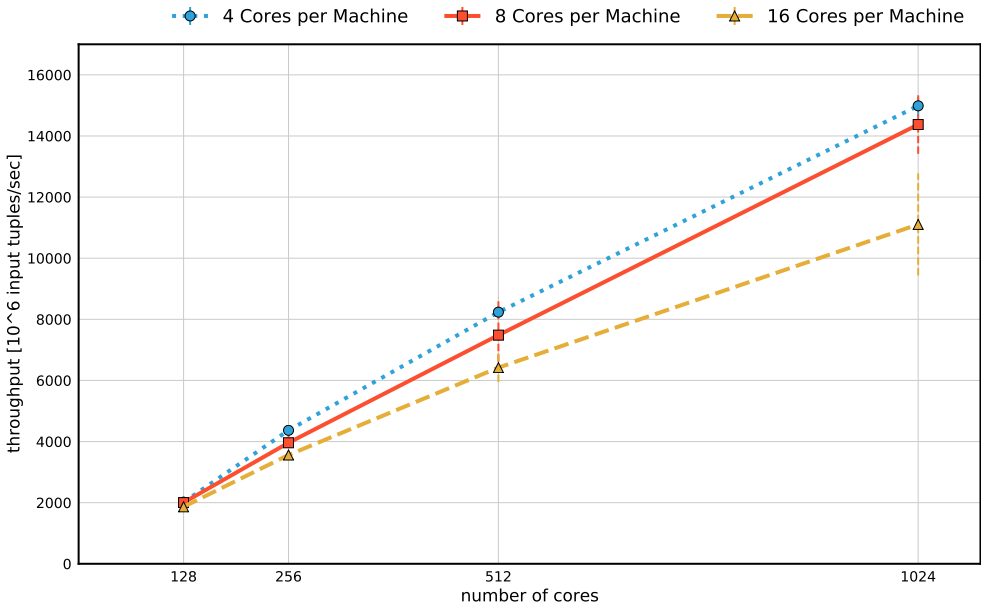
4.3.4 Scale-Up Experiments

When designing a distributed system, one is confronted with two design choices: scale-out and scale-up. In order to determine which of the two options is better suited for our implementations, we ran both algorithms on the Cray XC40 system, which allows us to increase the number of processes to 16 cores per node. In addition, we performed experiments on the Cray XC30 machine with 4 cores per node.

For both algorithms, we observe that the configuration with 4 cores per machine yields the highest throughput. As seen in Figure 4.6, the radix hash join benefits from the reduced interference as it is more memory intensive in its partitioning phase than the sorting operation of sort-merge join. The performance of both algorithms suffers when



(a) Radix hash join



(b) Sort-merge join

Figure 4.6: Scale-out and scale-up experiments with different number of cores per compute node for 40 million tuples per relation per core.

increasing the number of processes to 16 cores per node. We measured that considerably more time is spent executing the `MPI_Put` and `MPI_Flush` operations. More processes per machine put more load on each individual network card, which makes it difficult to fully interleave computation and communication.

In general, the more processes share the same network card, the more state the network card needs to hold (connections, memory translations, etc.). This is an important observation because this phenomenon is difficult to observe in conventional clusters. We conclude that the performance of both joins is directly related to the performance of the network and the number of processes that share the same network card.

4.4 Evaluation of the Performance Models

Using the model of both algorithms, we can compare the estimated and measured execution time. We use the models developed for the rack-scale experiments that are presented in Section 3.3 and evaluated in Section 3.5. Table 4.2 summarizes the results of the experiments along with the predictions of the model for both algorithms on 1024 cores. To instantiate both models, we use performance numbers gathered through component-level micro benchmarks on the Cray supercomputer.

For the hash join, we can see that the model predicts the performance of phases not involving any network operation. The model does not account for the cost associated with window allocation and registration. A significant difference comes from the noise inherent to large systems. This is reflected in the compute imbalance and the waiting time after the data exchange. From this observation, we can conclude that reducing the costs of the network operations would significantly speed up the hash join.

Similar observations can be made for the sort-merge join. The difference between measured and predicted execution time is due to the compute imbalance and the network wait time. We observe that despite these two factors, the execution time of the sort-merge join is close to the time predicted by the model, and the communication pattern of the sort-merge join is well suited for the underlying hardware.

Table 4.2: Evaluation of the performance models of the radix hash and sort-merge join algorithms for 1024 cores and 40 million tuples per relation per core.

Radix Hash Join			
Phase	Exec. Time	Model	Diff.
Histogram Comp.	0.34s	0.36s	−0.02s
Window Allocation	0.21s	—	+0.21s
Network Partitioning	2.08s	0.67s	+1.41s
Local Partitioning	0.58s	0.67s	−0.09s
Build-Probe	0.51s	0.38s	+0.13s
Imbalance	0.62s	—	+0.62s
Total	4.34s	2.08s	+2.26s
Sort-Merge Join			
Partitoning	1.20s	1.00s	+0.20s
Window Allocation	0.06s	—	+0.06s
Sorting	1.99s	1.78s	+0.21s
Merging	1.81s	1.78s	+0.03s
Matching	0.26s	0.22s	+0.04s
Imbalance	0.38s	—	+0.38s
Total	5.70s	4.78s	+0.92s

Parameters [million tuples per second]

RHJ: $P_{\text{scan}} = 225$, $P_{\text{Part}} = 120$, $P_{\text{net}} = 1560$, $P_{\text{build-probe}} = 210$

SMJ: $P_{\text{part}} = 120$, $P_{\text{sort}} = 45$, $P_{\text{net}} = 1560$, $P_{\text{merge}} = 270$, $P_{\text{scan}} = 370$, $d_{\{l,r\}} = 3$

4.5 Discussion

In this section, we discuss the outcome of the experiments focusing on the relative performance of hashing and sorting, the costs of communication along with the importance of network scheduling for the types of workloads used in the experiments.

***Hash v.s. Sort at Large Scale*:** We look at the behavior of sort-based and hash-based join algorithms on large scale-out architectures. Our findings show that the hash join is still the algorithm of choice in terms of raw throughput. However, they also reveal that several shortcomings prevent the algorithm from reaching an even higher throughput. One significant disadvantage lies in the uncoordinated all-to-all communication pattern during the first partitioning pass. Addressing this issue requires significant changes to the structure of the algorithm, potentially resulting in a new type of algorithm. Although the raw throughput is lower, the sort-merge join has several inherent advantages over its competitor. The interleaving of sorting and communication creates a steady load on the network. The fact that at each point in time every node has exactly one communication partner makes more efficient processing on the network possible. This implicit scheduling can be maintained up to a thousand cores, after which more sophisticated scheduling methods are required. In addition, the sort-merge join outputs sorted tuples, which might be advantageous later on in the query pipeline.

***Network Scheduling*:** Issuing `MPI_Put` requests is significantly more costly for the radix hash than for the sort-merge join. This is caused by the fact that the underlying hardware can only handle a limited number of simultaneous requests. To improve performance, these operations need to be coordinated. The results show that the performance of the hash join suffers from not having an effective scheduling technique. This problem is aggravated as more processes share the same network card. The sort-merge join avoids this problem at small scale as each process starts sorting a different range of the input. Despite this implicit network schedule, we observe that significantly more time is spent in the network calls as the number of cores increases. In essence, light-weight scheduling techniques are needed for both algorithms in order to maintain good performance while scaling out.

4.6 Related Work

***Database Systems and MPI*:** Liu et al. [LYB17] study the challenges of efficient data shuffling operators over RDMA capable networks. To that end, the authors propose six data exchange algorithms and evaluate them on a modern InfiniBand network. Besides testing different connection configuration parameters, e.g., reliable and unreliable transport services, they also evaluate different communication libraries, including an MPI implementation. The proposed MPI-based exchange algorithms use synchronous as well as asynchronous send/receive and broadcast primitives. Their findings indicate that algorithms that use low-level communication interfaces can significantly out-perform MPI-based implementations. This observation highlights the importance of having a specialized MPI implementation tuned for a specific network (e.g., foMPI [GBH13] for Cray Aries) instead of using a generic MPI implementation running on many different network technologies. Vectorwise [ZvdWB12] is an analytical database which originated from the MonetDB column-store project and VectorH [CIR+16] brings SQL to MapReduce environments by building on the multi-core support of Vectorwise. This system has been extended to a distributed system using an MPI-based exchange operator.

***Data Processing on Supercomputers*:** Some supercomputer vendors are increasingly offering software for advanced data processing on their hardware. The Cray Graph Engine [RHMM18] is an advanced platform for searching and traversing very large graph-oriented structures and querying interconnected data. The engine is designed to scale to supercomputer-sized problems. In their evaluation, the authors were able to process queries with a trillion of triples, and, among other operations, performed join operations on 512 nodes within seconds. Alchemist [GRW+18] is a framework for interfacing Apache Spark applications with MPI implementations. Alchemist calls MPI-based libraries from within Spark applications, enabling them to run on a supercomputer. Smart [WABJ15] is a re-implementation of a MapReduce framework directly using MPI as its communication abstraction, and the Spark-MPI [MCJ+18] project adds an MPI-based communication layer to the driver-executor model of Spark.

4.7 Summary

In this chapter, we proposed distributed hash and sort-merge join algorithms that use MPI as their communication abstraction. Just as with the rack-scale algorithms, these joins are optimized to use one-sided memory operations in order to take full advantage of modern high-speed networks. Using MPI addresses several challenges arising from large-scale distribution, primarily the automatic selection of the underlying communication method and the management of communication buffers. We evaluated both join implementations on two different distributed environments and showed that having the right balance of compute and communication resources is crucial to reach maximum performance and scalability. The proposed models show that the sort-merge join reaches its peak throughput. Reducing the network overhead would significantly speed up the radix hash join. Despite this fact, the performance of the radix hash join is superior to that of the sort-merge join.

Executing joins over large data sets in real-time has many applications in analytical data processing, machine learning, and data sciences. Therefore, it is crucial to understand the behavior of distributed joins at large scale. We showed that the radix hash and sort-merge join algorithms scale to 4096 processor cores, achieving a peak throughput of 48.7 billion input tuples per second.

5

Large-Scale Transaction Processing

Concurrency control is a cornerstone of distributed database engines and storage systems. An efficient coordination mechanism that supports a high throughput of transactions is a critical component for distributed database systems. Recently, the dramatic increase in parallelism arising from multi-socket, multi-core servers and cloud platforms has motivated both researchers and practitioners to explore alternative concurrency control implementations and weaker forms of consistency. Many of these proposals exhibit significant differences in throughput when running on a large number of cores or machines. These systems apply a wide range of optimizations that impose restrictions on the workloads the engine can support. For example, they give up serializability in favor of snapshot isolation [ZBKH17], impose restrictions on long-running transactions [KN11, TZK+13, DNN+15], assume partitioned workloads [KKN+08], or require to know the read and write sets of transactions ahead of time [KKN+08, TDW+12]. Due to the very different assumptions made and the wide range of performance levels achieved, these systems are difficult to compare to each other. However, one common underlying assumption is that Two-Phase Locking (2PL) and Two-Phase Commit (2PC) – the primary components of a textbook implementation of a database lock manager – do not scale.

A recent evaluation of several distributed concurrency control mechanisms suggests that a tight integration of concurrency control and modern networks is needed to scale out distributed transactions [HAPS17]. While the costs of synchronization and coordination might be significant on conventional networks, modern interconnects and new communication mechanisms, such as Remote Direct Memory Access (RDMA), have significantly lowered these costs.

5.1 Problem Statement and Novelty

In this chapter, we explain how to design a lock-based concurrency control mechanism for high-performance networks, establish a new baseline for running a lock manager on a system with thousands of cores, and show that the low latency offered by modern networks makes a concurrency control mechanism based on 2PL and 2PC a viable solution for large-scale database systems. This approach makes our findings relevant for scaling out existing database engines that use similar mechanisms.

The lock table used in this experimental evaluation supports all the conventional lock modes used in multi-level granularity locking. The system operates following a traditional design, as explained for instance in the Gray & Reuter book on transaction management [GR92]. We introduce neither optimizations and nor restrictions on transaction structure, operations, or presume any advance knowledge of the transactions or sequence of submission. We also do not use any pre-ordering mechanism such as an agreement protocol. Through the use of strict 2PL, the system provides *strict serializability*. To ensure that transactions leave the database in a consistent state, the system uses conventional 2PC. The questions we seek to answer are how to implement 2PC and 2PL on modern networks in order to achieve low-latency communication, how well the proposed mechanisms can scale with the number of machines, and whether or not the algorithms can take advantage of large parallel systems with hundreds of machines and thousands of cores.

Although, we focus on strong consistency in the form of *strict serializability* implemented through strict 2PL, we also make sure that the system can be used by weaker isolation

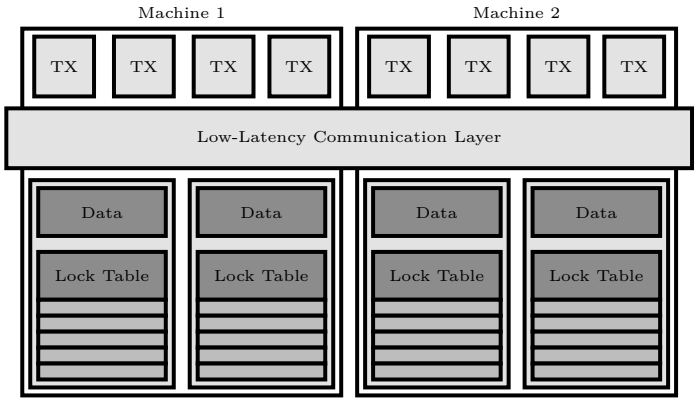


Figure 5.1: Architecture of the transaction processing system.

levels such as *read-committed*, a common isolation level used in database systems. In the process, we establish a new baseline for running a combination of TPC-C and synthetic workloads with different isolation levels on a supercomputer.

5.2 Distributed Transaction Processing using MPI

We develop a distributed lock table supporting all the standard locking modes used in database engines (see Section 2.3.5). Figure 5.1 depicts the system that has three main components: (i) the transaction processing layer is responsible for executing transaction on behalf of the clients, (ii) the lock table and data layer contains the data that is being manipulated as well as the data structures needed to synchronize data accesses, and (iii) the communication layer designed for low-latency communication.

Similar to the results presented by Wei at al. [WDCC18], we observed that, given the performance characteristics and programming abstractions of current networks, a hybrid approach composed of one-sided, two-sided, and atomic operations is need to build a scalable and performing system. To overcome these limitations, we propose new communication primitives targeting data-intensive applications, including operations seeking to accelerate transaction processing, in Chapter 6.

5.2.1 Transaction Processing Layer

The transaction processing agents are responsible for executing the transactions. Each agent runs in its own process, executes one transaction at a time, and is independent of other transaction processing agents. There is no direct communication between the transaction processing agents. Coordination is done exclusively through the lock-based concurrency control mechanism.

Upon start-up, the transaction processing agent discovers all available lock servers as well as the range of locks for which they are responsible. Each lock server is responsible for a fixed number of locks. With this information, the transaction agent can forward lock requests to the appropriate section of the lock table.

When a new transaction starts executing, a local identifier is assigned to it. System-wide, a transaction is uniquely identified by the combination of the transaction agent identifier and the local transaction number. Apart from assigning an identifier to a transaction, no additional setup is required. Next, the transaction acquires the required locks. To request a lock, the transaction generates a lock request message that is transmitted to the target lock server using a single one-sided RMA write operation. Each lock request contains a predefined *Lock request* message tag. Furthermore, a lock request contains the identifier of the lock, the identifier of the transaction processing agent, and the requested mode. Corresponding response messages are identified by a specific *Response* message tag. A response messages contains the same information as the request message, with the addition of a flag indicating if the lock has been granted or not. When contacting a lock server, the transaction agent stores the identifier of the lock server in order to be able to inform it when the transaction is ready to commit or has been aborted. Once all the locks have been successfully acquired, the transaction processing agent can access the data layer through one-sided read and write operations. At commit time, the transaction decides if the 2PC protocol needs to be executed. This is the case if data has been modified on at least one remote machine, i.e., when one or more remote items have been locked in exclusive (X) mode. If a vote is required, the transaction processing agent starts the Two-Phase Commit protocol among all involved processes. Processes that did not contribute

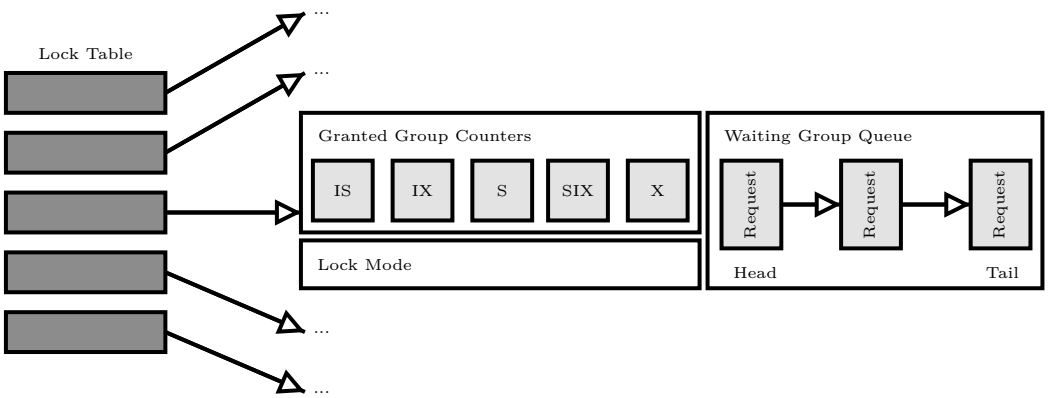


Figure 5.2: Lock table entry layout.

to a transaction do not participate in the vote. The transaction processing agent takes the role of coordinator, registering how many positive and how many negative votes have been collected. Once every participant has voted, the transaction processing agent informs them about the outcome through the use of a *End of transaction* message. We do not use any optimizations such as Presumed-Abort or Presumed-Commit [BHG87].

