Technische Universität München
Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme

Scalable Distributed Query Processing in Parallel Main-Memory Database Systems

Wolf-Steffen Rödiger

Technische Universität München
Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme

Scalable Distributed Query Processing in Parallel Main-Memory Database Systems

Wolf-Steffen Rödiger
Master of Science with honours

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr.-Ing. Georg Carle

Prüfer der Dissertation:

1. Prof. Alfons Kemper, Ph.D.
2. Prof. Gustavo Alonso, Ph.D.
3. Prof. Dr. Thomas Neumann

Die Dissertation wurde am 25.02.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17.05.2016 angenommen.

Abstract

The continuous increase in compute speed and main-memory capacity of modern servers triggered the development of a new generation of in-memory database systems. These systems completely rewrote the traditional database architecture to use main memory as primary storage. Discarding several now obsolete abstractions of disk-based database systems enabled unprecedented query performance on a single server. However, network communication slows down queries as soon as multiple servers are involved. The result is a significant performance gap between local and distributed query processing. Still, a scale out to a cluster becomes inevitable when the workload exceeds the capacity of a single server.

This thesis seeks to further the state-of-the-art of distributed query processing in parallel main-memory database systems by addressing the performance barrier introduced by network communication. Thus, instead of concentrating on an isolated problem, we design a novel distributed query engine that adapts to the available network bandwidth as well as unexpected workload characteristics that hinder scalability. It exploits locality to speed up query processing over commodity networks and implements a novel parallelism model to fully leverage modern high-speed interconnects. We prove the feasibility of our design with a prototypical implementation for the high-performance in-memory database system HyPer. Using redo log multicasting and global transaction-consistent snapshots, the engine further enables query processing on fresh transactional data. An extensive evaluation with the renowned TPC-H analytical benchmark demonstrates that HyPer with our novel distributed query engine not only outperforms competing parallel database systems but also scales its query performance with the cluster size.

Kurzfassung

Die kontinuierliche Steigerung der Rechengeschwindigkeit und Hauptspeicherkapazität von modernen Servern hat zur Entwicklung einer neuen Generation von Hauptspeicher-Datenbanksystemen geführt. Bei diesen Systemen löst der deutlich schnellere Hauptspeicher die Festplatte als Primärspeicher ab. Dabei wurde die traditionelle Datenbankarchitektur tiefgreifend verändert, um eine erhebliche Leistungssteigerungen bei analytischen Anfragen zu erreichen. Allerdings beschränkt nun teure Netzwerkkommunikation den Durchsatz von analytischen Anfragen, sobald bei der Bearbeitung mehrere Rechner involviert sind. Dadurch ergibt sich ein signifikanter Leistungsunterschied zwischen lokalen und verteilten Anfragen. Der Einsatz eines Rechnerverbunds bleibt jedoch unvermeidlich, wenn die verarbeiteten Daten die Kapazität eines einzelnen Rechners überschreiten.

Diese Dissertation zielt auf die Verbesserung des aktuellen Stands der Technik bezüglich der verteilten Anfrageverarbeitung in parallelen Hauptspeicher-Datenbanksystemen ab. Sie adressiert dabei primär die negativen Auswirkungen von Netzwerkkommunikation auf die Anfrageleistung dieser Systeme. Anstatt ein isoliertes Problem zu betrachten, beschreiben wir den Entwurf einer neuen verteilten Anfrageeinheit, die sich an die verfügbare Netzwerkbandbreite und an unerwartete Eigenschaften der verarbeiteten Daten anpassen kann. Sie nutzt Lokalität in der Datenplatzierung um die Anfrageverarbeitung über herkömmliche Netzwerke zu beschleunigen. Weiterhin setzt sie ein neuartiges Modell zur Parallelisierung ein, so dass das Potenzial moderner Hochgeschwindigkeits-Netzwerke voll ausgenutzt werden kann. Wir belegen die Machbarkeit unseres Entwurfs anhand eines Prototyps für das hochperformante Hauptspeicher-Datenbanksystem HyPer. Durch die Verwendung von Redo-Log-Multicasting und globalen transaktionskonsistenten Schnappschüssen ermöglicht unsere Anfrageeinheit weiterhin analytische Anfrageverarbeitung auf aktuellen transaktionalen Daten. Eine ausführliche Evaluation unter Verwendung des renommierten TPC-H Benchmarks zeigt, dass HyPer mit unserer neuen verteilten Anfrageeinheit vergleichbare Systeme schlägt und seine Anfrageleistung mit der Größe des Rechnerverbunds steigern kann.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An in-memory database system needs to scale to a cluster of servers to process workloads that exceed the main-memory capacity of a single server. However, a scale-out introduces network communication as distributed queries have to read data from different servers. For today's systems network bandwidth is the clear bottleneck for distributed query processing. Distributed joins and aggregations are slowed down by expensive all-to-all data shuffling. This thesis focusses on four important problems that a distributed query engine for a modern parallel main-memory database system has to address when it aims at scalable distributed query processing: the reduced network bandwidth of commodity networks, the new bottlenecks revealed when modern high-speed networks replace commodity networks, the load imbalance caused by skew, and the performance penalty of distributed transactions when queries are processed on fresh transactional data.

1.1 Problem Statement

Modern in-memory database systems completely rewrite the traditional database architecture to accommodate for switching primary storage from disk to main memory. They are able to offer unprecedented single-node query performance by removing several now obsolete abstractions of traditional disk-based database systems and utilizing the high compute power and parallelism level of today's many-core servers. However, the main-memory capacity of a single server is limited to a few terabytes. A scale out to a cluster of machines is necessary to process workloads that do not fit into the main memory of a single machine. Perfect partitioning of the workload is unrealistic, which makes distributed processing inevitable. Commodity networks such as Gigabit Ethernet slow down query processing due to the large performance gap between CPU speed and network bandwidth. However, even special high-speed networking hardware such as InfiniBand alone cannot achieve scalable query processing. Instead, new bottlenecks surface such as the inflexibility of the classic exchange operator model and CPU-intensive TCP/IP stack processing. A parallel database system thus has to adapt query processing differently depending on whether commodity or high-speed interconnects are used. Query processing performance is further impaired by attribute value skew and the resulting load imbalances. Traditional skew handling approaches add significant processing overheads for non-skewed workloads. These become immediately visible when high-speed networks are used that are too fast to hide any extra computational work. Modern businesses require query processing on fresh transactional data. However, moving from a single server to a cluster of machines will in general severely degrade transaction throughput due to the overheads incurred by distributed commit handling, global lock management, and deadlock detection.

A distributed query engine for a parallel hybrid in-memory database system that aims at scaling the query processing performance with the cluster size therefore has to address the following issues:

1. **Insufficient bandwidth of commodity networks.**

   In the past, the query performance of a parallel database system was bound by processing costs and disk I/O, while network I/O played only a minor role [15, 20]. Modern main-memory database systems offer unprecedented single-node query processing performance by rewriting the database architecture to fully utilize the large main-memory capacities and many cores of today's servers. Consequently, the network bandwidth has become the dominant bottleneck for distributed query processing. In fact, a single machine easily outperforms a cluster of servers connected via Gigabit Ethernet as soon as data has to be shuffled across the network for distributed joins and aggregations. Uncoordinated network communication further reduces the throughput and thereby query performance due to switch contention.

2. **New bottlenecks revealed by high-speed networks.**

   While special high-speed networking hardware is not yet commonly available in data centers such as Amazon EC2 and the Google Compute Engine, it has become economically viable for custom-built database clusters. However, using faster networking hardware alone does not solve the scalability problem of distributed query processing where a single server easily outperforms a cluster of machines. When high-speed networks replace commodity networks as the cluster interconnect but the distributed query engine remains unchanged, new bottlenecks surface that limit query performance. This includes the inflexibility of the classic exchange operator model and the high overhead of TCP/IP stack processing [13, 27]. Both prevent a main-memory database system from leveraging the full potential of modern high-speed networks.

3. **Load imbalances caused by skew.**

   Skew threatens the scalability of distributed query processing. Attribute value skew manifests itself in a large numbers of duplicate values, so called heavy hitters. These create load imbalances during distributed joins so that one server has to process a much larger part of the input than its fair share. Other servers have to wait for this one straggler, increasing the query response time. Traditional approaches either require a separate analysis phase to detect skew or extensive statistics that might not be available or might be overly inaccurate, in particular for intermediate results. High-speed networks are simply too fast to hide any substantial amount of extra computational work. Hence, a novel lightweight skew detection and handling approach is required that adds only minimal overhead for non-skewed workloads.

4. **Performance penalty of distributed transactions.**

   Modern businesses require real-time analytics on the most recent state of the transactional data. Traditional approaches that extract data from the transactional system to a dedicated data warehouse in regular intervals, e.g., every night, inevitably suffer from the problem of data staleness. HyPer [45] is a hybrid in-memory database system that processes transactions and queries on the same data at the same time, hence providing the desired real-time business analytics. However, HyPer is limited by the query throughput and main-memory capacity of a single server. A new approach is required to scale a hybrid main-memory database system such as HyPer to a cluster and still sustain its excellent transaction throughput of more than 100 000 TPC-C transactions per second without sacrificing the high consistency guarantees.

1.2 Research Questions

This thesis focusses on the following research questions that result directly from the preceding problem statement:

1. Is it possible to reduce the expensive all-to-all data shuffles incurred by distributed query processing over slow commodity networks?
2. Which changes to the distributed query engine are required to address the new bottlenecks that modern interconnects reveal?
3. Is it possible to achieve scalable query processing in the presence of skew without sacrificing the performance for non-skewed workloads?
4. Can a parallel database system achieve scalable query processing on fresh transactional data without sacrificing transaction throughput?

1.3 Contributions

We make the following contributions to the area of distributed query processing in main-memory database systems:

1. **Locality-aware data shuffling for commodity networks.**

   The disparity between the large processing capacities and limited network bandwidth of commodity servers is so pronounced that compute-intensive optimizations that minimize the network duration become worthwhile. We developed the distributed join algorithm Neo-Join that exploits locality in the data distribution to speed up distributed query processing. It optimizes the assignment of join partitions to servers via linear programming to achieve minimal query response times. Neo-Join further avoids the detrimental effects of cross traffic via intelligent network scheduling.

2. **High-speed query processing over high-speed networks.**

   We developed a novel distributed query engine that addresses the new bottlenecks that surface when a high-speed interconnect replaces a slow commodity network. The engine implements a new hybrid parallelism model. It leverages morsel-driven parallelism [52] for NUMA-aware processing and InfiniBand's highly-efficient remote direct memory access (RDMA) in combination with low-latency network scheduling for fast network communication. Our engine scales the excellent query performance of the HyPer main-memory database system not only with the number of cores per server but also with the number of servers in the cluster—something that was not possible before.

3. **Low-overhead skew handling.**

   We propose a novel skew handling scheme for distributed joins called Flow-Join. It detects heavy hitter skew using small approximate histograms that add only a minimal processing overhead. Join key values that exceed a skew threshold are kept local to avoid a load imbalance. Skew detection and skew handling happen at runtime and do not require additional statistics or materialization of the join inputs. Most importantly, Flow-Join enables the database system to perform always-on skew detection without paying a noticeable performance penalty for non-skewed inputs.

4. **Query processing on fresh transactional data.**

   We present ScyPer, a new architecture for hybrid distributed query and transaction processing. It consists of a primary node that processes transactions and secondary nodes that replay its redo log to keep their transactional state up-to-date. Queries are processed by secondaries using a round-robin load balancing scheme to linearly scale the query throughput with the number of servers. At the same time, it sustains the excellent transaction throughput of the hybrid in-memory database system HyPer. Global transaction-consistent snapshots separate query and transaction processing, avoiding consistency problems. Secondaries act as fast fail-over nodes that can replace the primary in case of unexpected failures. We also give an outlook beyond full replication by drafting a distributed architecture designed to leverage the combined main-memory capacity of the cluster, scale query performance, and still sustain HyPer's excellent transaction throughput.

1.4 Outline

The remainder of this thesis is structured as follows:

- Chapter 2 introduces Neo-Join, a join algorithm for slow commodity networks that utilizes locality-aware data shuffling to exploit locality in the data distribution and thereby speeds up query processing over commodity interconnects such as Gigabit Ethernet. It achieves the minimal query execution time by optimizing the assignment of join partitions to nodes. Neo-Join further utilizes an intelligent network scheduling approach to avoid cross traffic that would otherwise significantly reduce network throughput.

- In Chapter 3 we describe our novel distributed query engine that is specifically tailored for high-speed networks. Our new hybrid approach to parallelism employs the existing NUMA-aware morsel-driven parallelism for local processing while utilizing an efficient RDMA-enabled communication multiplexer for fast network communication. In combination, our holistic approach enables query performance that scales on both levels: the number of cores inside a single server and the number of servers in the cluster.

- Chapter 4 describes Flow-Join, our new lightweight skew handling approach for distributed joins. It uses small approximate histograms to detect skew at runtime while adding only a minimal overhead to the query response time, even when high-speed networks are used. A dedicated data redistribution scheme for the detected heavy hitters avoids the load imbalances usually caused by skew and even improves performance by reducing communication.

- Chapter 5 presents ScyPer, a distributed architecture for HyPer that enables query processing on fresh transaction data. ScyPer uses redo log multicasting to keep secondaries up-to-date and global transaction-consistent snapshots to separate query and transaction processing. A round-robin load-balancing scheme scales query performance linearly with the cluster size. At the same time, HyPer's excellent transaction throughput is sustained.

Chapter 2

Commodity Networks

Parts of this chapter were previously published in [71].

The growth in compute speed has outpaced the growth in network bandwidth over the last decades for commodity networking hardware such as Gigabit Ethernet, which is predominant in today's data centers. This has led to an increasing performance gap between local and distributed query processing. A parallel database system thus has to maximize the locality of query processing. Relations are commonly co-partitioned to avoid expensive data shuffling across the network. However, this is limited to one attribute per relation and is expensive to maintain in the face of updates. Other attributes often exhibit a *fuzzy* co-location due to correlations with the distribution key or *time-of-creation* clustering, but current approaches do not leverage this.

In this chapter, we introduce *locality-sensitive data shuffling*, which can dramatically reduce the amount of network communication for distributed operators such as joins and aggregations. We present four novel techniques: (i) *optimal partition assignment* exploits locality to reduce the network phase duration; (ii) *communication scheduling* avoids bandwidth underutilization caused by cross traffic; (iii) *adaptive radix partitioning* retains locality during data repartitioning and handles value skew gracefully; and (iv) *selective broadcast* reduces network communication in the presence of extreme value skew or large numbers of duplicates. We present comprehensive experimental results, which show that our techniques can improve performance by up to factor of 5 if the data distribution exhibits locality and a factor of 3 for inputs with value skew.

2.1 Motivation

Parallel databases are a well-studied field of research, which attracted considerable attention in the 1980s with the Grace [31], Gamma [20], and Bubba [10] systems.

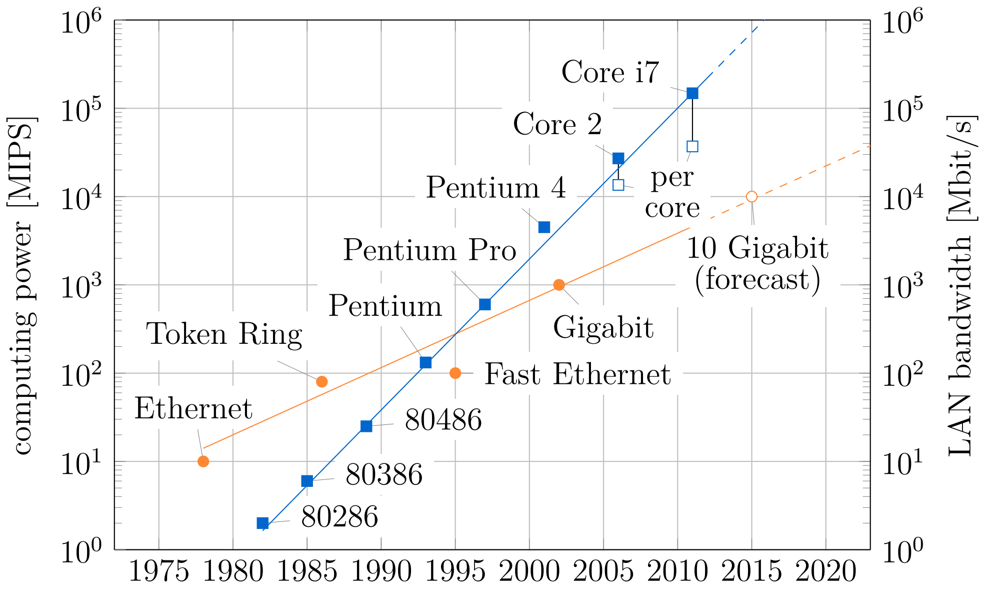


Figure 2.1: Evolution of CPU speed and network bandwidth[1]

At the time, the network bandwidth was sufficient for the query performance of these disk-based database systems. For example, DeWitt et al. [20] showed that Gamma took only 12 % additional execution time when data was redistributed over the network—compared to an entirely local join of co-located data. Copeland et al. [15] similarly stated in the context of the Bubba system in 1986:

"[...] on-the-wire interconnect bandwidth will not be a bottleneck."

The situation has changed dramatically since the 1980s. For today's commodity network hardware, this formerly small overhead of 12 % to compute a distributed instead of a local join has turned into a significant performance penalty of 500 % (cf., Section 2.5.1 for the detailed experimental results). This performance gap is caused by the development depicted in Figure 2.1: Over the last decades, CPU performance has grown much faster than the network bandwidth, at least for commodity hardware that is used in data centers such as Amazon EC2 or the Google Compute Engine. High-speed networks such as InfiniBand are not yet commonly available and still several times more expensive than the predominant Gigabit Ethernet [42]. This chapter therefore focusses on optimizing distributed operators for commodity network hardware. We address InfiniBand in Chapter 3 of this thesis, building a high-speed query engine that takes advantage of these special high-speed networks.

[1]CPU MIPS from Hennessy and Patterson [40]. Dominant network speed of new servers according to the IEEE NG BASE-T study group [42].

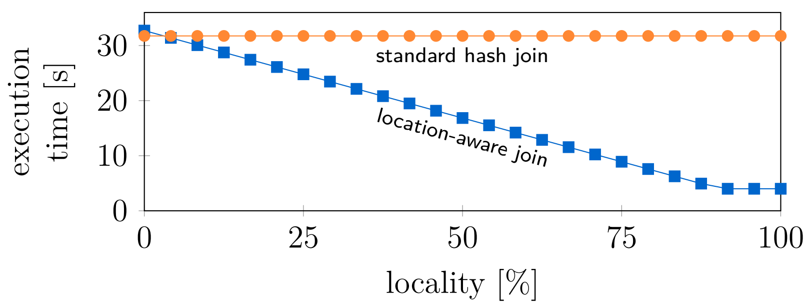
Modern main-memory database systems such as HyPer [45], MonetDB [55], or Vectorwise [88] achieve an unprecedented single-node query processing performance that widens the gap to distributed processing further. Slow network transfers cannot be hidden behind other bottlenecks anymore (e.g., disk access) so that any significant data transfer becomes immediately visible as an overhead to the query execution time.

The current situation is highly unsatisfactory for parallel in-memory database systems: Important distributed operators such as join, aggregation, and duplicate elimination are slowed down by overly expensive data shuffling. A common technique to reduce network communication is to explicitly co-partition frequently joined tables by their join key [86]. However, co-partitioning of relations is limited to one attribute per relation (unless the data is also being replicated), requires prior knowledge of the workload, and is expensive to maintain in the face of updates. Other attributes often exhibit a fuzzy co-location due to strong correlations with the distribution key or time-of-creation clustering but current approaches do not leverage this.

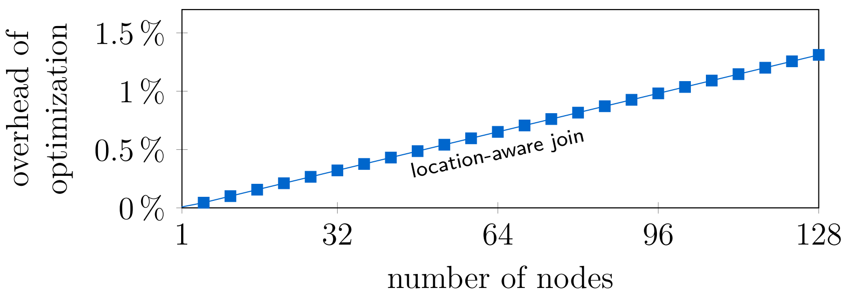
In this chapter, we introduce *locality-sensitive data shuffling*, four novel techniques that exploit workload characteristics to reduce the amount of network communication incurred by distributed operators. Its cornerstone is the *optimal partition assignment*, which allows operators to benefit from data locality. Most importantly, the optimization overhead is small even when data exhibits no locality. We employ *communication scheduling* to use all the available network bandwidth of the cluster. Uncoordinated communication would otherwise lead to cross traffic and thereby reduce bandwidth utilization. We propose an *adaptive radix partitioning* for the repartitioning to retain locality in the data and to handle value skewed inputs gracefully. *Selective broadcast* is an extension to the partition assignment that decides dynamically for every partition between shuffle and broadcast. With selective broadcast, our approach can benefit from extreme value skew and reduce communication further.

We primarily target clusters of high-end commodity machines, which consist of few but *fat* nodes with large amounts of main memory. This typically represents the most economic choice for parallel main-memory database systems. MapReduce-style systems are, in contrast, designed for larger clusters of low-end machines. Recent work [86] has extended the MapReduce processing model with the ability to co-partition data. However, this does not cover fuzzy co-location, in which case MapReduce-style systems still have to perform a network-intensive data redistribution between map and reduce tasks. They would thus also benefit from locality-sensitive data shuffling.

The immense savings that locality-sensitive data shuffling can achieve compared to a standard hash join (shown in Figure 2.2a) easily compensate the asso-



(a) **Benefit:** Join time for 128 nodes, 1 GbE, and 200 M tuples per node



(b) **Cost:** Overhead to optimize the partition assignment for data locality

Figure 2.2: Benefit and cost of the location-aware join

ciated optimization costs (shown in Figure 2.2b). Even when the data distribution exhibits no locality and a large cluster of 128 nodes is used, partition assignment and network scheduling still increase query execution time only by a mere 1.3 %.

In summary, this chapter makes the following contributions:

- *Optimal partition assignment*: We devise a method that allows operators to benefit from data locality. It applies to all operators that reduce to item matching, e.g., join, aggregation, and duplicate elimination.

- *Communication scheduling:* Our communication scheduler prevents cross traffic, which would otherwise reduce network throughput dramatically.

- *Adaptive radix partitioning:* An efficient repartitioning scheme that retains existing locality in the data redistribution and handles value skewed inputs gracefully.

- *Selective broadcast:* We extend the partition assignment to selectively broadcast small partitions. This significantly improves performance for inputs with high attribute value skew.

We describe our techniques based on Neo-Join, a network-optimized join operator. They can be similarly applied to other distributed relational operators that reduce to item matching, e.g., aggregation and duplicate elimination.

2.2 Related Work

As outlined in the introduction, distributed joins have first been considered in the context of database machines. Fushimi et al. introduced the Grace hash join [50] of which a parallel version was evaluated by DeWitt et al. [20]. These algorithms are optimized for the disk as bottleneck. However, parallel join processing in main-memory database systems avoids the disk and is instead limited by the network bandwidth of today's commodity network hardware.

Wolf et al. proposed heuristics for distributed sort-merge [84] and hash join [85] algorithms to achieve load balancing in the presence of skew. However, their approach targets CPU-bound systems and does not apply when the network is the bottleneck. Wilschut et al. [83] devised a distributed hash join algorithm with fewer synchronization requirements. Again, CPU costs were identified as the limiting factor. Stamos and Young [74] improved the fragment-replicate (FR) join [24] by reducing its communication cost. However, partition-based joins still outperform FR in the case of equi-joins. Afrati and Ullman [1] optimized FR for MapReduce, while Blanas et al. [8] compared joins for MapReduce. MapReduce focuses especially on scalability and fault-tolerance, whereas we target per-node efficiency. Frey et al. [29] designed a join algorithm for non-commodity high-speed InfiniBand networks with a ring topology. They state that the network was not the bottleneck. We will address high-speed InfiniBand networking hardware in Chapter 3.

Systems with non-uniform memory access (NUMA) distinguish expensive remote from cheaper local reads similar to distributed systems. Teubner et al. [78] designed a stream-based join for NUMA systems. However, it does not fully utilize the bandwidth of all memory interconnect links. Albutiu et al. [3] presented MPSM, a NUMA-aware sort-merge-based join algorithm. Li et al. [53] applied data shuffling to NUMA systems and in particular to MPSM. They showed that a coordinated round-robin access pattern increases the bandwidth utilization by up to 3× but reduces the join execution time by only 8 % as sorting dominates the runtime. We show that data shuffling applied to distributed systems achieves much higher benefits. In contrast to Li et al. we also consider skew in the data placement.

Bloom filters [9] are commonly used to reduce the network traffic. They are orthogonal to our approach and can be used as a preliminary step. However, they should not be applied in all cases due to their incurred computation and communication costs. Dynamic bloom filters [6] lower the computation costs as they can be maintained continuously for common join attributes. Still, the exchange of the filters itself causes network traffic that increases quadratically in the number of nodes. They should therefore only be used for joins that are highly selective while our approach is independent of join selectivity.

CloudRAMSort [48] introduced the idea to split tuples into key and payload. This is compatible with our approach and could be applied to reduce communication costs further. A second data shuffling phase would then merge result tuples with their payloads.

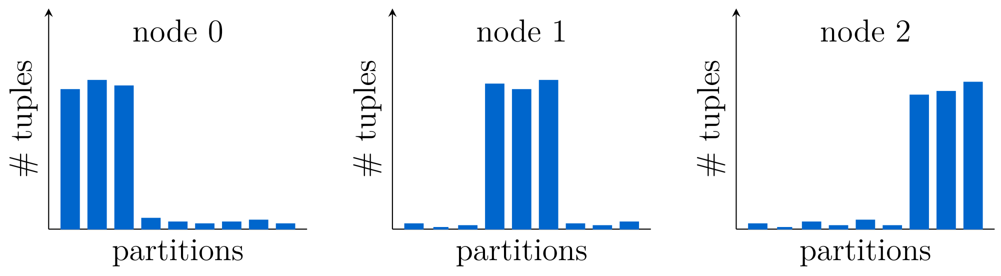
2.3 Neo-Join

Neo-Join is a distributed join algorithm based on locality-sensitive data shuffling. It computes the equi-join of two relations R and S, which are horizontally fragmented across the nodes of a distributed system. Neo-Join exploits locality and handles value skew gracefully using optimal partition assignment. Communication scheduling allows it to avoid cross-traffic. The algorithm proceeds in four phases: (1) *repartition* the data, (2) *assign* the resulting partitions to nodes for a minimal network phase duration, (3) *schedule* the communication to avoid cross traffic, and (4) *shuffle* the partitions according to the schedule while *joining* incoming partition chunks in parallel. In the following we describe each of these phases in detail.

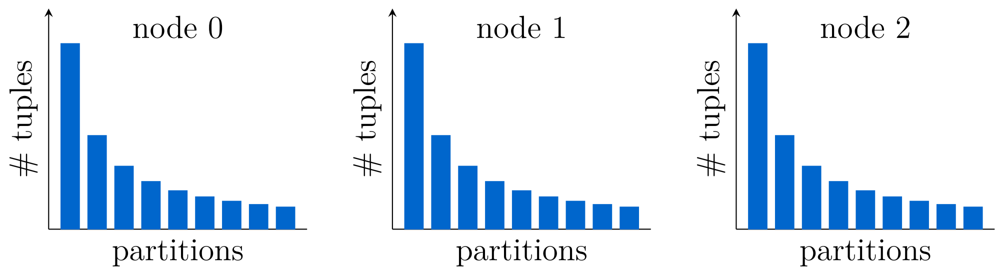
2.3.1 Data Repartitioning (Phase 1)

The idea to split join relations into disjoint partitions was introduced by the Grace [31] and Gamma [20] database machines. Partitioning ensures that all tuples with the same join key end up in the same partition. Consequently, partitions can be joined independently on different nodes. We first define locality and skew before covering different choices to repartition the inputs.

Locality. We use the term locality to describe the degree of local clustering for a specific partitioning of the data. Figure 2.3a shows an example with high locality. We specify locality in percent, where $x\,\%$ denotes that on average for each partition the node with its largest part has $x\,\%$ of its tuples with an additional $1/n$-th of the remaining tuples (for n nodes). $0\,\%$ thus corresponds to a uniform distribution where all nodes own equal parts of all partitions and $100\,\%$ to the other extreme where nodes own partitions exclusively. Locality in the data distribution can have many reasons, e.g., a (fuzzy) co-partitioning of the two relations, a distribution key used as join key, a correlation between distribution and join key, or time-of-creation clustering. Locality can be created on purpose during load time to



(a) **Locality:** the partition fragments exhibit a high degree of local clustering



(b) **Skew:** the partition sizes are skewed according to a Zipfian distribution

Figure 2.3: Data locality vs. attribute value skew

benefit from the significant savings possible with locality-sensitive operators. One may also let tuples wander with queries to create fuzzy co-location for frequently joined attributes.

Skew. The term skew is commonly used to denote the deviation from a uniform distribution. Skewed inputs can significantly affect the join performance and therefore should be considered during the design of parallel and distributed join algorithms (e.g., [84, 3, 47]). We use the term *value skew* to denote inputs with skewed value distributions. Skewed inputs can lead to skewed partitions as shown in Figure 2.3b. We will first focus on locality and afterwards address skew in Section 2.4.

In the following, we assume the general case that distribution and join key differ. Otherwise, one or both relations would already be distributed by the join attribute and repartitioning becomes straightforward: The existing partitioning of one relation can be used to repartition the other relation accordingly. Our optimal partition assignment will automatically exploit the locality in this way.

There are several options to repartition the inputs. Optimal partition assignment (cf., Section 2.3.2) and communication scheduling (cf., Section 2.3.3) apply to any of them. However, repartitioning schemes such as the proposed radix partitioning that retain locality in the data placement can improve the join performance significantly, as we describe in the following.