5.2.2 Lock Table and Data Layer

The lock table server processes are responsible for receiving and executing requests from the transaction processing layer. They manipulate all relevant data structures used to manage the locks: (i) the lock table containing the individual locks, (ii) the transaction table, which contains lists of locks held by each transaction, and (iii) the deadlock detection list that contains all the locks that can be part in a potential deadlock situation.

The data guarded by the locks in the lock table is co-located on these processes. It is accessed by the transaction layer through one-sided memory operations. Apart from loading the data and registering the buffers with the network card at start-up, the lock table agent is not involved in data retrieval and manipulation operations.

The lock table contains all available locks together with their pending and granted requests. Although it is logically one table, it can be distributed across multiple lock table agents on

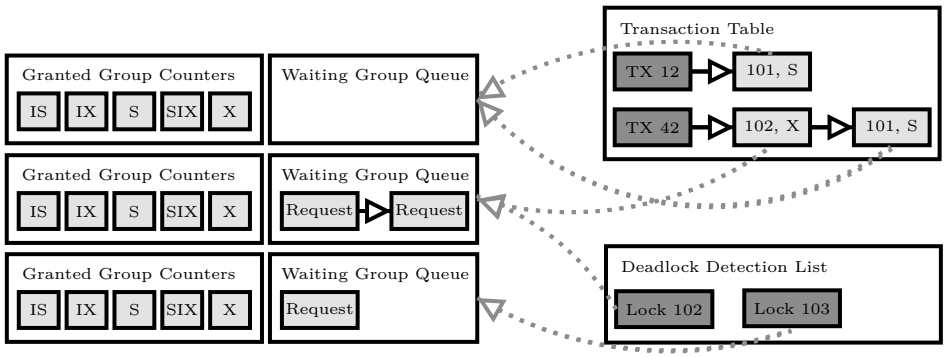


Figure 5.3: Auxiliary lock table data structures.

different physical machines. Each agent is assigned to an exclusive non-overlapping range of consecutive lock identifiers. Therefore, accesses to the lock table entries do not need to be synchronized. The ranges are chosen such that each server process is responsible for an equal number of locks. This information is broadcast to the transaction processing agents at system start-up. Each entry in the table corresponds to one lock. As seen in Figure 5.2, the lock data structure is composed of a queue of pending requests (waiting group) and a set of counters (granted group). The lock table supports multi-level granularity locking. For each mode, there is exactly one counter indicating how many requests of that mode have been granted. From this information, the lock mode can be computed. This enables the lock server to quickly determine if the head of the request queue is compatible with the granted requests.

The transaction table holds information of each running transaction. It implements a multiset, i.e., for each transaction, the table contains a collection of all acquired locks together with their request modes (see Figure 5.3). To ensure that a transaction can be uniquely identified, this table operates on global transaction numbers. These 64-bit global transaction numbers are based on the combination of the identifier of the transaction processing agent (upper 32 bits) and a local transaction number (lower 32 bits) issued by the transaction processing agent for each transaction that it executes. Although individual locks can be released at any point in time, the primary purpose of the transaction table is to implement an efficient strict 2PL system. In strict 2PL, there is no shrink phase

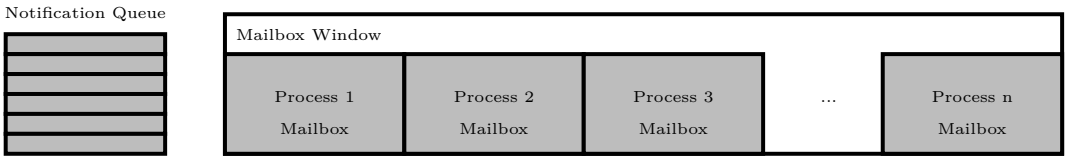


Figure 5.4: Low-latency mailbox buffer management.

in which locks are progressively unlocked. Rather, all acquired locks are released by a transaction upon commit or abort. Using this data structure, the lock table agent can release all the locks held by a transaction without having to receive multiple or variable-sized *Unlock* messages. This information is also useful to speed up the recovery phase in case of failures [AAE94].

The lock table agents perform a time-based deadlock detection. To that end, the lock table agent adds the current timestamp to all incoming requests before adding them to the waiting group of the requested lock. Furthermore, each table agent keeps a list of local locks that have pending requests (see Figure 5.3). The agent iterates over this list to determine how long the head of the queue has been waiting to acquire the lock. A lock must be acquired within a predefined time frame (e.g., 100ms). If a timeout occurs, the transaction is informed about the unsuccessful lock attempt with a negative acknowledgment message and the request is removed from the waiting group. This bounded-wait deadlock detection mechanism (DL-BW) enables to resolve deadlocks while also avoiding an excessive abortion rate in case of workload contention. When the last request has been removed from the waiting group, either because it has been granted or because it timed out, the lock table agent removes the lock entry from the list of locks with pending requests to no longer include it in the computation of deadlock detection mechanism.

5.2.3 Low-Latency Communication Layer

In order to support a variety of high-performance networks, the communication between the transaction processing layer and the lock table agents uses the Message Passing Interface (Mpi) [Mes12] as an abstraction. This has the advantage that the interface is

identical for communication between local and remote processes, which hides the complexities arising from large-scale distribution, while still delivering good performance by using the most appropriate communication method based on the relative distance of the processes involved in the communication. In our system, we use foMPI-NA [BH15], an extention of foMPI [GBH13] (see Chapter 4.3) that introduces *notified access* communication primitives that aim to reduce synchronization latencies, especially for fine- and medium-grained data transfers.

Communication between the transaction processing agents and the lock table agents is performed exclusively using one-sided RMA operations. Upon start-up, each process allocates a set of two buffers and registers them with the network card using `MPI_Win_alloc`. This operation is a collective operation, which means that every process involved in the communication needs to execute this operation. During window allocation, the access information to these buffers is exchanged between all processes, such that every component of the system is able to read and write to these regions of memory using RDMA operations. The first of these buffers is used as a mailbox for incoming messages. The second one is used in the voting phase of the 2PC protocol.

Since the lock table agents can potentially receive requests from any transaction, their mailbox is wide enough that it can accommodate one message from each transaction processing agent. Each process in the transaction processing layer can have at most one pending lock request that needs to be granted before it can continue processing. Therefore, its mailbox size is such that it can hold a single message. *Lock request*, *Response*, and *End of transaction* messages are transmitted by issuing a `MPI_Put_notify` call. This extended MPI interface is available in the notified access extension [BH15] of foMPI [GBH13]. This call triggers a remote write operation similar to a `MPI_Put` with the addition of a notification on the remote machine. Some network implementations refer to this operation as a *write with immediate*. In order to avoid synchronization when writing to the mailbox, the i-th transaction processing agent will write its content at the i-th slot in the mailbox.

On the target side, the lock table process can start listening for incoming notifications by initializing the notified access support (`MPI_Notify_init`) and activating a request

handle (`MPI_Start`). Using this handle, a process can either wait for messages (`MPI_Wait`) or perform a non-blocking test to verify if a new notification has been created (`MPI_Test`). Once a notification is ready, the target can read out the origin of the request and consume the content at the respective message slot. Using notified access operations, avoids that the target process has to iterate over all message slots, which would impact the scalability of the communication mechanism. Furthermore, using a mailbox is beneficial for small messages as the content of a request can directly be placed in a specific pre-allocated region memory, which avoids any dynamic allocation of RDMA send and receive buffers during execution. When a request is granted, the corresponding notification is placed in the mailbox of the transaction using the same mechanism. The lock server agents use the non-blocking test to check for incoming messages. If there is no new request to process, they check for deadlocks. The transactions on the other hand use the blocking wait operation as they cannot continue processing before the lock has been granted.

The second memory window is used during 2PC . It is wide enough to accommodate a single 64-bit integer. Before broadcasting a vote request message to the lock servers involved in a transaction, the transaction processing agent zeroes out this memory. Upon receiving the vote request, the lock servers perform a remote atomic operation on this memory, either incrementing the lower 32 bits to signal a positive vote, or the upper 32 bits to trigger an abort. This is done by issuing an `MPI_Fetch_and_op` operation combined with the `MPI_SUM` argument. Using an atomic operation makes use of the hardware-acceleration available in these network cards and avoids expensive processing in software.

5.3 Performance Model

The throughput of the concurrency control system is dependent on the time required to acquire locks. Acquiring a single lock requires (i) a message to be transmitted from the transaction processing layer to the lock table server, (ii) the lock table servers checking their notification queue for incoming messages, (iii) processing them accordingly, and (iv) sending a message back to the origin of the request.

In our model, a request spends t_{comm} amount of time in the communication layer. This amount of time is dependent on the relative distance of the two processes.

$$t_{\text{comm}}(\text{source, target}) = \begin{cases} t_{\text{local\_comm}}, & \text{source \& target on the same machine} \\ t_{\text{remote\_comm}}, & \text{otherwise} \end{cases} \tag{5.1}$$

The expected time in the queue depends on the contention of the locks given the workload w: $t_{\text{queue}}(w)$. The workload also dictates the probability that a local lock is taken $P_{\text{local\_lock}}(w)$ and the amount of locks a transaction takes $N_{\text{locks}}(w)$. Assuming independent accesses, the probability that all locks are taken can therefore be determined.

$$P_{\text{all\_local}}(w) = P_{\text{local\_lock}}(w)^{N_{\text{locks}}(w)} \tag{5.2}$$

When at least one lock is remote, a two-phase commit protocol is executed at the end of the transaction. The time required to execute the voting operation is t_{vote}. And the time required to clean up a transaction is denoted as $t_{\text{clean\_up}}$. From the above, we can determine the execution time of an individual transaction. Given that acquiring a lock requires two messages (a request and a response), the communication time has to be counted twice.

$$\begin{aligned} t_{\text{tx}}(w) = {} & N_{\text{locks}}(w) \cdot 2 \cdot \Big(P_{\text{local\_lock}}(w) \cdot t_{\text{local\_comm}} + (1 - P_{\text{local\_lock}}(w)) \cdot t_{\text{remote\_comm}} \Big) \\ & + N_{\text{locks}}(w) \cdot t_{\text{queue}}(w) + (1 - P_{\text{all\_local}}(w)) \cdot t_{\text{vote}} + t_{\text{clean\_up}} \end{aligned} \tag{5.3}$$

Each of the $N_{\text{tx\_core}}$ cores executes one transaction at a time. Therefore, the expected throughput $Tp(w)$ can be determined.

$$Tp(w) = \frac{1}{N_{\text{tx\_core}} \cdot t_{\text{tx}}(w)} \tag{5.4}$$

The above model, does not include the time required to access data and therefore only represents the maximum throughput the lock table can support. In case data accesses should also be considered, additional time $t_{\text{data}}(\text{query})$ needs to be added to the execution time of the transaction that is specific for the given query.

5.4 Experimental Evaluation

In the experimental evaluation we study the performance of our concurrency control mechanism on a large-scale compute infrastructure with up to 5040 cores.

5.4.1 Workload and Setup

The experiments are conducted on the same supercomputer as the large-scale join experiments in Chapter 4. We use the compute nodes of the XC40 partition of the system described in the previous chapter. The machines contain two Intel Xeon E5-2695 v4 processors with up to 128 GB of main memory. The network remains unchanged. The compute nodes are connected through a Cray Aries routing and communications ASIC [AKR12] connected through a Dragonfly [KDSA08] topology.

Our concurrency control mechanism implements a conventional lock table conceptually similar to the one used in many existing databases systems. In order to gain insights into scaling out conventional database architectures, we augmented the lock management mechanism of the *MySQL* database server in order to get a detailed trace of all the locks that get acquired. This information includes the transaction number, the identifier of the acquired lock, and the requested lock mode. Using this modified database system, we generated lock traces of the Tpc-C benchmark using different isolation levels: *serializable* and *read committed*. In a distributed database system, we envision that different lock table agents are responsible for managing locks belonging to different Tpc-C warehouses. To be able to scale to thousands of cores, we configured the benchmark to simulate 2520 warehouses. The augmented lock manager provided us with a set of locks and their corresponding lock mode that each transaction was granted. Using the official Tpc-C description, we access data on that warehouse using one-sided read and write operations once all the locks have been acquired.

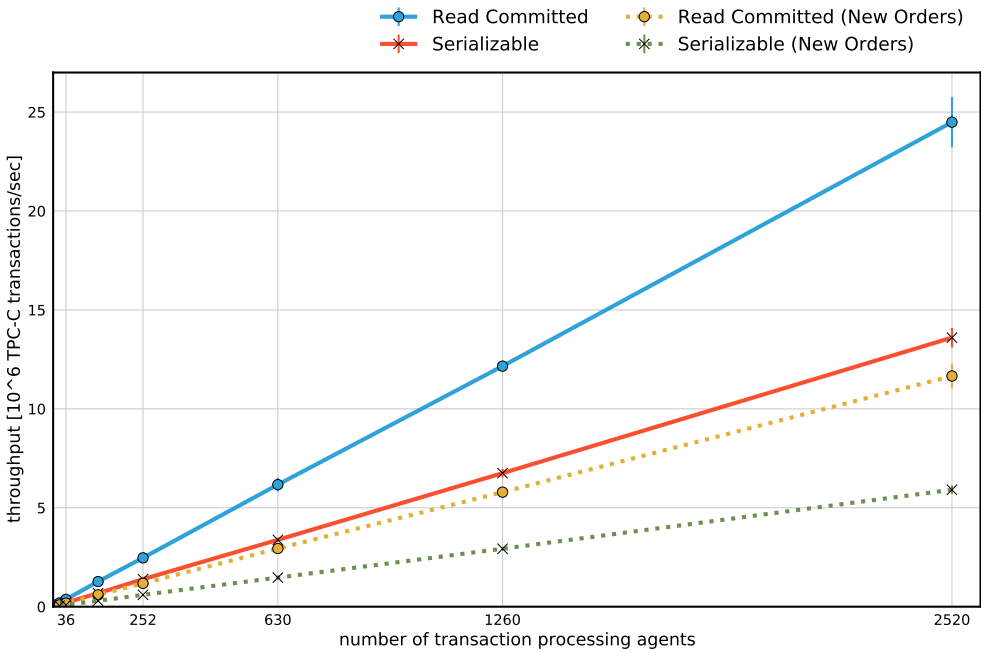
To be able to study the effect of remote lock accesses, we designed a synthetic benchmark. Each transaction acquires on average ten locks. Contrary to the Tpc-C benchmark, we

vary the probability of a lock not residing on a specific machine, thus changing the ratio of local and remote locks that need to be acquired. In our evaluation, we acquire up to 50 percent remote locks. The lock mode for each request is taken uniformly at random. The system does not access any data while running the synthetic workload.