Hash Partitioning

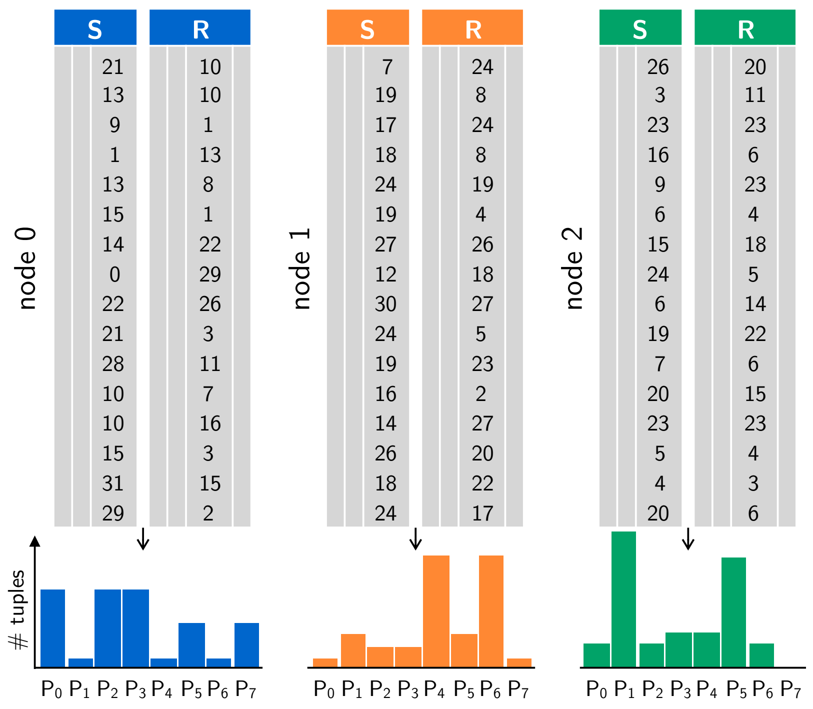
Hashing is commonly used for partitioning as it generally achieves balanced partitions. However, it can be sub-optimal. An example is an input that is almost range-partitioned across nodes (e.g., caused by time-of-creation clustering as is the case for the *orders* and *orderline* relations in TPC-H). With range-partitioning, the input could be assigned to nodes in a way that only the few tuples that violate the range partitioning are shipped over the network. Hash partitioning destroys this fuzzy range partitioning and instead generates n balanced partitions. Nodes keep only $1/n$-th of their input data, leading to unnecessary network communication.

Radix Partitioning

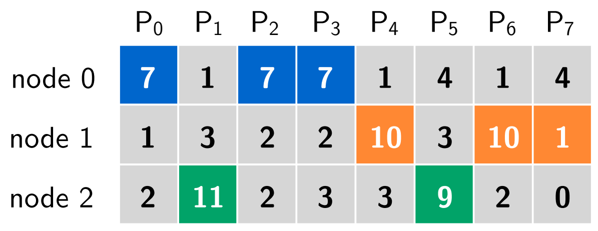
We propose *radix partitioning* [54] of the join key based on the most significant bits[2] (MSB) to retain locality in the data placement. MSB radix partitioning is a special, "weak" case of hash partitioning, which uses the b most significant bits as the hash value. Of course, using the key directly does not produce such balanced partitions as proper hash partitioning. But more importantly, MSB radix partitioning is order-preserving and thereby a restricted case of range partitioning, which allows the algorithm to assign the partitions to nodes in a way that reduces the communication costs significantly. By partitioning the input into many more partitions than there are nodes, one can still handle value skew, e.g., when there is a bias towards small keys. Section 2.4 covers techniques that handle moderate and extreme cases of value skew while keeping the number of partitions low.

Figure 2.4 depicts a simple example with 5 bit join keys ($0 \leq \text{key} < 32$). First, the nodes compute histograms for their local input by radix-clustering the tuples into eight partitions $P_0, \ldots, P_7$ according to their 3 most significant bits $\mathbf{b_4 b_3 b_2} b_1 b_0$ as shown in Figure 2.4a. Next, the algorithm assigns these eight partitions to the three nodes so that the duration of the network phase is minimal. As we show in Section 2.3.3, the network phase duration is determined by the maximum straggler, i.e., the node that needs the most time to receive or send its data. An optimal assignment, which minimizes the communication time of the maximum straggler, is shown in Figure 2.4b. Figure 2.4c depicts the send and receive cost for every node to transfer the tuples according to the optimal assignment. With this assignment both node 0 and node 2 are maximum stragglers with a cost of 12 (node 0 receives 5 tuples from node 1 and 7 from node 2, node 2 sends 7 tuples to node 0 and 5 to node 1). In the best case for hash partitioning, every node sends $1/n$-th of its tuples to every other node (≈ 21) and receives $1/n$-th of the tuples from every other node (also ≈ 21). In this simplified example, radix partitioning thus reduced the duration of the network phase by almost a *factor of two*.

[2]More precisely, the most significant *used* bits to avoid leading zeros.

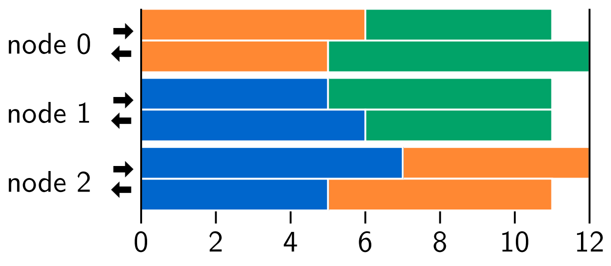


(a) Histograms according to the three most significant bits



	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
node 0	7	1	7	7	1	4	1	4
node 1	1	3	2	2	10	3	10	1
node 2	2	11	2	3	3	9	2	0

(b) Optimal partition assignment



(c) Network phase duration for the assignment

Figure 2.4: Optimal partition assignment

The next two sections explain how the histograms resulting from the chosen partitioning scheme can be used to first compute an *optimal assignment* of partitions to nodes and afterwards the corresponding *communication schedule*.

2.3.2 Optimal Partition Assignment (Phase 2)

The previous section described how to repartition the input relations so that tuples with the same join key fall into the same partition. In general, the new partitions are *fragmented* across nodes. Therefore, all fragments of one specific partition have to be transferred to the same node for joining. We now describe how to determine an assignment of partitions to nodes that minimizes the network phase duration.

We define the *receive cost* r_i of a node i as the number of tuples it receives from other nodes for the partitions that were assigned to it. Similarly, its *send cost* s_i is defined as the number of tuples node i has to send to other nodes. Section 2.3.3 shows that the minimum network phase duration is determined by the node with the maximum send or receive cost. The assignment is therefore optimized to minimize this maximum cost.

A naïve approach would assign a partition to the node that owns its largest fragment. However, this is not optimal in general. Consider the assignment for the running example in Figure 2.4b. Partition 7 is assigned to node 1 even though node 0 owns its largest fragment. While the assignment of partition 7 to node 0 reduces the send cost of node 0 by 4 tuples, it also increases its receive cost to a total of 13 tuples. As a result, the network phase duration would increase from 12 to 13 (see Figure 2.4c). As we will show in Section 2.3.2, the problem of computing an optimal assignment of partitions to nodes is in fact NP-hard.

Mixed-Integer Linear Programming

We phrase the partition assignment problem as a mixed-integer linear program. As a result, we can use an integer programming solver to solve it. The linear program computes a configuration of the decision variables $x_{ij} \in \{0, 1\}$. These decision variables define the assignment of the p partitions to the n nodes: $x_{ij} = 1$ determines that partition j is assigned to node i, while $x_{ij} = 0$ specifies that partition j is not assigned to node i.

Each partition has to be assigned to exactly one node. We model this in our linear program by adding the constraint that the sum of all decision variables x_{ij} for a partition j has to be 1:

$$\sum_{i=0}^{n-1} x_{ij} = 1 \qquad\qquad \text{for } 0 \leq j < p \tag{2.1}$$

We want to compute the optimal partition assignment using our linear program. Therefore, we need to specify an objective function that minimizes the duration of the network phase. This duration is equal to the maximum send or receive cost over all nodes as mentioned before and explained in detail in Section 2.3.3. We denote the send cost of node i as s_i and its receive cost as r_i. Using these placeholders, the objective function becomes the following:

$$\min \max_{0 \leq i < n} \{s_i, r_i\} \tag{2.2}$$

Using the decision variables x_{ij} and the size of partition j at node i—denoted with h_{ij}—we can now express the amount of data each node has to send (s_i) and receive (r_i), defining the placeholders in the objective function:

$$s_i = \sum_{j=0}^{p-1} h_{ij} \cdot (1 - x_{ij}) \qquad \text{for } 0 \leq i < n \tag{2.3}$$

$$r_i = \sum_{j=0}^{p-1} \left(x_{ij} \sum_{k=0, i \neq k}^{n-1} h_{kj} \right) \qquad \text{for } 0 \leq i < n \tag{2.4}$$

Equation 2.3 computes the send cost of node i as the size of all local fragments of partitions that are not assigned to it. These fragments need to be sent to a remote node and thus constitute the send cost. Equation 2.4 sums up the size of remote fragments of partitions that were assigned to node i. These remote fragments have to be sent to node i and thus make up its receive cost.

Mixed-integer linear programs require a linear objective, which unfortunately minimizing a maximum is not. However, we can rephrase the objective and instead minimize a new variable w. Additional constraints take care that w assumes the maximum over all send and receive costs:

$$\textbf{minimize } \boldsymbol{w}\textbf{, subject to}$$

$$\text{(OPT-ASSIGN)} \qquad \begin{aligned} w &\geq \sum_{j=0}^{p-1} h_{ij}(1 - x_{ij}) && 0 \leq i < n \\ w &\geq \sum_{j=0}^{p-1} \left(x_{ij} \sum_{k=0, i \neq k}^{n-1} h_{kj} \right) && 0 \leq i < n \\ 1 &= \sum_{i=0}^{n-1} x_{ij} && 0 \leq j < p \end{aligned}$$

One can obtain an optimal solution for a specific partition assignment problem (OPT-ASSIGN) by passing the mixed-integer linear program to an optimizer such as Microsoft Gurobi[3] or IBM CPLEX[4]. These solvers can be linked as a library to create and solve linear programs with minimal overhead.

Argument Size Balancing

In general, one would like to avoid that large fractions of the argument relations are assigned to a single node. This could lead to exhaustion of the resources on this node (e.g., main memory) during join computation or when a (potentially very large) result set is generated. The following constraint restricts the input size for all nodes i to a multiple of the ideal input size:

$$\sum_{j=0}^{p-1} x_{ij}(h_j^R + h_j^S) \leq (1+o) \cdot \frac{|R| + |S|}{n} \qquad \text{for } 0 \leq i < n \tag{2.5}$$

where $o \in [0, n-1]$ is the overload factor, which is allowed in addition to the ideal input size, $|R|$ and $|S|$ denote the size of the argument relations, and h_j^R and h_j^S are the total size of partition P_j for relation R and S, respectively. Argument size balancing is not used in the experiments.

NP-hardness

We provide a proof sketch to show that OPT-ASSIGN is NP-hard. We show that its *decision variant* (ASSIGN) is NP-complete by reducing the known NP-complete *partition problem* (PARTITION) to it. We recall from [32] that as a consequence OPT-ASSIGN is NP-hard. ASSIGN decides whether the objective function of OPT-ASSIGN is smaller or equal to a given constant k. PARTITION determines whether a given bag B of positive integers can be partitioned into bags S_1 and S_2 with equal sum.

The polynomial-time reduction is achieved as follows: Every integer c_i of the bag B corresponds to a partition P_i of size $2 \cdot c_i$ where two nodes n_1 and n_2 both own a fragment of size c_i. The send and receive cost for partition P_i is by construction equal to c_i for both nodes. ASSIGN can be used to decide whether an assignment exists in which both nodes have the same send and receive cost ($r_1 = r_2 = s_1 = s_2 = \text{sum}(B)/2$) by choosing $k = \text{sum}(B)/2$. If this is possible, the partitions assigned to node n_1 represent the subset S_1 and those for node n_2 the subset S_2. Thus, a solution to the assignment problem is also a solution to the partition problem. Figure 2.5 shows an example.

[3]http://www.gurobi.com

[4]http://ibm.com/software/integration/optimization/cplex

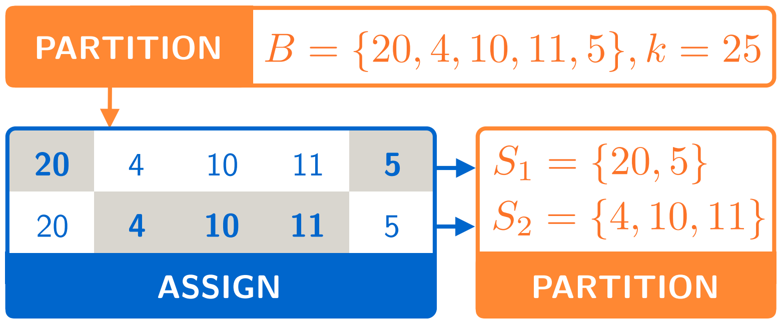


Figure 2.5: Reduction of PARTITION to ASSIGN

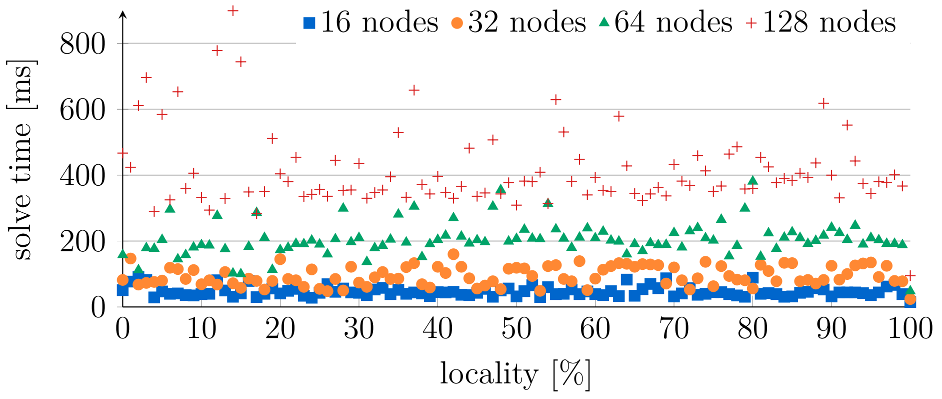
ASSIGN is in NP, with the partition assignment as a certificate. PARTITION is NP-complete [32] and we have constructed a reduction to ASSIGN. Therefore, ASSIGN is NP-complete and OPT-ASSIGN NP-hard. □

Optimization Time

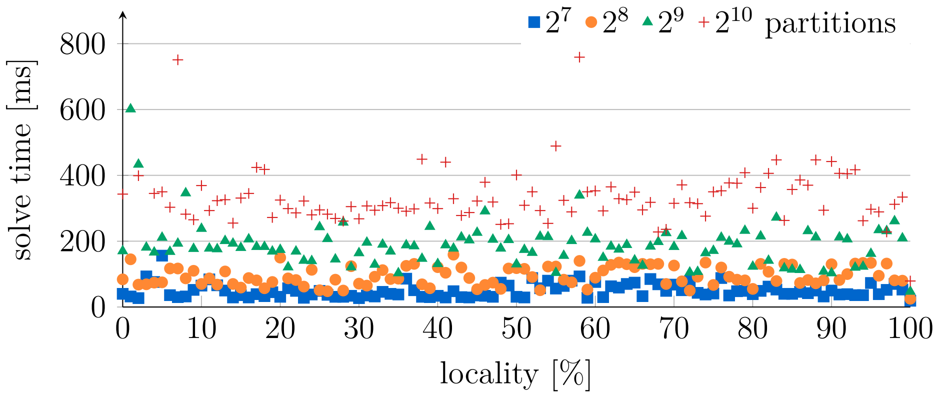
Despite the NP-hardness of the partition assignment problem it is possible to solve real-world instances in reasonable time, i.e., much faster than the potential savings in communication time. Figure 2.6 depicts the solve time using the linear programming solver CPLEX for variations of the problem with increasing locality.

In general, the solve time for a linear program increases with the number of variables. For the partition assignment, every combination between nodes and partitions is represented by a variable. Consequently, there is a direct correlation between the number of nodes/partitions and the solve time as shown in Figure 2.6a and 2.6b. These figures further show that the degree of locality does not influence optimization time. The number of tuples per node has also no impact on the solve time as expected and shown in Figure 2.6c.

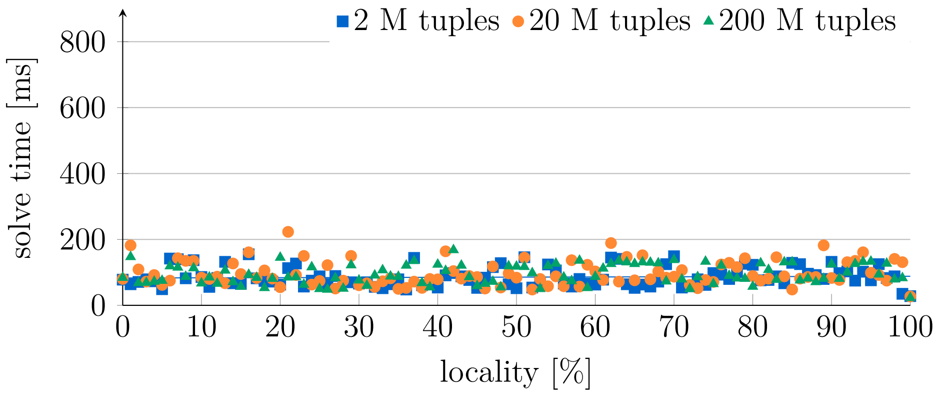
Figure 2.6 indicates that the optimal assignment becomes expensive for clusters with hundreds or even thousands of nodes. As outlined in the introduction, our target are mainly clusters with fewer but fatter nodes as this is typically the most economic choice for parallel main-memory database systems. Consequently, we only focus on the optimal solution. Efficient approximations should be possible to support larger clusters. For example, a heuristic could first assign a partition to the node which owns its largest part, then randomly try to swap partitions from the maximum straggler to the node whose cost increases minimally. This could be implemented using a meta-heuristics such as simulated annealing. We further expect an even slower network performance for very large clusters due to a shared network infrastructure, which would also increase the savings possible with locality-sensitive operators.



(a) Varying the number of *nodes* with 256 partitions and 200 M tuples per node



(b) Varying the number of *partitions* with 32 nodes and 200 M tuples per node



(c) Varying the number of *tuples* per node with 32 nodes and 256 partitions

Figure 2.6: Runtime analysis for the optimal partition assignment

There are several options to minimize the optimization time: One can reduce the number of partitions by adaptively combining small partitions (cf., Section 2.4.1). The assignment can be precomputed eagerly or cached for recurring queries to avoid the runtime optimization overhead completely. Lastly, the locality can be estimated to invest the optimization time only when the expected savings are big enough.

Giving up Optimality

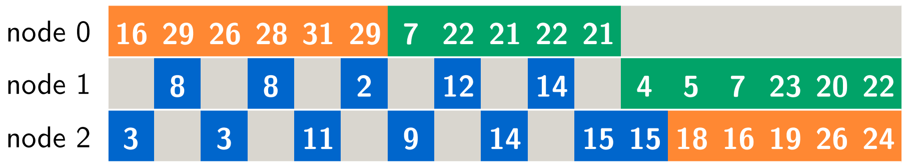
Integer linear programming (ILP) solvers first construct a feasible solution. This solution is then improved iteratively until it is proven optimal. A priori, the objective value of the optimal solution is unknown. Solvers approximate it by solving the linear program (LP) relaxation of the problem which is significantly faster to solve as it allows fractional values for all variables. The objective value for the relaxation is an upper bound for the integer program and allows the solver to estimate the distance to the optimal integer solution (called the optimality gap). A solution is considered optimal when the distance is lower than a given threshold (e.g., 0.01 %) or when it is otherwise proven to be optimal.

In the case of partition assignment, it is not essential that we find an optimal solution. Instead, we are interested in the best solution that can be found in a certain period of time. Solvers support this use case with a configurable time limit after which the currently best solution is returned. This takes care of the occasional outlier which takes longer to optimize (cf., Figure 2.6a and 2.6b) and allows us to compute approximate solutions for even larger problem instances.

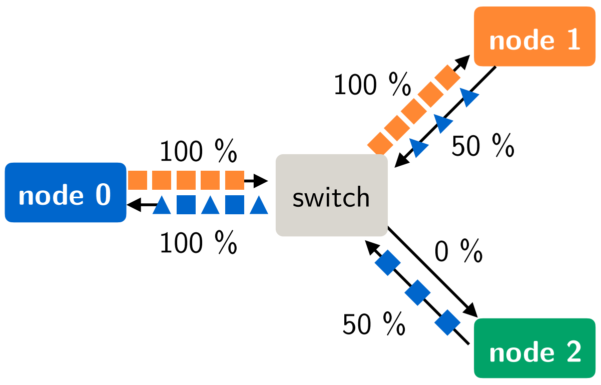
Another option for solving larger problems are heuristics. *Randomized rounding* [66] constructs a solution for the integer linear program from the linear program relaxation by repeatedly rounding the fractional variables to integer values. The obtained rounded values for the variables may violate constraints. Randomized rounding therefore has to check whether the modified solutions are still feasible. In our experiments, an integer programming solver with a time limit performed consistently better in both runtime and solution quality than randomized rounding.

2.3.3 Communication Scheduling (Phase 3)

The previous section described how to compute an optimal assignment of partitions to nodes. The next step is to distribute the partitions according to this assignment. However, when the nodes use the network without coordination, the available bandwidth is utilized poorly. The scheduling of communication tasks can improve the bandwidth utilization significantly. We assume a star topology with uniform bandwidth that is common for small clusters. Our approach can be extended to non-uniform bandwidths by adjusting the partition assignment problem.



(a) A naïve schedule which uses only 67 % of the available bandwidth



(b) The naïve schedule leads to network congestion as node 1 and 2 send to node 0 at the same time

Figure 2.7: Naïve schedule

Network Congestion

A naïve data shuffling scheme would let all the nodes send their tuples to the first node, then to the second node, and so on. Figure 2.7a depicts a naïve schedule for the running example. This simple scheme leads to significant network congestion since the nodes compete for the bandwidth of a single link while other links are not fully utilized. Figure 2.7b visualizes the reason for the network congestion: Both, node 1 and node 2 send data to node 0 at the same time and therefore share the bandwidth of the link that connects node 0 to the switch. Node 0 can send with only 1 Gbit/s to either node 1 or node 2 although both could receive simultaneously. Ultimately, 1 Gbit/s of network bandwidth remains unused.

Network congestion can be avoided entirely by dividing the communication into distinct phases. In each phase a node has a single target to which it sends, and likewise a single source from which it receives. However, it is not obvious how to determine the phases so that a schedule with minimum finish time is realized. In practice, the nodes send different amounts of data to different nodes, which renders a simple round-robin scheme impractical. The problem to devise a communication schedule with minimum finish time corresponds to the open-shop scheduling problem [36]. It can be solved in polynomial time when preemption is allowed, which in our case corresponds to splitting network transfers.

Open Shop	**Auto Shop**	**Network Transfer**
job	car	sender
task	check engine	data transfer
processor	engine test bench	receiver
execution time	time for the check	message size
preemption	suspend the check	split message

Table 2.1: Terminology mapping

The Open Shop Scheduling Problem

The open shop scheduling problem is defined for abstract jobs, tasks, and processors. We introduce it by using the example of an auto repair shop. Afterwards, we will translate our original problem of computing an optimal network communication schedule into an open shop problem. This will allow us to use the polynomial-time algorithm that solves open shop problems to compute communication schedules.

An auto repair shop consists of m processors each dedicated to perform a specific repair task, e.g., the engine test bench, the wheel alignment system, and the exhaust test facility. There are multiple jobs, i.e., cars that need maintenance, consisting of m tasks, which need to be performed, e.g., check the engines, align wheels, and test the exhaust system. Each task of a job is performed by the corresponding processor. Every task has a specific processing time. The tasks may be performed in any order as it is irrelevant if the engine or the wheel alignment is checked first. However, two repair tasks cannot be performed for the same car simultaneously, as processors are located in different buildings. Similarly, a processor can check only one car at a time. Suspension of tasks is allowed. The goal is to find an optimal schedule with minimal total processing time.

The network scheduling problem can be translated to an *open shop scheduling problem* with preemption as summarized in Table 2.1: A *task* is the data transfer from one node to another node and it has an *execution time* corresponding to the size of the data transfer. The *job* to which the task belongs is the sending node and the *processor* is the receiving node. A node should not send to several other nodes simultaneously, similarly to the tasks of a job, which cannot be processed at different processors at the same time. A node should receive from at most one other node, just as a processor can execute only one task. The data transfer between two nodes can be split into multiple smaller transfers, just as tasks can be *preempted*.

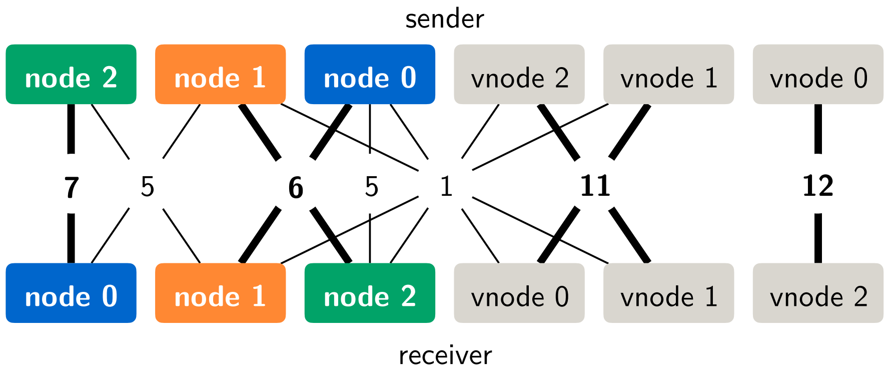


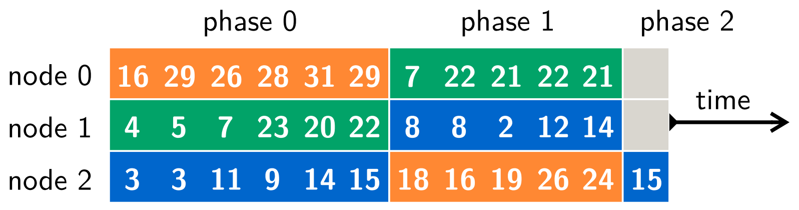
Figure 2.8: Bipartite graph with initial matching

Solving Open Shops

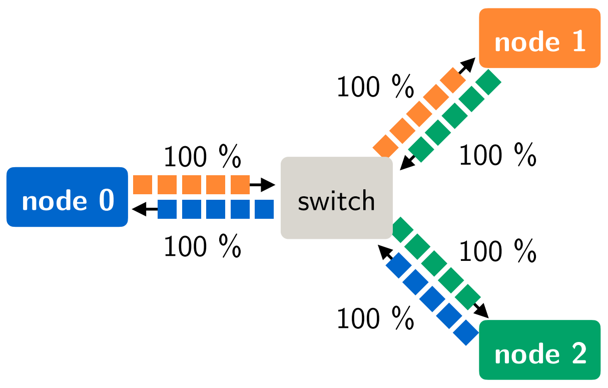
Gonzales and Sahni [36] describe a polynomial time algorithm that computes a minimum finish time schedule for open shops. The algorithm is based on finding perfect matchings in bipartite graphs. We explain it on the basis of our running example with three nodes.

The algorithm starts by generating the two vertex sets of the bipartite graph. The first set of vertices consists of a vertex for each sender and an equal number of additional vertices to represent virtual senders. Similarly, the second set has vertices for normal and virtual receivers. In the example, there are $N = 3$ nodes and therefore a total of 12 vertices in the bipartite graph as shown in Figure 2.8. Each network transfer is represented as an edge connecting a sender with a receiver. Every edge is weighted with the transfer size, e.g., node 2 has to send 7 tuples to node 0. For optimality, it is important that all nodes have a weight equal to α, the maximum send or receive cost across nodes (12 for the running example as shown by Figure 2.4c). More specifically, α is defined as $\max_{0\leq i<n}\{s_i, r_i\}$ where s_i is the send cost of node i, i.e., the sum over the message sizes of all of its outgoing traffic, and r_j is the receive cost of node j, i.e., the sum over the message sizes of all incoming traffic. Gonzales and Sahni describe in [36] how to insert edges between nodes and their virtual partners to ensure this.

The second step of the algorithm repeatedly finds perfect matchings. Every matching corresponds to a network phase and the edges of the matching define which nodes communicate in this phase. The minimal edge weight in the matching determines its duration. All matching edges are decreased by this amount, removing edges with weight zero. Senders that are matched to virtual receivers do not send in this phase and receivers matched to virtual senders do not receive. This process is repeated until no edges remain.



(a) An optimal schedule which uses 94 % of the available bandwidth



(b) The optimal schedule avoids harmful cross traffic

Figure 2.9: Optimal schedule

The matching highlighted in Figure 2.8 corresponds to the first phase of the schedule. Every transfer in this phase sends 6 tuples, as this is the minimum edge weight. The edges of the matching specify that node 0 sends to node 1, node 1 to node 2, and node 2 to node 0. The resulting optimal schedule is shown in Figure 2.9a. It consists of three phases and achieves a network bandwidth utilization of 94 %, in contrast to the 67 % for the naïve schedule.

Optimality

A schedule with a duration of less than α is not possible since α is the maximum send or receive cost across all nodes. At least one node has this send or receive cost and cannot finish earlier. Surprisingly, the algorithm always finds an optimal schedule with duration equal to α.

We give an outline of the proof given by Gonzales and Sahni [36]: They first show that it is possible to find perfect matchings in every step of the algorithm, which we take as given in the following. Furthermore, the weights of the nodes are by construction initially all equal to α. Each phase has a duration which corresponds to the minimal edge weight of the matching found in this iteration of the algorithm. The weights of the matching edges are reduced by this quantity and—since the matchings are always perfect—the weights of all nodes are decreased by the same amount. The algorithm stops when all edges are removed and the nodes have a weight of 0. The total duration of all phases is therefore α and the algorithm computes an optimal schedule. □

Time Complexity

The runtime of the algorithm is in $\mathcal{O}(r^2)$ where r is the number of non-zero tasks [36]. Every transfer of the communication schedule translates to a non-zero task. A system with n nodes has no more than $n(n-1)$ transfers because each of the n nodes sends to at most all other nodes. The runtime is therefore in $\mathcal{O}(n^4)$.

Figure 2.10a compares the time needed to schedule the communication for a varying number of nodes. It is apparent that the problem size increases with the number of nodes. Figure 2.10b and 2.10c show that the number of partitions or tuples do not affect the schedule time. The error bars show the standard deviation.