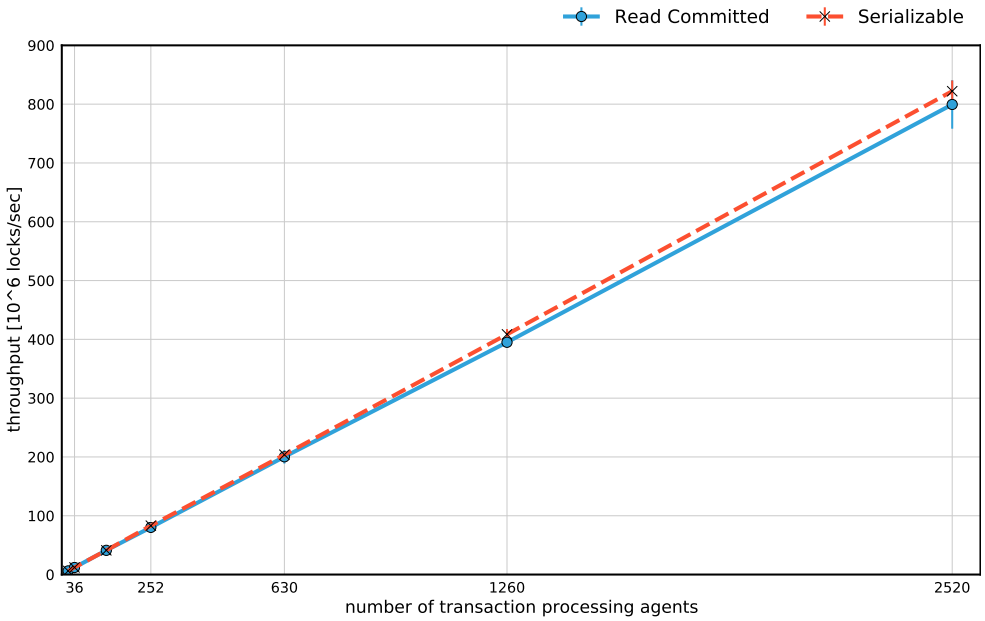
5.4.2 Scalability and Isolation Levels

In these experiments, we deploy multiple configurations of the system. Each node in the system has 36 processor cores that are either assigned to the lock table layer or the transaction processing layer. We found that deploying 18 lock table agents together with 18 transaction processing agents per compute node yields the highest throughput (13.6 million transactions per second in serializable mode), compared to using 12 or 6 cores for the lock table layer (10.2 million transactions per second and 4.72 million transactions per second, respectively). Each process is bound to a dedicated core and the processes are distributed equally over both sockets. A lock table agent is responsible for managing one or more warehouses, while the transaction processing agents execute queries and transactions on behalf of the clients. In Tpc-C, each client has a home warehouse which is accessed most frequently. Therefore, it is reasonable to assume that clients connect to a transaction processing agent that is located on the same physical machine as the data belonging to its home warehouse. Requests targeting a specific warehouse originate from a single source in the transaction processing layer. This setup also reduces the number of conflicts and aborts as transactions targeting the same home warehouse are partly serialized within the transaction processing layer.

We scale our implementation from a single machine up to 140 physical compute nodes, which corresponds to a total of 5040 processor cores. In the execution of the Tpc-C benchmark used to collect the traces containing the history of acquired locks by the transactions, we used a total of 2520 warehouses. Although our concurrency control system is agnostic to the workload and can support an arbitrary number of warehouses, using the traces we collected, the transaction processing agents are limited to replaying transactions that target up to the maximum number of available warehouses.



(a) Transaction throughput



(b) Lock table throughput

Figure 5.5: Throughput of the TPC-C workload for two different isolation levels: read committed and serializable.

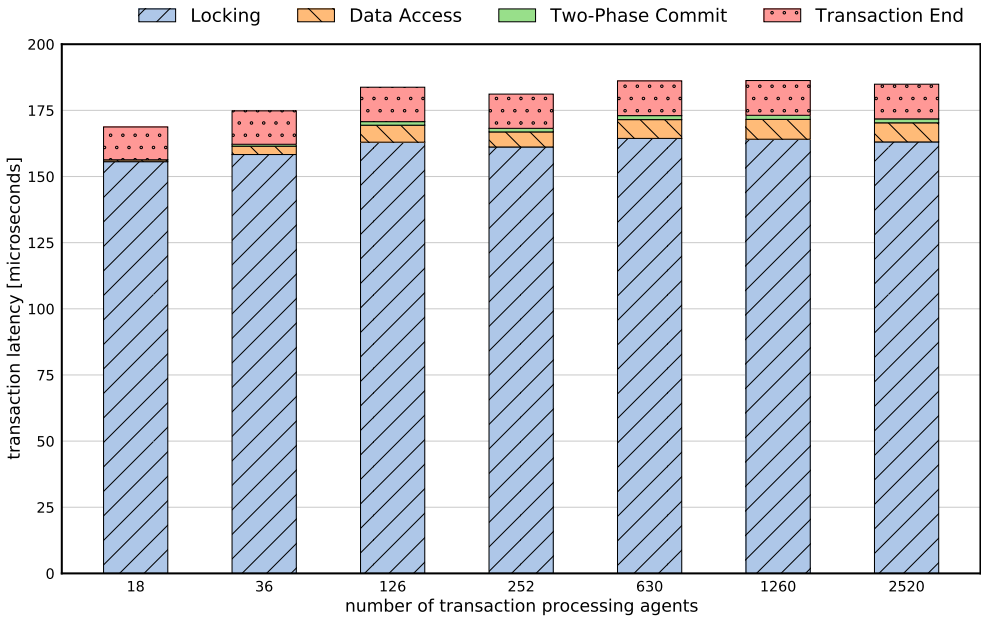
In Figure 5.5a, we see two executions of the TPC-C trace, using different isolation levels. The error bars represent the 95% confidence intervals. We can see that the system is able to take advantage of the increased core count and is able to scale to thousands of cores. We observe a linear performance increase as we scale out both layers of the system simultaneously. At full scale, the table can support over 13.6 million transactions per second in serializable mode. The throughput in terms of lock requests is independent of the isolation level (see Figure 5.5b). This is an important aspect of the system as it makes the behavior of the lock table independent of the provided isolation level. In both cases, the lock table can sustain a throughput of over 800 million lock requests per second. This in turn gives us a predictable throughput in terms of lock table operations and is part of the reason why an isolation level requiring fewer locks (i.e., read committed) can achieve a higher throughput (24.5 million transaction per second) in terms of transactions.

When adding compute nodes, both layers can be scaled out in the right proportions, thus ensuring that no component is becoming the bottleneck. As resources are added, the lock table can either be distributed with a finer granularity such that each table agent is responsible for fewer locks. Alternatively the higher core count can be used to serve more locks overall.

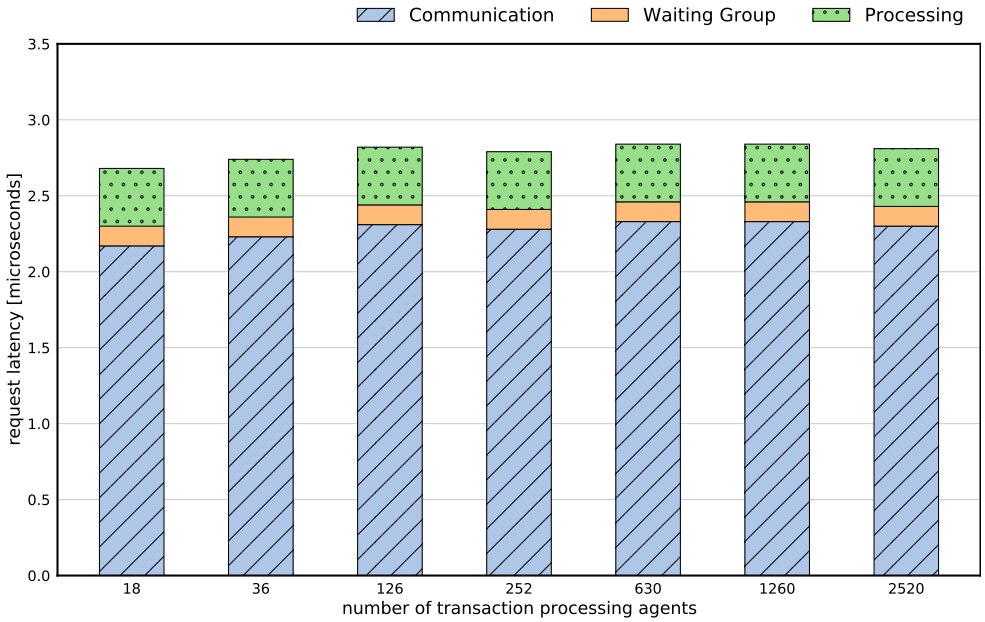
Furthermore, we can observe that a stricter isolation level requires taking significantly more locks, which results in a lower transaction throughput. The serializable execution of TPC-C takes on average 53.2 locks per transaction, while running a transaction in the *read committed* mode requires only 27.9 locks on average. The reported throughput in serializable mode of 13.6 million – 5.9 million *new order* – transactions is significantly higher than the official benchmark results [Tra18b].

5.4.3 Execution Time Breakdown

The majority of the execution time of the TPC-C workload is dedicated to acquiring locks. Around 12 percent of the time is needed for accessing the data, executing the 2PC protocol, and informing the lock table agents that a transaction has ended (see Figure 5.6a). There are multiple reasons for this behavior. First, transactions request multiple locks, while



(a) Transaction



(b) Lock request

Figure 5.6: Latency breakdown for a transaction and lock request for the TPC-C workload in serializable mode.

there is at most one vote operation per transaction. Second, locks are acquired one after the other as they are needed. For systems that require a deterministic behavior of the workload, this time could be lowered by either requesting multiple locks in quick succession or by issuing requests that target multiple locks, thus amortizing the round-trip latency. Vote requests can always be issued and collected in parallel. The time required to execute a vote is dependent on the slowest participant, not the number of participants. Third, the majority of transactions targets the home warehouse of the client. Since transactions are executed by a processing agent co-located with the locks and the data, most transactions modify items in local memory and acquire only local locks. Transactions that do not modify data on remote machines do not execute a 2PC protocol. Transactions that need to execute the commit protocol often have a small number of participants in the voting phase. A TPC-C transaction needs to contact on average 1.1 lock table agents.

As seen in Figure 5.6b, acquiring a lock takes on average 2.8 microseconds. The majority of the time is needed for inter-process communication (round trip time) and message queuing within the communication. The more the lock table servers become the bottleneck, the longer requests are being queued in the communication layer before they are being processed by the lock table layer. Once the request has been received and is being processed, it is added to the waiting group, in which it spends 0.13 microseconds, indicating that there is only a small amount of contention in the workload. Finally, the remaining 0.38 microseconds are required by the lock table agent for checking if the request is compatible with the current state of the granted group, updating the lock mode, and for preparing the response message in order to inform the client about the outcome of the request.

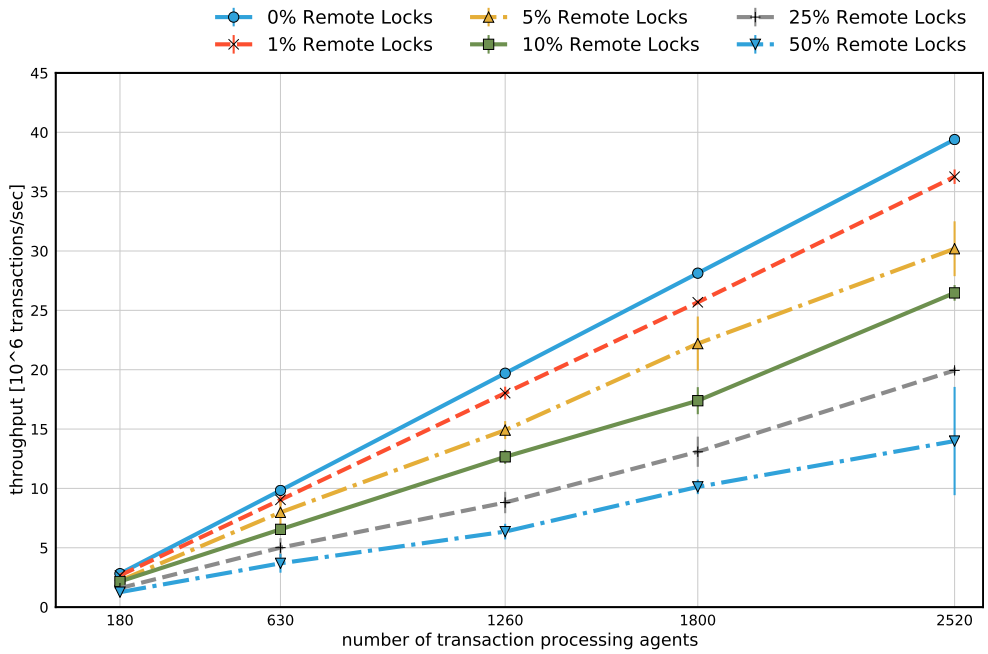
The vote of the 2PC protocol requires 8.5 microseconds on average, which is longer than the time required to acquire a lock. The reason for this behavior is that multiple servers need to be contacted which does not happen completely in parallel. The more servers are involved, the higher the chance that a single straggler will delay the outcome. Finally, in our system, atomic operations are cached in fast memory on the network card and updates to these values only become visible after an expensive synchronization call by the initiator of the vote. In general, we observe that updates to atomic values become visible to the processor faster in networks that do not rely heavily on caching intermediate values.

Given that the TPC-C workload, with its concept of a warehouse, can be partitioned across many physical nodes, we observe that the latency of both operations does not change significantly as we add more cores. This shows that our system exhibits predictable and scalable performance for partitionable workloads.

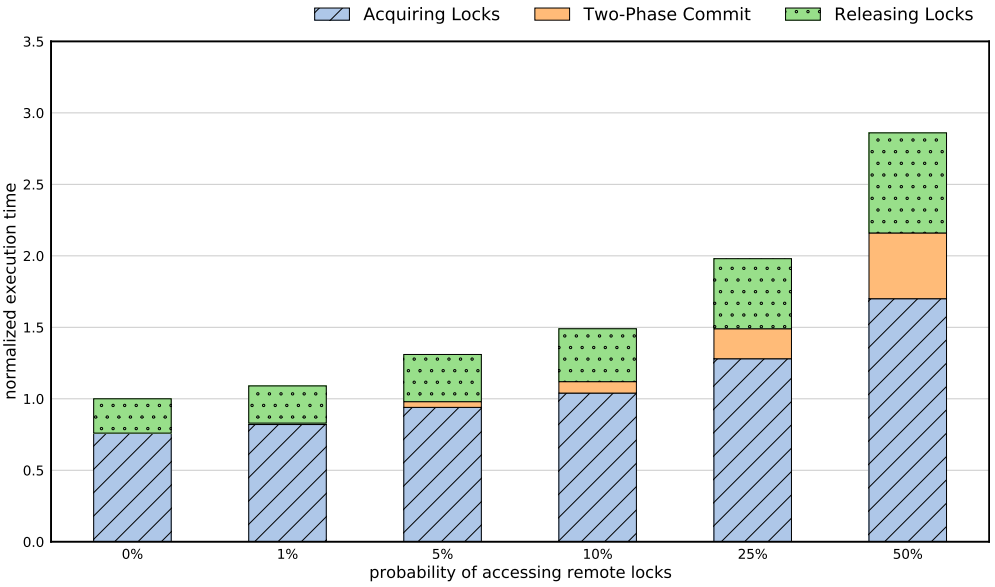
5.4.4 Local and Remote Access Ratios

The ratio of local and remote locks that are acquired depends heavily on the workload. Workloads that can be partitioned usually have a higher proportion of local accesses than workloads that need to access data from different partitions. In TPC-C, *new order* transactions have a certain probability of accessing an item not belonging to the home warehouse of the client that issues the transaction (1 percent per item). To investigate the impact of remote lock accesses on the performance of the concurrency control mechanism, we use the synthetic workload and vary the probability of accessing remote locks.

In Figure 5.7, we scale the synthetic workload to the same scale as the TPC-C traces, namely 2520 concurrent transactions. The error bars represent the 95% confidence intervals. Similar to previous experiments, we dedicate 18 cores per machine to serve the lock table, and assign 18 cores to the transaction processing layer. We observe that the throughput in terms of transactions decreases as more remote elements need to be locked (see Figure 5.7a). This drop in performance has multiple reasons. The latency of a remote lock request is higher than that of a local lock. In Figure 5.7b, we see that more time is required to acquire the same number of locks. In addition, to more expensive lock operations, more overall time is needed for committing the transactions as more 2PC protocols need to be executed before it is safe for the transactions to commit their changes. Furthermore, a transaction needs to collect votes from a higher number of participants. It also needs inform more table agents that the transaction has ended. Although the clean-up process is supposed to be executed in parallel, the costs of this phase increase as more lock table servers are involved in managing the transaction and its locks. The combination of these effects leads to an increase in execution time by a factor of 2.8 (see Figure 5.7b).



(a) Transaction throughput



(b) Execution time for 2520 transaction agents. Normalized to the execution with only local locks.

Figure 5.7: Effects of accesses to remote locks with the synthetic workload.

Table 5.1: Evaluation of the performance model of the concurrency control mechanism for 1260 and 2520 concurrent transactions.

Remote Lock Probability	**0%**	**1%**	**5%**	**10%**	**25%**	**50%**
Predicted Exec. Time [us]	67.500	68.833	73.511	78.236	88.521	101.992
1260 - Predicted Tp [M TX/s]	18.67	18.31	17.14	16.11	14.23	12.35
1260 - Measured Tp [M TX/s]	19.70	18.03	14.91	12.66	8.81	6.36
2520 - Predicted Tp [M TX/s]	37.33	36.61	34.28	32.21	28.47	24.71
2520 - Measured Tp [M TX/s]	39.39	36.26	30.19	26.47	19.94	13.99

5.5 Evaluation of the Performance Model

The synthetic workload is useful as it enables us to evaluate the validity of the performance model proposed in Section 5.3 under a variety of remote access probabilities. Table 5.1 shows the expected transaction runtime and throughput for 1260 and 2520 concurrent transactions. When the probability of accessing a remote lock is low, the model is able to accurately predict the measured throughput of the system. The difference between the model prediction and the measurement is around 5 percent. In cases where many remote locks are accessed, the model provides a lower execution time and the predicted throughput is higher than the one measured on the supercomputer.