While the network scheduling is quite fast for up to 64 nodes where it takes about 50 ms, this increases to 623 ms for 128 nodes and even further for more nodes. Still, this is not a problem as the communication schedule can be computed incrementally and in parallel to the actual data shuffling. The nodes can start communicating as soon as the first phase is computed—which takes less than a millisecond. The remaining phases are then computed during the data shuffling.

Simultaneous Communication

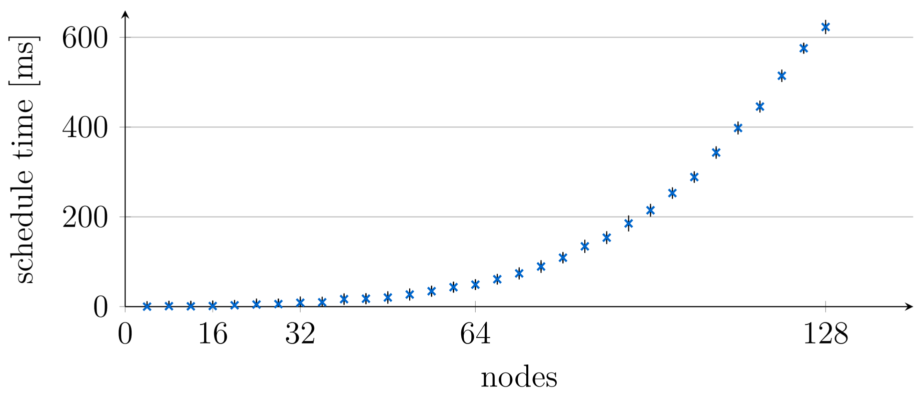
We have until now assumed that no other communication happens over the network during the data shuffling. In the general case where simultaneous communication takes place, all network traffic needs to be scheduled to utilize all the available bandwidth of the database cluster. In this case the communication scheduling should be extended to guarantee fairness, so that no operator can "starve", e.g., by using a simple round-robin scheme between operators.

2.3.4 Partition Shuffling and Local Join (Phase 4)

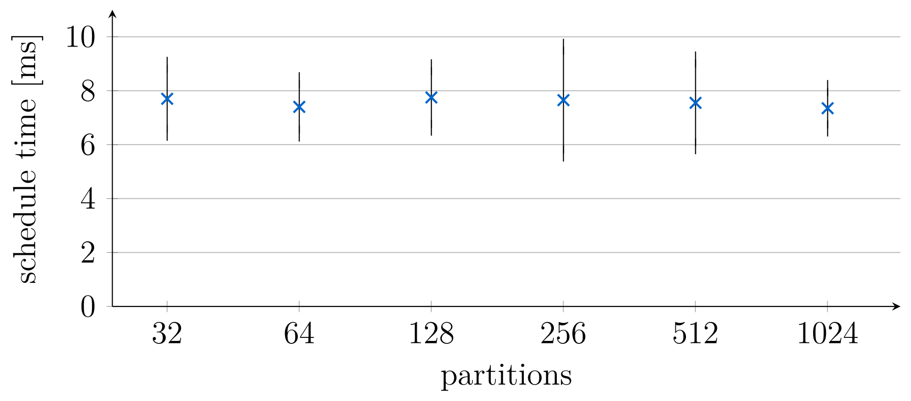
At this point, we have a partition assignment and a schedule, which together describe how to redistribute the partitions. The only task that remains is to actually transfer the partitions over the network and join incoming partition chunks in parallel.

Partition Shuffling

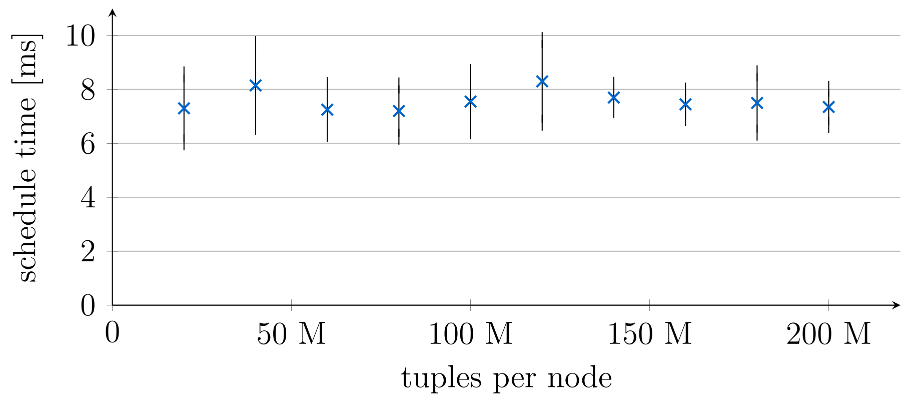
In theory, there is no need to synchronize between the phases of the communication schedule. All nodes that participate in a phase send the exact same amount of data and should therefore also finish together. In reality, some nodes stop sending a



(a) Varying the number of *nodes* with 256 partitions and 200 M tuples per node



(b) Varying the number of *partitions* with 32 nodes and 200 M tuples per node



(c) Varying the number of *tuples* per node with 32 nodes and 256 partitions

Figure 2.10: Runtime analysis for network scheduling

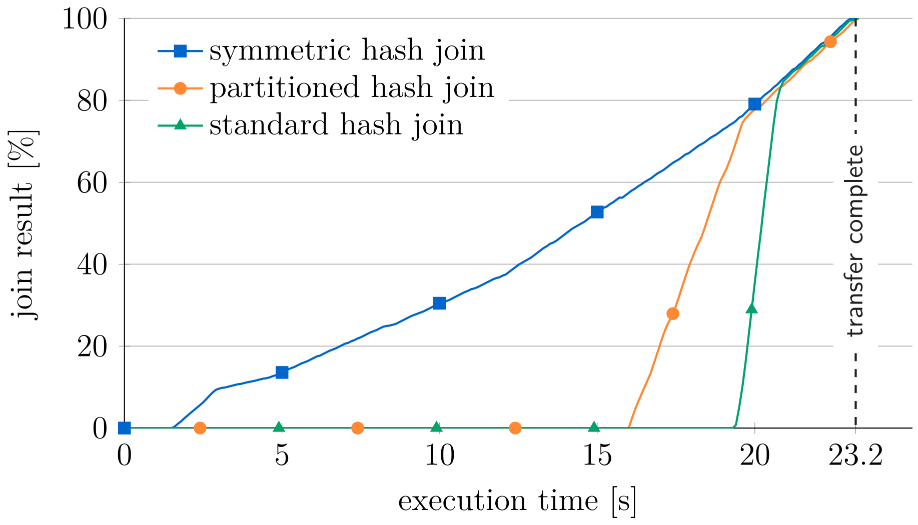


Figure 2.11: Availability of the join result over time

little bit earlier than others due to variations in the TCP throughput, e.g., caused by packet loss. These nodes then send to their next target, which still receives from another node. As a result, both nodes share the bandwidth of the same link and are slowed down. The problem intensifies when other nodes start to use the links still occupied by the slower nodes. The situation is similar to a traffic jam.

To mitigate this problem, we let nodes synchronize before they begin the next phase. While this avoids cross traffic, it introduces a synchronization barrier at which nodes could be forced to wait for a node with temporarily less bandwidth. Waiting for synchronization did not noticeably impact the performance in our experiments. Nevertheless, we propose the following solution for it: The nodes stop sending when they exceed the time limit for the current phase and report their remaining tuples. The communication scheduler updates the bipartite graph, which represents the network transfers, accordingly. It then computes a perfect matching to determine the next phase.

Local Join

CPU performance and network bandwidth have grown at different speeds over the last decades. In today's systems with commodity network hardware the runtime of a distributed join is dominated by the network transfers, whereas the local join computation on the nodes is less critical. Arriving tuples can be joined in parallel to the network transfer. Figure 2.11 shows the result generation for three hash join variants over time using 4 nodes and 200 M tuples per node. It shows that the join finishes immediately after the last tuple has arrived independent of the

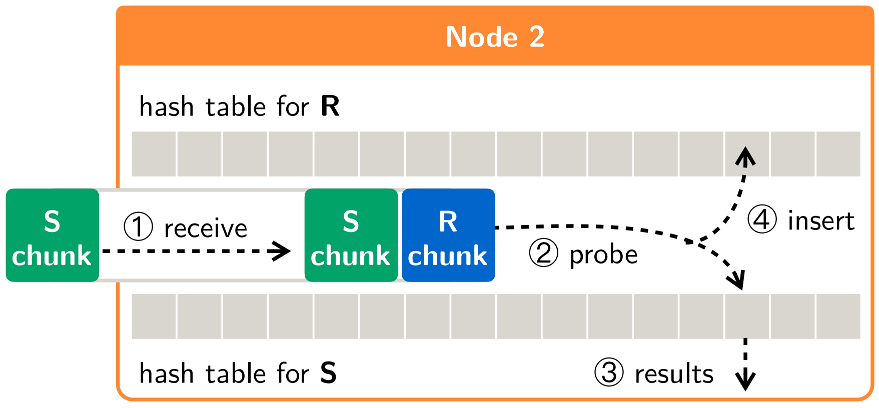


Figure 2.12: Symmetric hash join

choice for the local join. Consequently, we have so far focused on optimizing the data shuffling. Still, there are several choices for the local join.

The *standard hash join* [21] first builds a hash table for the smaller input, which is then probed with the larger input. This two-phase approach effectively blocks for the probe input. In a distributed join, the probe input needs to be cached until after the entire build input has been received and processed. Only then can probing of the hash table start to produce result tuples. This is visible as a steep incline in the result size about 20 s into the join computation.

The *symmetric hash join* [83], a variant thereof known as XJoin [81], consists of only one phase: An incoming tuple is first probed into the hash table of the other input and then added to the hash table of its own input as shown in Figure 2.12. The symmetric hash join processes each tuple immediately, thereby avoids to postpone work, and still computes the correct result. It finishes at the same time as the standard hash join even though it processes every tuple twice, once for each hash table. The network transfer dominates the runtime to such a large extent that this additional work has no impact. The continuous processing of incoming tuples is reflected in the steady incline in the number of result tuples as shown in Figure 2.11.

The *partitioned hash join* is a third hash join variant that performs no more work than the standard hash join and can produce results earlier. It maintains a hash table for every partition and can probe those hash tables that are fully built, while build tuples for other partitions are still missing. This is evident in Figure 2.11 as result tuples are generated already 16 s into the join.

Neo-Join supports all three hash joins. It uses the partitioned hash join by default since it is able to perform its work earlier than the standard hash join. Further, the partitioned hash join processes every tuple only once in contrast to the symmetric hash join, which maintains two hash tables. This becomes important for inputs with high locality since Neo-Join reduces the network transfer time considerably—potentially even to zero. Partitioned and standard hash join are in this case up to twice as fast as the symmetric hash join. All subsequent experiments use the partitioned hash join.

2.3.5 Shuffling the Join Result

So far, we have not considered the cost for reshuffling a large join result in preparation for the next join or aggregation operator in the query plan. There is no need to redistribute the join result when the subsequent operator references the same attribute as the current join. In fact, locality-sensitive data shuffling identifies and exploits the resulting co-location automatically. In all other cases, reshuffling is necessary and can have a significant impact on the total query execution time. All techniques described in this chapter should be applied for the optimization of this additional data shuffling phase. A combined optimization of successive operators could result in further improvements. The assignment of partitions to nodes influences the result sizes on the nodes, which in turn determine the cost for shuffling the result.

Instead of a separate phase, the result could already be redistributed during the join computation. However, it is hard to estimate at which point in time result tuples are available at specific nodes. Moreover, the bandwidth of the nodes should already be fully utilized during the data shuffling for the join. The redistribution of the result could instead start after the data shuffling for the join has finished, yet possibly before the end of the join computation. This further increases the overlapping of communication and computation. However, in a network-bound system this situation will only arise when the data shuffling phase of the join has a short duration due to data co-location.

2.4 Handling Skew

We extend our locality-sensitive data shuffling approach to handle value skew and high numbers of duplicates while keeping the complexity of the partition assignment and thus the solve time low. Using the newly extended model for the partition assignment, Neo-Join can benefit from extreme value skew and reduce network communication further.

2.4.1 Adaptive Radix Partitioning

The optimal partition assignment as presented so far handles skewed inputs by balancing larger partitions with many smaller ones. This handles those cases quite

	Zipf factor z				
	0.00	0.25	0.5	0.75	1.00
16 partitions	27 s	24 s	23 s	29 s	44 s
512 partitions	23 s	23 s	23 s	23 s	33 s
16 partitions (ARP)	23 s	24 s	24 s	24 s	24 s
16 partitions (selective broadcast)	24 s	24 s	23 s	20 s	10 s
16 partitions (selective broadcast + ARP)	23 s	23 s	24 s	20 s	10 s

Table 2.2: Join performance for skewed inputs

well where for example 80 % of the data lies in the first 20 % of the value range. However, for extreme cases of value skew a considerable number of partitions is needed for a balanced partition assignment. This is highly undesirable as the runtime of the partition assignment increases with the number of partitions.

We extend radix partitioning to handle inputs with extreme value skew by adaptively combining small partitions, hence called adaptive radix partitioning (ARP). With ARP the nodes create histograms with many buckets. The partition assigner aggregates these into a global histogram and combines buckets that are smaller than a certain threshold. The resulting partitions are better balanced than with standard radix partitioning. Most importantly, the number of partitions used in the optimal partition assignment is kept small.

We evaluate ARP with the Zipf distribution, which is commonly used to model extreme cases of value skew and high numbers of duplicates. The Zipf factor $z \geq 0$ controls the extent of skew, where $z = 0$ corresponds to a uniform distribution. Given n elements ranked by their frequency, a Zipf distribution with skew factor z denotes that the most frequent item (rank 1) accounts for $x = 1/H_{(n,z)}$ of all values, where $H_{(n,z)} = \sum_{i=0}^{n} 1/i^z$ is the nth generalized harmonic number. The element with rank r occurs x/r^z times. Zipf is known to model real world data accurately, including the size of cities and word frequencies [38].

Our micro-benchmark consists of two relations, *city* and *person*, where person has a foreign key *hometown* referencing the city relation. Both relations contain 400 M tuples. The *hometown* attribute of the *person* table is skewed to model the fact that most persons live in few cities. We varied the Zipf factor z from 0 to 1. $z = 0$ corresponds to a uniform distribution as mentioned before, while $z = 1$ implies that 77 % of the values are in the first 1 % of the value range. Note that for $z = 1$ the number of duplicates is also quite high: the value 0 occurs in 21 M tuples (5 %), the value 1 in 10 M (2.4 %), the value 2 in 7 M (1.6 %), etc. n denotes the number of elements, in this case $n = 400$ M.

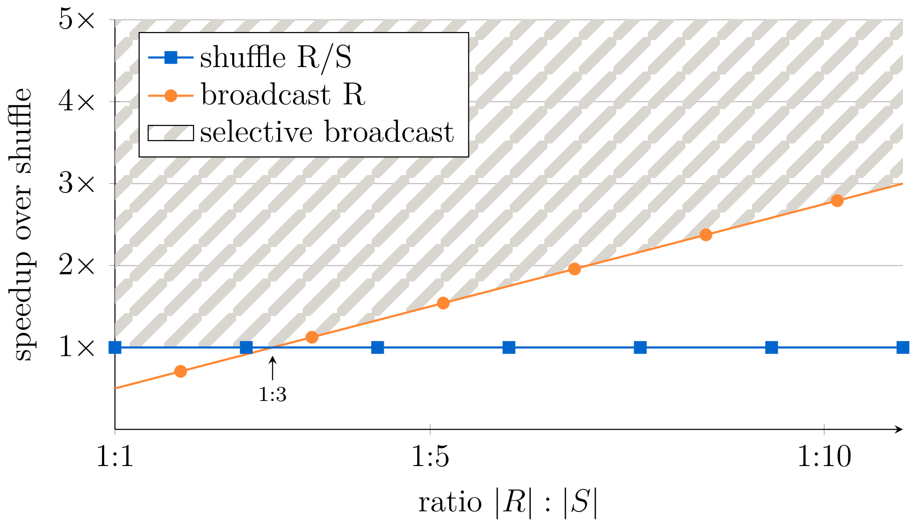


Figure 2.13: Selective broadcast outperforms broadcast and shuffle

Table 2.2 shows the join duration using 4 nodes for an increasingly skewed *person* table. With simple radix partitioning, even 512 partitions do not suffice to maintain the join duration for $z = 1$. On the other hand, adaptive radix partitioning needs only 16 partitions to sustain the duration of the uniform case. Since ARP produces better results for the same number of partitions, it should be used as a replacement for the standard radix partitioning we described in Section 2.3.1.

2.4.2 Selective Broadcast

Selective broadcast (SB) extends the optimal partition assignment model so that it can additionally decide for every partition whether to assign it to a node or broadcast one of its relation fragments instead. This achieves two things: First, it covers the case where one relation is significantly smaller than the other so that broadcasting it as a whole is more efficient than shuffling individual partitions. Second, the ability to decide between broadcast and shuffle for every single partition is highly beneficial for skewed inputs, as we will show in the following.

Shuffle or Broadcast

There are two fundamental options for distributed joins: (i) shuffle both relations so that tuples with the same key end up on the same node, or (ii) let one relation remain fragmented across the system and broadcast the other—also known as the fragment-replicate join [24]. Shuffling relations R and S incurs communication costs of $\frac{n-1}{n} \cdot (|R| + |S|)/n$ as each of the n nodes sends $1/n$-th of its fragments of

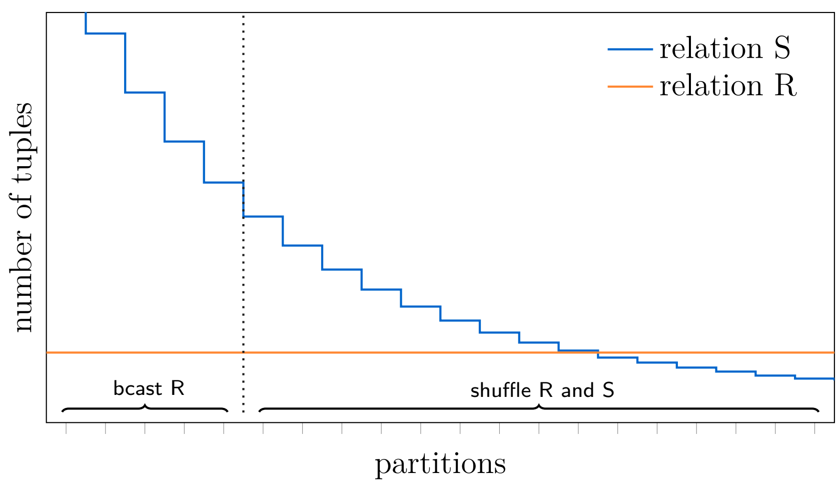


Figure 2.14: Selective broadcast adapts to skewed inputs

R and S to the other $n-1$ nodes. Broadcasting R costs $(n-1) \cdot |R|/n$ as every node sends its fragment of R to every other node. Broadcast thus performs better than shuffling when the ratio between relations is higher than $1 : (n-1)$.

The locality-sensitive data shuffling as presented so far only considers shuffling. Selective broadcast extends our approach so that it decides for every partition whether use shuffling or broadcast. In particular, this covers the case where one relation is much smaller than the other and should be broadcast as a whole. As a consequence, selective broadcast performs always at least as good as shuffling both relations or broadcasting the smaller as illustrated in Figure 2.13 for 4 nodes. Selective broadcast outperforms broadcast and shuffle for inputs with value skew as we explain in the next section.

Skew

Selective broadcast can lead to significant speed-ups in the case of highly skewed inputs. It broadcasts those partition fragments of one relation that are significantly smaller than their counterpart of the other relation. The remaining partitions are either broadcast by the other relation or assigned to nodes as before. Figure 2.14 illustrates this for two relations R and S where S is skewed towards small values. The first five partitions of R are broadcast as the corresponding partitions of S are much larger. The remaining partitions are assigned to nodes and shuffled as before. This has the additional benefit that partitions are kept local that are large due to a high number of duplicates, which is common for Zipf distributions. This avoids that all the duplicate values are assigned to a single node, which then has to process a much larger part of the input than the other nodes.

To show the potential savings with selective broadcast, we revisit the example of the *city* and *person* relations. For Zipf factor $z = 1$ and 16 partitions, the join performance improves by a factor of 2.8 due to the use of selective broadcast compared to the non-skewed case $z = 0$. In particular, the first three partitions are broadcast by *city*, the next four partitions are shuffled, while the remaining nine partitions are broadcast by *person*.

Model Extension

In the following, we describe how to extend the mixed-integer linear program of the partition assignment to support selective broadcast.

As before, we use h_j^R and h_j^S to denote all tuples of relation R and S that belong to partition j. Consider the simple case where all nodes i have equal-sized fragments h_{ij} of partition j for relations R and S (we assume $h_j^R < h_j^S$): We should broadcast the fragment of partition j for relation R instead of assigning the complete partition j to a node, whenever shuffling the fragments is more expensive than a broadcast:

$$h_j^R + h_j^S > n \cdot h_j^R$$

We have to model the choice between assigning a partition to a node or broadcasting it by either relation S or relation R. The existing binary variables $x_{ij} \in \{0, 1\}$ specify whether partition j is assigned to node i. We add two new variables $y_j, z_j \in \{0, 1\}$ per partition j that denote if its fragment for relation R respectively S is broadcast instead.

The constraints have to be updated using the new variables. Previously, the model included the restriction that every partition has to be assigned to exactly one node (cf., Equation 2.1). In the new model, each partition j is either assigned to a node or broadcast for one of the two relations. All variables that refer to the same partition therefore have to be mutually exclusive:

$$\left(\sum_{i=0}^{n-1} x_{ij}\right) + y_j + z_j = 1, \qquad \text{for } 0 \leq j < p - 1 \tag{2.6}$$

The constraints for the send and receive costs have to be updated as well. The send cost r_i of node i was previously defined as the sum of all partitions it has to send because they were assigned to other nodes (cf., Equation 2.3). For selective broadcast, one has to account for the additional cost for partitions that are broadcast instead: If a partition is broadcast, all nodes that own a relation fragment of this partition have to send it to all $n - 1$ other nodes. The additional send cost of node i for broadcasts s_i^B is therefore:

$$s_i^B = \sum_{j=0}^{p-1} \left(y_j (n-1) h_{ij}^R + z_j (n-1) h_{ij}^S \right) \qquad \text{for } 0 \leq i < n \tag{2.7}$$

where h_{ij}^R and h_{ij}^S denote the size of the relation fragments of partition j at node i for R and S, respectively. In addition to the receive cost r_i for partitions that were assigned to node i (cf., Equation 2.4), it also receives all relation fragments from the other nodes for partitions that are broadcast. The additional receive costs of node i for broadcasts r_i^B are:

$$r_i^B = \sum_{j=0}^{p-1} \left(y_j \sum_{k=0, i \neq k}^{n-1} h_{kj}^R + z_j \sum_{k=0, i \neq k}^{n-1} h_{kj}^S \right) \qquad \text{for } 0 \leq i < n \tag{2.8}$$

The objective (cf., Equation 2.2) remains unchanged, the program still minimize the maximum send or receive cost across all nodes. One can now update the linear program for the partition assignment with equations 2.6-2.8 to support selective broadcasts, including an additional small adjustment to the send cost:

$$\text{(SEL-BCAST)} \qquad \begin{array}{lll} & \textbf{minimize } \boldsymbol{w}\textbf{, subject to} & \\ & w \geq s_i + s_i^B - \sum_{j=0}^{p-1} (y_j h_{ij} + z_j h_{ij}) & 0 \leq i < n \\ & w \geq r_i + r_i^B & 0 \leq i < n \\ & 1 = \left(\sum_{i=0}^{n-1} x_{ij} \right) + y_j + z_j & 0 \leq j < p \end{array}$$

The runtime of the partition assignment increases by an average of 39 % with selective broadcast enabled (comparing the geometric mean of 720 experiments that cover combinations of 8 to 64 nodes, 4 to 16 partitions, and 20 levels of locality using 3 repetitions).

2.5 Evaluation

All experiments of this chapter were conducted on a shared-nothing [75] cluster of four identical machines except for the scale up experiment, which uses 16 nodes with older hardware. The cluster is connected via Gigabit Ethernet. Each of the four nodes has 32 GiB of RAM, an Intel Core i7-3770 processor with four cores at 3.4 GHz each. The machines run Linux 3.8 as operating system. The implementation links to the IBM CPLEX library for solving linear programs.

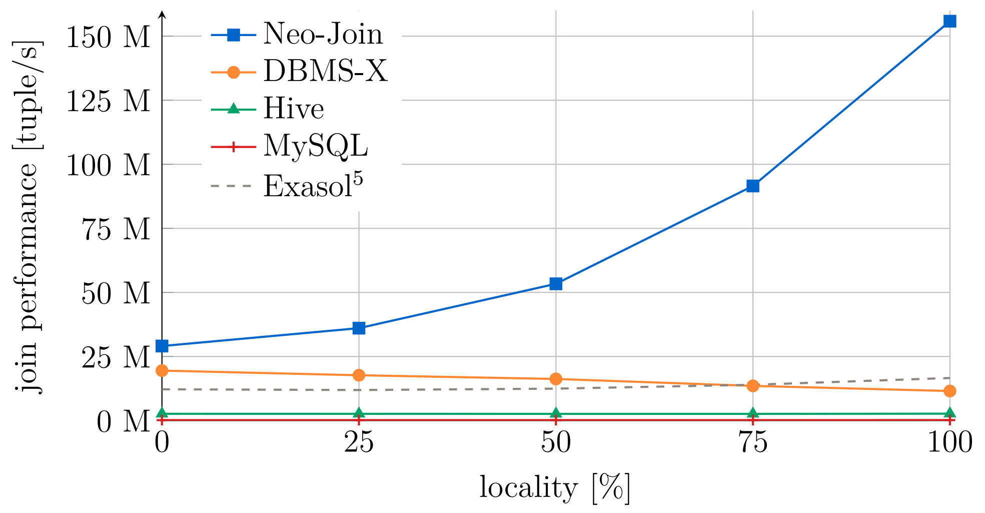


Figure 2.15: Join performance of several distributed SQL systems

2.5.1 Locality

Figure 2.15 compares Neo-Join to MySQL Cluster 7.2.10, Hive 0.10 on a Hadoop 1.1.1 cluster, and the commercial system DBMS-X for five different levels of locality. MySQL Cluster is a distributed variant of the MySQL database, which uses main memory as storage. Hive is a data warehouse system based on Hadoop, the open source implementation of MapReduce. For better comparability, we use the in-memory file system `ramfs` instead of a disk and tuned the number of map/reduce tasks. DBMS-X is a disk-based column store, which is configured likewise to use main memory as storage. However, whether disk or main memory were used did not result in noticeable differences in performance, which indicates that indeed the network bandwidth is the limiting factor. All systems were configured to use three data nodes and one coordinator.

The experiments use a total of 600 M tuples (100 M tuples per relation per node). Each tuple consists of a key and a payload, both defined as `DECIMAL(18, 0)`, which fits into a 64 bit integer. Neo-Join is in general independent of the data layout, but we implemented it for a row-store. The join key is constrained to $[0, 2^{32})$ so that the join result is not nearly empty as is the case for 64 bit uniform join keys. We varied the locality, where $x\,\%$ denotes that for each partition the node with its largest part owns $x\,\%$ of its tuples with an additional $1/n$-th of the remaining tuples for this partition (for n nodes). $0\,\%$ thus corresponds to a

[5]Exasol, current record holder in the TPC-H cluster benchmark, was measured with the same input on a different cluster with slightly lower CPU performance but the same network bandwidth—which is the bottleneck for distributed joins.

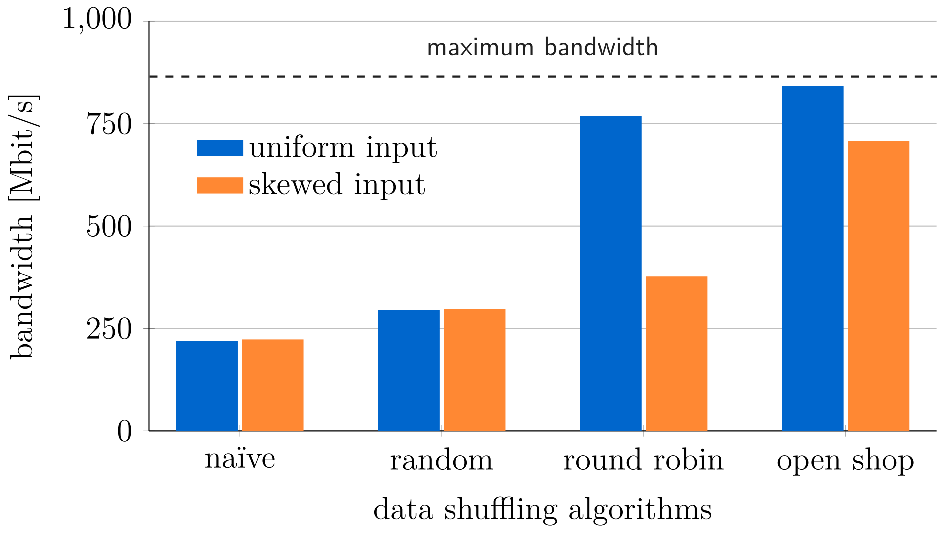


Figure 2.16: Throughput of data shuffling algorithms

uniform distribution where all nodes own equal parts of all partitions and 100 % to the other extreme where nodes own partitions exclusively.

MySQL takes an hour to compute the join on uniform data while Hive finishes in 4 min, DBMS-X in 30 s, and Neo-Join in 20 s. MySQL and Hive perform similar independent of locality while DBMS-X's performance deteriorates by 41 %. Neo-Join benefits significantly from increasing locality improving join performance by a factor of 5 from 30 M to more than 150 M tuples/s. These results emphasize the importance of locality-sensitive data shuffling. No contender was able to take advantage from locality.

2.5.2 Data Shuffling Alternatives

Figure 2.16 compares the bandwidth utilization of four different data shuffling schemes for uniform and skewed inputs on 4 nodes. For the naïve scheme all nodes first send to node 0, followed by node 1, and so on. The random data shuffling scheme selects targets at random. Both the naïve and the random scheme are not synchronized as this decreases their performance. The round-robin scheme orders the nodes in a cycle. Each node sends first to its clock-wise neighbor, then to the one after that, etc. The phases are synchronized to avoid cross traffic. Open shop is our data shuffling scheme, which is based on an open shop schedule and is synchronized similar to the round-robin scheme.

The naïve and random distribution scheme perform similar for both the uniform and the skewed input. The bandwidth utilization of round-robin deteriorates for the skewed input. The node that sends the most data determines the duration of

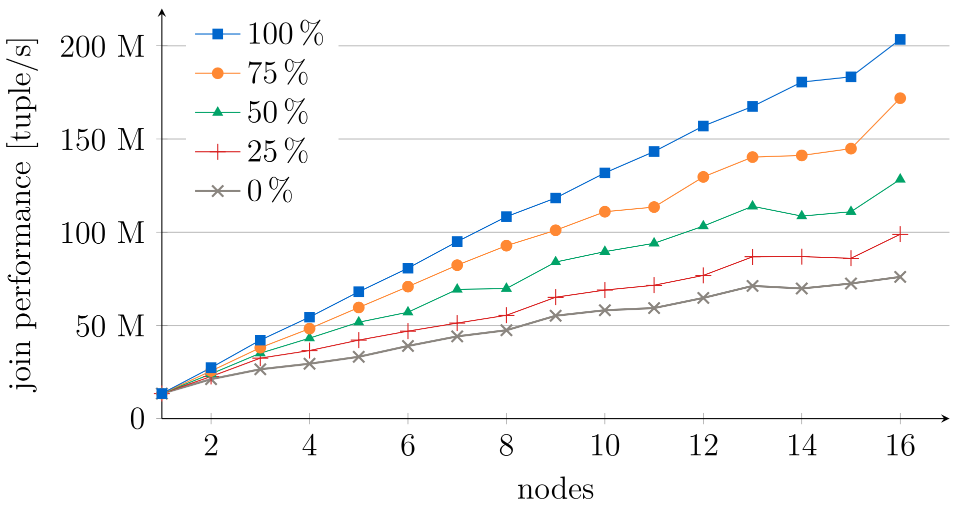


Figure 2.17: Scale up results for Neo-Join

a phase which is disadvantageous when the nodes send different amounts of data. Open Shop handles the skewed input better due to its optimal communication schedule.

2.5.3 Scale Up

The scale up experiment was conducted on a cluster of 16 nodes. Each node has an Intel Core 2 Quad Q6700 CPU with four cores at 2.66 GHz and 8 GiB of main memory. They are connected over Gigabit Ethernet—still the dominant connection speed for new servers [42]. The nodes are somewhat aged as visible in the rather inferior single-node join performance. Note that faster CPUs would also improve the join performance for joins with high locality.

A linear scale up is defined as a linear increase in join performance when the number of nodes and the size of the input is increased proportionally. In this case, the input is increased by 100 M tuples for every additional node. Figure 2.17 demonstrates that Neo-Join scales almost linearly with the number of nodes. Moreover, it shows that locality-aware data-shuffling still scales when there is no locality in the data, despite of the increasing overhead for partition assignment and network scheduling.

2.5.4 TPC-H

We chose the TPC-H benchmark to test our approach with a more realistic data set. We generated the relations for a scale factor of 100 and split them into four parts, one per node. The resulting data set has about 100 GiB. We compare our

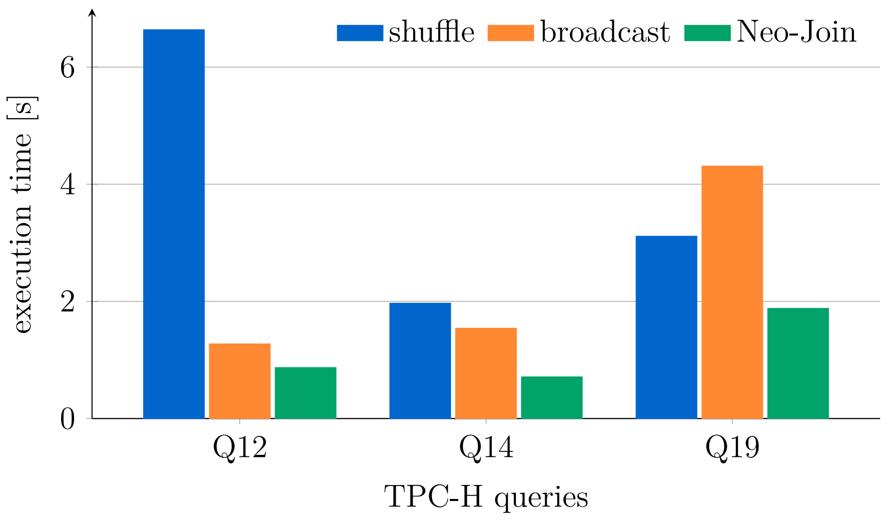


Figure 2.18: Selected TPC-H results for Neo-Join

approach to a hash-based shuffle of both relations and a broadcast of the smaller relation. These are the two state-of-the-art choices for the data shuffling phase of a distributed query. The results for the selected single-join queries are shown in Figure 2.18.

The selection predicate of Q12 is so restrictive that one of the join inputs is 45× larger than the other. Consequently, a broadcast of the smaller relation is much faster than shuffling both relations. Neo-Join still improves over this by exploiting the near-perfect co-location of the *orders* and *lineitem* tables caused by time-of-creation clustering. Note that Neo-Join does not assign the few partitions that violate the co-partitioning but instead selectively broadcasts them by the smaller relation, which is even faster. It achieves a speedup of 7.6× over shuffling and 1.5× over broadcasting. For Q14, Neo-Join is able to exploit the time-of-creation clustering of the *part* relation and repartitions the *lineitem* table, which exhibits no locality on the join attribute. This improves the execution time by a factor of 3.4 over shuffling respectively 1.2 for broadcast. The size of the input relations for Q19 differ only by 55 %, thus shuffling becomes faster than broadcast. Neo-Join again exploits the locality in the data placement of the *part* relation and is 1.7× faster than a shuffle and 2.3× faster than a broadcast.

2.6 Concluding Remarks

Over the last decades, compute speed has grown much faster than network speed, at least for commodity hardware commonly used in data centers such as Amazon EC2 and the Google Compute Engine. In such a setting, parallel main-memory database systems need to utilize locality in the data placement to speed up query processing. A common technique is to co-partition relations during schema design to reduce expensive data shuffling. However, co-partitioning is restricted to one attribute per relation (unless it is also being replicated) and expensive to maintain under updates. Other attributes often exhibit a *fuzzy* co-location or time-of-creation clustering but current approaches do not leverage this.

In this chapter, we have introduced *locality-sensitive data shuffling*, a set of four techniques that can automatically exploit these characteristics of the workload to dramatically reduce the amount of network communication of distributed operators. We have presented four novel techniques: (i) *optimal partition assignment* computes an assignment with minimum network phase duration given any repartitioning of the input while considering locality, skew, and the case that some nodes own larger parts of a relation than others; (ii) *communication scheduling* leverages all the available network bandwidth in a cluster; (iii) *adaptive radix partitioning* retains locality in the data and handles value skew gracefully; and (iv) *selective broadcast* allows to reduce network communication for cases with extreme value skew by dynamically deciding whether to shuffle or broadcast a partition. We have presented comprehensive experimental results, which show that our approach can improve performance by up to a factor of 5 for fuzzy co-location and a factor of 3 for inputs with value skew.

Chapter 3

High-Speed Networks

Parts of this chapter were previously published in [70].

Modern clusters entail two levels of networks: connecting CPUs and NUMA regions inside servers in the small and multiple servers in the large. The huge performance gap between these two types of networks slowed down distributed query processing to such an extent that a cluster performed in fact worse than a single server. The increased main-memory capacity of the cluster remained the sole benefit of such a scale-out.

The economic viability of high-speed interconnects such as InfiniBand has narrowed this performance gap considerably. However, InfiniBand's higher bandwidth alone does not improve query performance as expected when the distributed query processing engine is left unchanged. The scalability of distributed query processing is impaired by TCP overheads, switch contention due to uncoordinated network communication, and load imbalances resulting from the inflexibility of the classic exchange operator model.

This chapter presents the blueprint for a distributed query engine that addresses these problems by considering both levels of networks holistically. It consists of two parts: First, *hybrid parallelism*, which distinguishes between local and distributed parallelism to scale query performance both with the number of cores per server as well as the number of servers in the cluster. Second, a *novel communication multiplexer* tailored for analytical database workloads using remote direct memory access (RDMA) and low-latency network scheduling for high-speed communication with almost no CPU overhead. An extensive evaluation within the HyPer in-memory database system system using the renowned TPC-H ad-hoc analytical benchmark shows that our holistic approach enables *high-speed query processing over high-speed networks.*

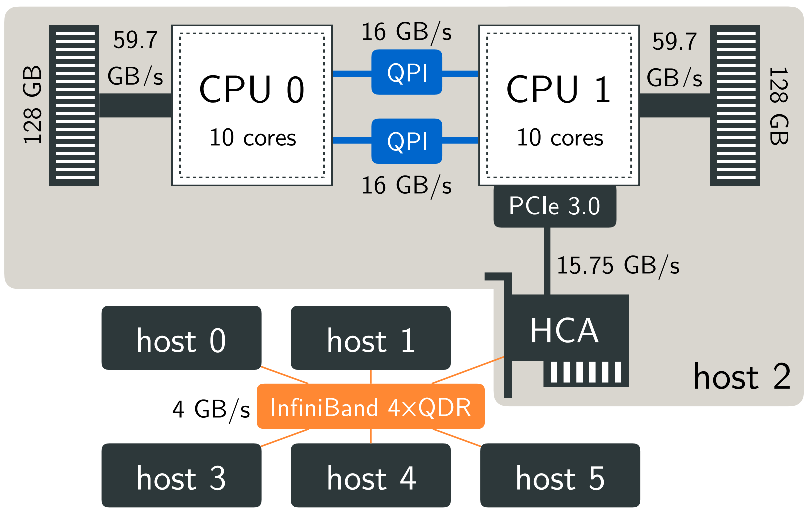


Figure 3.1: Compute cluster with two levels of networks

3.1 Motivation

In-memory database systems have gained increasing interest in academia and industry over the last years. The success of academic projects, including MonetDB [55] and HyPer [45], has led to the development of commercial main-memory database systems such as Vectorwise [88], SAP HANA [25], Oracle Exalytics [33], IBM DB2 BLU [67], and Microsoft Apollo.

This development is driven by a significant change in the hardware landscape: Today's many-core servers often have main-memory capacities of several terabytes. The advent of these brawny servers enables unprecedented single-server query performance. Moreover, a small cluster of such servers is often already sufficient for companies to analyze their business. For example, Walmart—the world's largest company by revenue—uses a cluster of only 16 servers with 64 TiB of main memory to analyze their business data [63].

Such a cluster entails two levels of networks as highlighted in Figure 3.1: The network in the small connects several many-core CPUs and their local main memory inside a single server via a high-speed QPI interconnect, while the network in the large connects separate servers. Main-memory database systems have to efficiently parallelize query execution across these many cores and adapt to the non-uniform memory architecture (NUMA) inside servers to avoid the high cost of remote memory accesses [53, 52]. Traditionally, exchange operators are used to introduce parallelism both locally inside a single server as well as globally between servers. However, the inflexibility of the classic exchange operator model introduces several scalability problems. We propose a new *hybrid approach* instead, that

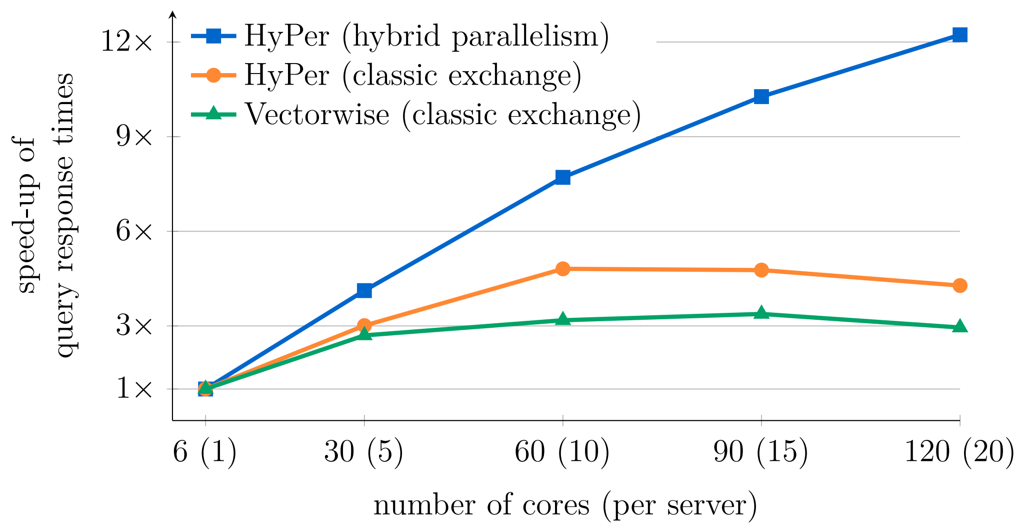


Figure 3.2: TPC-H results for different parallelism paradigms

combines special decoupled exchange operators for distributed processing with the existing intra-server morsel-driven parallelism [52] for local processing. Choosing the parallelism paradigm for each level that fits best, hybrid parallelism scales better with the number of cores per server than classic exchange operators as demonstrated by Figure 3.2. The experiment executes TPC-H with a scale factor 300 data set on a 6-server cluster for the database systems HyPer and Vectorwise Vortex while increasing the number of cores per server.

In the past, limited network bandwidth between servers actually *reduced* query performance when scaling out to a cluster. Consequently, previous research focused on techniques that avoid communication as much as possible [71, 65]. In the meantime, high-speed networks such as InfiniBand have become economically viable, offering link speeds of several gigabytes per second. However, faster networking hardware alone is not enough to scale query performance with the cluster size. Similar to the transition from disk to main memory, new bottlenecks surface when InfiniBand replaces Gigabit Ethernet. TCP/IP processing overheads and switch contention threaten the scalability of distributed query processing. Figure 3.3 demonstrates this by comparing two distributed query engines using the TPC-H benchmark. Both engines are implemented in our in-memory database system HyPer. The first uses traditional TCP/IP, while the second is built with remote direct memory access (RDMA). The experiment adds servers to the cluster while keeping the data set size fixed at scale factor 100. Using Gigabit Ethernet actually decreases performance by 6× compared to using just a single server of the cluster. The insufficient network bandwidth slows down query processing. Still, a scale out is inevitable once the data exceeds the main-memory capacity of a single

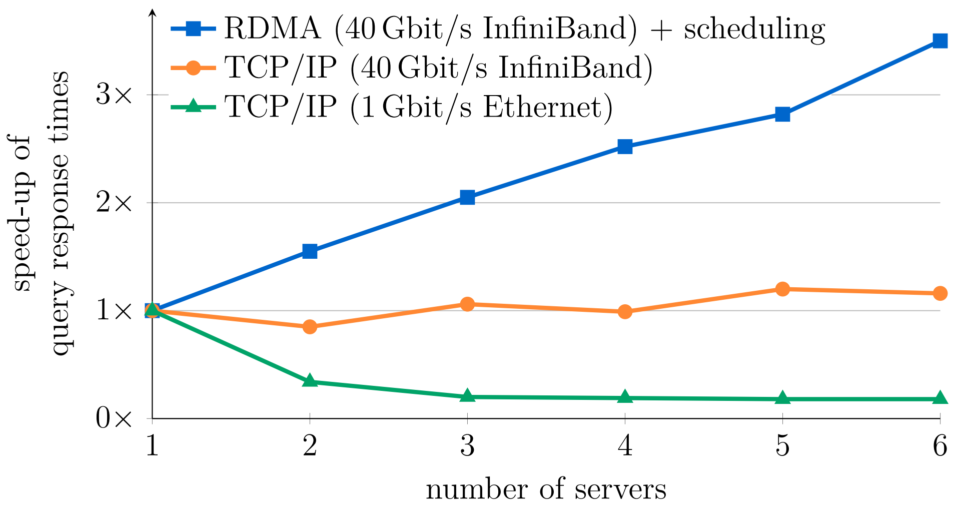


Figure 3.3: TPC-H results for HyPer's TCP and RDMA query engines

server. InfiniBand 4×QDR offers 32× the bandwidth of Gigabit Ethernet. However, Figure 3.3 also shows that simply using faster networking hardware is not enough. The distributed query engine has to be adapted to avoid TCP/IP stack processing overheads and switch contention. By combining RDMA and network scheduling in our novel distributed query engine we can scale query performance with the cluster size, achieving a speedup of 3.5× for 6 servers.

RDMA enables true zero-copy transfers at almost no CPU cost. Recent research has shown the benefits of RDMA for specific operators (e.g., joins [5]) and key-value stores [43]. However, we are the first to present the design and implementation of a complete distributed query engine based on RDMA that is able to process complex analytical workloads such as the TPC-H benchmark. In particular, this chapter makes the following contributions:

1. *Hybrid parallelism:* A NUMA-aware distributed query execution engine that integrates seamlessly with intra-server morsel-driven parallelism, scaling considerably better both with the number of cores as well as servers compared to standard exchange operators.

2. A novel *RDMA-based communication multiplexer* tailored for analytical database workloads that utilizes all the available bandwidth of high-speed interconnects with minimal CPU overhead; it avoids switch contention via low-latency network scheduling, improving all-to-all communication throughput by up to 40 % for an 8-server cluster.

3. A prototypical implementation of our approach in our full-fledged in-memory DBMS HyPer that scales both in the number of cores as well as servers.

	GbE	InfiniBand (4×)				
		SDR	DDR	**QDR**	FDR	EDR
throughput [GB/s]	0.125	1	2	**4**	6.8	12.1
latency [µs]	340	5	2.5	**1.3**	0.7	0.5
introduction	1998	2003	2005	**2007**	2011	2014

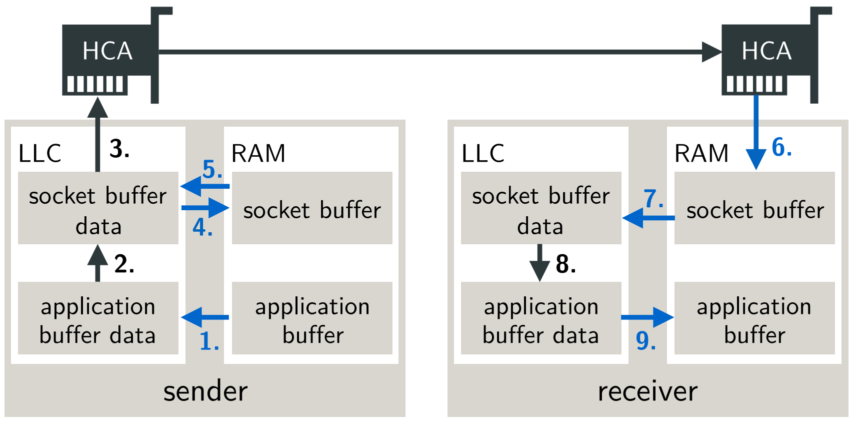
Table 3.1: Network data link standards

Section 3.2 evaluates high-speed cluster interconnects for typical analytical database workloads. Specifically, we study how to optimally configure the TCP and RDMA transport protocols for expensive all-to-all data shuffles that are common for distributed joins and aggregations. Building upon these findings, Section 3.3 presents a blueprint for our novel distributed query engine that is carefully tailored for both the network in the small and in the large. It consists of *hybrid parallelism* for improved scalability in both the number of cores and servers as well as our *optimized communication multiplexer* that combines RDMA and low-latency network scheduling for high-speed communication. Finally, Section 3.5 provides a comprehensive performance evaluation using the ad-hoc OLAP benchmark TPC-H, comparing a prototypical implementation of our approach within our full-fledged main-memory database system HyPer to several SQL-on-Hadoop as well as parallel main-memory database systems: HyPer improves TPC-H performance by 1421× compared to Apache Hive, 256× compared to Spark SQL, 168× to Cloudera Impala, 38× to MemSQL, and 5.4× to Vectorwise Vortex.

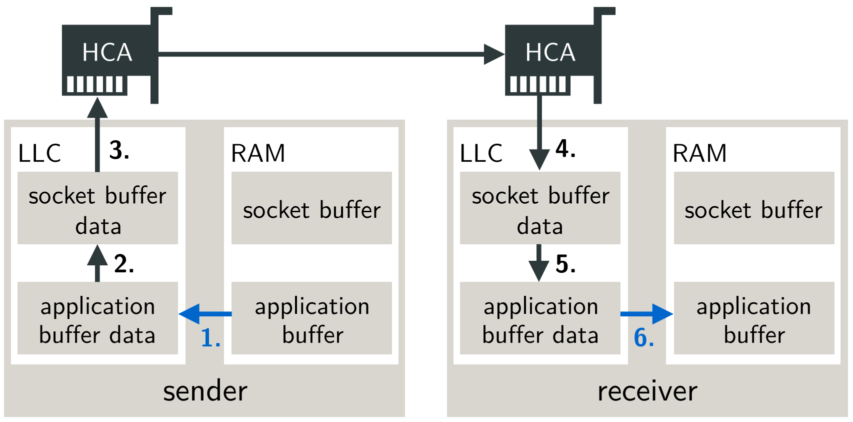
3.2 High-Speed Networks

InfiniBand is a high-bandwidth and low-latency cluster interconnect. Table 3.1 compares several InfiniBand data rates to Gigabit Ethernet (GbE). The following performance study uses InfiniBand 4×QDR hardware that offers 32× the bandwidth of GbE and latencies as low as 1.3 µs. We expect the findings to be valid for the faster data rates as well.

InfiniBand offers the choice between two transport protocols: TCP via *IP over InfiniBand* (IPoIB) and the native *ibverbs* interface for remote direct memory access (RDMA). In the following, we analyze and tune both protocols for analytical database workloads that require massive data shuffling for distributed joins and aggregations. In contrast, transactional database workloads typically involve much smaller messages and would thus shift the tuning target from high throughput to low latencies.



(a) Classic I/O involves three memory trips at sender/receiver



(b) Data direct I/O reduces this to only one memory trip each

Figure 3.4: TCP memory bus traffic for classic and data direct I/O

3.2.1 TCP

Existing applications that use TCP or UDP for communication can utilize InfiniBand via IPoIB. It is a convenient option to increase the network bandwidth for applications without changing their implementation.

Data Direct I/O and Non-Uniform I/O Access

Since the standardization of TCP in 1981 as RFC 793 the bandwidth provided by the networking hardware increased by several orders of magnitude. Yet, the socket interface still relies on the fact that message data is copied between application buffer and socket buffer [27]. The resulting multiple trips over the memory bus were identified as one of the main reasons hindering TCP scalability [13, 27, 28]. However, we noticed during our experiments that this is no longer the case for modern systems. Indeed, the number of memory trips required by TCP and similar protocols was reduced significantly when Intel introduced data direct I/O (DDIO) in 2012 with its Sandy Bridge processors. DDIO allows the I/O subsystem to directly access the last level cache of the CPU for network transfers. DDIO has no hardware dependencies and is invisible to drivers and software.

Figure 3.4a and 3.4b show the memory trips performed by the classic I/O model and data direct I/O, respectively. At the sender, classic I/O (1) reads the data from application buffer into the last level cache (LLC), (2) copies it into the socket buffer, and (3) sends it over the network forwarded from the LLC, which causes (4) cache eviction and (5) a speculative read. At the receiver, the data is (6) DMAed to the socket buffer in RAM, (7) copied into LLC, (8) copied into the application buffer, and (9) written to RAM.

DDIO instead targets the last level cache (LLC) of the CPU directly. At the sender, DDIO (1) reads the application data, (2) copies it into the socket buffer, and (3) sends it directly from LLC. At the receiver, the data is (4) allocated or overwritten in the LLC via Write Allocate/Update (restricted to 10 % of the LLC capacity to reduce cache pollution), (5) copied into the application buffer, and (6) written to main memory. DDIO reduces the number of memory bus transfers from 3 to 1 compared to the classic I/O model.

NUMA systems add a complication: A network adapter is directly connected to one of the CPUs. Consequently, there is a difference between local and remote I/O consumption. This is called Non-Uniform I/O Access (NUIOA). NUIOA is a direct consequence of the multi-CPU architecture of modern many-core servers. NUIOA systems restrict DDIO to threads running on the CPU local to the network card. We validated this by measuring the memory bus traffic in a micro-benchmark using Intel Performance Counter Monitor (PCM)[1]: Running the network thread on the local NUMA node caused every byte to be read 1.03× on the sender side and written 1.02× on the receiver side. Running the network thread on the remote NUMA node read every byte 2.11× for the sender while on the receiver side it was read 1.5× and written 2.33× (overheads might be due to TCP control traffic, retransmissions, cache invalidations, and prefetching [27]). This demonstrates that DDIO was only active for the NUMA-local thread running on the NUIOA-local CPU. Accordingly, our distributed query engine pins the network thread to the NUIOA-local CPU to avoid extra trips over the memory bus. This enables the I/O system to directly target the cache, reducing the number of memory trips for TCP from 3 to 1 and also further reduces the already lower memory bus traffic of RDMA.

[1]Intel Performance Counter Monitor (PCM) enables access to core and uncore performance monitoring units: `http://www.intel.com/software/pcm`

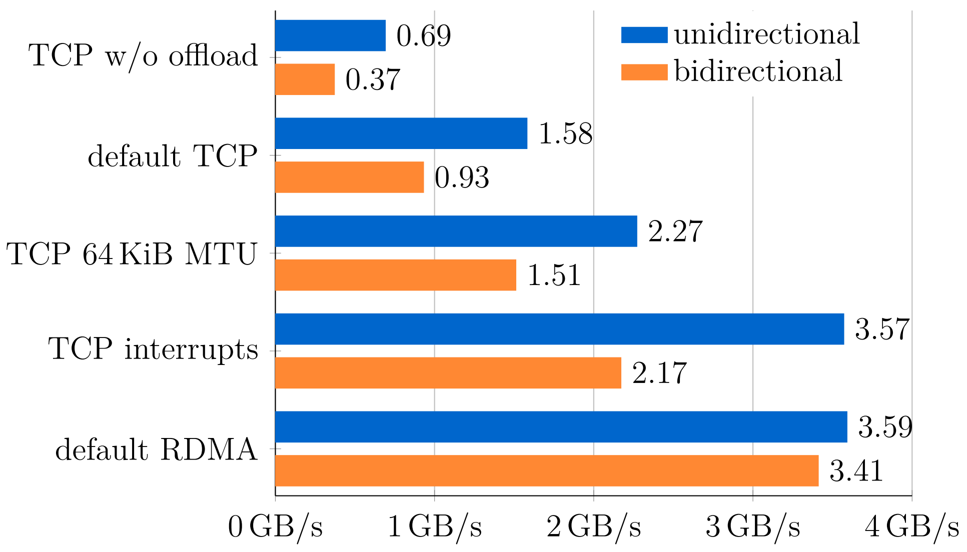


Figure 3.5: Uni- and bidirectional throughput for TCP and RDMA

Tuning TCP for Analytical Workloads

We designed a micro-benchmark that compares TCP with RDMA for database workloads. The performance of TCP is limited by different bottlenecks depending on the size of a message. For small messages, processing time is dominated by kernel overheads, sockets and protocol processing. For bulk transfers, data touching (i.e., checksums and copy) and interrupt processing account for most of the processing time [27]. Analytical query processing transfers large chunks of tuples during distributed joins and aggregations, we will thus focus on tuning TCP throughput for large packets, reducing the per-byte cost.

Our micro-benchmark sends 100k distinct messages of size 512 KiB between two machines using a single thread. From a variety of TCP options that should improve performance only SACK gave a measurable improvement. SACK enables fast recovery from packet loss, which is especially relevant for high-speed links. In a first experiment we transfer data only from sender to receiver, while in the second we use fully duplex communication. The results are shown in Figure 3.5.

The original specification of IPoIB in RFC 4391 and 4392 was limited to the datagram mode. This mode supports a 2044 byte MTU, TCP offloading to the network card, and IP multicast. The connected mode was later added in RFC 4755. While it allows a MTU of up to 65 520 bytes, it does not support TCP offloading or IP multicasting. Disabling TCP offloading in datagram mode decreases the throughput for bidirectional transfer by 60 % from 0.93 GB/s to 0.37 GB/s. The connected mode with the same MTU of 2044 bytes and without support for TCP offloading performs similar at 0.38 GB/s, as one might expect. However, the larger

MTU of 65 520 bytes available in connected mode more than offsets the missing TCP offloading features. The large MTU increases the throughput to 1.51 GB/s, an improvement of 62 % over datagram mode with offloading.

An important bottleneck for TCP is interrupt handling. The network card issues interrupt requests (IRQ) to which the kernel responds by executing an interrupt handler on a configured core. The kernel automatically schedules the network thread to this same core to reduce cache misses. However, TCP throughput increases by a further 44 % to 2.17 GB/s when the network thread is explicitly pinned to a different core. While this improves performance it also adds to the CPU overhead as now two cores are used. We further investigated the impact of NUIOA on TCP throughput. In our micro-benchmark, pinning the network thread to the local socket improves throughput by 15 % in datagram and 6 % in connected mode for bidirectional transfers. The interrupt handler should always run on the socket of the network thread as otherwise throughput drops by 50 %.

The bottleneck of TCP remains the CPU at the receiver. Receive and send thread as well as the interrupt handler add a significant CPU overhead. The receiver experiences 100 %–190 % CPU utilization for the unidirectional transfers. The peak of 190 % (i.e., two occupied cores) is reached in datagram mode when network thread and interrupt handler are pinned to different cores.

3.2.2 RDMA

RDMA is InfiniBand's asynchronous, zero-copy transport protocol that bypasses the CPU and thus frees resources for application processing.

Asynchronous Operation

InfiniBand's *ibverbs* interface is inherently asynchronous. Work requests are posted to send and receive work queues of the InfiniBand host channel adapter (HCA). The HCA processes work requests asynchronously and adds a work completion to a completion queue once it has finished. The application can process completion notifications when it sees fit to do so. This asynchronous interface makes overlapping of communication and computation easier than for TCP, which would require two threads or the use of non-blocking sockets.

Kernel Bypassing

The InfiniBand HCA reads and writes main memory directly without interacting with the operating system or application during transfers. This avoids the overhead of system calls and the copying between application and system buffers. Consequently, the application has to manage buffers explicitly. For this purpose, RDMA introduces the concept of a memory region. A memory region provides the mapping between virtual and physical addresses so that the HCA can access main memory at any time without involving the kernel. Memory regions have to be registered beforehand to pin the memory and avoid swapping to disk. Registering memory regions is a time-consuming operation [28] and regions should thus be reused whenever possible. Our distributed query engine implements this via a message pool.