The model is able to accurately predict the amount of time it takes the query to acquire all the locks. However, the execution times of the voting phase and of the clean-up phase – in which the transaction processing agent instructs all involved lock table servers to release the locks and clean up all associated data structures – are not constant. This phenomenon can also be seen in Figure 5.7b. More lock table servers are contacted in workloads with a high remote lock probability. In an ideal execution, the time to complete these two phases is independent on the number of lock table servers as the voting and clean-up processes are executed in parallel on all severs. However, small load imbalances cause this process to not happen simultaneously, thus resulting in higher transaction voting and clean-up

costs than predicted by the model, which in turn results in a lower throughput than what is theoretically possible given the parallelism of the machine.

5.6 Discussion

***Choice of Workloads*:** In the experimental evaluation, we used a combination of TPC-C and synthetic workloads to evaluate our lock table implementation. We observe that the workloads used in this evaluation provide very little contention. This can be seen by the short amount of time that lock requests spend in the waiting group. Note that the baseline we provide in this dissertation intends to test the scalability of concurrency control mechanism, not the scalability of the workload, a problem already pointed out in related work [ZBKH17]. It is important to distinguish between the scalability of the underlying mechanism that is offered by the database system and the characteristics of the workload: In the presence of high-speed networks, a lock-based concurrency control mechanism is a scalable approach for enforcing high transaction isolation levels. To translate this performance into a high throughput in terms of transactions, one requires a scalable workload. This is not the same as having a partitioned workload, but rather depends on the amount of contention present in the workload. Most database workloads do not have a single highly-contented item and thus not a single lock that every transaction seeks to lock in exclusive mode. However, if a workload exhibits such contention, for most concurrency control mechanisms, a lower overall throughput would be observed than what the mechanism could support. In such a scenario, we would not observe a degradation of the message passing latency, but rather an increased waiting time of requests in the queue or a high abort rate if the deadlock detection timeout is too short.

***One Mechanism for Many Isolation Levels*:** Using a weaker isolation level translates to fewer locks being taken and for a shorter period of time. This means that the load on the lock table decreases and the freed resources could be added to the transaction processing layer to process more transactions in parallel. The overall throughput can be further increased as the isolation level requirements are lowered. Locking mechanisms are

not only useful to implement pessimistic concurrency control. Snapshot isolation and optimistic concurrency control mechanisms can be implemented on top of a locking system, not to prevent concurrent access, but to detect conflicts. In such systems, even fewer locks are needed, e.g., in snapshot isolation, transactions do not take locks for reading data. As seen in the comparison between *serializable* and *read committed*, solutions that take fewer locks are expected to perform better.

***Detecting Deadlocks*:** The deadlock detection mechanism used by this system is based on timeouts (DL-BW). A request can only wait for a specific predefined period of time in the waiting group before it is canceled. The idea is to detect deadlocks while also not aborting too many transactions in case of light contention on one of the locks. In an alternative design, the lock table agent could also construct a wait-for graph in order to detect deadlock situations. Since two transactions can be conflicting on two locks managed by two different table agents such a mechanism would require an additional communication and synchronization protocol between the processes managing the lock table.

5.7 Related Work

In recent years we have seen a renewed interest in large-scale concurrency control due to the increasing amount of parallelism and the benefits that it entails.

Schmid et al. [SBH16] propose a topology-aware implementation of MCS locks optimized for high contention using one-sided network instructions. Yoon et al. [YCM18] design a locking protocol based on fetch-and-add operations that is fault-tolerant and starvation-free. Both approaches have in common that they only support two locking modes (shared and exclusive) and cannot easily be extended to the six modes we support as this would require wider machine words than those supported by atomic RDMA operations available on current hardware.

Spanner [CDE+12] is a large-scale distributed database system that focuses on geographic distribution. The system not only uses a lock table to implement concurrency control, but also relies on GPS and atomic clocks to serialize transactions at a global scale. This setup is

Table 5.2: Performance results for large-scale concurrency control mechanisms.

System/Paper	Mechanism	Cores Machines	TPC-C Perf. SNOT/s
This dissertation	2PL-BW	5.0k (140)	5.9M
HyPer [KN11]	TS+MVCC	8 (1)	171k
Silo [TZK+13]	MVOCC	32 (1)	315k
FaRM [DNN+15]	OCC	1.4k (90)	4.5M
NAM-DB [ZBKH17]	TS+MVCC	896 (56)	6.5M
DrTm [WSC+15]	HTM	480 (24)	2.4M
DistCC Eval* [HAPS17]	TS+MVCC	512 (64)	410k
	2PL-NW	512 (64)	300k
	OCC	512 (64)	100k
	TS	512 (64)	430k
	2PL-WD	512 (64)	340k
	Calvin [TDW+12]	512 (64)	380k
Abyss* [YBP+14]	2PL-DD	1k (1)	760k
	2PL-NW	1k (1)	670k
	2PL-WD	1k (1)	-
	TS	1k (1)	1.8M
	TS+MVCC	1k (1)	1.0M
	OCC	1k (1)	230k
	H-Store [KKN+08]	1k (1)	4.3M

* Marked systems only implement a subset of the Tpc-C workload.

different from the one used in our evaluation, where the focus is on using high-performance networks to achieve low-latency communication between all system components in a single geographic location. Chubby [Bur06] is lock service designed to provide coarse-grained reliable locking. The design emphasis is on ensuring high-availability for a small number of locks. This scenario is different from the locking mechanisms used in database systems that focus on achieving a high throughput for a large number of uncontended locks. Furthermore, using coarse-grained locks is not suitable for some database workloads, for example coarse-grained locking is sub-optimal for transactions that need to access a few specific items.

FaSST [KKA16] is an RDMA-based system that provides distributed in-memory transactions. Similar to our system, FaSST uses remote procedure calls over two-sided communication primitives. The authors pay special attention to optimizing their system to use unreliable datagram messages in an InfiniBand network. Unlike our implementation, this system uses a combination of optimistic concurrency control (OCC) and 2PC to provide serializability. Although the evaluation does not include the TPC-C benchmark, the system is able to outperform FaRM [DNCH14, DNN+15] on other workloads. DrTM [WSC+15] is an in-memory transaction processing system that uses a combination RDMA communication primitives and hardware transactional memory (HTM) support to run distributed transactions on modern hardware.

Table 5.2 shows an overview of selected related work. The performance numbers are taken from the original publications. As a best effort, for systems that only implement a subset of the TPC-C workload (marked in the table by '*'), we converted their results to number of successful *new order* transactions per second (SNOT/s) by assuming that the missing transactions execute at the same speed as the mix of the implemented ones. Since the performance of some schemes decreases with increasing core count, we take the highest achieved throughput as peak performance. For the paper by Harding et al. [HAPS17] ("DistCC Eval"), we use the numbers with 1024 warehouses, which are better than the numbers with 4 warehouses presented in the same paper. The paper provides an evaluation for the most popular concurrency control mechanism: Two-Phase Locking No-wait (2PL-NW), Two-Phase Locking Wait-die (2PL-WD), optimistic concurrency control (OCC),

multi-version concurrency control (MVCC), timestamp ordering (TS), and the mechanism used by Calvin [TDW+12].

In the following, we describe the compromises done by the systems in Table 5.2. Some databases, including FaRM [DNN+15] (4.5 million SNOT per second), implement optimistic concurrency control (OCC) without keeping multiple versions and verify at the end of each transaction that the read and write set does not intersect with that of concurrent transactions. This approach requires that the changes of all transactions are being kept during the lifetime of the longest-running concurrent query, which limits on how long that period can be [KR81]. If several versions of each record are stored (MVOCC), such as in Silo [TZK+13] (315 thousand SNOT per second), the read sets of read-only transactions do not need to be tracked. Read-only transactions can be arbitrarily long. However, this is not the case for read-write transactions. Multiversion concurrency control (MVCC) combined with timestamps (TS) handles long-running read-only transactions. Long-running read-write transactions may be problematic or impossible. For example, HyPer [KN11] (171 thousand SNOT per second) forks long-running transactions into a new process that sees the snapshot of the virtual memory at the time of its fork and cannot perform any updates. Furthermore, transactions must be written as stored procedures in order to classify them as long- or short-running in advance. NAM-DB [ZBKH17] (6.5 million SNOT per second) allows updates in long-running transactions, but only checks for write-write conflicts, thus giving up serializability in favor of snapshot isolation. While snapshot isolation is widely used, it is not without problems [WJFP17]. The other MVCC mechanisms from Table 5.2 achieve good serializability by locking new versions until commit time and aborting on updates of records with newer reads. This can lead to starvation in presence of contention because the longer a transaction runs, the more likely it is that other transactions access its (future) write set. If only a single version of the data is kept (TS without MVCC), this problem is even more pronounced.

Recent work on concurrency control proposes to deterministically order data accesses in order to avoid any form of synchronization. While in H-Store [KKN+08] (4.3 million SNOT per second), an early system following this idea, this approach did not work with unpartitioned workloads due to the coarse-grained partition locking, newer systems such as

Calvin [TDW+12] (380 thousand SNOT per second) overcome this issue. Both systems need to know the read and write sets of each transaction beforehand (or detect them in a dry-run). This assumption can only be made for stored procedures and is impractical for long-running queries. In contrast, locking avoids the above-mentioned compromises. It provides serializability, allows long-running read-write queries, and works with stored procedures as well as sequences of client requests. As the performance comparison in Table 5.2 shows, this mechanism does not introduce a significant overhead. Our throughput of 13.6 million transactions per second (5.9 million SNOT per second) is among the highest reported.

5.8 Summary

In this chapter, we provided a new baseline for distributed concurrency control at large scale. To that end, we implemented a conventional lock table and Two-Phase Commit protocol using state-of-the-art communication primitives as can be found in modern HPC systems. Our implementation relies on low-latency communication mechanisms offered by high-performance networks. In order to hide the complexity arising from large-scale distribution, we use MPI as our communication layer. We evaluated our prototype of this concurrency control mechanism on over a hundred physical machines and thousands of cores using a combination of TPC-C and synthetic workloads.

Even without any special architecture or optimizations, our distributed lock table can support well over 800 million lock operations per second on 5040 cores. Thus, this work shows that conventional Two-Phase Locking and Two-Phase Commit are a viable solution to implement the highest levels of transaction isolation, namely *serializability*, while also being scalable. Furthermore, this approach does not impose any restrictions on the workload in terms of lock modes supported, structure of the transactions, deterministic behavior, or support for long-running transactions. By using MPI to implement a low-latency message passing mechanism, we show that our implementation is able to take advantage of the scale-out architecture used in our evaluation. Provided that there is little contention, local as well as remote locks can be acquired within a few microseconds.

We have demonstrated that the proposed concurrency control mechanism can scale to thousands of cores, reaching a throughput 13.6 million TPC-C transactions (5.9 million *new order* transactions) per second. These numbers can be used as a new baseline to evaluate large-scale transaction processing systems and are competitive with all results published so far. Since many database systems use a conventional lock table, our findings can also be used to scale out existing systems requiring a low-overhead, distributed concurrency control mechanism that can sustain a high throughput and take advantage of the parallelism offered by large distributed systems.

6

Outlook on Future Networks

The performance increase of future networks needs to be kept in line with other system-level performance gains. To that end, most high-performance network manufacturers offer a roadmap that provides an outline of the progression of interconnect technologies. In the case of InfiniBand, the roadmap details the bandwidth of future iterations of the InfiniBand technology for different port widths. Figure 6.1 shows that the first implementations that reach 600 Gbits per second (12× HDR) are expected to appear on the market by 2019 and a 1.2 Tbits per second version (12× NDR) will be released in late 2020. This represents a tremendous increase in bandwidth compared to the 56 Gbits per second (4× FDR) used in the experimental evaluation in Chapter 3. Already with current state-of-the-art networks, we observe that the performance of distributed joins is in a similar ballpark as centralized algorithms. By installing multiple networks cards per machine (e.g., one for each processor), it is conceivable that in future systems RDMA-capable inter-machine networks outperform the internal interconnect in terms of bandwidth.

The trend towards a higher bandwidth has a lot of implications on the design of database systems and algorithms [BAH17]. Analytical workloads can benefit from this increase as the cost of data movement is a crucial factor. For example, the radix hash join is

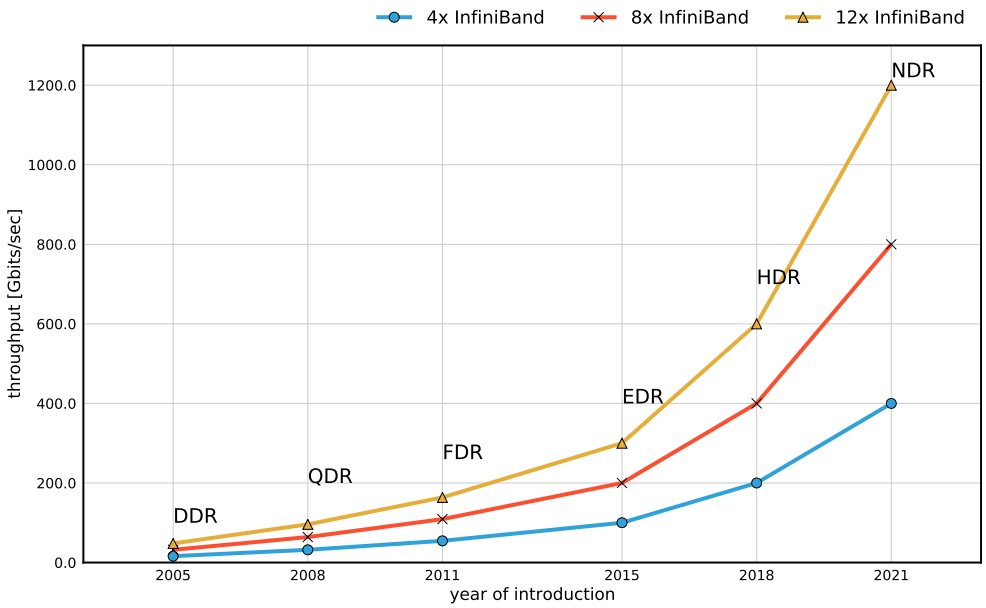


Figure 6.1: InfiniBand roadmap.

bound by the network bandwidth on the QDR cluster and can be accelerated by using the faster FDR network. In the evaluation of the performance model in Section 3.5, the optimal number of processor cores per machine is seven on the FDR network, meaning that the configuration used in the experiments (i.e., 8 cores per machine) is close to optimal. In this dissertation, we looked at the joins in isolation having the whole machine and network at their disposal. To saturate a faster network link, more cores would need to be dedicated to the join operation which poses its own set of coordination challenges as seen in Section 4.3. Therefore, instead of accelerating a single query, the additional bandwidth of future networks will be useful to run several joins or queries concurrently. The deployment of multiple concurrent queries is a difficult task on multi-core, multi-socket servers [GARH14] that requires a close integration with the operating system [GSS+13, GZAR16, Gic16] and is likely to be challenging in scale-out architectures.

For transaction processing, latency is the dominant performance factor. Current network technologies offer a latency in the order of a single-digit microsecond, a period of time that is sometimes difficult to interleave with useful processing [BMPR17]. In addition, many operations that database systems perform on a regular basis (e.g., appending data to a queue, traversing an index structure, locking items in multi-level granularity locking)

require multiple round-trips when using one-sided operations. This small programming surface of RDMA hardware is a major limitation of modern networks. Most network cards only implement message passing (*send* and *receive*), one-sided (*read* and *write*), and single-word atomic operations (*fetch-and-add* and *compare-and-swap*) efficiently. The interface that is exposed to software developers reflects the same abstractions. In the following sections, we will have a detailed look at the limitations of common interfaces that we identified during the development of the algorithms presented in this dissertation. We argue that future networks need to offer more sophisticated instructions to overcome some of these limitations. Finally, we will discuss how to implement this functionality in future networking hardware.

6.1 Current and Future Network Interfaces

For the algorithms evaluated in this dissertation, we used two network interfaces: the Verbs interface for the rack-scale join experiments and MPI for all experiments conducted on the supercomputers. These two interfaces differ significantly in their design, abstractions, and ease of use. While the Verbs interface is a low-level interface, MPI provides many high-level functions. Both interfaces have their own set of advantages and disadvantages when it comes to distributed data processing.