Memory vs. Channel Semantics

RDMA allows to remotely read and write the main memory of a remote server without involving its CPU. These so-called *memory semantics* requires that the initiator of the remote read or write has the memory key for the target memory region. A separate channel is required to exchange memory keys before communication can start. The alternative are *channel semantics* via two-sided send and receive operations. The receiver posts receive work requests that specify the target memory region for the next incoming message. This eliminates the requirement to exchange memory keys before the transfer can start. There is no performance difference between one- and two-sided operations [28]. An application should choose the semantics that fit best.

For our distributed query engine, it makes sense to use two-sided operations (channel semantics). First, two-sided operations do not require a separate communication channel to exchange memory keys. Second, the receiver is notified when new messages arrive and can process incoming tuples right away. One-sided operations (memory semantics) do not involve the receiver in the transfer at all. Making the receiver aware about incoming messages would thus require a separate communication channel or busy polling.

Polling vs. Events for Completion Notifications

RDMA with channel semantics provides two mechanisms to check for the availability of new messages. The first uses busy polling to check for new completion notifications. While this guarantees lowest latency it also occupies one core to 100 %. The second mechanism uses events to signal new completion notifications. The HCA raises an interrupt when a new message arrives and wakes threads that are waiting for this event. The event-based handling of completion notifications reduces the CPU overhead to a mere 4 % for a full-speed bidirectional transfer with 512 KiB messages at the cost of a potentially higher latency compared to polling. Fortunately, the latency increase is insignificant for analytical database workloads with large messages.

3.2.3 Discussion

In the previous sections we discussed how to tune the TCP and RDMA transport protocols for analytical database workloads that shuffle large amounts of data between the servers of a cluster. This stands in contrast to transactional database workloads that typically involve much smaller messages and thus shift the focus from achieving maximal throughput to minimizing the latency.

While it is possible to bring TCP's throughput closer to that of RDMA via careful parameter tuning, this comes at the cost of significantly increased CPU load already for a single communication stream (100 % to 190 % for TCP compared to 4 % for RDMA) and would further require multiple TCP streams to use all the available network bandwidth. RDMA is thus the better option as it enables truly asynchronous communication, requires less tuning, and frees the CPU for actual query processing. Our main findings for transmitting large messages over high-speed networks are the following:

1. *Reduce memory traffic* by pinning the network thread to the NUIOA-local CPU, allowing the network card to target the cache directly and take advantage of data direct I/O.

2. *For TCP:* Use the IPoIB connected mode with the maximum MTU of 65 520 bytes, pin the network thread to a different core than the interrupt handler.

3. *For RDMA:* Operate directly on message buffers for zero-copy communication, reuse buffers to avoid memory region registration costs, use channel semantics to simplify communication, use event-based completion notifications to minimize the CPU overhead.

3.3 High-Speed Query Processing

The exchange operator is traditionally used to introduce parallelism both locally inside a single server as well as globally between the machines of a cluster. However, it introduces unnecessary materialization overheads for local processing, is inflexible when it comes to dealing with load imbalances, making it vulnerable to attribute value skew, and faces serious scalability issues due to the sheer number of parallel units, especially for modern many-core servers.

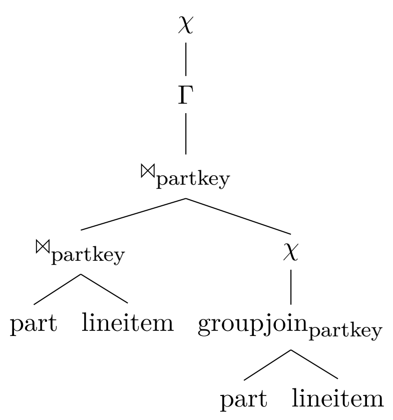
We propose a new hybrid approach instead, choosing the paradigm for each level that fits best. Locally, we use our existing morsel-driven parallelism [52] to parallelize queries across cores ensuring NUMA-local processing. Globally, we designed a new data redistribution scheme between servers that combines decoupled exchange operators and a RDMA-based communication multiplexer that uses low-latency network scheduling. Both levels of parallelism are seamlessly integrated into a new hybrid approach that avoids unnecessary materialization, reacts to load imbalances at runtime, and scales better with the number of cores inside a single machine as well as the number of servers in the cluster.

3.3.1 Classic Exchange Operators

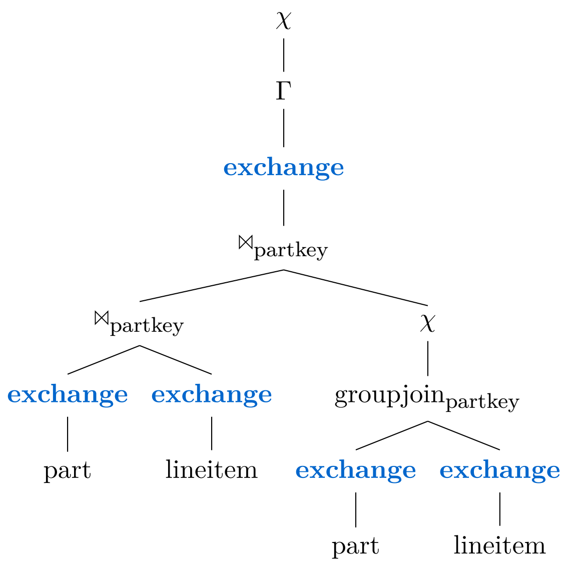
The exchange operator was introduced by Graefe for the Volcano database system [37]. It is a landmark idea as it allows systems to encapsulate parallelism inside an operator. All other relational operators are kept oblivious to parallel execution, making it straightforward to parallelize an existing non-parallel system. An example is shown in Figure 3.6: The single-server query plan shown in Figure 3.6a, which was unnested by the query optimizer, is transformed into the distributed plan of Figure 3.6b by adding exchange operators where required. Two common optimizations for exchange operators are then introduced in Figure 3.6c: First, instead of hash partitioning both inputs, the smaller input is broadcast when the inputs of a join have largely different sizes. Second, pre-aggregations are added as they significantly reduce the number of tuples that have to be shuffled—especially for aggregations with a small number of groups.

The exchange operator is commonly used to introduce parallelism both inside a single machine and between servers (e.g., Vectorwise Vortex [17] and Teradata [87]). Threads execute copies of the query plan and are seen as separate parallel units that operate independently of each other. Parallel units communicate only via exchange operators. There is no difference between two parallel units that operate on the same server or on different machines. While this simplifies parallelization, it also introduces a number of problems.

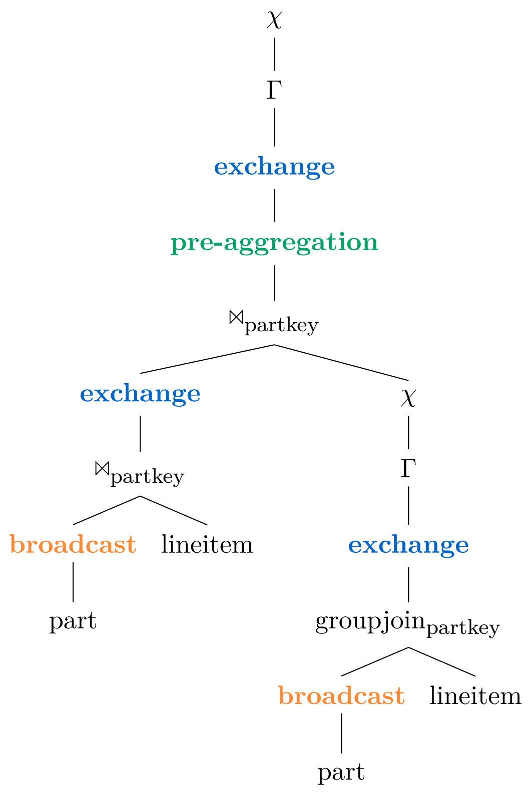
The exchange operator fixes the degree of parallelism in the query plan, which makes it hard to deal with load imbalances and increases the impact of attribute value skew. Each exchange operator splits its input into one partition per parallel unit, e.g., 240 for our relatively small 6-server cluster with 20 hyper-threaded cores and thus 40 threads per machine. If one of the resulting 240 partitions contains more than 1/240th of the input, all other parallel units have to wait for the straggler. A moderately skewed data set with Zipf factor $z = 0.84$ already more than doubles the input for the overloaded parallel unit. Hybrid parallelism instead distinguishes between local and remote parallel units and performs intra-server work stealing, reducing the number of parallel units to the number of servers. The same data set thus increases the input for the overloaded parallel unit by a mere 2.8 % for hybrid parallelism. The fewer parallel units, the lesser the impact of skew. This is orthogonal to the use of specific techniques that detect and deal with skew, e.g., the skew handling approach that we describe in Chapter 4.


χ
Γ
$\bowtie_{\text{partkey}}$
$\bowtie_{\text{partkey}}$
χ
part
lineitem
groupjoin$_{\text{partkey}}$
part
lineitem

(a) Query plan for local execution



(b) Query plan for distributed execution



(c) Optimized plan for distributed execution

Figure 3.6: Query plans for TPC-H query 17

The large number of exchange operators in the classic approach also reduces the applicability of the broadcast optimization for distributed joins. A broadcast join is faster than hash partitioning when one input is much smaller than the other. This limit is $n \times t - 1$ for n servers and t local exchange operators per server as each exchange operator has to send its tuples to every other exchange operator. Hybrid parallelism distinguishes between local and distributed parallelism and can reduce this limit to $n - 1$ as the tuples need only be sent once to every remote server in the cluster. For our 6-server cluster, hybrid parallelism can thus use broadcast instead of hash joins already when the input sizes differ by 5× compared to 239× for the classic exchange operator model.

The huge number of connections and buffers required by the classic exchange operator model leads to further scalability issues and increases memory consumption significantly. An exchange operator requires a connection for each of the $n \times t - 1$ other exchange operators as well as a message buffer to partition its tuples. This results in $n^2 \times t^2 - t$ connections in the cluster and $n \times t - 1$ buffers per operator. For our relatively small 6-server cluster, this requires already a total of 57 560 connections in the cluster and 239 buffers per exchange operator. Hybrid parallelism instead integrates the exchange operators with intra-server morsel-driven parallelism and uses a dedicated communication multiplexer on each machine. It thereby eliminates the unnecessary materialization of intermediate results, significantly reduces the impact of skew, and requires only $n \times (n - 1) = 30$ connections in the cluster and $n - 1 = 5$ buffers per exchange operator.

3.3.2 Hybrid Parallelism

Our novel approach splits distributed query processing into two parts. Decoupled exchange operators perform serialization and deserialization of tuples in a fully parallelized way leveraging query compilation for highest performance. They interact with a single communication multiplexer per server that manages network transfers and uses RDMA to achieve highest network throughput at minimal CPU overhead. All communication multiplexers of the cluster coordinate with one another to avoid switch contention.

Locally inside a server, query execution is parallelized according to the existing morsel-driven parallelism approach [52] using one worker thread per hardware context (two per core for hyper-threading). The input data—coming from either a pipeline breaker or a base relation—is split into work units of constant size, called morsels. Each worker pushes the tuples of its morsel all the way through the compiled query pipeline [62] until a pipeline breaker is reached. This keeps tuples in registers and low-level caches for as long as possible. All database operators—including our new decoupled exchange operators—are designed such that workers can process the same pipeline job in parallel.

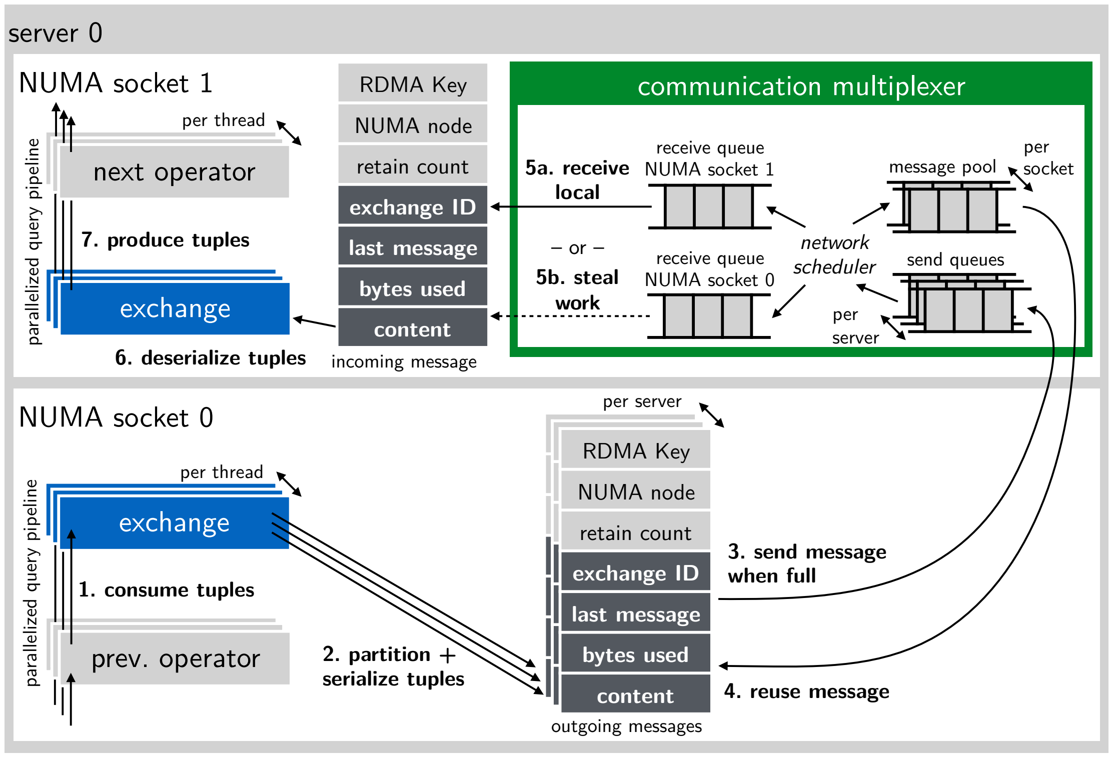


Figure 3.7: Hybrid parallelism

Figure 3.7 illustrates our novel approach: A decoupled exchange operator (1) consumes the tuples that are pushed to it by the preceding operator of its pipeline. It (2) partitions these tuples according to the CRC32 hash value of the join attributes into n messages, one for each of the n servers in the cluster. Broadcast exchange operators differ in that they instead serialize the tuples into a single message, using a retain counter to avoid the higher memory consumption of multiple copies. A message consists of two parts: The first part includes its RDMA memory key, the NUMA node where the message resides, and said retain counter. Only the second part of a message is sent over the network: It consists of an identifier for the corresponding logical exchange operator, an indicator whether this is the last message for this operator, the number of bytes used, and the actual serialized tuples. Once a message is full or the exchange operator has processed all of its input, the message is (3) passed to the communication multiplexer and queued for sending. The exchange operator needs a new empty message that it (4) reuses from a memory pool, ensuring that it is NUMA-local to the CPU core on which the worker thread executes. The RDMA multiplexer sends and receives messages according to a round-robin network schedule to avoid link sharing and the resulting reduced network throughput. Only the used part of a partially-filled

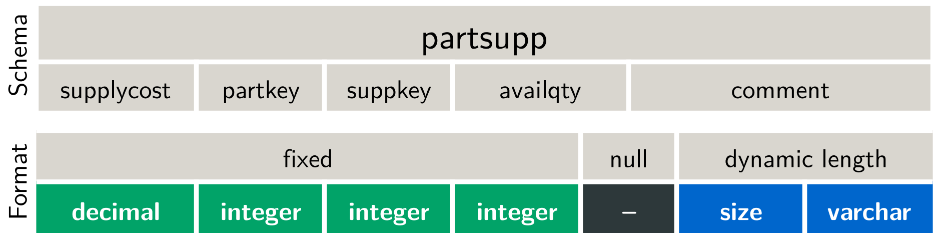


Figure 3.8: Serialization format for the partsupp relation

message is sent over the network. Once a message was successfully sent, it is put into the correct message pool for reuse. The multiplexer receives messages for every NUMA region in turn and notifies waiting exchange operators. These (5a) process NUMA-local messages. Only when there are none available, do they (5b) steal work from other NUMA regions. After (6) deserialization, the tuples are at last (7) pushed to the next operator in the pipeline.

Decoupled Exchange Operators

In contrast to the classic model, our decoupled exchange operator is unaware of all other exchange operators whether local or remote and only interacts with its local communication multiplexer. This has several advantages: Our multiplexer sends broadcast messages only once to every remote server. In contrast, classic exchange operators have to send it to every other exchange operator, reducing the applicability of broadcasts. The classic exchange operator is also inflexible in dealing with load imbalances as each operator is considered a separate parallel unit. Thus, skew has a much higher impact. Our hybrid approach instead treats servers as parallel units and uses work stealing inside the servers to handle local load imbalances. Further, in the classic model each exchange operator needs a buffer for every other exchange operator compared to only one buffer per server for hybrid parallelism, which reduces memory usage significantly.

Our decoupled exchange operator uses LLVM code generation to efficiently serialize and deserialize tuples, minimizing the overhead of materialization. The code is expressly generated for the specific schema of the input tuples and thus does not need to dynamically interpret a schema. This reduces branching, improving code and data locality. Columns that are not required by subsequent operators are pruned as early as possible to reduce network transfer size. An example for our densely-packed, binary serialization format is shown in Figure 3.8 for the *partsupp* relation of the TPC-H benchmark. The format has three parts: The first part contains the values for all fixed-size attributes (e.g., decimal, integer, date) that are defined as *not null*, ordered first by the data type and second according to the schema. The second part consists of null indicators followed by the attribute values

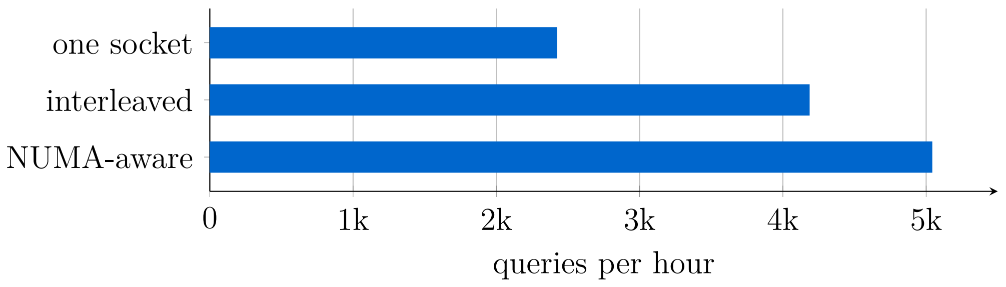


Figure 3.9: TPC-H results for different NUMA allocation strategies

in case the attribute is not null for the current tuple. The third part contains the values for attributes of dynamic length (e.g., varchar, blob, text), which are stored as size and data content.

RDMA-based, NUMA-aware Multiplexer

Our novel communication multiplexer connects the decoupled exchange operators for distributed query processing. It uses RDMA and low-latency network scheduling for high-speed communication and ensures NUMA-local processing.

The multiplexer is a dedicated network thread per server that performs the data transfer between local and remote exchange operators by continuously sending messages according to a global round-robin schedule. Local workers are not connected to all remote workers as this would lead to an excessive number of connections. Instead, only the multiplexers are connected with each other. Any available worker can process any incoming message. This enables work stealing and greatly alleviates the effect of skew. The multiplexer manages the send and receive queues as well as the reuse of messages via reference counting. Instead of deallocating messages when they are no longer needed, they are placed in a message pool. This avoids repeated memory allocation and deallocation during query processing as well as the expensive pinning of new messages to memory and registering them with the InfiniBand HCA to enable RDMA [28].

Modern servers with large main-memory capacities feature a non-uniform memory architecture (NUMA). Every CPU has its own local memory controller and accesses remote memory via QPI links that connect CPUs. As QPI speed is lower than local memory bandwidth and has a higher latency, a remote access is more expensive than a local access. The query execution engine has to take this into account and restrict itself to local memory accesses as much as possible. Our communication multiplexer exposes NUMA characteristics to the database system to avoid incurring this performance penalty. The multiplexer has one receive queue for every NUMA socket as shown in Figure 3.7 and alternatively receives mes-

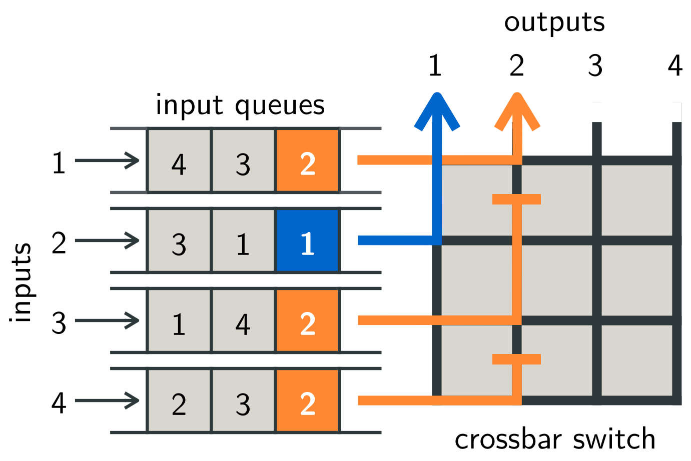


Figure 3.10: Head-of-line blocking

sages for each of them. This also means that NUMA is hidden inside the server so that servers in the cluster could have heterogenous architectures. The multiplexer supports work-stealing: workers take messages from remote queues when their NUMA-local queue is empty.

For our 6-server cluster, allocating messages on a single socket reduces TPC-H performance for the hash join plans by a mere 8 % for a scale factor 100 data set. However, these servers have only two sockets that are further well-connected via two QPI links. This explains the minimal NUMA effects. Figure 3.9 shows the measurement for a 4-socket Sandy Bridge EP server with 15 cores per socket and 1 TiB of main memory. Sockets are fully-connected with one QPI link for each combination of sockets. Interleaved allocation of the network buffers reduces TPC-H performance by 17 % compared to NUMA-aware allocation, allocating messages on a single socket even by 52 %. This demonstrates NUMA-aware allocation of message buffers can have a huge impact on performance. Our novel communication multiplexer therefore provides NUMA-local message buffers to the decoupled exchange operators.

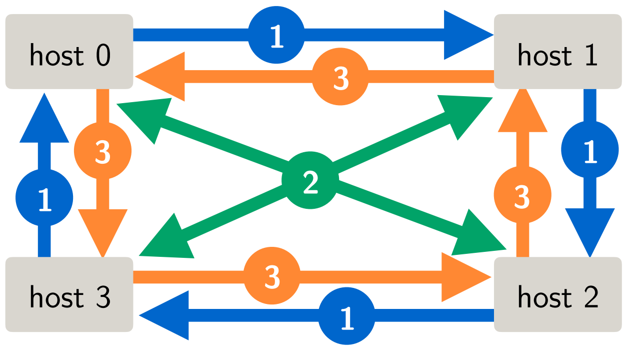
Application-Level Network Scheduling

Uncoordinated all-to-all network traffic can cause switch contention and reduce throughput significantly—even for non-blocking switches that have enough capacity to support all ports simultaneously at maximum throughput. In the case of Ethernet switches, input queuing in the switch can cause head-of-line (HOL) blocking as illustrated by Figure 3.10: Input port 2 sends to output port 2 and thereby blocks input ports 3 and 4. Input port 3 could send to output port 4 and port 3 to port 4, but they are both blocked by packets they need to send to port 2.

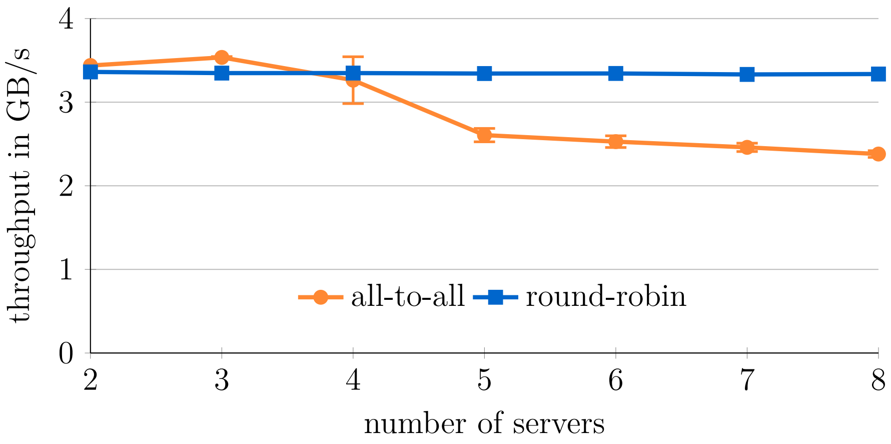
InfiniBand implements flow control differently than TCP. While TCP drops packets during congestion, InfiniBand specifies a lossless fabric to achieve minimal latency. It uses a credit-based link-level flow control to guarantee lossless transport. Credit passes via management packets and is issued per virtual lane to enable prioritization. Each credit granted by the receiver to the sender guarantees that 64 bytes can be received. No data is transmitted unless the available credits indicate sufficient buffer space. While this prevents head-of-line blocking, switch contention is still possible: When several input ports transmit data to the same output port, the credits from the corresponding receiver run out faster than they are granted. Other packets from the same input ports could still be processed, however, the buffer space for input ports is limited. Thus, it is possible that all outstanding packets of a port run out of credits. This creates back pressure and the switch cannot receive more packets for this input port until it obtains new credits from the receiver.

Network scheduling with global knowledge of all active flows has been proposed before to solve the problem of switch contention. Hedera [2] uses a central coordinator that regularly collects flow statistics and moves data flows from congested to underutilized links. However, flow estimation and scheduling is performed only every 5 s—much too infrequent for high-speed networks where transfers take only a few milliseconds and a complete TPC-H run finishes in less than 5 s at scale factor 100. In Chapter 2 we introduced a network scheduling approach that solves the Open Shop problem to minimize join execution time. While this scheduling algorithm can yield huge benefits for commodity networks, its overhead of several hundred milliseconds is simply too high for high-speed networks.

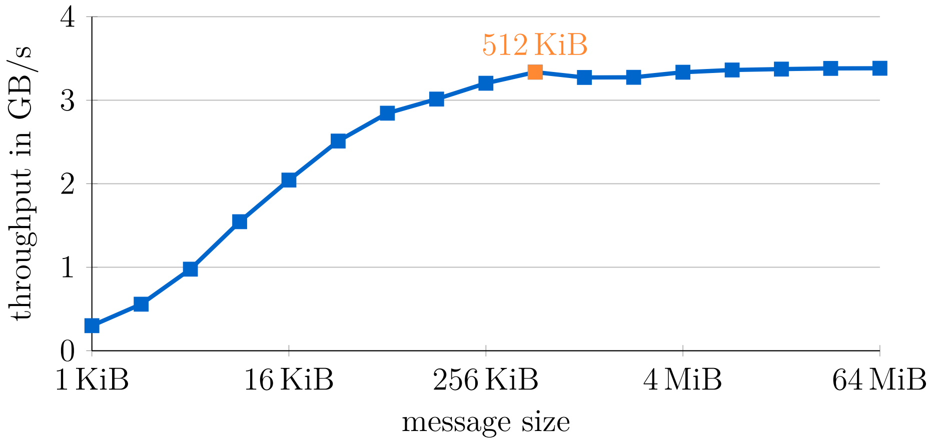
High-speed networks require a new approach to network scheduling that reacts fast and incurs latencies of at most a few microseconds similar to NUMA shuffling inside a single server [53]. For this reason we decided to implement a simple but very efficient round-robin network scheduling algorithm that makes use of special low-latency RDMA operations. It avoids HOL blocking for Ethernet and credit starvation for InfiniBand by dividing communication into distinct phases that prevent link sharing. In each phase a server has one target to which it sends, and a single source from which it receives as shown in Figure 3.11a for four servers and three phases. Round-robin scheduling improves throughput by up to 40 % for an 8-server InfiniBand 4×QDR cluster as demonstrated by the micro-benchmark in Figure 3.11b. In this experiment, we added two smaller servers to our cluster to fully utilize our 8-port InfiniBand switch. Each server transmits 1680 messages of size 512 KiB. After sending 8 messages to a fixed target, all servers synchronize via low-latency ($\approx 1\,\mu s$) inline synchronization messages before they send to the next target. The data transfer between synchronizations has to be large enough to amortize the time needed for synchronization as illustrated in Figure 3.11c. For



(a) Round-robin schedule with conflict-free phases



(b) Application-level network scheduling improves throughput by up to 40 %



(c) 512 KiB messages hide synchronization cost completely (6 servers)

Figure 3.11: Application-level network scheduling

	Q2	Q3	Q4	Q5	Q7	Q8	Q9	Q10	Q11	Q12	Q17	Q18	Q20	Q21	Q1–22
hash join [s]	0.24	0.53	0.18	1.00	0.52	1.07	1.61	0.71	0.22	0.21	1.03	1.39	0.18	1.01	10.99
broadcast [s]	0.04	0.34	0.17	0.22	0.32	0.17	0.59	0.47	0.06	0.12	0.09	0.70	0.13	0.47	4.97
improvement	6.3×	1.5×	1.1×	4.5×	1.6×	6.4×	2.7×	1.5×	3.7×	1.7×	11.0×	2.0×	1.4×	2.2×	2.2×

Table 3.2: TPC-H results for hash join and broadcast plans

our distributed query processing engine we thus use a message size of 512 KiB. It is important to reduce the CPU overhead of handling completion notifications for synchronization messages by processing only every nth notification. We use the maximum of 16k active work requests supported by our hardware to keep the synchronization latency at a few microseconds.

3.4 Distributed Operator Details

Query plans with exchange operators leave room for optimizations. Figure 3.6 illustrates this for TPC-H query 17: The single-server query plan shown in Figure 3.6a, which was unnested by the query optimizer, is transformed into the distributed plan of Figure 3.6b by adding exchange operators where required. Figure 3.6c applies broadcasts and pre-aggregations as optimizations to reduce network traffic.

3.4.1 Join

A distributed equi-join requires that either both of its inputs are hash partitioned by one of the join keys or that one input is broadcast to all servers. The former is known as Grace hash join [50] or hybrid hash join [19] and the latter as fragment-replicate join [24] or broadcast join. The broadcast join is cheaper than the hash join when one input is at least $(n-1)\times$ larger than the other, where n is the number of servers in the cluster. Switching to the broadcast join when inputs have largely different sizes yields significant performance benefits as shown in Table 3.2. The experiment executes TPC-H on a scale factor 100 data set with HyPer on our 6-server cluster comparing broadcast with hash join query execution plans. The overall runtime for all TPC-H queries improves by 2.2× when hash joins are replaced by broadcast joins where beneficial. The overall amount of data that needs to be materialized and reshuffled decreases by a factor of 6 from 154 GiB to 25 GiB when broadcast joins are used.

Using the broadcast redistribution method requires an additional redistribution of the join result to guarantee the correct result for some types of non-equi joins. While a hash join computes semi, anti, and outer joins correctly without modification, a broadcast join might produce duplicates and false results for left semi, left anti, and left outer join. The left semi join may produce duplicate result tuples at different join sites. The result therefore has to be redistributed to eliminate these duplicates. The anti join will produce valid results n times, once for each join site. A replicated tuple may join only at a subset of the join sites and thus cause spurious results at remaining sites, where it does not find a join partner. Again, the result needs to be redistributed and a tuple is to be included in the result only if it occurs exactly n times (i.e., if it found no join partner at any join site). The left outer join may produce duplicates and false results similar to the anti join. Dangling tuples have to be redistributed and counted. Only dangling tuples that found no partner across sites are kept in the final result.

The broadcast join can further compute theta joins with arbitrary join conditions as every combination of tuples from the two relations is compared at some join-site. This is not the case for the hash join that only compares tuples that fall into the same partition.

3.4.2 Grouping/Aggregation

Distributed aggregations require that either their input is partitioned by one of the grouping keys or that all input tuples are sent to a single server. However, both options are overly expensive for large inputs. It is instead often beneficial to introduce a pre-aggregation prior to the exchange operator. The actual aggregation operator following the exchange operator then combines the resulting partial aggregates (cf., Figure 3.6c). This can significantly reduce the number of tuples redistributed across the cluster—especially if the number of groups is small. It is straightforward to implement the pre-aggregation for sum and count. The average of an expression can be computed based on sum and count using a subsequent map operator. A pre-aggregation cannot be used when the aggregation computes a distinct count.

HyPer's distributed query plans use a standard aggregation operator for most pre-aggregations. A dedicated in-cache pre-aggregation operator improves performance further (e.g., for TPC-H query 18). It spills partially aggregated results to the exchange operator whenever its buffer exceeds the size of the cache—similar to the IBM DB2 BLU [67] implementation of the aggregation operator for a single machine. A spilling pre-aggregation may produce multiple output tuples for the same aggregation key. However, the final aggregation after the exchange operator takes care of these duplicates.

Query 1 is an excellent example that demonstrates the benefits of using pre-aggregation operators. It finishes in 6.4 s when the aggregation is computed without pre-aggregation on the server that is also responsible for producing the result. A distributed aggregation that uses hashing to repartition the input according to the grouping key reduces the execution time to 3.3 s. The introduction of a pre-aggregation operator prior to the exchange operator reduces the runtime considerably to a mere 0.08 s. This is an improvement of 80× compared to aggregating on the final server and 40× compared to a distributed hash-based aggregation without pre-aggregation.

3.4.3 Sorting/Top-k

A sort operator can be distributed across servers by sorting the input locally on every server and merging the resulting sorted runs on one server. HyPer's distributed query plans currently do not apply this optimization due to the lack of a merge operator in HyPer. However, the plans use local sorts to reduce the number of tuples redistributed for top-k queries. Every server computes a local top-k and transfers its k result tuples to a single server that then finalizes the distributed top-k computation with a final top-k operator.

This optimization applies to query 2, query 3, and query 18 of the TPC-H benchmark. However, it does not result in measurable improvements as the number of input tuples for the top-k operators are too small in these cases. Query 2 computes a top-100 on 47 107 tuples, query 3 a top-10 on 131 041 tuples, and query 18 a top-100 on 6398 tuples (SF 100). However, this optimization should reduce the number of redistributed tuples significantly when the input is larger. Distributed sorting can be further optimized using specialized techniques. One option is to split sort key and payload prior to actual sorting similar to CloudRAMSort [48] in order to overlap computation and communication further.

3.5 Evaluation

We integrated our distributed query engine in HyPer, a full-fledged main-memory database system that supports the SQL-92 standard. HyPer's excellent single-server performance makes it hard to increase performance when scaling out to a cluster as communication overheads become easily visible as an increase in execution time. The experiments focus on ad-hoc analytical distributed query processing performance.

3.5.1 Experimental Setup

We conducted all experiments on a cluster of six identical servers connected via ConnectX-3 host channel adapters (HCAs) to an 8-port QSFP InfiniScale IV InfiniBand IS5022 switch operating at 4× quad data rate (QDR) resulting in a theoretical network bandwidth of 4 GB/s per link. Each Linux server (Ubuntu 14.10, kernel 3.16.0-41) is equipped with two Intel Xeon E5-2660 v2 CPUs clocked at 2.2 GHz with 10 physical cores (20 hardware contexts due to hyper-threading) and 256 GiB of main memory—resulting in a total of 120 cores (240 hardware contexts) and 1.5 TiB of main memory in the cluster. The RDMA host channel adapter runs firmware version 2.32.5100 and driver version 2.2-1. It is connected via 16× PCIe 3.0. The hardware setup is illustrated in Figure 3.1. The thickness of a line in the diagram corresponds to the respective bandwidth of the connection.

HyPer offers both row and column-wise data storage; all experiments were conducted using the columnar format and only primary key indexes were created. On each server, HyPer transparently distributes the input relations over all available NUMA sockets (two in our case). Execution times include memory allocation (from the OS), page faulting, and deallocation for intermediate results, hash tables, etc.

TPC-H joins relations mostly along key/foreign-key relationships and thus benefits considerably when relations are partitioned accordingly. This enables partially or even completely local joins that avoid network traffic and thus improve query response times. Still, we decided against partitioning relations for HyPer as this is a manual process that requires prior knowledge of the workload. Instead, we assign relation chunks to servers as generated by *dbgen* without initial redistribution for all experiments in this chapter except where otherwise noted.

3.5.2 Hybrid Parallelism

This section evaluates the performance of our new approach by analyzing the scalability of the individual TPC-H queries for different distributed query execution engines. We further analyze the impact of the available network bandwidth and our low-latency network scheduling approach.

Scalability

Figure 3.12 shows how the individual TPC-H queries scale in HyPer when servers are added to the cluster. The experiment uses a scale factor 100 data set. It is apparent that the queries do not scale for Gigabit Ethernet. The only exceptions

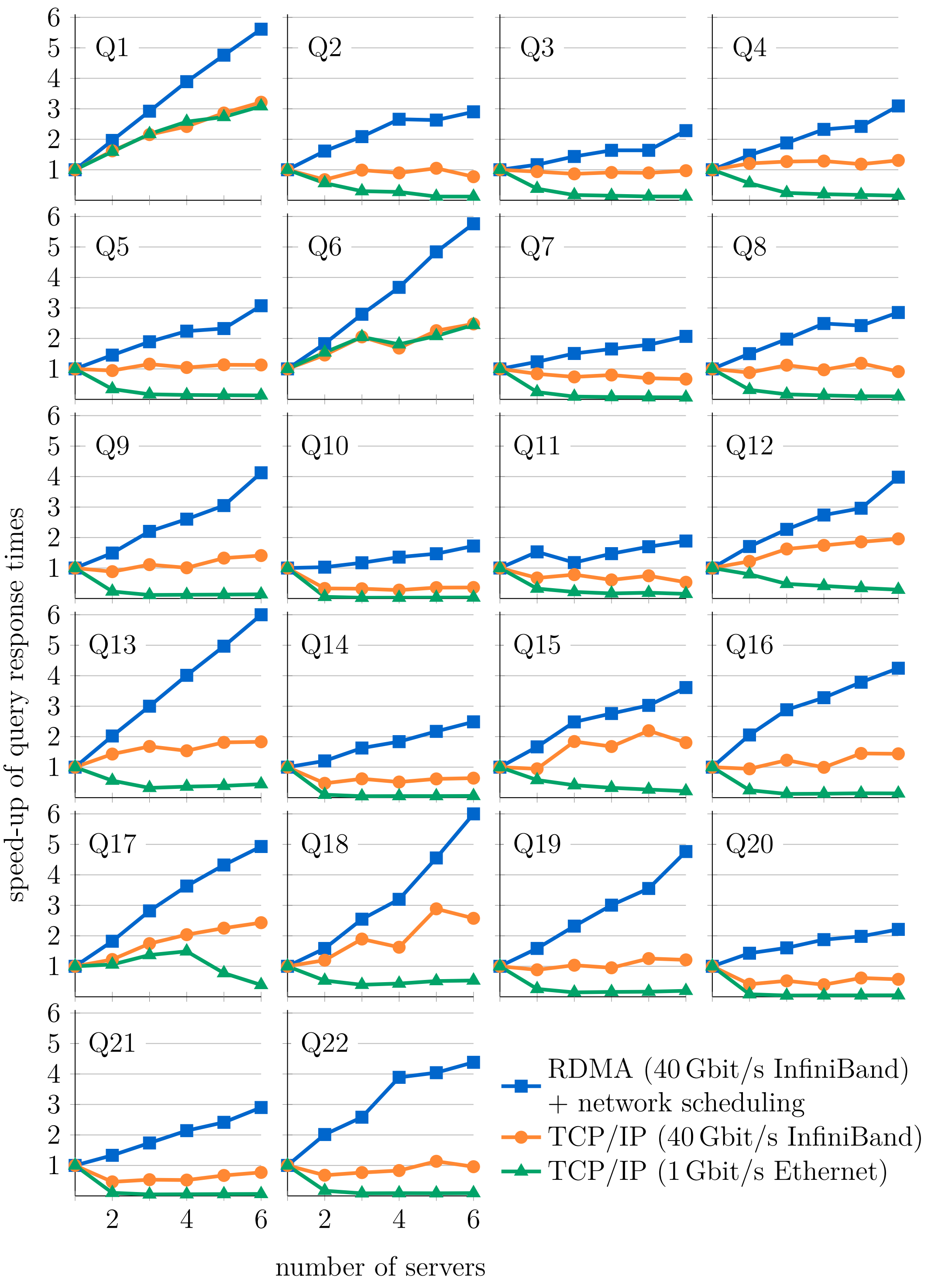


Figure 3.12: Scalability of the individual TPC-H queries in HyPer

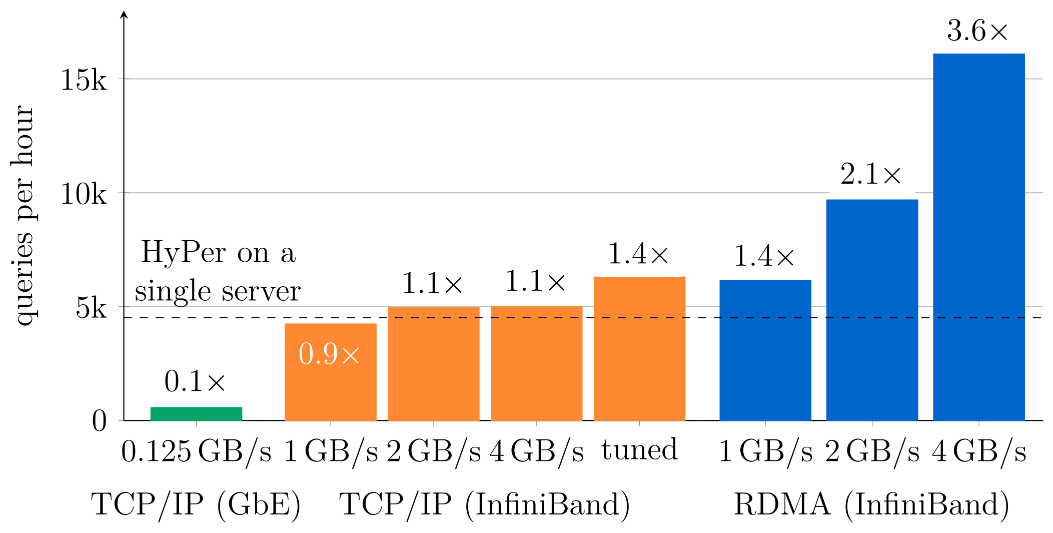


Figure 3.13: Impact of the interconnect on HyPer's TPC-H performance

are query 1 and query 6, which transfer almost no data over the network. The scalability graph of query 17 illustrates the impact of switch contention: For more than 4 servers the effective bandwidth drops significantly and thus also the query performance. TCP/IP over InfiniBand (4×QDR) performs better than TCP over Gigabit Ethernet but still does not scale well, mostly staying close to single-server performance or even performing worse (e.g, for queries 10, 11, and 20). Only our RDMA-based communication multiplexer with network scheduling can improve the performance for all queries, with an overall speed-up of 3.5× for 6 servers (cf., Figure 3.3).

Impact of Network Bandwidth

Figure 3.13 shows the impact of the network bandwidth and different transport protocols on HyPer's TPC-H performance. We compare the TPC-H performance (measured in queries per hour) of a 6-server cluster with that of a single server using a scale factor 100 data set. When the cluster is connected via Gigabit Ethernet, it actually performs 8× *worse* than a single sever due to the large overhead of network transfers. Our InfiniBand hardware allows to configure the data rate, enabling us to measure the TCP and RDMA variants of our network multiplexer for 1 GB/s (single data rate), 2 GB/s (double data rate), and 4 GB/s (quad data rate). HyPer's TPC-H performance stagnates at 1.1× of the single-server performance for both DDR and QDR rate when TCP is used with default settings (IPoIB datagram mode with 2 KiB MTU). Tuning TCP (IPoIB connected mode with 64 KiB MTU and a separate core for interrupt handling) improves performance by 24 % to 1.4×

the performance of a single server when QDR is used. However, only our novel RDMA-based communication multiplexer with network scheduling is able to scale HyPer's TPC-H performance with the available network bandwidth, processing 3.6× more queries per hour than a single server for InfiniBand 4× QDR.

Impact of Network Scheduling

We analyzed the impact of network scheduling on HyPer's TPC-H performance for our 6-server cluster. Scheduling improves HyPer's TPC-H performance by 230 % when Gigabit Ethernet is used to connect the servers. Due to the high CPU overhead of TCP stack processing for TCP/IP over InfiniBand, network scheduling does not improve performance in this setting. For RDMA, network scheduling improves HyPer's TPC-H performance by 12 %. We expect the impact of network scheduling on TPC-H performance to increase further with the cluster size.

3.5.3 Distributed SQL Systems

We compare HyPer with five state-of-the-art distributed SQL systems: Apache Hive 1.1, Spark SQL 1.3, Cloudera Impala 2.2 [51], MemSQL 4.0, and Vectorwise Vortex [17]. HyPer, MemSQL, and Vectorwise use custom data storage. We configured Hive to use main memory as scratch space. We ensure that Spark SQL caches the HDFS input as deserialized Java objects in main memory before query execution (cache level `MEMORY_ONLY`) to avoid deserialization overheads. Impala processes HDFS-resident Parquet files during query execution. We ensured a hot Linux buffer cache to avoid expensive disk accesses. Still, Impala has to perform deserialization during query execution. We conducted a micro-benchmark analyzing multiple TPC-H queries and found that deserialization makes up less than 30 % of the query execution time.

Apache Hive, Spark SQL, and Impala are installed as part of the Cloudera Hadoop distribution (CDH 5.4). We set the HDFS replication factor to 3 (so that every block is replicated to three data nodes) and enabled short-circuit reads. The cluster is configured so that all systems use the high-speed InfiniBand interconnect instead of Gigabit Ethernet. We use unmodified TPC-H queries except for Hive and Spark SQL—which required rewritten TPC-H queries that avoid correlated subqueries [26]—and a modified query 11 for Impala, which does not support subqueries in the having clause.

Figure 3.14 compares the six systems for the TPC-H benchmark on a scale factor 100 data set using our 6-server cluster ($\approx$ 110 GiB data on disk). A full TPC-H run takes 1 h 30 min 29 s for Hive, 16 min 19 s for Spark SQL (without query 11), 10 min 42 s for Impala, 2 min 26 s for MemSQL, 20.5 s for Vectorwise,

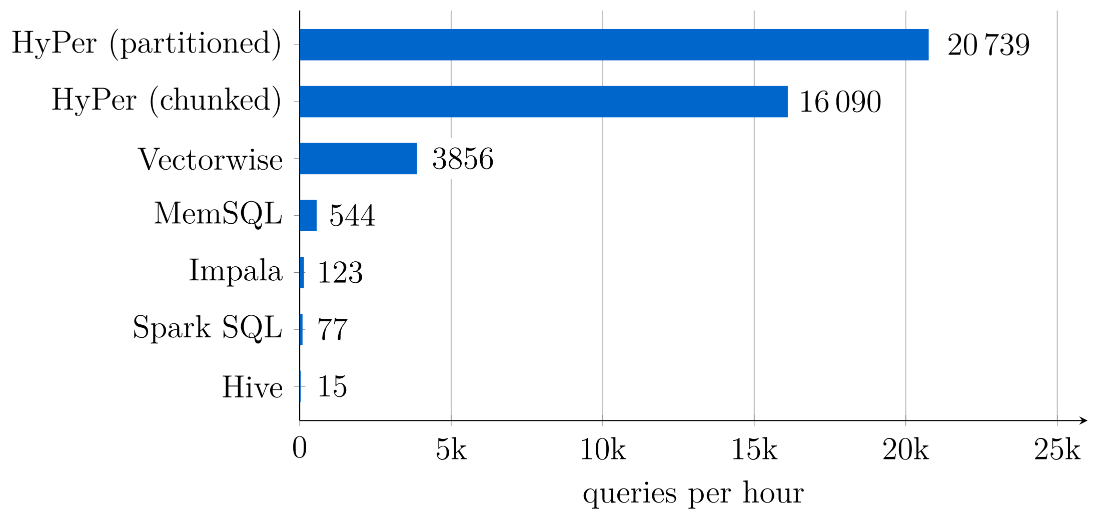


Figure 3.14: TPC-H results for several distributed SQL systems

4.9 s for HyPer with chunked data placement and 3.8 s when relations are partitioned by the first attribute of the primary key. Queries are run 10 times, keeping the median. The Linux file system cache is not flushed between runs to keep data hot in main memory and avoid disk accesses.

Tables 3.3 and 3.4 show the detailed query execution times with the fastest highlighted in bold. They also show network and disk I/O metrics for each system. Hive, Spark SQL, and Impala do not support partitioning relations by a specific attribute. Thus, Table 3.3 compares these systems with HyPer for a chunked data placement. In this setting, distributed joins and aggregations are always required and shuffle large amounts of data across the network. MemSQL and Vectorwise support partitioning, therefore Table 3.4 compares these two systems to HyPer for a partitioned data placement. We partitioned the relations by the first attribute of the primary key. Distributed joins can often exploit partitioned relations to reduce the amount of data shuffled between servers. Note that HyPer with chunked data placement still outperforms MemSQL and Vectorwise with partitioned data placement even though it has to shuffle much more data for distributed joins and aggregations. This is at least partly owed to its efficient RDMA-based query engine.

Configuration Details

We tuned all compared systems according to their documentation, previous publications, and direct advice from their developers to ensure their best possible performance.

	Hive	Spark SQL	Impala	HyPer
Q1	1 min 19 s	7.09 s	7.04 s	**0.08 s**
Q2	3 min 14 s	16.92 s	8.91 s	**0.04 s**
Q3	3 min 59 s	26.04 s	23.79 s	**0.34 s**
Q4	2 min 44 s	10.21 s	25.53 s	**0.16 s**
Q5	5 min 14 s	1 min 13 s	23.57 s	**0.22 s**
Q6	53.24 s	1.97 s	3.39 s	**0.03 s**
Q7	7 min 56 s	59.67 s	43.57 s	**0.33 s**
Q8	6 min 4 s	1 min 20 s	21.04 s	**0.17 s**
Q9	13 min 30 s	2 min 52 s	1 min 11 s	**0.60 s**
Q10	4 min 28 s	18.63 s	8.86 s	**0.51 s**
Q11	2 min 40 s	–	4.20 s	**0.06 s**
Q12	2 min 41 s	18.30 s	8.97 s	**0.12 s**
Q13	2 min 26 s	12.36 s	25.97 s	**0.36 s**
Q14	1 min 40 s	7.13 s	6.00 s	**0.06 s**
Q15	2 min 24 s	12.92 s	4.83 s	**0.08 s**
Q16	2 min 41 s	11.37 s	7.35 s	**0.20 s**
Q17	3 min 25 s	2 min 20 s	1 min 28 s	**0.09 s**
Q18	4 min 25 s	1 min 40 s	1 min 7 s	**0.58 s**
Q19	2 min 54 s	8.73 s	1 min 43 s	**0.20 s**
Q20	3 min 55 s	24.97 s	16.27 s	**0.14 s**
Q21	8 min 56 s	2 min 50 s	1 min 8 s	**0.49 s**
Q22	3 min 4 s	15.69 s	6.35 s	**0.07 s**
packets sent	151 million	107 million	176 million	7.3 million
data shuffled	401 GiB	211.3 GiB	140.52 GiB	27.95 GiB
disk I/O	833 GiB	0.23 GiB	0.04 GiB	0.00 GiB
total time	1 h 30 min	16 min 19 s	10 min 42 s	**4.92 s**
geometric mean	204.84	24.32	17.21	**0.16**
queries per hour	15	77	123	**16 090**

Table 3.3: Detailed TPC-H results for chunked data placement

	MemSQL	Vectorwise	HyPer
Q1	8.38 s	0.80 s	**0.10 s**
Q2	1.65 s	0.37 s	**0.05 s**
Q3	13.90 s	0.24 s	**0.18 s**
Q4	0.81 s	**0.06 s**	0.18 s
Q5	8.83 s	0.84 s	**0.13 s**
Q6	2.50 s	**0.05 s**	**0.05 s**
Q7	2.76 s	0.36 s	**0.16 s**
Q8	2.48 s	2.01 s	**0.10 s**
Q9	11.92 s	1.44 s	**0.56 s**
Q10	1.50 s	1.56 s	**0.26 s**
Q11	0.53 s	0.22 s	**0.12 s**
Q12	1.76 s	0.10 s	**0.09 s**
Q13	4.48 s	3.61 s	**0.41 s**
Q14	2.29 s	0.69 s	**0.05 s**
Q15	13.00 s	0.95 s	**0.08 s**
Q16	3.44 s	0.69 s	**0.16 s**
Q17	0.75 s	0.53 s	**0.11 s**
Q18	51.30 s	1.63 s	**0.37 s**
Q19	0.60 s	0.81 s	**0.23 s**
Q20	8.53 s	0.51 s	**0.13 s**
Q21	2.08 s	1.91 s	**0.24 s**
Q22	2.19 s	1.18 s	**0.06 s**
packets sent	7.02 million	7.06 million	2.4 million
data shuffled	13.96 GiB	19.36 GiB	8.88 GiB
disk I/O	0.03 GiB	0.01 GiB	0.00 GiB
total time	2 min 26 s	20.54 s	**3.82 s**
geometric mean	3.23	0.59	**0.14**
queries per hour	544	3856	**20 739**

Table 3.4: Detailed TPC-H results for partitioned data placement

Hive. Hive is measured with generated statistics processing ORC files using a tuned Hadoop YARN (optimized main-memory limits, JVM reuse enabled, output compression via Snappy, increased I/O buffers). We used the rewritten TPC-H queries from a recent study [26] that avoid correlated subqueries and enable Hive's correlation optimization, predicate push down, and map-side join/aggregation. Early measurements revealed that Hive writes 2 TiB of data to disk for a TPC-H SF 100 run. We configured it to use main memory as scratch space but it still writes over 800 GiB of intermediate results to HDFS owing to the MapReduce processing model.

Spark SQL. Apache Spark SQL, the follow-up to Shark, is still early in development. We ensure in-memory processing by specifying the tables as temporary and explicitly caching them. We further configured Spark to use main memory for its scratch space and disabled spilling to disk. We increased the per-executor memory to 250 GiB per server and the parallelism level to twice the number of cores in the cluster as recommended. For queries 2, 15, and 22 we had to use rewritten queries from [26] that avoid correlated subqueries. Spark SQL takes more than an hour for query 11, we thus measured all metrics for the remaining 21 queries.

Impala. Impala is run with generated statistics on Parquet files with disabled pretty printing. Short-circuit reads and runtime code generation are enabled, operator spilling disabled. Using the HDFS cache did not improve query performance compared to the Linux file system cache.

MemSQL. We configured MemSQL to replicate nation and region by specifying them as reference tables. All other relations are partitioned by the first column of their primary key. We used one partition per CPU core in the cluster as recommended in the MemSQL documentation for maximum parallelism. We further created foreign key indexes to enable index-nested-loop joins that improve performance significantly.

Vectorwise. We measure Vectorwise Vortex with statistics, primary keys, and foreign keys. We set the maximum parallelism level to 120 and the number of cores to 20 as recommended by an Actian engineer for our cluster. Similar to MemSQL, all relations except nation and region are partitioned by the first column of their primary key. Replicating customer and supplier as recommended by Actian for an optimal TPC-H performance reduces the runtime by 19 % to 16.56 s and the geometric mean to 0.5.

HyPer. For HyPer we measured both chunked data placement to compare against Apache Hive, Spark SQL, and Impala as well as partitioned data placement to compare against MemSQL and Vectorwise. For chunked relations, HyPer has to use distributed joins and aggregations more often and thus shuffles considerably more data across the network—28 GB instead of the 9 GB for partitioned data

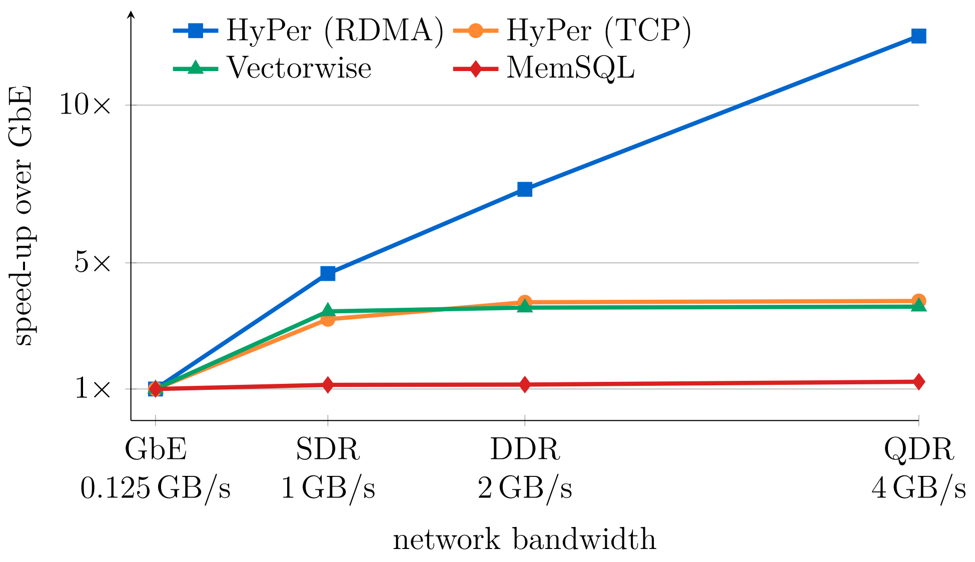


Figure 3.15: Impact of network bandwidth on TPC-H performance

placement. Note that HyPer with chunked relations (shuffling 28 GB of data) still outperforms Vectorwise and MemSQL when they use partitioned data placement and shuffle only 14 GB respectively 19 GB of data between servers. This is mainly due to our fast RDMA-based query engine.

Network Bandwidth

Our InfiniBand hardware supports data rates of 1 GB/s (single data rate), 2 GB/s (double data rate), and 4 GB/s (quad data rate). Figure 3.15 shows the impact of the network bandwidth on the TPC-H performance of the parallel main-memory database systems HyPer, Vectorwise Vortex, and MemSQL. For each system, we show the speed up compared to its own performance when Gigabit Ethernet is used instead. MemSQL only improves by 23 % when using the 32× faster InfiniBand 4×QDR interconnect. Vectorwise Vortex and a variant of HyPer using TCP achieve a speed up of 4× for InfiniBand but cannot scale performance substantially when increasing the data rate. Our RDMA-enabled communication multiplexer enables HyPer to scale its TPC-H performance with the network bandwidth, processing 12× more queries per hour for InfiniBand 4×QDR compared to Gigabit Ethernet.

Larger Scale Factor

We further ran TPC-H at scale factor 300 for the three fastest systems to see how they scale to larger inputs: HyPer takes 12 s (geometric mean 0.42) to process the ≈ 320 GiB of data. This is 3.1× (3×) more than the 3.8 s (geometric mean 0.14) for SF 100. Vectorwise Vortex takes 44.2 s (geometric mean 1.15) for SF 300, an increase of 2.2× (2×) compared to 20.5 s (geometric mean 0.59) for SF 100. MemSQL processes a SF 300 data set in 8 min 46 s (geometric mean 10.41), this is 3.4× (3.2×) more than the 2 min 26 s (geometric mean 3.23) for SF 100.

3.6 Related Work

Parallel databases are a well-studied field of research that attracted considerable attention in the 1980s/90s with Grace [31], Gamma [20], Bubba [10], Volcano [37], and Prisma [4]. Several parallel database systems of the time, commercial and academic, are discussed in an influential survey [18]. Today's commercial parallel database systems include Teradata, Exasol Exasolution, IBM DB2, Oracle Database In-Memory [61], Greenplum, SAP HANA [34], HP Vertica (which evolved from C-Store [76]), MemSQL, Cloudera Impala [51], and Vectorwise Vortex [17].

The comparatively low bandwidth of standard network interconnects such as Gigabit Ethernet creates a bottleneck for distributed query processing. Consequently, recent research focused on minimizing network traffic: Neo-Join [71] and Track Join [65] decide during query processing how to redistribute tuples to exploit locality in the data placement. High-speed networks such as InfiniBand and RDMA remove this bottleneck and have been applied to database systems before. Frey et al. [28] designed the Cyclo Join for join processing within a ring topology. Goncalves and Kersten [35] extended MonetDB with a novel distributed query processing scheme based on continuously rotating data in a modern RDMA network with a ring topology. Mühleisen et al. [60] pursued a different approach, using RDMA to utilize remote main memory for temporary database files in MonetDB. Kalia et al. [43] used RDMA to build a fast key-value store. Barthels et al. [5] provide a detailed analysis of a distributed radix join using RDMA for rack-scale InfiniBand clusters. Costea and Ionescu [17] extended Vectorwise, which originated from the MonetDB/X100 project [88], to a distributed system using MPI over InfiniBand. The project is called Vectorwise Vortex and is included in our evaluation.