6.1.1 A Critique of RDMA Verbs

The RDMA Verbs interface is an abstract interface to RDMA-enabled network cards. It provides a set of calls for queue and memory management. Being a low-level interface is both its biggest strength and weakness. While any application can create its set of connections (i.e., queue pairs) and buffers (i.e., memory regions) as it is most desirable, the interface places a significant burden on the application developer to implement the required functionality. The interface was designed with no particular use-case in mind and does not offer any high-level functions targeting a particular purpose. High-level operations need to be implemented by the database system directly. These primitives would also need to

be tuned to the underlying interconnect technology as different network implementations have vastly different performance characteristics.

In addition, the interface has limited security features. Applications that know the address of a remote buffer and have access to a queue pair within the same protection domain can arbitrarily read or write data without any additional security checks.

Most network cards have a limited cache in which they hold queue pair and buffer address translation information. As a consequence, applications need to keep the number of active connections at a minimum. One way to address this issue in the context of a data center or cloud computing provider is to share these resources among multiple applications. When using RDMA Verbs, it is currently not possible to use existing connections, buffers, and queues from within multiple applications in a safe way. To overcome these limitations, new abstractions have been proposed, some of which suggest using an indirection tier within the operating system to virtualize and manage the RDMA Verbs interface [TZ17].

6.1.2 A Critique of MPI

Although MPI offers many advanced communication features, using it in the context of a database system comes with its own set of challenges. For example, the degree of parallelism of an MPI application can be specified at start-up time by indicating the desired number of MPI processes. Processes are identified by an integer number, the *rank*, making them the fundamental unit of parallelism. Addressing processes directly instead of generic endpoints is challenging for database systems and algorithms that are highly multi-threaded. With the current standard, it is not possible to address a specific thread. As a result, the implementations of the join algorithms proposed in this dissertation had to be significantly changed before MPI could be used (see Section 4.2).

Many MPI operations are implemented as collective operations, meaning that every process of the communication group, i.e., the context in which the collective call is executed, has to participate in the call. Common examples of collective operations are reduce, gather, scatter, and broadcast operations, but the list also includes management operations

such as window allocation and deallocation, i.e., when a process needs to allocate a new window, every other process that might potentially access its content has to participate in the collective operation, regardless whether it will ever access the window at a later point in time or not. In some MPI implementations, these operations represent a point of synchronization in the execution. To avoid synchronization, in the case of our join implementations, all windows are allocated in an early phase of the algorithm.

In a database system, some messages are latency critical while others are not. For example, control messages and messages belonging to a transaction are usually small, while analytical queries might require large data exchanges. In this dissertation, we look at join processing and transaction processing in isolation. However, in a system handling a hybrid workload of transactions and queries, it is crucial to avoid that small, latency-critical messages are scheduled behind large bulk transfers. The current version of the MPI standard does not enable the developer to prioritize specific operations or flows.

Typical HPC applications such as scientific applications, physics simulations, or weather predictions, are started with a specific lifespan in mind. Their execution time lasts from a few minutes to several hours. This is a different model than the one used for building database systems. Ideally, a database system runs for an unlimited time. During its execution it is therefore likely that parts of the system fail and new components join. Failures have to be contained and recovered from. However, MPI does not have the necessary functionality to support adding and removing processes from a running application and to notify the application in case of failures.

Despite these disadvantages, using standard communication libraries such as MPI instead of hand-tuned code makes the application code portable. Given that MPI is an interface description, an MPI application can be linked against many different library implementations, each tailored to a specific network. Using operations that have a rich semantic meaning makes it possible to reason about the intentions of the application and enables the developer of the communication abstraction to choose the right set of network primitives and hardware-specific features in order to implement high-level operations efficiently.

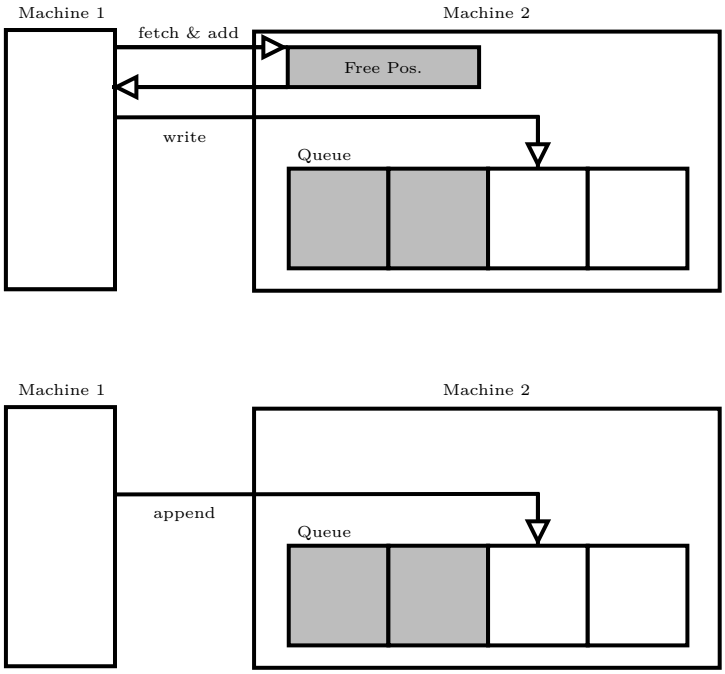


Figure 6.2: Append operation adding data to a partially filled remote buffer or remote queue.

6.1.3 Beyond Read and Write Operations

MPI as well as RDMA Verbs offer a limited set of one-sided RMA operations. In a nutshell, both interfaces are limited to read and write (i.e., `MPI_Get` and `MPI_Put`) operations that transfer data between local and remote memory regions. Both interfaces also offer simple atomic operations. In many network implementations, such as InfiniBand, only compare-and-swap and fetch-and-add are offered and accelerated by the hardware. To facilitate the development of future systems, this set of operations needs to be extended. The primitives offered by the network should be designed to meet the needs of a variety of data processing applications. This section of the dissertation lists several important operations that manipulate, transform, and filter data while it is moving through the network.

Advanced One-Sided Operation

Similar to previous work, we found that one-sided operations are often limited in their flexibility and ease of use [Hoe16, DNC17]. The operations presented in this section will be useful in future networks to eliminate several round trips that are needed with current network interfaces to accomplish specific tasks. We envision that these operations will be implemented directly in hardware. Unlike a Remote Procedure Call (Rpc), these operations will not be able to invoke arbitrary program functions, initiate system calls, or issue additional network requests. Unlike a network instruction set architecture (Nisa), these methods are not intended to support user-defined programs.

***Remote Append*:** During the hashing and sorting operation of the join algorithms, the processes use one-sided Rma operations. Using one-sided write operations reduces the amount of synchronization in these phases, as the target process does not need to be actively involved in the communication in order for the transfer to complete. However, the benefits of Rma do not come for free and require up-front investment in the form of a histogram computation phase. Although computing and exchanging these histograms can be done with great efficiency, this operation can be avoided in future Rma systems. For example, a remote append operation, which would sequentially populate a buffer with the content coming from different Rma operations would significantly simplify the design of the algorithms and speed up the join implementations evaluated in this dissertation. This operation would also be useful for any system manipulating remote, queue-like data structures. For example, with the current generations of networks, adding an element to a queue requires at least two round-trips: First, the end of the queue needs to be identified and a slot for writing the data needs to be reserved. Afterwards, the actual content can be added to the queue. The proposed *append* operation would enable the system to perform the same operation in a single request as seen in Figure 6.2.

***Remote Selection*:** Snapshot isolation is a concurrency control mechanism used in many database engines. The advantage of this approach is that queries, i.e., read-only operations, do not need to be synchronized with other concurrent transactions. Locks are only needed to resolve write-write conflicts at commit time. The advantages of snapshot

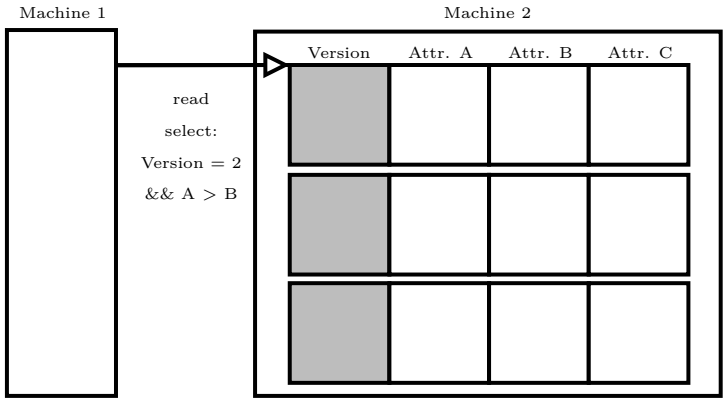


Figure 6.3: Selection of items in a remote buffer based on specific attributes and version numbers.

isolation can only be leveraged through careful memory management and version control. When using one-sided operations to access data, following pointers to fetch a specific version of a record can cause a significant amount of random remote memory accesses and requires multiple round-trips. Instead, a remote selection operator could be used to only return data belonging to the same version, thus enabling consistent reads. In general, a remote section operator is not only useful in the context of snapshot isolation, but can also be used to filter data before it is transmitted over the network, thus saving valuable network bandwidth as seen in Figure 6.3.

***Remote Aggregation*:** Similar to the remote selection, an aggregation algorithm can save significant amounts of bandwidth. While data is processed by the remote network card, it can be aggregated instead of being transmitted in its entirety. A remote aggregation operator can include simple aggregation operations (e.g., sum, max, min, avg). With these primitives many meta-data information, such as histograms, can be computed directly by the networking hardware with great efficiency. These operations do not need to be implemented inside network cards, but can also be placed inside routers and switches in order to combine data coming from multiple streams and locations.

Type-and Schema-Aware Operations

Traditional database systems operate primarily on structured data. Pushing down the schema information to the network enables novel in-network processing applications and operations that take the data layout into consideration.

***Data Transformations*:** BatchDB [MGBA17, Mak17] is a database engine designed to reduce the interference caused by hybrid workloads running in the same system while maintaining strict guarantees with respect to performance, data freshness, consistency, and elasticity. To that end, BatchDB uses multiple workload-specific replicas. These copies of the data can either be co-located on the same machine or distributed across multiple compute nodes. Transactions operate exclusively on a write-optimized version of the data, i.e., the primary copy. Updates to this component are propagated to the satellite replicas. The replicas are responsible for converting the data into their respective format, applying the updates, and signaling a management component that the updates have been applied. In order to meet strict data freshness requirements, low-latency communication is essential. To ensure that the system can be extended with future workload-specific formats, it is the responsibility of each replica to convert the change set, which is sent out in row-store format, to its internal representation. Given that this data transformation typically involves a significant amount of processing and copying, doing this step in software impacts the performance of the satellite component. To that end, we propose that future networking technology enables the destination node to push down simple rewrite rules to the network card. The networking hardware should be able to change the data layout while writing the incoming updates to main memory. Transforming data while it is transmitted is a general mechanism which is useful to any system that requires many different data formats during its processing.

***Compression*:** A special case of data transformation is compression and de-compression of data. Many databases store their data in compressed format and on-the-fly compression and de-compression could be done by the network card as data is read from or written back to remote memory, thus eliminating the CPU overhead of compression, avoiding unnecessary copy operations, and reducing the overall storage requirements. Such a functionality

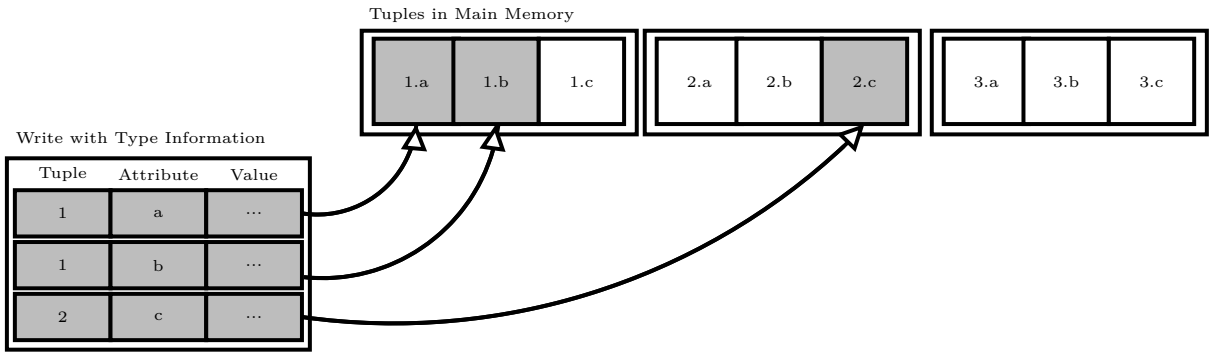


Figure 6.4: Translation of a *write* command with type information into attribute-level accesses by the remote network card.

is not only important when accessing main memory, but can also be used in systems where data is directly accessed from persistent storage, e.g., RDMA over Fabrics.

***Attribute-Level Access*:** Many analytical queries are only interested in specific attributes of a tuple. Having data stored in column-major format is useful in such cases, as the operator only needs to access the specific memory regions where the desired attributes are stored. In a row-major format, data belonging to the same tuple is stored consecutively in memory. Although the majority of networks offers gather-scatter elements, in large databases, it is not feasible to create one gather-scatter element for each individual tuple. In addition, variable-length data often prevents computing the necessary offsets at the origin of the request. Specifying a data layout and the set of attributes that need to be read would enable the network card to determine a generic access pattern and only transmit the desired attributes. This operation corresponds to a remote projection. When writing data, this mechanism makes it possible to consolidate different attributes of different tuples in a single operation. For example, BatchDB forwards attribute-level change sets. Since the updates need to be propagated to all replicas, having sufficient bandwidth on the machine hosting the primary copy is important. To keep the bandwidth require-

ments at a minimum, the transactional component only forwards the attributes of the tuples that have changed. With precise type information, the network card could directly update the corresponding attributes for each tuple individually as illustrated in Figure 6.4.

Conditional Operations

The only conditional operation that is offered by many state-of-the-art networks is the remote atomic compare-and-swap operation. Conditional operations enable the developer to create simple *if-then-else* operations. For an operation with condition check, the remote network card will first evaluate if the remote data is in a specific state before applying the operation, thus eliminating a round trip over the network and reducing the need for running expensive synchronization or agreement protocols.

Conditional operations can be used to significantly accelerate the locking system proposed in Chapter 5 of this dissertation. As described in Section 5.2, each lock consists of a request queue and a granted group. A conditional operation is an efficient way to first check the status of the lock and the queue. If the request can be granted (*if*-branch) the lock counter is being incremented (using a fetch-and-add operation), otherwise (*else*-branch), the request is added to the queue using the previously proposed *append* operation. All these operations would require a single round-trip without involving the remote processor.

6.2 Current and Future Network Cards

Recent work on Field Programmable Gate Arrays (FPGAs) has analyzed how common database operators can be accelerated. In combination with advanced network cards, FPGAs can be used for bump-in-the-wire processing as well as to implement many of the advanced network primitives described in Section 6.1.3. Furthermore, routers and switches are additional places where advanced processing capabilities can be integrated into the network infrastructure.

Research on database hardware accelerators has shown that database tables can be analyzed as they are retrieved from persistent storage and histograms can be computed at virtually no extra performance cost. The same type of histogram computation can prove useful for running distributed join operators that, as explained in Section 3.2, rely on histogram data to distribute the data using one-sided RMA operations. Furthermore, several skew handling techniques mentioned in Section 3.6 use approximate or exact histograms to detect frequent items. Using the techniques described by István et al. [IWA14], these histograms can be computed on the fly as the data flows through the network.

Furthermore, future network cards could be used to offload many data redistribution operations, e.g., the radix partitioning of the hash join can be implemented in hardware. Kara et al. [KGA17] accelerate the hash join by offloading the partitioning operation to an FPGA. This approach allows the join operator to use larger fan-outs than what is typically possible on modern processors. As explained in Section 4.2, a large fan-out is required when operating at massive scale in order to generate enough data partitions, at least one for each core. Not only partitioned hash joins can be accelerated through the use of FPGAs. FPGA-based acceleration of the sort operator, the integral part of the sort-merge join and many other database operators, has gained significant attention [KT11, MTA12]. Casper et al. [CO14] propose an implementation of a sort-merge join on FPGAs. We postulate that future network cards, or combinations of FPGAs and network cards, can be used to offload significant parts of the join algorithms proposed in this dissertation.

Direct implementations of relational operators in hardware will not be limited to joins. Many database operations (e.g., selection, aggregation, projection, etc.) can be performed by a specialized ASIC or FPGA. For example, the matching of regular expressions by a run-time parametrizable regex operator can be used to implement an in-network selection operator for specific string expressions [ISA16, BWT18]. In addition to individual operators, systems and architectures are being proposed for accelerating entire or parts of query pipelines and data streams using FPGAs [MTA09, MTA10, STM+15, SIOA17].

To be able to offload a variety applications, abstract machine models of offload-enabled network architectures have been proposed [GJUH16] and low-latency network stack imple-

mentations are available for FPGAs [SIA16]. Given the trend towards processing data as it moves through the network, manufacturers are increasingly offering programmable high-performance network cards. For example, Mellanox BlueField [Mel18a] is highly integrated system-on-a-chip offering a multi-core processor, NVMe storage, and RDMA networking functionality. Mellanox Innova-2 [Mel18b] is an InfiniBand and Ethernet network card with an on-board FPGA. Enzian is a research computer that uses a combination of off-the-self components and reconfigurable hardware to explore the design space of future systems [ETH18].

7

Conclusions

Modern database systems are challenged to process ever-increasing volumes of data. Given the limited resources of a single machine, distributed systems are required. However, to utilize the full potential of modern compute clusters, efficient data movement is critical for both query and transaction processing. Recently, high-speed interconnects with RDMA support have become economically viable beyond HPC systems and are being introduced in many clusters and data centers. These networks significantly reduce the costs of communication by offering high bandwidth and low latency. However, these performance advantages do not come for free and can only be leveraged through careful design of the database algorithms.

The dissertation took several important steps towards understanding the impact fast interconnects have on large-scale query and transaction processing. This work studied hardware-conscious, main-memory, relational joins – one of the most complex and communication-intensive database operations – on modern compute clusters in which the machines are connected by a state-of-the-art, high-performance network. We discussed two widely-used approaches for implementing join: hash- and sort-based algorithms. Our findings show that algorithms need to be optimized not only for modern processor architec-

tures but also for high-speed networks. In particular, these algorithms need to interleave computation and communication, efficiently manage RDMA communication buffers, and lay out their data structures in such a way that one-sided direct data access and placement mechanisms can be used. Using these optimizations, distributed join algorithms achieve the same performance as their single-machine counterparts on the same number of cores while having the added benefit that they can scale to hundreds of machines.

Although the latency of accessing remote data is still higher than that of a local memory access, modern networks and communication frameworks make it possible to reach every node in the system within a few microseconds. In this document, we showed that this latency is small enough to implement a high-performance, lock-based concurrency control mechanism that can be used to implement the highest levels of transaction isolation – namely *read committed* and *serializable* – without imposing any restrictions on the types of workloads the database system can support.

We evaluated all our algorithms not only on a rack-scale cluster but also on large supercomputers that provided us with thousands of cores and a fast network. This dissertation is one of the first research projects that combined traditional database algorithms with the technologies used in HPC systems. Both communities can benefit from this interaction. HPC researchers have a lot of experience in creating scalable applications that can run on thousands of processor cores. On the other hand, many large scientific computations can be broken down into basic relational operators (e.g., selection, filtering, aggregation, join) for which efficient algorithms exist. The work presented in this dissertation is one of the first attempts in combining both aspects of computer science and the experiments conducted on the supercomputers are one of the largest deployments of traditional database algorithms to-date.

In Chapter 3, we analyzed distributed joins algorithms on a cluster connected by InfiniBand QDR and FDR. The performance of the distributed algorithms is comparable to that of optimized, single-machine implementations. The two algorithms place a different load on the network. While the radix hash join with its all-to-all data exchange can easily saturate the QDR network, the more compute-intensive sorting operation of the

sort-merge join places an even load on the network. The importance of the compute-to-communication ratio become visible in large deployments in Chapter 4. While the radix hash join can out-perform the sort-based approach in a rack-scale cluster, on 4096 cores, both algorithms achieve comparable performance. Despite this fact, the experimental results generally show that both algorithms scale well with the number of cores and are able to reach a very high throughput of 48.7 billion input tuples per second. Many of our findings are useful beyond the acceleration of joins. For example, the presented communication mechanisms are also applicable to other database algorithms such as large-scale aggregation operations. In Chapter 5, we investigated the low-latency aspect of modern networks in the context of a transactional database system with a lock-based concurrency control mechanism. We showed that, given the latency of modern interconnects, Two-Phase Locking (2PL) and Two-Phase Commit (2PC) are viable solutions for implementing a large-scale transaction processing system. Traditional workloads, such as TPC-C, can be scaled to thousands of cores and millions of transactions can be processed each second.

In order to gain further insights into the behavior of the algorithms, we presented detailed performance models for all algorithms. As described in Section 2.1.1, performance modeling is an important aspect when developing applications for large HPC machines. These models provide us with tight bounds on the execution of each individual phase of the algorithms and enabled us to quickly identify inefficiencies in the implementations.

Throughout this dissertation, we explored different communication interfaces: (i) the low-level RDMA Verbs interface for the rack-scale systems, and (ii) the high-level interface offered by MPI for large-scale experiments on the HPC machine. In recent work, the database community has been experimenting with MPI in various contexts. Our conclusion is that both interfaces come with their own set of challenges. With RDMA alone, completing the most basic remote operations often requires multiple round-trips over the network or up-front processing (e.g., histogram computation). Therefore, a natural question to ask is which functionality future communication abstractions need to expose. In Chapter 6, we provided a set of instructions that manufacturers and researchers can use as a guideline to steer the development of novel interconnects optimized for communication- and data-intensive applications such as database systems.

7.1 Research Outlook

The work presented in this dissertation opens up several interesting directions for future research in the area of database systems, high-performance computing, and future networking technology.

***Joins in Cloud Environments*:** While high-performance networks could only be found in HPC systems and special niche solutions (e.g., database appliances) at the beginning of my doctoral studies, today, these networks are being deployed in many data centers. At the time of writing this dissertation, the first cloud computing providers started offering virtual machines with RDMA-capable networks on an hourly basis. Cloud computing has significantly changed the economic aspect of computing and hardware provisioning. It is therefore an interesting direction for future research to study how the proposed algorithms behave inside a cloud computing environment and what the exact cost/performance tradeoffs are. Having affordable access to a large number of machines with an advanced network means that conducting large-scale experiments like the ones presented in this dissertation will no longer require special access to a national supercomputer.

***RDMA in a Query Pipeline*:** In this dissertation, we used RDMA and RMA to accelerate join operations. It is important to note that the majority of our findings is not limited to joins or individual database operators. Apart from accelerating other operations (e.g., aggregation), one should also explore the use of RDMA between operators, i.e., inside a query pipeline. In this dissertation, we assumed a column-store layout and focused on processing narrow tuples consisting of a join key and a record identifier, without materializing the final result. In many column-store database systems, this materialization step is usually performed in one of the final processing stages in order to fetch as little data as possible. However, in queries that analyze and return large datasets, this step can potentially be as expensive as computing the query. It is likely that materialization can also benefit from many of the optimization techniques discussed in this dissertation. Therefore, the usage of RDMA between operators in general, and in the result materialization step in particular, represents a challenging direction for future investigations.

***A Database for Supercomputers*:** This work is one of the first building blocks towards developing a database that can run at the scale of a supercomputer. Our results show that even the most complex operators and concurrency control mechanisms can scale to thousands of cores. Given that supercomputers often require significant investments at a national level, the users of these systems are reluctant to use non-optimized, generic data processing frameworks as they often exhibit sub-optimal performance. Therefore, in combination with the query processing pipeline mentioned above, creating such a hardware-conscious distributed database system for HPC computers would be useful for many large-scale scientific applications that often re-implement many common database operators from scratch.

***Concurrent Data Processing*:** In this dissertation, we analyzed query and transaction processing in isolation. Furthermore, we focused on executing a single join operator at a time. However, in many database systems, several queries are executed concurrently. Until now, the limited bandwidth of commodity networks represented a major bottleneck, especially in cases where several queries competed for the same network resources. Modern high-performance networks significantly reduced the impact this limitation. Our results have also shown that many cores are required for a join to fully saturate the bandwidth of a high-performance network. As networks become faster, one has therefore two options: One can either dedicate more cores to a single query (i.e., to a single join operator), or make use of the higher bandwidth to run two join operations concurrently. Both options are worth an in-depth exploration.

***Implementation of New Communication Primitives*:** In Chapter 6, we presented several new communication primitives that future networks should support. At the same time, FPGAs are being increasingly adopted as the research platform of choice to accelerate database operations. Since many manufacturers offer combined network card and FPGA solutions, the implementations and evaluation of new communication abstractions represents an interesting direction for future work. We estimate that a significant part of the algorithms presented in this dissertation can be offloaded to and accelerated by specialized network cards and switches.

7.2 Concluding Remarks

This dissertation seeks to further the state-of-the-art in distributed query and transaction processing for parallel in-memory database systems. We explored how join algorithms and lock-based concurrency control mechanisms behave when high-bandwidth, low-latency networks are used. We evaluated these algorithms on rack-scale clusters with a similar architecture to that of commercial database appliances as well as on large HPC clusters. By combining traditional database algorithms with the technologies used in HPC systems, we were able to conduct experiments on thousands of processor cores, a scale usually reserved to massively parallel scientific applications or large map-reduce batch jobs. Operating at large scale requires careful process orchestration and efficient communication. Our results show that this setup poses several challenges when scaling out database systems. For example, the algorithms need to keep track of data movement between the compute nodes, use many different communication primitives offered by the underlying hardware, and interleave the communication and processing. At large scale, the performance of the algorithms is dependent on having a good communication abstraction and future networks are likely to significantly expand the set of instructions they expose to the applications.

In the future, high-performance networks with RDMA support will offer an even higher bandwidth and a lower latency, further reducing the costs of communication. In addition, these RDMA-capable interconnects will become omnipresent. For example, cloud computing providers are starting to extend their offerings in this direction. In light of these trends, this dissertation re-evaluates multiple database algorithms used in query and transaction processing, and proposes novel techniques and design principles that enable future distributed database systems to take full advantage of this new generation of networks.

List of Figures

List of Tables

Bibliography

[AAE94] G. Alonso, D. Agrawal, and A. El Abbadi. "Reducing Recovery Constraints on Locking based Protocols." In *Proceedings of the 13th Symposium on Principles of Database Systems*, pp. 129–138. 1994.

[ABF+10] L. Adhianto, S. Banerjee, M. W. Fagan, M. Krentel, G. Marin, J. M. Mellor-Crummey, and N. R. Tallent. "HPCTOOLKIT: tools for performance analysis of optimized parallel programs." *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, 685–701, 2010.

[AKN12] M. Albutiu, A. Kemper, and T. Neumann. "Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems." *Proceedings of the VLDB Endowment*, vol. 5, no. 10, 1064–1075, 2012.

[AKR12] B. Alverson, E. F. L. Kaplan, and D. Roweth. "Cray XC Series Network." *Cray Inc. Whitepaper*, 2012.

[Apa18a] Apache Foundation. "Apache Cassandra." http://cassandra.apache.org/, 2018. Online, accessed July 2018.

[Apa18b] Apache Foundation. "Apache CouchDB." http://couchdb.apache.org/, 2018. Online, accessed July 2018.

[Apa18c] Apache Foundation. "Apache Flink." http://flink.apache.org/, 2018. Online, accessed July 2018.

[Apa18d] Apache Foundation. "Apache H-Base." http://hbase.apache.org/, 2018. Online, accessed July 2018.

[Apa18e] Apache Foundation. "Apache Hadoop." http://hadoop.apache.org/, 2018. Online, accessed July 2018.

[Apa18f] Apache Foundation. "Apache Spark." http://spark.apache.org/, 2018. Online, accessed July 2018.

[AU11] F. N. Afrati and J. D. Ullman. "Optimizing Multiway Joins in a Map-Reduce Environment." *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 9, 1282–1298, 2011.

[AV06] S. R. Alam and J. S. Vetter. "A framework to develop symbolic performance models of parallel applications." In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*. 2006.

[BAH17] C. Barthels, G. Alonso, and T. Hoefler. "Designing Databases for Future High-Performance Networks." *IEEE Data Engineering Bulletin*, vol. 40, no. 1, 15–26, 2017.

[Bal14] C. Balkesen. "In-memory parallel join processing on multi-core processors." Ph.D. thesis, ETH Zurich, 2014.

[Bat68] K. E. Batcher. "Sorting Networks and Their Applications." In *Proceedings of the 1986 American Federation of Information Processing Societies AFIPS Conference*, pp. 307–314. 1968.

[BATÖ13] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. "Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited." *Proceedings of the VLDB Endowment*, vol. 7, no. 1, 85–96, 2013.

[BBD+09] A. Baumann, P. Barham, P. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. "The multikernel: a new OS architecture for scalable multicore systems." In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, pp. 29–44. 2009.

[BCF+95] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. "Myrinet: A Gigabit-per-Second Local Area Network." *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, vol. 15, no. 1, 29–36, 1995.

[BCG+16] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. "The End of Slow Networks: It's Time for a Redesign." *Proceedings of the VLDB Endowment*, vol. 9, no. 7, 528–539, 2016.

[BEG+15] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang. "Analytics in Motion: High Performance Event-Processing AND Real-Time Analytics in the Same Database." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 251–264. 2015.

[BH14a] M. Besta and T. Hoefler. "Slim Fly: A Cost Effective Low-Diameter Network Topology." In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 348–359. 2014.

[BH14b] A. Bhattacharyya and T. Hoefler. "PEMOGEN: Automatic Adaptive Performance Modeling During Program Runtime." In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 393–404. 2014.

[BH15] R. Belli and T. Hoefler. "Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization." In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium*, pp. 871–881. 2015.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman, 1987.

[BKG+18] C. Balkesen, N. Kunal, G. Giannikis, P. Fender, S. Sundara, F. Schmidt, J. Wen, S. R. Agrawal, A. Raghavan, V. Varadarajan, A. Viswanathan,

B. Chandrasekaran, S. Idicula, N. Agarwal, and E. Sedlar. "RAPID: In-Memory Analytical Query Processing Engine with Extreme Performance per Watt." In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, pp. 1407–1419. 2018.

[BLAK15] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. "Rack-Scale In-Memory Join Processing using RDMA." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1463–1475. 2015.

[BLMR02] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen. "Portals 3.0: Protocol Building Blocks for Low Overhead Communication." In *Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium*. 2002.

[BLP11] S. Blanas, Y. Li, and J. M. Patel. "Design and evaluation of main memory hash join algorithms for multi-core CPUs." In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pp. 37–48. 2011.

[BMAH] C. Barthels, I. Müller, G. Alonso, and T. Hoefler. "Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores." [Under submission].

[BMPR17] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan. "Attack of the killer microseconds." *Commununications of the ACM*, vol. 60, no. 4, 48–54, 2017.

[BMS+17] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. "Distributed Join Algorithms on Thousands of Cores." *Proceedings of the VLDB Endowment*, vol. 10, no. 5, 517–528, 2017.

[BTAÖ13] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware." In *Proceedings of the 29th IEEE International Conference on Data Engineering*, pp. 362–373. 2013.

[BTAÖ15] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. "Main-Memory Hash Joins on Modern Processor Architectures." *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, 1754–1766, 2015.

[Bur06] M. Burrows. "The Chubby Lock Service for Loosely-Coupled Distributed Systems." In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 335–350. 2006.

[BWT18] A. Becher, S. Wildermann, and J. Teich. "Optimistic regular expression matching on FPGAs for near-data processing." In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, pp. 4:1–4:3. 2018.

[CDE+12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. "Spanner: Google's Globally-Distributed Database." In *Proccedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pp. 261–264. 2012.

[CDG+06] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. "Bigtable: A Distributed Storage System for Structured Data." In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 205–218. 2006.

[CDM+05] C. Coarfa, Y. Dotsenko, J. M. Mellor-Crummey, F. Cantonnet, T. A. El-Ghazawi, A. Mohanti, Y. Yao, and D. G. Chavarría-Miranda. "An evaluation of global address space languages: co-array fortran and unified parallel C." In *Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 36–47. 2005.

[CEH+11] D. Chen, N. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. D. Steinmacher-Burow, and J. J. Parker.

"The IBM Blue Gene/Q interconnection network and message unit." In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 26:1–26:10. 2011.

[CIR+16] A. Costea, A. Ionescu, B. Raducanu, M. Switakowski, C. Bârca, J. Sompolski, A. Luszczak, M. Szafranski, G. de Nijs, and P. A. Boncz. "VectorH: Taking SQL-on-Hadoop to the Next Level." In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pp. 1105–1117. 2016.

[CKJE14] W. Culhane, K. Kogan, C. Jayalath, and P. Eugster. "LOOM: Optimal Aggregation Overlays for In-Memory Big Data Processing." In *6th USENIX Workshop on Hot Topics in Cloud Computing*. 2014.

[CKJE15] W. Culhane, K. Kogan, C. Jayalath, and P. Eugster. "Optimal communication structures for big data aggregation." In *Proceedings of the 2015 IEEE Conference on Computer Communications*, pp. 1643–1651. 2015.

[CO14] J. Casper and K. Olukotun. "Hardware acceleration of database operations." In *Proceedings of the 2014 ACM SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 151–160. 2014.