The problem of switch contention has been addressed in the literature before. Hedera [2] applies heuristics to move data flows from overloaded links to free links using a central coordinator with global knowledge. However, flow estimation and scheduling is performed only every 5 s—much too infrequent for high-speed networks where transfers take a few milliseconds and a complete TPC-H run finishes in less than 5 s at scale factor 100. Neo-Join [71] uses application-level network scheduling solving the Open Shop problem to minimize join execution time. However, its scheduling algorithm requires prior knowledge of data transfer sizes and does not scale well as its runtime is in $\mathcal{O}(n^4)$ for n servers. High-speed networks require a different approach to network scheduling with much less overhead. The network scheduling approach described in this chapter reduces synchronization time to just a few microseconds by utilizing low-latency InfiniBand operations.

Several papers discuss the implications of NUMA for database systems. Albutiu et al. [3] devised MPSM, a NUMA-aware sort-merge join algorithm. Li et al. [53] applied data shuffling to NUMA systems and particularly to MPSM. Our application-level network scheduling is similar to NUMA shuffling in that we also use a simple round-robin schedule to keep synchronization overheads at a few microseconds.

There has been various research analyzing TCP performance [13, 27, 28]. We refer the reader to [28] for a detailed discussion. Previous studies have found that TCP performance does not scale well to higher network bandwidths, as the receiver becomes CPU-bound. For large messages, data touching operations such as checksums and copying cause a high CPU load and the network interface card raises interrupts leading to many context switches [27, 28]. TCP offloading, already proposed in the 1980s to alleviate the CPU bottleneck [13], has since been implemented in hardware. A second problem identified in the literature is TCP's high memory bus load [13]: Every byte sent and received over the network causes 2 to 4 bytes traversing the memory bus [27]. However, our experiments have shown that the introduction of data direct I/O reduced memory bus traffic considerably for NUMA-aware applications (cf., Section 3.2.1).

IBM DB2 differentiates between local and global parallelism similar to hybrid parallelism to overcome some of the problems of the classic exchange operator. However, instead of decoupled exchange operators that enable work stealing, DB2 uses a special exchange operator that merges the results of threads to reduce the number of parallel units.

3.7 Concluding Remarks

Remote direct memory access (RDMA) currently receives increasing interest in the database research community. Binnig et al. [7] have shown that database systems need to adopt RDMA to fully leverage the high bandwidth of InfiniBand. It is not enough to just use faster networking hardware, the software has to change as well to address the new bottlenecks that emerge. Recent work has shown the benefits of RDMA for specific relational operators (e.g., joins [5]) and key-value stores [43]. However, we are the first to present the design and implementation of a complete distributed query engine based on RDMA that is capable of processing complex analytical workloads such as the TPC-H benchmark.

Our engine uses RDMA to avoid the overheads of TCP processing, low-latency network scheduling to address switch contention, and a flexible approach to parallelism that overcomes the inflexibility of the classic exchange operator model. In combination, this allows us to scale the high single-server performance of a state-of-the-art in-memory database system with the number of servers in the cluster.

Chapter 4

Skew Handling

Parts of this chapter were previously published in [69].

Modern high-speed interconnects such as InfiniBand are crucial to achieve scalable distributed query processing where adding a server to the cluster increases performance. However, the scalability of distributed joins is threatened by unexpected data characteristics: Skew can cause severe load imbalances such that a single server has to process a much larger part of the input than its fair share and by this slows down the entire distributed query.

We introduce *Flow-Join*, a novel distributed join algorithm that handles attribute value skew with *minimal overhead*. Flow-Join detects heavy hitters *at runtime* using small approximate histograms and adapts the redistribution scheme to resolve load imbalances before they impact the join performance. Previous approaches often involve expensive analysis phases that slow down distributed join processing for non-skewed workloads. This is especially the case for modern high-speed interconnects that are too fast to hide the extra computation. Other skew handling approaches require detailed statistics that are often not available or overly inaccurate for intermediate results. In contrast, Flow-Join with its novel lightweight skew handling scheme executes at the full network speed of more than 6 GB/s for InfiniBand 4×FDR, joining a skewed input at 11.5 billion tuples/s with 32 servers. This is 6.8× faster than a standard distributed hash join using the same hardware. At the same time, Flow-Join does not compromise the join performance for non-skewed workloads.

4.1 Motivation

Today's many-core servers offer unprecedented single-server query performance and main-memory capacities in the terabytes. Yet, a scale-out to a cluster is still necessary to increase the main-memory capacity beyond a few terabytes. For

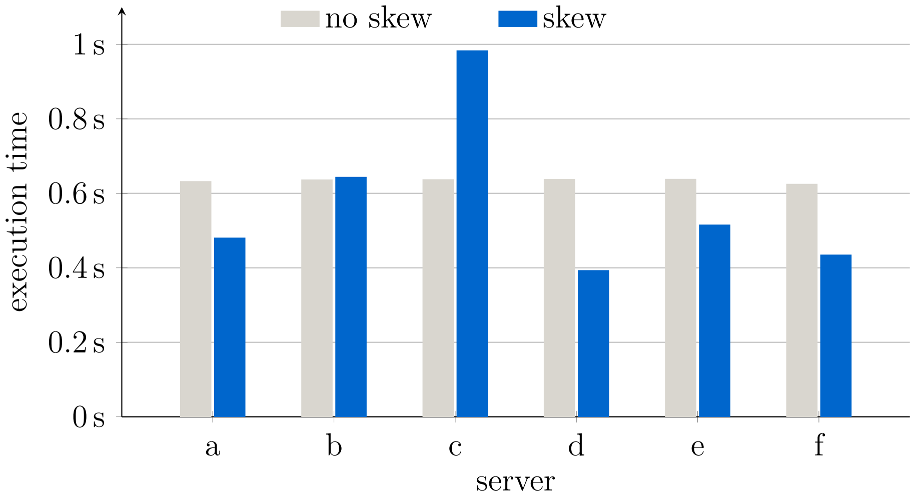


Figure 4.1: Join execution time for skewed and non-skewed inputs

example, Walmart—the world's largest company by revenue—uses a 16 server cluster with a total of 64 TiB of main memory to perform analytical queries on their business data [63].

Data is commonly partitioned across servers so that users can utilize the combined main-memory capacity of the cluster. Consequently, query processing requires network communication between servers. Network bandwidth used to be a bottleneck for distributed query processing so that a cluster actually performed worse than a single server. Previous work thus focussed on avoiding communication as much as possible [71, 65]. However, the economic viability of high-bandwidth networks has changed the game: Modern high-speed interconnects with link speeds of several gigabytes per second enable scalable query processing where adding servers to the cluster in fact improves query performance [7, 70].

However, skew can cause load imbalances during data shuffling and thus again threatens the scalability of distributed query processing as highlighted by Figure 4.1. The experiment shows the join execution time for each of the 6 servers in a cluster comparing skewed and non-skewed inputs. The cluster is connected via InfiniBand 4×QDR, which offers a theoretical throughput of 4 GB/s. The skewed input (Zipf factor $z = 1.25$) causes server c to process many more tuples than the others and by this increases the join execution time by 54 %.

Heavy hitter skew is particularly harmful for partitioning-based operators (e.g., distributed joins and aggregations), as all tuples with the same partitioning key are assigned to the same server. A distributed operator is only as fast as its slowest instance, overloading a single server will thus cause the whole operator to significantly underperform. With more servers the situation gets even worse:

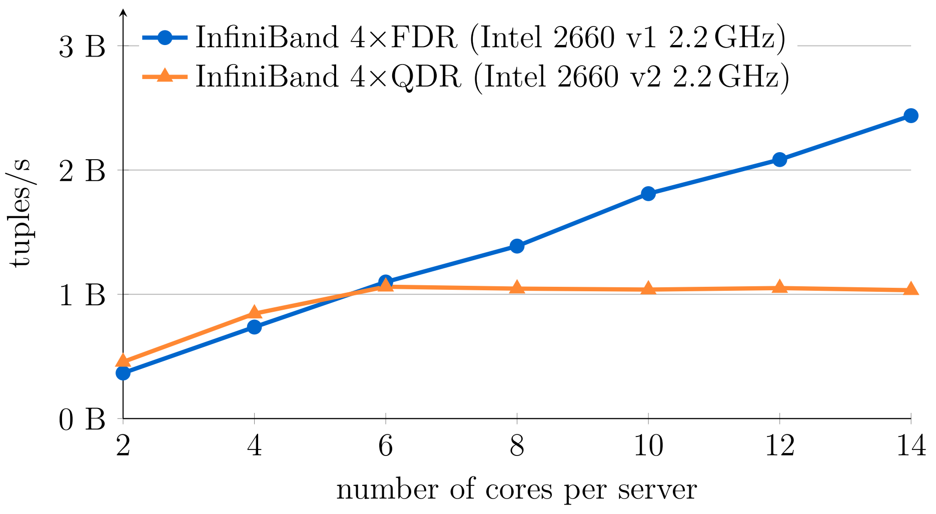


Figure 4.2: CPU-bound vs. network-bound distributed join processing

The larger the cluster, the larger the negative impact of skew. For a constant join input, the input size per server decreases when servers are added to the cluster while the number of heavy hitter tuples stays the same. The heavy hitter tuples assigned to a single server will thus increase its input by a larger factor.

While distributed aggregations can generally handle heavy hitter skew effectively using a fast in-cache pre-aggregation, there is no simple remedy for distributed joins. A distributed join partitions both inputs into as many partitions as there are servers. Only tuples from corresponding partitions will join and these partition pairs are thus assigned to servers. As tuples with the same key are assigned to the same server, heavy hitters will cause serious load imbalances as shown in Figure 4.1. In the example, the third server is assigned many more tuples than the other servers. This impacts performance in two ways: First, network communication becomes irregular, congesting the link to this server. Second, the third server must process many more tuples than the other servers. Ultimately, the overloaded server takes much longer to process its part of the input and thus slows down the entire join.

Heavy hitter skew causes load imbalances during distributed query processing whether the network is slow or fast. However, for slow networks the actual join computation accounts for such a small fraction of the total execution time that even expensive skew handling approaches are a viable choice. For example, a distributed join over Gigabit Ethernet using 6 servers with 5040 build and 105 M probe tuples per server has a runtime of 13.2 s. The actual join computation takes less than 250 ms while the remaining time is spent waiting for network transfers.

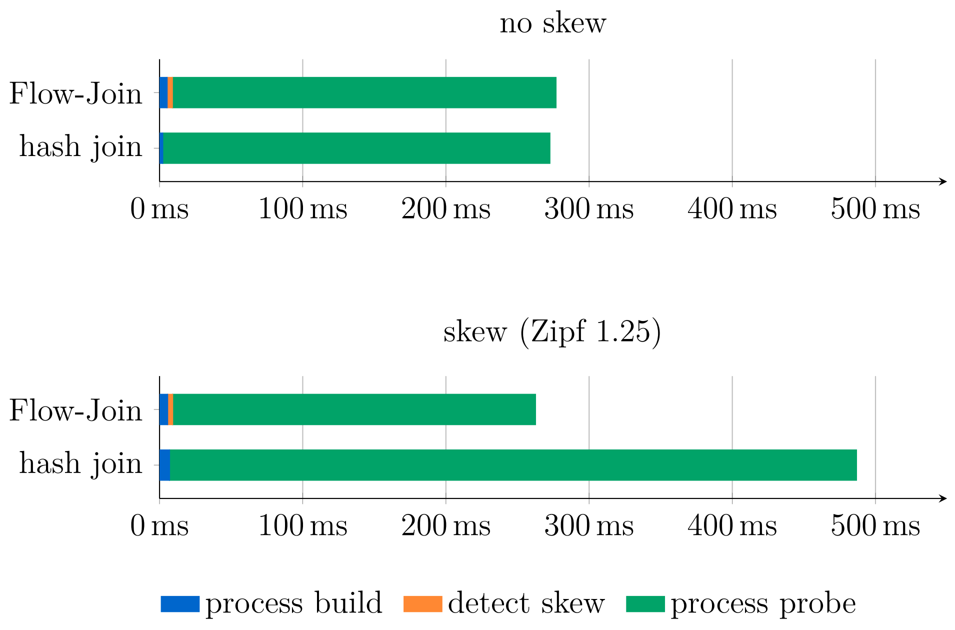


Figure 4.3: Breakdown of join execution time

In this case, a preceding skew detection phase would increase runtime only by 2 % even if it performs as much computational work as the actual join.

Modern high-speed interconnects have closed the gap between network bandwidth and compute speed: A distributed join is not necessarily network-bound anymore. Figure 4.2 shows that using additional cores for join processing keeps improving the join performance of a 6-server cluster that is connected via InfiniBand 4×FDR. This stands in contrast to the slower 4×QDR where six cores already suffice to saturate the available network bandwidth. In this experiment the size of the build input is intentionally chosen to fit into cache for fastest join processing (5040 build and 105 M probe tuples per server). InfiniBand 4×FDR is still fast enough to not be the bottleneck. The experiment shows that skew detection cannot be hidden behind slow network transfers anymore. Instead, any additional work will directly translate into a visible increase in execution time. Modern high-speed networks thus require fast skew detection. This applies even more for the upcoming InfiniBand EDR hardware that will offer almost twice the bandwidth of FDR.

Previous approaches to handle skew depend either on detailed statistics [22, 87] that are often not available or overly inaccurate for intermediate results, or on analysis phases [49, 41, 85, 71] that are too expensive for high-speed interconnects. Flow-Join instead performs a lightweight heavy hitter detection alongside partitioning. It avoids load imbalances by broadcasting tuples that join with heavy hitters. Flow-Join's skew detection and handling adds almost no overhead—even for high-speed networks such as InfiniBand 4×FDR, which is more than 50× faster than Gigabit Ethernet. Figure 4.3 compares Flow-Join to a standard hash join. The experiment runs on a 6-server cluster connected via 4×FDR at 6.8 GB/s using 5040 build and 105 M probe tuples per server. Both join algorithms perform similar for the non-skewed input with Flow-Join's skew detection adding only a minimal overhead of 1.6 %. For skewed inputs, the hash join takes 79 % longer, while Flow-Join actually performs 5 % faster compared to the non-skewed input as it keeps heavy hitter tuples local, thereby reducing network communication. Flow-Join does not have a separate skew detection phase. It detects skew while it partitions the first 1 % of the probe input, after which a global consensus on skew values is formed. Both implementations overlap computation (partitioning, build, and probe) with network communication so that the join finishes shortly after the last network transfer. Our adaptive skew handling approach enables pipelined execution of joins and thus avoids the materialization of the probe side, reducing main-memory consumption significantly. In particular, this chapter makes the following contributions:

1. A novel mechanism to detect heavy hitters alongside partitioning of the inputs that incurs only minimal overhead, comparing several implementations of the employed approximate histograms.

2. A method to adapt the redistribution scheme at runtime for a subset of the keys identified as heavy hitters. Broadcasting corresponding build tuples avoids that all probe tuples for a specific heavy hitter value are assigned to a single server.

3. Based on these two techniques, a highly-scalable implementation of Flow-Join that utilizes remote direct memory access (RDMA) to shuffle the data at full network speed. The implementation further distinguishes between local and distributed parallelism to avoid the inflexibility of the classic exchange operator model.

4. A generalization of Flow-Join beyond key/foreign-key equi joins that employs the Symmetric Fragment Replicate redistribution scheme to optimally handle correlated skew in both inputs.

5. Finally, an extensive evaluation including a scalability experiment on a cluster of 32 servers using Zipf-generated data as well as a real workload from a large commercial vendor.

4.2 Flow-Join

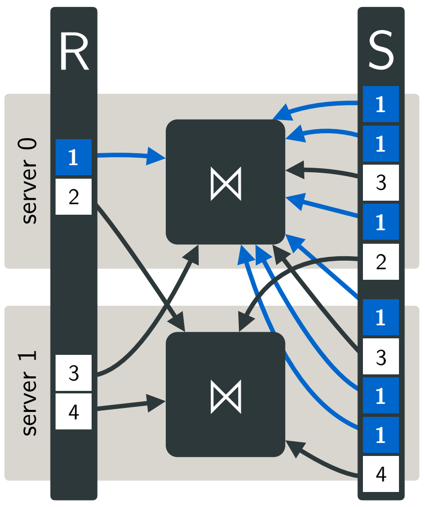
It is hard to detect skew at runtime without causing a significant overhead for non-skewed workloads. Materializing all tuples and computing histograms to decide on an optimal assignment takes time and will provide no benefit when the input is not skewed. While the additional computation might not add a noticeable overhead to the overall query execution time for slow interconnects such as Gigabit Ethernet, this is no longer the case for InfiniBand 4×FDR that is more than 50× faster. Any substantial computation in addition to core join processing is likely to increase query execution time when such high-speed networks are used.

Flow-Join is a skew-resilient distributed join algorithm with negligible overhead even for high-speed interconnects. It computes small, approximate histograms alongside partitioning to detect skew early. When a small percentage of the input has been processed, the frequencies in the histograms are checked. Servers exchange the approximate counts for join key values when they exceed a skew threshold—i.e., the expected frequency—by a large factor. Afterwards, all servers in the cluster know the heavy hitters. The tuples that join with heavy hitters are broadcast to avoid load imbalances before they arise. This can be further refined as more of the input is processed, which becomes important when the heavy hitters in the input vary over time.

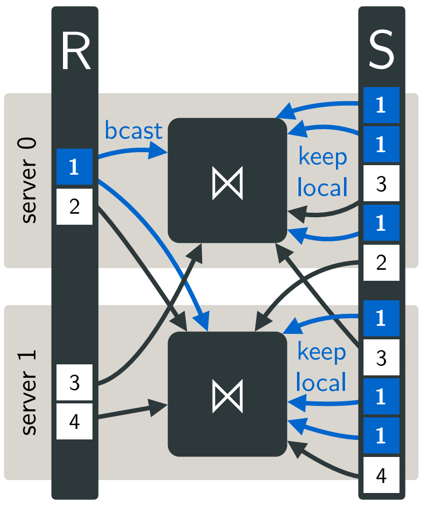
To simplify the presentation we restrict the discussion in this section to the common case of key/foreign-key equi joins. Section 4.4 generalizes Flow-Join to arbitrary joins without a key/foreign-key relationship as well as non-equi joins. For key/foreign-key joins, heavy hitter skew is by definition limited to the foreign-key side as the attribute values of the other side are necessarily unique as a consequence of the primary key property. Therefore, correlated skew—i.e., both inputs are skewed on the same join key value—is also not covered here but as part of Section 4.4.

4.2.1 Selective Broadcast

A standard hash join redistributes tuples between servers according to the hash value of the join key. Tuples with the same join key value will thus all end up at the same target server. An example is shown in Figure 4.4a: Server 0 is assigned all tuples with the skewed join key 1 and as a result receives 4× more tuples than server 1. This impacts the performance in two ways: First, network communication becomes irregular as most data is sent over the link to server 0. Second, server 0 must process many more tuples than its fair share. The execution time of distributed operators is determined by the slowest server. Consequently, the increased load at server 0 will slow down the entire query. This also affects the scalability of the system: Adding more servers to the cluster will not improve the



(a) Standard hash join



(b) Flow-Join

Figure 4.4: Selective Broadcast

performance as expected. The heavy hitter is still assigned to a single server and the query execution time thus remains largely unchanged.

An effective way to handle heavy hitter skew is the Selective Broadcast [87, 71, 65] redistribution method for distributed hash joins that is known as Subset-Replicate [22] in the case of range partitioning. The key idea of Selective Broadcast is to keep tuples with skewed join keys local to avoid overloading a single server. Instead, the corresponding tuples of the other input are broadcast to all servers so that the heavy hitter tuples can be processed locally. This is shown in Figure 4.4b where the R tuples with join key 1 are broadcast, while S tuples with join key 1 are kept local. An equi join algorithm that uses Selective Broadcast still computes the correct join result and at the same time avoids the load imbalance caused by a standard hash join (for the required changes to support non-equi joins see Section 4.4.3). Selective Broadcast yields performance results for skewed workloads that are on par with those for non-skewed inputs. For current systems, a distributed join using Selective Broadcast will in fact achieve higher performance for skewed than for non-skewed workloads as the heavy hitter tuples can be kept local and do not need to be materialized and sent over the network.

4.2.2 Heavy Hitter Detection

Even though Selective Broadcast seems to solve the problem of heavy hitter skew, there is one important caveat: The heavy hitter values have to be known beforehand. Previous approaches either assumed that heavy hitter elements can be

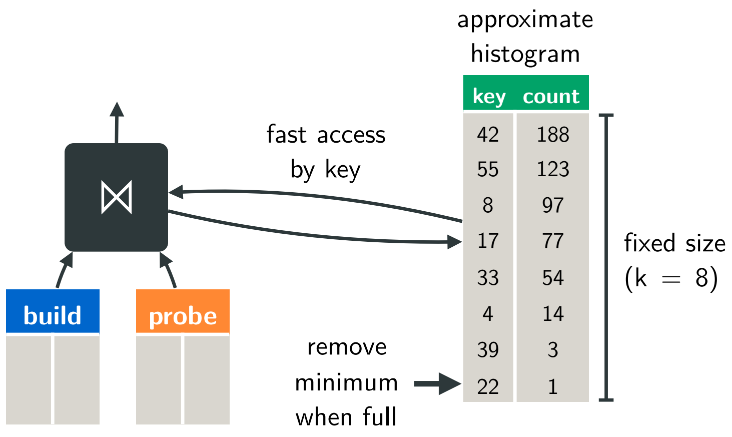


Figure 4.5: Heavy hitter detection

deduced from existing statistics [22, 87]—which is often difficult and expensive for intermediate results—or are computed during a separate analysis phase that requires a complete materialization of both join inputs and additional processing [49, 41, 85, 71].

Flow-Join identifies heavy hitters alongside partitioning and thus does not need detailed statistics or a separate analysis phase to detect skew. This enables pipelined join processing and thus avoids materialization of the probe side. The high-level idea of Flow-Join's algorithm to detect heavy hitters is shown in Figure 4.5. The join maintains approximate—and therefore extremely efficient—histograms for the probe input. Histograms are updated during join processing by incrementing the count for each probe tuple as it is processed. The tuple is only forwarded to the target server when its count is below the skew threshold. Otherwise, the probe tuple is kept local and the corresponding build tuple is broadcasted to ensure the correct result.

Frequent Items Problem

Detecting heavy hitters during distributed join processing corresponds to finding frequent items in a data stream. This is known as the *frequent items problem.* The subsequent definitions are adapted from a recent survey [16] on algorithms that compute frequent items:

Definition 1. Exact Frequent Items Problem:
Given a stream $\mathcal{S}$ of n items $t_1, ..., t_n$, the frequency of item i is $f_i = |\{j \mid t_j = i\}|$, i.e., the number of indices j where the jth item is i. The exact ϕ-frequent items for a frequency threshold of ϕn are then defined as the set $\{i \mid f_i > \phi n\}$ that contains all items that occur more than ϕn times.

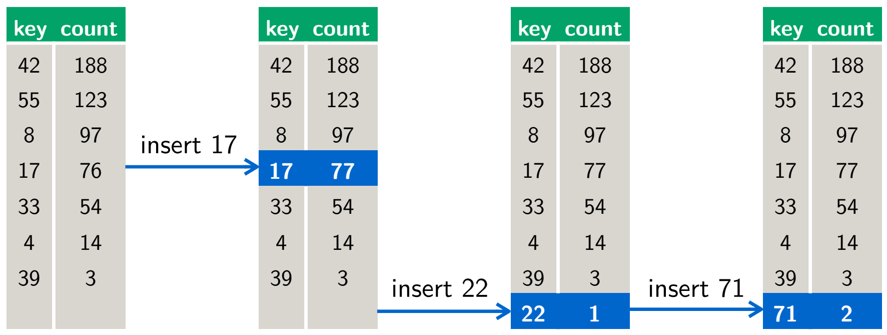


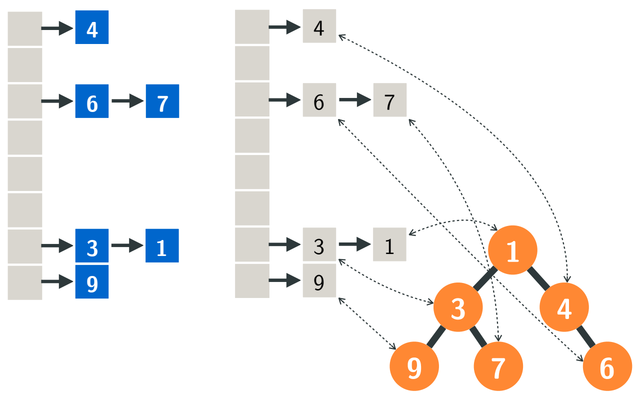
Figure 4.6: SpaceSaving

However, solving the frequent items problem *exactly* has been shown to require space linear in the size of the input [16]. Basically, a counter per join value would be required to detect the k most frequent items. Apart from the large amount of main memory needed, this would also be very compute-intensive to maintain due to cache misses and related issues. Thus, we instead focus on the *approximate* version of the frequent items problem with error tolerance ϵ:

Definition 2. Approximate Freq. Items Problem:
Given a stream $\mathcal{S}$ of n items, compute the set F of approximate frequent items so that each reported item $i \in F$ has frequency $f_i > (\phi - \epsilon)n$. This allows a reported item to occur ϵn times less often in the stream than the specified threshold of ϕn. Further, there should be no $i \notin F$ such that $f_i > \phi n$, i.e., all items i with a frequency larger than ϕn have to be reported.

Flow-Join's heavy hitter detection is based on the SpaceSaving algorithm [56], which solves the frequency estimation problem with error $\epsilon = 1/k$, where k is the histogram size. The frequency of a reported item is off by at most a factor of $1/k$. Solving the frequent items problem *approximately* with the SpaceSaving algorithm reduces the space requirements significantly compared to the exact version. For example, a histogram with 100 entries is sufficient to compute all items with frequency 1 % or higher. This already suffices to detect all heavy hitters that could potentially impact the performance of a cluster with dozens of machines.

SpaceSaving reports all frequent items. However, there is no guarantee that a reported item is indeed frequent. This is tolerable for our use case, as broadcasting a few additional tuples will not impact performance noticeably. The aforementioned survey on algorithms for the frequent items problem has shown that SpaceSaving is faster, more accurate, and requires less space than other counter-, quantile-, and sketch-based alternatives [16], which we will thus not consider. Figure 4.6 shows an exemplary execution of the SpaceSaving algorithm for a histogram with space for $k = 8$ elements. In the example there are already 7 elements in



(a) Hash table (b) Hash table + heap (c) Hash table + sorted array

Figure 4.7: Implementation alternatives for the approximate histogram

the histogram from the beginning. The insertion of the existing key 17 simply increments its count. The subsequent insertion of the new value 22 fills the last remaining free slot. Consequently, there is no free slot for key 71, which instead replaces the element with the smallest count. The count of the previous element 22 is kept and incremented by one, yielding a total count of 2 for the new element 71. Keeping the count of the previous element when a new element is inserted into a full histogram is necessary to ensure an item is never underestimated. This approach bounds the error for the frequency estimation problem to $1/k$ [56].

Data Structures

The main challenge introduced by the SpaceSaving algorithm is that elements are accessed in two different ways: via their *key* for updates and their *count* to remove the minimum. The approximate histogram has to support both operations in a very efficient manner. To this end, we compared three successively-refined data structures characterized in Figure 4.7. The first data structure, depicted in Figure 4.7a, is a hash table that enables key access in $O(1)$. However, removing the minimum is expensive as it incurs a full scan of the hash table with $O(k)$ operations. Figure 4.7b combines the hash table with a heap to reduce the cost for accessing the minimum. However, the heap requires $\log k$ moves for every update and removal. Thus, insert, update, and remove minimum are now all in $O(\log k)$. Figure 4.7c replaces the heap with a sorted array. This enables a remove minimum that is in $O(1)$. While "increment key" is now worst case $O(k)$, this only occurs for the rightmost element when all elements have the same count. In practice, however, an element rarely moves more than one position to the left.

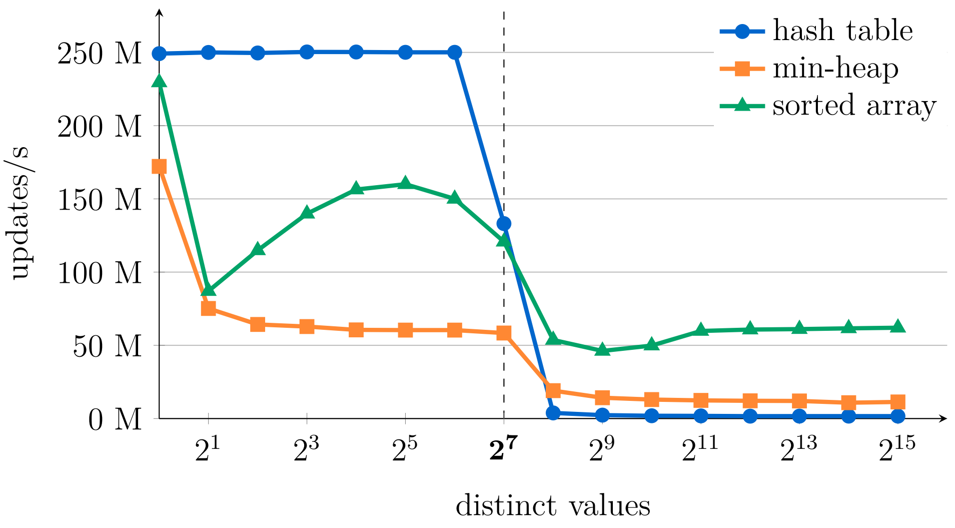


Figure 4.8: Update rates of the approximate histogram alternatives

The experiment shown in Figure 4.8 inserts 10 million random keys for a uniform distribution without skew into histograms that have capacity for $k = 128$ elements. The number of distinct values increases along the x-axis. Note that the x-axis is logarithmic so that the hash table outperforms the alternatives only as long as there are very few distinct values. The update rate of the hash table drops severely as soon as the number of distinct values exceeds the capacity $k = 128$. At this point the remove minimum operation becomes necessary, which incurs an expensive scan of the hash table. Combining the hash table with a heap to support a fast remove minimum operation improves performance by up to 7× once there are more than $k = 128$ distinct values. The sorted array implementation further improves over this and outperforms the hash table by at least 28× and up to 37× when there are more than $k = 128$ distinct values. Increasing k shifts this limit but does not change the qualitative result. A reasonably small k is necessary to avoid expensive cache misses.