[Cra18] Cray Supercomputer Company. "XC Series Supercomputers." http://www.cray.com/products/computing/xc-series/, 2018. Online, accessed July 2018.

[DFI+13] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. "Hekaton: SQL server's memory-optimized OLTP engine." In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1243–1254. 2013.

[DG04] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation*, pp. 137–150. 2004.

[DGG+86] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. "GAMMA - A High Performance Dataflow Database Machine." In *Proceedings of the 12th International Conference on Very Large Data Bases*, pp. 228–237. 1986.

[DGS+90] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. "The Gamma Database Machine Project." *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 1, 44–62, 1990.

[DLHV16] A. M. Dan, P. Lam, T. Hoefler, and M. Vechev. "Modeling and Analysis of Remote Memory Access Programming." In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 129–144. 2016.

[DNC17] A. Dragojevic, D. Narayanan, and M. Castro. "RDMA Reads: To Use or Not to Use?" *IEEE Data Engineering Bulletin*, vol. 40, no. 1, 3–14, 2017.

[DNCH14] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. "FaRM: Fast Remote Memory." In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, pp. 401–414. 2014.

[DNN+15] A. Dragojevic, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. "No compromises: distributed transactions with consistency, availability, and performance." In *Proceedings of the 25th ACM SIGOPS Symposium on Operating Systems Principles*, pp. 54–70. 2015.

[DNS91] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. "Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting." In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems*, pp. 280–291. 1991.

[ETH18] ETH Systems Group. "The Enzian Research Computer." http://enzian.systems/, 2018. Online, accessed October 2018.

[FA09] P. W. Frey and G. Alonso. "Minimizing the Hidden Cost of RDMA." In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, pp. 553–560. 2009.

[Fac18] Facebook Inc. "RocksDB." http://rocksdb.org/, 2018. Online, accessed July 2018.

[FGKT09] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. "Spinning relations: high-speed networks for distributed join processing." In *Proceedings of the 5th International Workshop on Data Management on New Hardware*, pp. 27–33. 2009.

[FGKT10] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. "A Spinning Join That Does Not Get Dizzy." In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, pp. 283–292. 2010.

[FM70] W. D. Frazer and A. C. McKellar. "Samplesort: A Sampling Approach to Minimal Storage Tree Sorting." *Journal of the ACM*, vol. 17, no. 3, 496–507, 1970.

[FML+12] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. "The SAP HANA Database – An Architecture Overview." *IEEE Data Engineering Bulletin*, vol. 35, no. 1, 28–33, 2012.

[Fre10] P. W. Frey. "Zero-copy network communication: An applicability study of iWARP beyond micro benchmarks." Ph.D. thesis, ETH Zurich, 2010.

[GARH14] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. "Deployment of Query Plans on Multicores." *Proceedings of the VLDB Endowment*, vol. 8, no. 3, 233–244, 2014.

[GBH13] R. Gerstenberger, M. Besta, and T. Hoefler. "Enabling highly-scalable remote memory access programming with MPI-3 one sided." In *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 53:1–53:12. 2013.

[GHS+15] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres. "A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency." In *Proceedings of the 23rd IEEE Annual Symposium on High-Performance Interconnects*, pp. 34–39. 2015.

[GHTL14] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press, 2014.

[Gic16] J. Giceva. "Database/Operating System Co-Design." Ph.D. thesis, ETH Zurich, 2016.

[GJUH16] S. D. Girolamo, P. Jolivet, K. D. Underwood, and T. Hoefler. "Exploiting Offload-Enabled Network Interfaces." *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, vol. 36, no. 4, 6–17, 2016.

[GK10] R. Goncalves and M. L. Kersten. "The Data Cyclotron query processing scheme." In *Proceedings of 13th International Conference on Extending Database Technology*, pp. 75–86. 2010.

[GKP+10] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. "HYRISE - A Main Memory Hybrid Storage Engine." *Proceedings of the VLDB Endowment*, vol. 4, no. 2, 105–116, 2010.

[Gon13] R. P. Goncalves. "The Data Cyclotron: Juggling data and queries for a data warehouse audience." Ph.D. thesis, University of Amsterdam, 2013.

[GR92] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 1992.

[GRW+18] A. Gittens, K. Rothauge, S. Wang, M. W. Mahoney, J. Kottalam, L. Gerhardt, Prabhat, M. F. Ringenburg, and K. J. Maschhoff. "Alchemist: An Apache Spark MPI Interface." *Computing Research Repository*, 2018.

[GSS+13] J. Giceva, T. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe. "COD: Database / Operating System Co-Design." In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research*. 2013.

[GZAR16] J. Giceva, G. Zellweger, G. Alonso, and T. Roscoe. "Customized OS support for data-processing." In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, pp. 2:1–2:6. 2016.

[HAPS17] R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker. "An Evaluation of Distributed Concurrency Control." *Proceedings of the VLDB Endowment*, vol. 10, no. 5, 553–564, 2017.

[HB15] T. Hoefler and R. Belli. "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results." In *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 73:1–73:12. 2015.

[HCPR12] J. Hilland, P. Culley, J. Pinkerton, and R. Recio. *RDMA Protocol Verbs Specification*. Internet Engineering Task Force, 2012.

[HDT+15] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. D. Underwood. "Remote Memory Access Programming in MPI-3." *ACM Transactions on Parallel Computing*, vol. 2, no. 2, 9:1–9:26, 2015.

[HGTT10] T. Hoefler, W. Gropp, R. Thakur, and J. L. Träff. "Toward Performance Models of MPI Implementations for Understanding Application Scaling Issues." In *Recent Advances in the Message Passing Interface - 17th European MPI Users' Group Meeting*, pp. 21–30. 2010.

[HMLR07] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm. "Netgauge: A Network Performance Measurement Framework." In *Proceedings of the 3rd International Conference on High Performance Computing and Communications*, pp. 659–671. 2007.

[Hoe10] T. Hoefler. "Bridging Performance Analysis Tools and Analytic Performance Modeling for HPC." In *Proceedings of Workshop on Productivity and Performance*. 2010.

[Hoe16] T. Hoefler. "Active RDMA - new tricks for an old dog.", 2016. Invited talk at Salishan Meeting.

[HSL10] T. Hoefler, T. Schneider, and A. Lumsdaine. "Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale." *International Journal of Parallel, Emergent and Distributed Systems*, vol. 25, no. 4, 241–258, 2010.

[HY83] J. Huang and Y.C.Chow. "Parallel sorting and data partitioning by sampling." In *Proceedings of the 1983 Computer Software and Applications Conference*. 1983.

[IBY+07] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks." In *Proceedings of the 2nd European Conference on Computer Systems*, pp. 59–72. 2007.

[Inf07] InfiniBand Trade Association. *InfiniBand Architecture Specification, Volume 1, Release 1.2.1*, 2007.

[Inf10] InfiniBand Trade Association. *Supplement to the InfiniBand Architecture Specification Volume 1, Release 1.2.1: Annex A16: RDMA over Converged Ethernet*, 2010.

[ISA16] Z. István, D. Sidler, and G. Alonso. "Runtime Parameterizable Regular Expression Operators for Databases." In *Proceedings of the 24th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 204–211. 2016.

[IWA14] Z. István, L. Woods, and G. Alonso. "Histograms as a side effect of data movement for big data." In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 1567–1578. 2014.

[JSL+11] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. "Memcached Design on High Performance RDMA Capable Interconnects." In *Proceedings of the 2011 International Conference on Parallel Processing*, pp. 743–752. 2011.

[KAH+01] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. L. Gittings. "Predictive performance and scalability modeling of a large-scale application." In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pp. 37:1–37:12. 2001.

[KDSA08] J. Kim, W. J. Dally, S. Scott, and D. Abts. "Technology-Driven, Highly-Scalable Dragonfly Topology." In *Proceedings of the 35th International Symposium on Computer Architecture*, pp. 77–88. 2008.

[KGA17] K. Kara, J. Giceva, and G. Alonso. "FPGA-based Data Partitioning." In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, pp. 433–445. 2017.

[KK93] L. V. Kalé and S. Krishnan. "A Comparison Based Parallel Sorting Algorithm." In *Proceedings of the 1993 International Conference on Parallel Processing*, pp. 196–200. 1993.

[KKA14] A. Kalia, M. Kaminsky, and D. G. Andersen. "Using RDMA efficiently for key-value services." In *Proceedings of the 2014 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 295–306. 2014.

[KKA16] A. Kalia, M. Kaminsky, and D. G. Andersen. "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs." In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 185–201. 2016.

[KKN+08] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. "H-

store: a high-performance, distributed main memory transaction processing system." *Proceedings of the VLDB Endowment*, vol. 1, no. 2, 1496–1499, 2008.

[KN11] A. Kemper and T. Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." In *Proceedings of the 27th IEEE International Conference on Data Engineering*, pp. 195–206. 2011.

[KNPZ16] A. Kumar, J. F. Naughton, J. M. Patel, and X. Zhu. "To Join or Not to Join?: Thinking Twice about Joins before Feature Selection." In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pp. 19–34. 2016.

[KR81] H. T. Kung and J. T. Robinson. "On Optimistic Methods for Concurrency Control." *ACM Transactions on Database Systems*, vol. 6, no. 2, 213–226, 1981.

[KSC+09] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs." *Proceedings of the VLDB Endowment*, vol. 2, no. 2, 1378–1389, 2009.

[KT11] D. Koch and J. Tørresen. "FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting." In *Proceedings of the ACM SIGDA 19th International Symposium on Field Programmable Gate Arrays*, pp. 45–54. 2011.

[KTM83] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. "Application of Hash to Data Base Machine and Its Architecture." *New Generation Computing*, vol. 1, no. 1, 63–74, 1983.

[LBB+] F. Liu, C. Barthels, S. Blanas, H. Kimura, and G. Swart. "Beyond RDMA: Towards a New Communication Abstraction for Data-Intensive Computing." [Under submission].

[LCC+15] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T. Lee, J. Loaiza, N. MacNaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zaït. "Oracle Database In-Memory: A dual format in-memory database." In *Proceedings of the 31st IEEE International Conference on Data Engineering*, pp. 1253–1258. 2015.

[LCX16] H. Li, T. Chen, and W. Xu. "Improving Spark performance with zero-copy buffer management and RDMA." In *Proceedings of the IEEE Conference on Computer Communications Workshops*, pp. 33–38. 2016.

[LDSN16] F. Li, S. Das, M. Syamala, and V. R. Narasayya. "Accelerating Relational Databases by Leveraging Remote Memory and RDMA." In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pp. 355–370. 2016.

[LLA+13] H. Lang, V. Leis, M. Albutiu, T. Neumann, and A. Kemper. "Massively Parallel NUMA-aware Hash Joins." In *Proceedings of the 1st International Workshop on In-Memory Data Management and Analytics*, pp. 1–12. 2013.

[LLS+16] J. Lagraviere, J. Langguth, M. Sourouri, P. H. Ha, and X. Cai. "On the performance and energy efficiency of the PGAS programming model on multicore architectures." In *Proceedings of the 2016 International Conference on High Performance Computing and Simulation*, pp. 800–807. 2016.

[LMK+17] J. Lee, S. Moon, K. H. Kim, D. H. Kim, S. K. Cha, W. Han, C. G. Park, H. J. Na, and J. Lee. "Parallel Replication across Formats in SAP HANA for Scaling Out Mixed OLTP/OLAP Workloads." *Proceedings of the VLDB Endowment*, vol. 10, no. 12, 1598–1609, 2017.

[LMV15] S. Lee, J. S. Meredith, and J. S. Vetter. "COMPASS: A Framework for Automated Performance Modeling and Prediction." In *Proceedings of the 29th ACM International Conference on Supercomputing*, pp. 405–414. 2015.

[LPEK15] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. "On the Design and Scalability of Distributed Shared-Data Databases." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 663–676. 2015.

[LSBS18] F. Liu, A. Salmasi, S. Blanas, and A. Sidiropoulos. "Chasing Similarity: Distribution-aware Aggregation Scheduling." *Proceedings of the VLDB Endowment*, vol. 12, no. 3, 292–306, 2018.

[LWI+14] X. Lu, M. Wasi-ur-Rahman, N. S. Islam, D. Shankar, and D. K. Panda. "Accelerating Spark with RDMA for Big Data Processing: Early Experiences." In *Proceedings of the 22nd IEEE Annual Symposium on High-Performance Interconnects*, pp. 9–16. 2014.

[LYB17] F. Liu, L. Yin, and S. Blanas. "Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems." In *Proceedings of the 12th European Conference on Computer Systems*, pp. 48–63. 2017.

[Mak17] D. Makreshanski. "Systems and Methods for Interactive Data Processing on Modern Hardware." Ph.D. thesis, ETH Zurich, 2017.

[MBK02] S. Manegold, P. A. Boncz, and M. L. Kersten. "Optimizing Main-Memory Join on Modern Hardware." *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 4, 709–730, 2002.

[MBN04] S. Manegold, P. A. Boncz, and N. Nes. "Cache-Conscious Radix-Decluster Projections." In *Proceedings of the 13th International Conference on Very Large Data Bases*, pp. 684–695. 2004.

[McC95] J. D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers." *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, 1995.

[MCJ+18] N. Malitsky, A. Chaudhary, S. Jourdain, M. Cowan, P. O'Leary, M. D. Hanwell, and K. K. van Dam. "Building Near-Real-Time Processing Pipelines with the Spark-MPI Platform." *Computing Research Repository*, 2018.

[Mel18a] Mellanox Technologies. *Mellanox BlueField - Multicore System on Chip*, 2018.

[Mel18b] Mellanox Technologies. *Mellanox Innova 2 Flex - Open Programmable SmartNIC*, 2018.

[Mes12] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*, 2012.

[MGBA17] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. "BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications." In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 37–50. 2017.

[MIC97] Microsoft Corporation, Intel Corporation, and Compaq Computer Corporation. *Virtual Interface Architecture Specification*, 1997.

[MMI+13] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. "Naiad: a timely dataflow system." In *Proceedings of the 24th ACM SIGOPS Symposium on Operating Systems Principles*, pp. 439–455. 2013.

[Mon18] MongoDB Inc. "MongoDB." https://mongodb.com/, 2018. Online, accessed July 2018.

[MRR+13] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. "ScyPer: A Hybrid OLTP&OLAP Distributed Main Memory Database System for Scalable Real-Time Analytics." In *Datenbanksysteme für Business, Technologie und Web (BTW)*, pp. 499–502. 2013.

[MTA09] R. Müller, J. Teubner, and G. Alonso. "Streams on Wires - A Query Compiler for FPGAs." *Proceedings of the VLDB Endowment*, vol. 2, no. 1, 229–240, 2009.

[MTA10] R. Müller, J. Teubner, and G. Alonso. "Glacier: a query-to-hardware compiler." In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pp. 1159–1162. 2010.

[MTA12] R. Müller, J. Teubner, and G. Alonso. "Sorting networks on FPGAs." *VLDB Journal*, vol. 21, no. 1, 1–23, 2012.

[Neo18] Neo Technology. "Neo4J." http://neo4j.com/, 2018. Online, accessed July 2018.

[NMF10] F. D. Neeser, B. Metzler, and P. W. Frey. "SoftRDMA: Implementing iWARP over TCP kernel sockets." *IBM Journal of Research and Development*, vol. 54, no. 1, 5, 2010.

[Ope18a] Open MPI Development Team. "Open MPI." https://www.open-mpi.org/, 2018. Online, accessed July 2018.

[Ope18b] OpenFabrics Alliance - OFI Working Group. "OpenFabrics Interfaces." http://libfabric.org/, 2018. Online, accessed July 2018.

[OR11] A. Okcan and M. Riedewald. "Processing theta-joins using MapReduce." In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pp. 949–960. 2011.

[Ora12] Oracle Corporation. "A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server." *Oracle Corp. Whitepaper*, 2012.

[PBB+17] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquin, and D. Kossmann. "Fast Scans on Key-Value Stores." *Proceedings of the VLDB Endowment*, vol. 10, no. 11, 1526–1537, 2017.

[PFH+02] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. "The Quadrics Network: High-Performance Clustering Technology." *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, vol. 22, no. 1, 46–57, 2002.

[PSR14] O. Polychroniou, R. Sen, and K. A. Ross. "Track join: distributed joins with minimal network traffic." In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 1483–1494. 2014.

[RCP17] L. Rupprecht, W. Culhane, and P. R. Pietzuch. "SquirrelJoin: Network-Aware Distributed Join Processing with Lazy Partitioning." *Proceedings of the VLDB Endowment*, vol. 10, no. 11, 1250–1261, 2017.

[RHMM18] C. D. Rickett, U.-U. Haus, J. Maltby, and K. J. Maschhoff. "Loading and Querying a Trillion RDF triples with Cray Graph Engine on the Cray XC." In *Cray User Group*. 2018.

[RIKN16] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. "Flow-Join: Adaptive skew handling for distributed joins over high-speed networks." In *Proceedings of the 32nd IEEE International Conference on Data Engineering*, pp. 1194–1205. 2016.

[RMKN15] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. "High-Speed Query Processing over High-Speed Networks." *Proceedings of the VLDB Endowment*, vol. 9, no. 4, 228–239, 2015.

[RMU+14] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. "Locality-sensitive operators for parallel main-memory database clusters." In *Proceedings of the 30th IEEE International Conference on Data Engineering*, pp. 592–603. 2014.

[Röd16] W. Rödiger. "Scalable Distributed Query Processing in Parallel Main-Memory Database Systems." Ph.D. thesis, Technical University Munich, 2016.

[SBH16] P. Schmid, M. Besta, and T. Hoefler. "High-Performance Distributed RMA Locks." In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pp. 19–30. 2016.

[SBK+17] A. Salama, C. Binnig, T. Kraska, A. Scherp, and T. Ziegler. "Rethinking Distributed Query Execution on High-Speed Networks." *IEEE Data Engineering Bulletin*, vol. 40, no. 1, 27–37, 2017.

[SCW+02] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. M. Badia, and A. Purkayastha. "A framework for performance modeling and prediction." In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pp. 24:1–24:17. 2002.

[SD89] D. A. Schneider and D. J. DeWitt. "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment." In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pp. 110–121. 1989.

[Sha14] N. Shamgunov. "The MemSQL In-Memory Database System." In *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics*. 2014.

[SIA16] D. Sidler, Z. István, and G. Alonso. "Low-latency TCP/IP stack for data center applications." In *Proceedings of the 26th International Conference on Field Programmable Logic and Applications*, pp. 1–4. 2016.

[SIOA17] D. Sidler, Z. István, M. Owaida, and G. Alonso. "Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures." In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 403–415. 2017.

[SK10] E. Solomonik and L. V. Kalé. "Highly scalable parallel sorting." In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12. 2010.

[SKN94] A. Shatdal, C. Kant, and J. F. Naughton. "Cache Conscious Algorithms for Relational Query Processing." In *Proceedings of 20th International Conference on Very Large Data Bases*, pp. 510–521. 1994.

[STM+15] B. Sukhwani, M. Thoennes, H. Min, P. Dube, B. Brezzo, S. W. Asaad, and D. Dillenberger. "A Hardware/Software Approach for Database Query Acceleration with FPGAs." *International Journal of Parallel Programming*, vol. 43, no. 6, 1129–1159, 2015.

[SW13] M. Stonebraker and A. Weisberg. "The VoltDB Main Memory DBMS." *IEEE Data Engineering Bulletin*, vol. 36, no. 2, 21–27, 2013.

[Swi18] Swiss National Supercomputing Centre. "Piz Daint Supercomputer." http://user.cscs.ch/computing_systems/piz_daint/index.html, 2018. Online, accessed July 2018.

[SWL13] B. Shao, H. Wang, and Y. Li. "Trinity: a distributed graph engine on a memory cloud." In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 505–516. 2013.

[SWS+12] H. Shan, N. J. Wright, J. Shalf, K. A. Yelick, M. Wagner, and N. Wichmann. "A preliminary evaluation of the hardware acceleration of the Cray Gemini interconnect for PGAS languages and comparison with MPI." *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 2, 92–98, 2012.

[tBR10] M. ten Bruggencate and D. Roweth. "Dmapp - An API for One-sided Program Models on Baker Systems." In *Cray User Group*. 2010.

[TDW+12] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. "Calvin: fast distributed transactions for partitioned database systems." In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 1–12. 2012.

[The18] The Ohio State University. "MVAPICH." http://mvapich.cse.ohio-state.edu/, 2018. Online, accessed July 2018.

[Top18] Top500. "The Top 500 List." http://www.top500.org/, 2018. Online, accessed July 2018.

[Tra10] Transaction Processing Performance Council. *TPC Benchmark C - Standard Specification*, 2010.

[Tra17] Transaction Processing Performance Council. *TPC Benchmark H - Standard Specification*, 2017.

[Tra18a] Transaction Processing Performance Council. *TPC Benchmark DS - Standard Specification*, 2018.

[Tra18b] Transaction Processing Performance Council. "TPC-C Benchmark Results." http://www.tpc.org/tpcc/results/tpcc_results.asp, 2018. Online, accessed July 2018.

[TZ17] S.-Y. Tsai and Y. Zhang. "LITE Kernel RDMA Support for Datacenter Applications." In *Proceedings of the 26th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '17, pp. 306–324. 2017.

[TZK+13] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. "Speedy transactions in multicore in-memory databases." In *Proceedings of the 24th ACM SIGOPS Symposium on Operating Systems Principles*, pp. 18–32. 2013.

[WABJ15] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang. "Smart: a MapReduce-like framework for in-situ scientific analytics." In *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 51:1–51:12. 2015.

[WDCC18] X. Wei, Z. Dong, R. Chen, and H. Chen. "Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!" In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, pp. 233–251. 2018.

[WIL+13] M. Wasi-ur-Rahman, N. S. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. "High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand." In *Proceedings of the 2013 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1908–1917. 2013.

[WJFP17] T. Wang, R. Johnson, A. Fekete, and I. Pandis. "Efficiently making (almost) any concurrency control mechanism serializable." *VLDB Journal*, vol. 26, no. 4, 537–562, 2017.

[WSC+15] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. "Fast in-memory transaction processing using RDMA and HTM." In *Proceedings of the 25th ACM SIGOPS Symposium on Operating Systems Principles*, pp. 87–104. 2015.

[YBP+14] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores." *Proceedings of the VLDB Endowment*, vol. 8, no. 3, 209–220, 2014.

[YCM18] D. Y. Yoon, M. Chowdhury, and B. Mozafari. "Distributed Lock Management with RDMA: Decentralization without Starvation." In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, pp. 1571–1586. 2018.

[YIF+08] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language." In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pp. 1–14. 2008.

[ZBKH17] E. Zamanian, C. Binnig, T. Kraska, and T. Harris. "The End of a Myth: Distributed Transaction Can Scale." *Proceedings of the VLDB Endowment*, vol. 10, no. 6, 685–696, 2017.

[ZvdWB12] M. Zukowski, M. van de Wiel, and P. Boncz. "Vectorwise: A Vectorized Analytical DBMS." In *Proceedings of the 28th IEEE International Conference on Data Engineering*, pp. 1349–1350. 2012.

Appendices

A

Programming with RDMA Verbs

A.1 Connection Setup

Setting up an RDMA connection using the RDMA Verbs Interface involves several steps and is relatively cumbersome. Furthermore, the connection setup varies slightly between different network implementations. In this dissertation, we therefore limit our explanations to the InfiniBand network that is used in the experimental evaluation in Chapter 3.

The fundamental connection abstraction is the queue pair. Before a queue pair can be created, the necessary completion queues have to be instantiated, using the `ibv_create_cq` call. Afterwards, the `ibv_qp_init_attr` struct has to be populated before it can be passed to the `ibv_create_qp` method that will instantiate the queue pair. The data structure contains pointers to the completion queues, the maximum number of work requests these queues can hold, the maximum number of a scatter-gather elements per request, and various other information relevant to the creation of a connection.

```
struct ibv_qp_init_attr {
        void *qp_context;
```

```
        struct ibv_cq  *send_cq;
        struct ibv_cq *recv_cq;
        struct ibv_srq *srq;
        struct ibv_qp_cap cap;
        enum ibv_qp_type qp_type;
        int sq_sig_all;
};
struct ibv_qp *ibv_create_qp(struct ibv_pd *pd, struct ibv_qp_init_attr
   *qp_init_attr);
```

Once the queue pairs have been created, they need to be connected pairwise. The connection setup happens in several steps. First, the queue pair is transitioned to the initial `IBV_QPS_INIT` state and the queue pair number, sequence number, and device identifier are transmitted to the other side. This exchange usually happens through an out-of-band connection. This information is used to partially fill a `ibv_qp_attr` struct that can be used to alter specific aspects of the queue pair.

```
struct ibv_qp_attr {
        enum ibv_qp_state qp_state;
        enum ibv_qp_state cur_qp_state;
        enum ibv_mtu path_mtu;
        enum ibv_mig_state path_mig_state;
        uint32_t qkey;
        uint32_t rq_psn;
        uint32_t sq_psn;
        uint32_t dest_qp_num;
        int qp_access_flags;
        struct ibv_qp_cap cap;
        struct ibv_ah_attr ah_attr;
        struct ibv_ah_attr alt_ah_attr;
        uint16_t pkey_index;
        uint16_t alt_pkey_index;
        uint8_t en_sqd_async_notify;
        uint8_t sq_draining;
        uint8_t max_rd_atomic;
        uint8_t max_dest_rd_atomic;
        uint8_t min_rnr_timer;
```

```
        uint8_t port_num;
        uint8_t timeout;
        uint8_t retry_cnt;
        uint8_t rnr_retry;
        uint8_t alt_port_num;
        uint8_t alt_timeout;
};
int ibv_modify_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr, int
    attr_mask);
```

Using the exchanged information, the queue pair is first transitioned into the *ready to receive* (`IBV_QPS_RTR`) state using the `ibv_modify_qp` call. After successful completion of this step, the queue pair is then put in the *ready to send* (`IBV_QPS_RTS`) mode. Afterwards, it is connected to the other queue pair and fully operational. A queue pair can also be linked to itself and operate as a *loop-back* queue.

In order to make the setup phase less dependent on the actual network implementation, the Rdma Connection Management Abstraction (Rdma-Cma) has been developed. It allows for the creation of *event channels* over which the connection requests can be received and the necessary queue pair information are being exchanged automatically. On InfiniBand networks, this abstraction operates in a similar manner as described in this section.

A.2 Memory Registration

When using Rdma, the application has to manage all communication buffers manually in user-space, register them with the network card, and distribute the access information to the relevant system components.

An application first has to create buffers of appropriate size by using standard memory allocation mechanisms such as `malloc` and `mmap`. As the majority of networks requires the memory to be registered with the network card, Rdma Verbs provides the `ibv_reg_mr` call that, given a protection domain, address, size, and access flags, pins the memory such that

it cannot be swapped to disk and installs the necessary address translation information on the network card.

```
struct ibv_mr *ibv_reg_mr(struct ibv_pd *pd, void *addr, size_t length,
    int access);
```

After the registration call, the function returns the memory region information, in particular the sKey and rKey of the registered buffer. Once the registration is complete, this information cannot be altered. Changing the size of a region is possible in some network implementations, but often requires a complete re-registration of the buffer. It is left to the application to transmit the necessary access information to other processes. It is also worth noting that the same piece of memory can be registered multiple times, for example for security purposes when using two different protection domains.

A.3 Synchronizing Access to Remote Memory

Once memory is accessible to Rdma operations, adequate synchronization mechanisms are needed to prevent concurrent accesses to the same piece of memory, similar to developing multi-threaded algorithms and thread-safe data structures.

When using the Rdma Verbs Api, it is left to the application to synchronize concurrent accesses to the memory that is being shared over the network. It does not provide any build-in mechanisms to grant accesses to a specific memory region. Any part of the application that is in possession of the necessary access information can issue Rdma operations, provided that the buffer is in a protection domain it has access to.

A.4 Remote Read, Write, and Atomic Operations

Once the necessary connections between the systems elements have been established and memory has been registered with the network card, operations can be performed on these regions of remote memory. Of interest are one-sided Rma operations that allow one

process to read from or write to remote memory without the involvement of the processor on the target machine. The RDMA Verbs interface provides a single interface, i.e., the `ibv_post_send` call, for submitting all work requests. The user needs to fill out the `ibv_send_wr` work request struct. The struct contains the operation code (`opcode`) that has to be set to `IBV_WR_RDMA_WRITE` for write or `IBV_WR_RDMA_READ` for read operations. In order to atomically compare and swap a remote 64-bit value, the `opcode` value has to be equal to `IBV_WR_ATOMIC_CMP_AND_SWP`. Setting the field to `IBV_WR_ATOMIC_FETCH_AND_ADD` will trigger an atomic increment of the remote value.

In addition to the operation code, the application needs to specify the local buffers that are used in the operation. Multiple buffers can be used as one logical buffer through the use of a scatter-gather list. Each element of the list (`ibv_sge`) contains the virtual address, length, and local key of the targeted buffer.

In case of one-sided operations, the request needs to contain the address and key of the remote memory region. In addition, for atomic operations, the work request has to include the value to compare the remote number against, the replacement value, and/or the value by which the remote counter needs to be increased or decreased.

```
struct ibv_sge {
        uint64_t addr;
        uint32_t length;
        uint32_t lkey;
};
struct ibv_send_wr {
        uint64_t wr_id;
        struct ibv_send_wr *next;
        struct ibv_sge *sg_list;
        int num_sge;
        enum ibv_wr_opcode opcode;
        int send_flags;
        uint32_t imm_data;
        union {
                struct {
                        uint64_t remote_addr;
                        uint32_t rkey;
```

```
                } rdma;
                struct {
                        uint64_t remote_addr;
                        uint64_t compare_add;
                        uint64_t swap;
                        uint32_t rkey;
                } atomic;
                struct {
                        struct ibv_ah *ah;
                        uint32_t remote_qpn;
                        uint32_t remote_qkey;
                } ud;
        } wr;
};
int ibv_post_send(struct ibv_qp *qp, struct ibv_send_wr *wr, struct
   ibv_send_wr **bad_wr);
```

B

Programming with MPI

B.1 Connection Setup

When developing an application with Mpi, the complexity of establishing connections is the responsibility of the library and the Mpi runtime. To set up the library, including connections and data structures, each process executes a single `MPI_Init` call at the start of its execution.

```
int MPI_Init(int *argc, char ***argv);
```

It is worth noting, that many Mpi implementations running on InfiniBand or RoCE networks internally use the Rdma Verbs Interface and use a similar method than the one described in Appendix A for establishing connections.

B.2 Memory Registration

In MPI, registered memory is referred to as a *window*. Memory can either be allocated through the default allocation mechanisms offered by the operating system (i.e., `malloc` and `mmap`) or through MPI itself by calling `MPI_Mem_alloc`. The memory is registered with the network by executing the `MPI_Win_create` method. This call is a collective call, which means that it has to be executed by every process in the communication group that wants to perform RMA operations, even if the process does not register memory itself. A second method, named `MPI_Win_alloc`, combines memory allocation and registration.

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info
    info, MPI_Comm comm, MPI_Win *win);
```

The `MPI_Win` object returned by the function represents the collection of memory windows that were the input to the collective call by all the processes belonging to the specified communication group.

B.3 Synchronizing Access to Remote Memory

Before any operation can be executed on a window, the processes need to be properly synchronized. MPI provides multiple synchronization mechanisms: `MPI_Win_fence` synchronizes all RMA calls on a specific window, such that all incoming and outgoing operations will complete before the call returns. The period in-between two fence calls is referred to as an *epoch*. Because `MPI_Win_fence` is a collective call, this type of synchronization is called active target synchronization. It is useful for applications designed to operate in distinct rounds where every process goes through the exact same number of epochs.

```
int MPI_Win_fence(int assert, MPI_Win win);
```

To allow for applications with more complex communication patterns, MPI provides passive target synchronization mechanisms through a lock-based mutual exclusion mechanism. Before an RMA operation on a specific window can be executed, it needs to be locked. The

lock provides either exclusive (MPI_LOCK_EXCLUSIVE) or concurrent (MPI_LOCK_SHARED) access to a buffer. When releasing a lock, the library ensures that all pending RMA operations have completed both at the origin and at the target before the call returns.

```
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win);
```

To amortize the costs of synchronization, the user should initiate multiple data transfers per epoch. For shared access, a call exists to lock all memory windows associated with the window object instead of specifying a target rank and locking each window individually.

B.4 Remote Read, Write, and Atomic Operations

MPI provides multiple communication functions, one for each RMA operation. In order to write data to a remote window, the application invokes MPI_Put, providing the address of the local buffer, local and remote size and data type, target rank and window object. The data is then being transferred to the buffer that was registered by the target process during the collective window allocation operation (see Section B.2). MPI_Get provides a similar interface, but triggers a read operation.

For atomic operations on the other hand, MPI provides a generic method in which the user can define the function that needs to be executed. In order to create a fetch-and-add operation, the user invokes the MPI_Fetch_and_op function with the MPI_SUM argument. Combining the call with MPI_REPLACE exchanges the remote value atomically. In addition to predefined operations, the interface allows the application to use any user-defined function as an argument to the MPI_Fetch_and_op call.

```
int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype
    origin_datatype, int target_rank, MPI_Aint target_disp, int
    target_count, MPI_Datatype target_datatype, MPI_Win win);
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
    origin_datatype, int target_rank, MPI_Aint target_disp, int
    target_count, MPI_Datatype target_datatype, MPI_Win win);
```

```
int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
    MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Op
    op, MPI_Win win);
```