The sorted-array implementation is very fast, processing about 60 million keys/s *per thread*. The implementation of Flow-Join uses one thread per core to leverage the available parallelism of modern many-core servers. Worker threads do not share a single approximate histogram but instead each use their own private histogram. This avoids the cost of locking or atomic operations required when sharing a data structure. Flow-Join uses the sorted-array implementation with a capacity of $k = 128$ elements and 256 hash-table entries. This results in a hash table load factor of at most 1/2 and a memory footprint of only 2.5 KiB, ensuring fast in-cache processing. This allows to process skewed and non-skewed inputs with minimal overhead.

4.2.3 Early and Iterative Detection

Early detection of heavy hitter values is important to adapt the redistribution scheme as soon as possible such that imbalances in network communication and CPU utilization can be resolved before they impact the join performance. Flow-Join combines histograms on a per-server level after processing approximately 1 % of the probe input. Afterwards, the servers combine the resulting skew lists on the cluster level to reach a global consensus on the heavy hitter values. This is quite cheap: Each server sums up the counts for the join key values of its local per-thread approximate histograms. It then forwards those join key values together with their counts that still exceed the skew threshold to one of the servers. This one server again adds up the counts and thus determines the global list of heavy hitter skew values as those join key values that still exceed the threshold. It notifies the servers responsible for the corresponding build tuples (identified via the join's hash function). These servers then broadcast the build tuples across the cluster.

Sampling the first 1 % of the input ensures the early detection of the heavy hitter values. Once the build tuples are broadcasted, the materialized skewed probe tuples can be joined. This process can be repeated periodically during join processing to iteratively refine the heavy hitter detection in case the heavy hitters in the probe input vary over time.

4.2.4 Fetch on Demand

Combining histograms in the cluster to detect heavy hitters did not introduce a noticeable overhead in our experiments even for 32 servers. Still, the global consensus requires synchronization between servers. It is possible to avoid synchronization and reduce the materialization of heavy hitters to a minimum: Instead of creating a shared global list of skewed join values, servers (or even individual worker threads) can decide on the heavy hitter values on their own. Once a join key value exceeds the skew threshold, they fetch the corresponding build tuple asynchronously from the responsible server (identified via the join's hash function) similar to the PRPQ scheme by Cheng et al. [12]. A requesting server does not even need to wait for the response but can continue processing its input and simply materialize the probe tuples for the outstanding heavy hitter. Once the remote server has responded with the build tuple, the requesting server can join the probe tuples for this heavy hitter value. We call this refined approach *fetch on demand*.

Fetch on demand has a second benefit apart from avoiding the synchronization of the global consensus approach. Different servers (or even worker threads) could encounter different heavy hitters in their part of the input. Fetch on demand allows them to request only the build tuples for heavy hitters relevant to them while global consensus always broadcasts the build tuples for all heavy hitters across the whole cluster.

4.2.5 Hash Join Algorithm

There are different ways to implement a distributed hash join including the classic Grace-style hash join [50] and distributed radix joins [5]. Both suffer from load imbalances caused by heavy hitter skew as they have to assign each heavy hitter join key value to a single server. Barthels et al. [5] have shown that a distributed radix join experiences a 3.3× slow down for a skewed workload with Zipf factor $z = 1.2$ on an 8-server cluster. We observed similar results for the standard partitioned hash join, which suffers from a 2.1× slow down for Zipf factor 1.25. The effect of skew further intensifies with the cluster size—a standard hash join slows down by 5.2× on a 32-server cluster for a skewed workload with Zipf factor $z = 1.25$. This highlights the need for low-overhead skew handling techniques whether the underlying join algorithm is a standard Grace-style hash join or a distributed radix join.

Flow-Join is independent of the actual hash join algorithm used. During partitioning of the tuples Flow-Join uses approximate histograms to detect skew and Selective Broadcast to handle the detected heavy hitters. This applies to Grace-style hash joins [50] as well as radix joins [5]. We decided to implement the former to evaluate Flow-Join. In the experiments we focus on the worst case for Flow-Join where the build relation is small enough to fit into cache. In this setting, the overhead of skew detection and skew handling has the largest visible impact as the join performs fastest. A larger build side would cause cache misses, slowing down the join and making the overhead of Flow-Join less visible.

4.3 Implementation Details

Our implementation of Flow-Join uses remote direct memory access (RDMA) for network communication to utilize all the available network bandwidth offered by modern high-speed interconnects such as InfiniBand 4×FDR. Flow-Join combines exchange operators for distributed processing with work stealing across cores and NUMA regions for local processing to be able to scale to large clusters of many-core machines.

4.3.1 High-Speed Networks

InfiniBand is a high-bandwidth, low-latency cluster interconnect, which offers several different data rates that have been standardized over the years since its in-

	GbE	InfiniBand (4×)				
		SDR	DDR	QDR	**FDR**	EDR
bandwidth [GB/s]	0.125	1	2	4	**6.8**	12.1
compared to GbE	1×	8×	16×	32×	**54×**	97×
latency [µs]	340	5	2.5	1.3	**0.7**	0.5
introduction	1998	2003	2005	2007	**2011**	2014

Table 4.1: Network data link standards

troduction in 2001 as shown in Table 4.1. Most experiments in this chapter were performed on a 32-server cluster that is connected via InfiniBand 4×FDR hardware that provides more than 50× the bandwidth of Gigabit Ethernet and latencies as low as 0.7 µs.

InfiniBand offers the choice between two transport protocols: standard TCP via *IP over InfiniBand* (IPoIB) and an InfiniBand-native *ibverbs* interface that enables remote direct memory access (RDMA). Figure 4.9 compares the throughput achieved via TCP/IP over Gigabit Ethernet and InfiniBand with that of RDMA for a single full-duplex stream of 512 KiB messages. The experiment shows that RDMA is necessary to achieve a network throughput near the theoretical maximum of 6.8 GB/s for InfiniBand 4×FDR. This stands in contrast to TCP/IP, which causes significant CPU load—fully occupying one core—and would further require complex tuning as well as the use of multiple data streams processed on several cores to come close to the throughput of RDMA [70]. For the 32-server cluster used in the evaluation of this chapter with 8 cores per CPU and a fast 4×FDR interconnect, using all the available bandwidth with TCP/IP would likely occupy most of the cores of one of the two CPUs—compared to virtually no CPU overhead for RDMA. Flow-Join therefore facilitates native InfiniBand network communication via RDMA instead of standard TCP/IP.

InfiniBand's *ibverbs* interface is inherently asynchronous and thus requires a distinctly different application design. The InfiniBand network card processes the work requests that the application posts to its work queues asynchronously and inserts a completion notification to a completion queue once it is done. As its name suggests, RDMA reads and writes memory without involving the operating system or application during transfers. The performance-critical data transfer path thus involves no CPU overhead at all. The application has to manage network buffers explicitly to enable these zero-copy network transfers. Registering the network buffers with the network card before communication is necessary as the kernel has to pin them to main memory to avoid swapping to disk. Network buffers

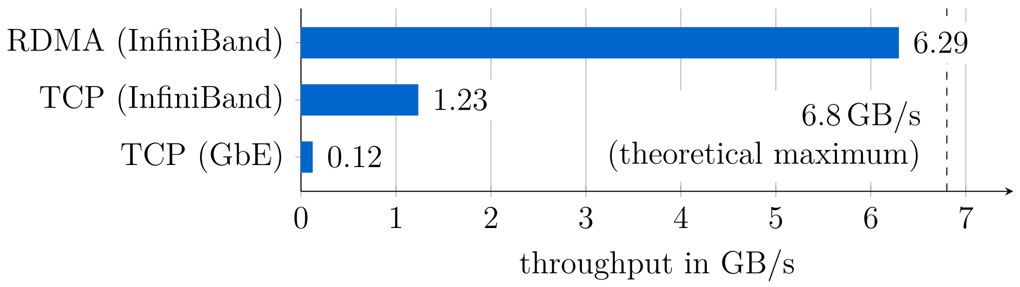


Figure 4.9: RDMA vs. TCP throughput

should be reused as much as possible as registration with the network card is a costly operation [28]. Our implementation uses a message pool to reuse network buffers and avoid the cost of repeated memory allocation and registration with the InfiniBand card.

RDMA further offers two different application semantics: Memory semantics allow the sender to read and write the receiver's memory without its involvement, requiring that both exchange information beforehand to identify the target memory region. With channel semantics both sender and receiver post work requests that specify the memory regions, rendering the preceding information exchange unnecessary. Channel semantics offer the additional benefit that not only the sender but also the receiver is notified when the data transfer has finished. For these reasons, we used channel semantics to implement Flow-Join's communication multiplexer.

4.3.2 Local and Distributed Parallelism

The exchange operator [37] is a landmark idea as it allows systems to encapsulate parallelism inside an operator. All other relational operators are kept oblivious to parallel execution, making it straightforward to parallelize existing non-parallel systems. The exchange operator is commonly used to introduce parallelism both inside a single machine and between servers (e.g., Vectorwise Vortex [17] and Teradata [87]). However, the classic exchange operator has several disadvantages as it introduces unnecessary materialization overhead during local processing, is inflexible in dealing with load balances and thus especially vulnerable to attribute value skew, and further faces scalability problems due to the large number of connections required for large clusters of many-core servers [70].

Flow-Join instead implements local and distributed parallelism differently. On the local level, instead of using traditional exchange operators, we parallelize Flow-Join similar to the morsel-driven approach by Leis et al. [52]. Workers process small NUMA-local chunks of tuples to avoid expensive remote memory accesses across

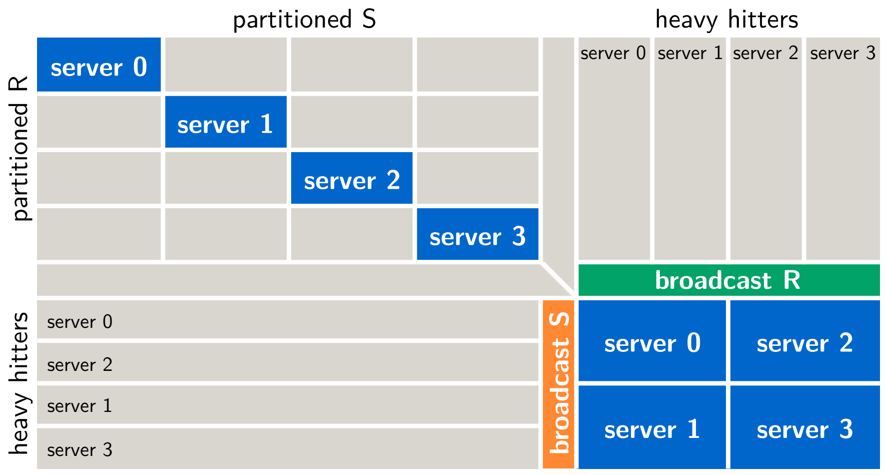


Figure 4.10: Generalized Flow-Join

CPU sockets. Work stealing allows faster workers to steal work units from workers that lag behind and by this effectively resolves local load imbalances. The impact of work stealing grows with the number of cores. Threads are pinned to cores to avoid expensive thread migrations. On the global level, exchange operators are connected via RDMA-based communication multiplexers instead of directly to each other. The multiplexer ensures that there is always at least one packet in flight for every target server to use all the available bandwidth in the network. Both levels of parallelism are seamlessly integrated into a new approach that avoids unnecessary materialization, is flexible in dealing with load imbalances, and offers near-linear scalability in the number of servers in the cluster.

4.4 Generalized Flow-Join

This section generalizes Flow-Join beyond key/foreign-key equi joins. The high-level idea is shown in Figure 4.10: Tuples with a join value that is neither skewed in R nor in S are partitioned, R tuples that join with heavy hitters in S are broadcast and S tuples that join with heavy hitters in R are handled similarly. Tuples with a join value that is both a heavy hitter in R and in S are redistributed according to the Symmetric Fragment Replicate (SFR) [74] data shuffling scheme.

4.4.1 Algorithm

In the following we will describe the generalized Flow-Join in more detail. We denote the build input as R and the probe input as S. Typically, the smaller input is used as build side to keep the hash table for the local join small, we therefore follow this convention. However, Flow-Join's skew detection and handling does not require this. The build input R is distributed before the probe input S to enable pipelined probing. Flow-Join avoids materialization as much as possible to minimize memory consumption. In detail, the generalized Flow-Join proceeds as follows:

1. **Exchange $\boldsymbol{R}$:** For each tuple in R:
   - Update approximate histogram
   - Compare heavy hitter count to threshold:
     (a) *skewed:* insert tuple into local hash table
     (b) *otherwise:* send tuple to target server
   - Create global list of heavy hitters in R
2. **Exchange $\boldsymbol{S}$:** For each tuple in S:
   - Update approximate histogram
   - Compare heavy hitter count to threshold:
     (a) *skewed in S:* materialize S tuple as the corresponding R tuple is not broadcast at this point in time
     (b) *skewed in R (but not S):* broadcast S tuple to all servers
     (c) *otherwise:* send S tuple to target server
   - Create global list of heavy hitters in S
3. **Handle skew:** (necessary if skew is detected in S)
   (a) Broadcast R tuples that join with heavy hitters in S that are not also heavy hitters in R, join the corresponding materialized S tuples
   (b) Redistribute tuples whose join key is a heavy hitter in both R and S via the Symmetric Fragment Replicate redistribution scheme

The generalized Flow-Join computes the correct join result as explained in the following paragraphs:

- The algorithm partitions tuples with join key values that are **not skewed** in R nor in S in step 1(b) and 2(c). These tuples are joined correctly at their target server.

- R tuples for heavy hitters **skewed only in $\boldsymbol{R}$** are kept local after the skew threshold is met in step 1(a) while before that they were sent to the target server in step 1(b). In both cases the tuples are joined correctly as the corresponding S tuple for the heavy hitter is broadcast in step 2(b).

- S tuples for heavy hitters **skewed only in *S*** are materialized after the skew threshold is met in step 2(a) while before that they were sent to the target server in step 2(c). The S tuples sent to the target server are joined correctly as the corresponding R tuple was previously sent there in step 1(b). The materialized S tuples are probed locally after the corresponding R partition has been broadcast in step 3(a).
- R tuples for heavy hitters **skewed in both *R* and *S*** are kept local after the skew threshold is met in step 1(a) while before that they were sent to the target server in step 1(b). In both cases the tuples are redistributed using SFR in step 3(b). Note that these tuples were not replicated in step 3(a) and exist only on a single server, avoiding spurious results.
- S tuples for heavy hitters **skewed in both *R* and *S*** are materialized after the skew threshold is met in step 2(a). Note that S tuples broadcast in step 2(b) before the skew threshold was met were probed correctly and are not considered in this step. The materialized S tuples and corresponding R tuples are redistributed and joined via SFR in step 3(b).

The following section describes step 3(b) and the handling of correlated skew with SFR in detail.

4.4.2 Correlated Skew

A join key value that is a heavy hitter for both inputs is even more problematic than tuples skewed for only one input. The join for such a heavy hitter is basically a cross product and thus causes not only a severe load imbalance during data redistribution but also generates a quadratic number of result tuples on a single server. While broadcasting the tuples from one of the two inputs—as proposed by previous approaches [22, 87]—balances data redistribution and result generation across servers, it still incurs a significant cost for broadcasting a large number of heavy hitters.

The Symmetric Fragment Replicate (SFR) [74] data shuffling scheme handles correlated skew at least as good as a broadcast. In many cases SFR is able to reduce query execution time significantly. Instead of assigning all heavy hitter tuples to a single server or broadcasting them, the servers are logically arranged in a grid. Servers replicate heavy hitters for one input across rows, while those of the other input are replicated across columns. An example with $n = 9$ servers is shown in Figure 4.11. This scheme ensures that every heavy hitter tuple from one input is joined exactly once with every heavy hitter tuple of the other side. The join site for two heavy hitter tuples from relation R and S, respectively, is the server at the intersection of the corresponding row and column. SFR reduces

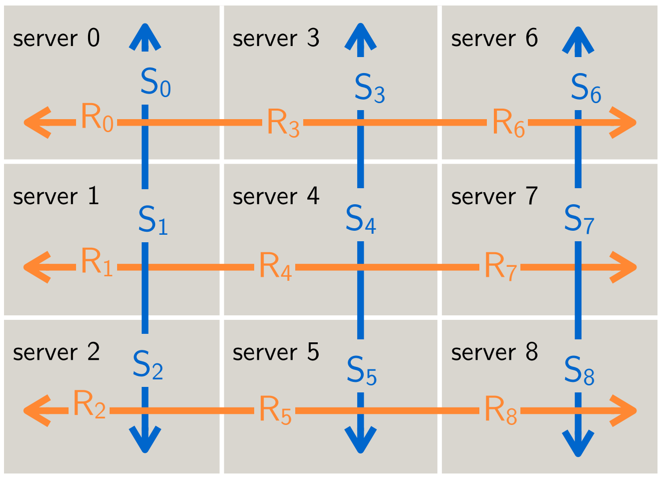


Figure 4.11: Symmetric Fragment Replicate

query execution time and network traffic by up to a factor of $(\sqrt{n}+1)/2$ compared to broadcasting.

Assuming a cluster of n servers and a heavy hitter that occurs x times in R and y times in S per server: Assigning the heavy hitter to a server costs $(n-1)(x+y)$ time units as every server has to send the $x+y$ heavy hitter tuples to the target server (except the target server itself), causing a congestion on the receive link of the target server. Broadcasting R costs $(n-1)x$ time units as every server sends its x heavy hitters of the smaller side R to every other server using all network links in parallel. SFR reduces this to $(n_1-1)x+(n_2-1)y$ time units, where n_1 and n_2 are the number of rows and columns of the SFR rectangle (with $n_1 \times n_2 = n$): The heavy hitter tuples of R are replicated across n_1 rows and those of S across n_2 columns. Again, all links of the network are used in parallel.

The optimal grid shape depends only on the relative frequency of the heavy hitter in both inputs, e.g., a quadratic shape is best when the heavy hitter occurs in both inputs with roughly the same frequency. The other extreme is a rectangle with only one row (or column) for a heavy hitter that occurs mostly in one of the two inputs. In this case SFR degenerates to a broadcast. Figure 4.12 shows the optimal rectangles for $n = 36$ servers for different heavy hitter frequency ratios and the corresponding speedup of SFR over a broadcast. For example, when frequencies differ only by up to a factor of 1.5, the quadratic 6×6 shape is the best choice and improves performance by up to $3.5\times$ (general case: $\frac{\sqrt{n}-1}{2}\times$) compared to broadcasting. On the other hand, when the frequency for one input is more than $18\times$ (general case: $\frac{n}{2}\times$) larger than for the other, SFR degenerates to a broadcast with a 1×36 shape (general case: $1 \times n$). The potential rectangles are

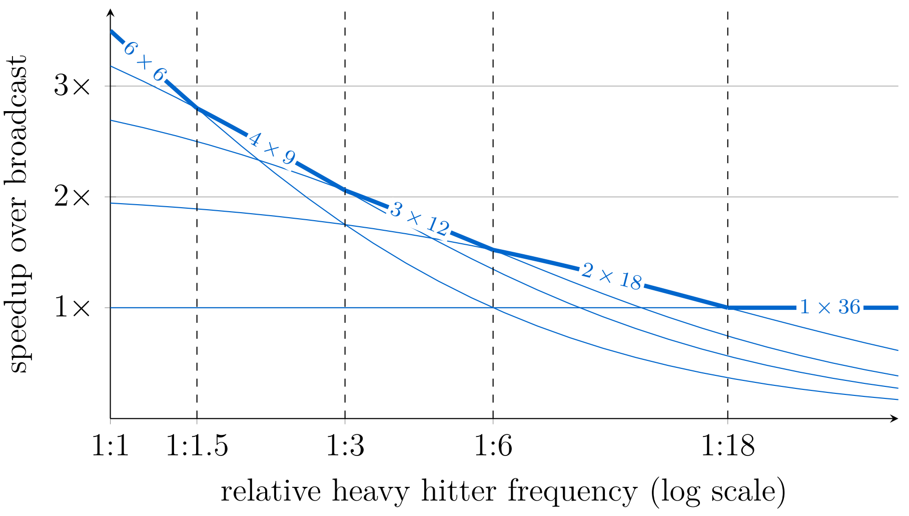


Figure 4.12: Optimal SFR rectangles

limited to the integer divisors of the number of servers n. For example, a quadratic shape is only possible when the number of servers is a square number.

SFR reduces the load imbalance, i.e., query execution time, but not necessarily the amount of network traffic. While SFR never sends more tuples than broadcasting, the assignment of the heavy hitter to a single server can reduce the network traffic at the cost of a significantly increased execution time. The assignment of the heavy hitter to a single server transfers $(n-1)(x+y)$ tuples as every server sends their heavy hitter tuples to the target server (except the target server itself). A broadcast transfers $n(n-1)x$ tuples as every server sends the heavy hitter tuples for the smaller input to every other server. SFR reduces network traffic to $n((n_1-1)x+(n_2-1)y)$ tuples as every server sends its heavy hitter tuples for one input across the n_1 rows and for the other input across n_2 columns.

4.4.3 Non-Equi Joins

Similar to a broadcast join [24] that replicates the smaller input across all servers while it keeps the larger one fragmented, Flow-Join has to be adapted to compute the correct result for semi, anti, and outer joins. This is owed to the Selective Broadcast and SFR redistribution schemes, which replicate tuples and can thus create spurious and duplicate results.

The semi join may produce duplicate result tuples at different join sites. The result therefore has to be redistributed to eliminate these duplicates. The anti join will produce valid results m times, once for each of the m join sites the tuple was replicated to. A tuple may find a join partner at only a subset of its join sites and

	$R \ltimes S$	$R \rtimes S$	$R \triangleright S$	$R \triangleleft S$	$R ⟖ S$	$R ⟕ S$
hash join	✓	✓	✓	✓	✓	✓
Flow-Join	✓[1]	✓[1]	✓[1]	✓[1]	✓[2]	✓[2]

Table 4.2: Non-equi joins

thus cause spurious results at the remaining sites, where it does not find a join partner. The result needs to be redistributed and a tuple is to be included in the result only if it occurs exactly m times, i.e., if it found no join partner at any of its join sites. The outer join may produce duplicates and false results similar to the anti join. Dangling tuples have to be redistributed and counted. Only dangling tuples that found no partner across all of their join sites have to be kept in the final result. Table 4.2 summarizes the changes required for Flow-Join to support non-equi joins.

4.5 Evaluation

This section evaluates our approach with two scenarios: In the first scenario, we generate skewed data that follows a Zipf distribution, which allows us to scale the input size easily with the number of servers for a scalability experiment. The second scenario uses a real workload from a large commercial vendor, demonstrating that Flow-Join can improve query processing performance in practice. At last, we will evaluate several parameters, including the build input size, the Zipf factor z, the skew threshold, and the maximum number of heavy hitters each node reports.

4.5.1 Experimental Setup

The experimental setup is illustrated in Figure 4.13 where the line width of each connection corresponds to its available bandwidth. We conducted most experiments in this chapter on a cluster of 32 servers connected via Connect-IB InfiniBand cards at a 4× fast data rate (FDR) resulting in a theoretical network bandwidth of 6.8 GB/s per incoming and outgoing link. The aggregate bandwidth of the cluster for all-to-all data shuffles is thus 218 GB/s. Each Linux server is equipped with two Intel Xeon E5-2660 CPUs clocked at 2.2 GHz with 8 physical cores each (16 hardware contexts per CPU due to hyper-threading) and 64 GiB of

[1]Requires redistribution of the result.

[2]Requires redistribution of dangling tuples.

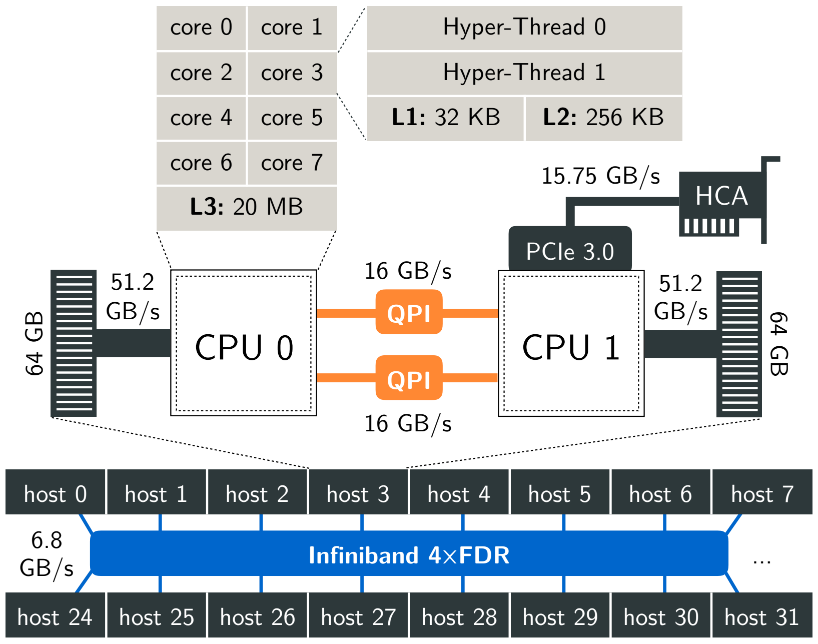
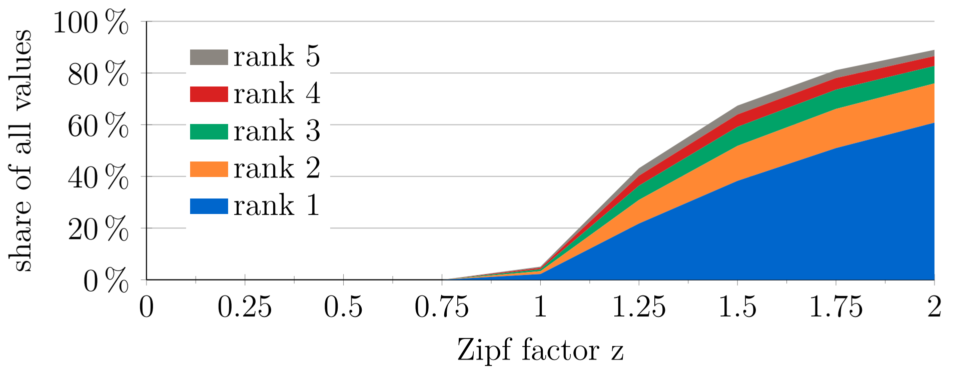


Figure 4.13: Experimental setup

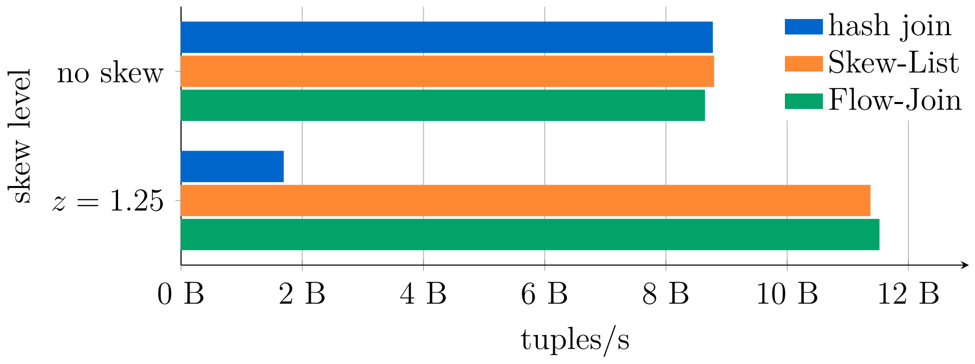
main memory per CPU—resulting in a total of 512 cores (1024 hardware contexts) and 4 TiB of main memory in the cluster.

4.5.2 Scalability

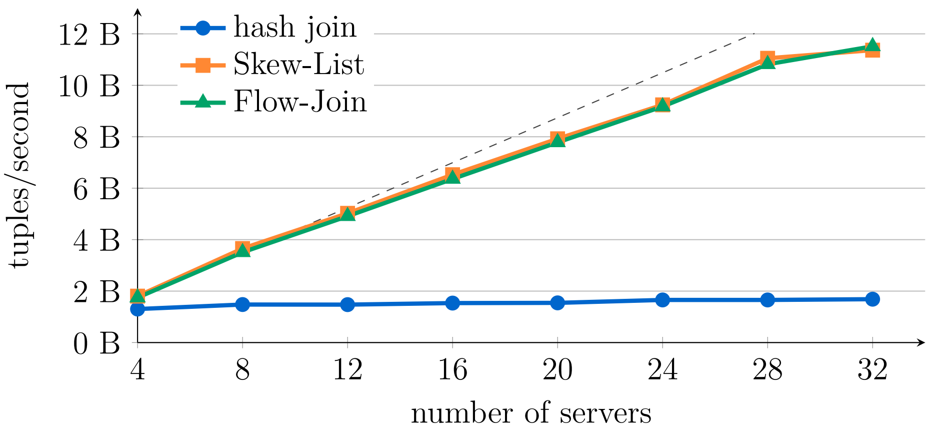
The first scenario depicted in Figures 4.14a, 4.14b and 4.14c compares Flow-Join to a standard hash join and Skew-List. Skew-List is an omniscient variant of Flow-Join that knows all heavy hitter values beforehand. It takes a predefined list of heavy hitter values as part of its input instead of performing skew detection at runtime. Tuples consist of 64 bit key and 64 bit payload. The keys of the probe tuples follow a Zipf distribution with Zipf factor $z = 1.25$. The Zipf distribution is known to model real world data accurately, including the size of cities and word frequencies [38]. Given n elements ranked by their frequency, a Zipf distribution with skew factor z denotes that the most frequent item with rank 1 accounts for $x = 1/H_{(n,z)}$ of all values, where $H_{(n,z)} = \sum_{n}^{i=0} 1/i^z$ is the nth generalized harmonic number. The element with rank r occurs x/r^z times. Figure 4.14a illustrates the impact of the Zipf factor z on the share of the five most frequent elements for 64 bit integers. For $z = 1$, these five values occur in 5 % of all tuples, increasing to 43 % for $z = 1.25$, 67 % for $z = 1.5$ and 89 % for the extreme case of $z = 2$. Our experiments use Zipf factor $z = 1.25$. The input consists of 5040 build and 105 M probe tuples per server. A build input that fits into cache represents the worst



(a) Ratio of the five most frequent keys for increasing Zipf factor z (64 bit integers)



(b) Minimal overhead for inputs without skew, 6.8× faster with skew (32 servers)



(c) Flow-Join scales near-linearly before the switch is a bottleneck ($z = 1.25$)

Figure 4.14: Performance results for Flow-Join on synthetic inputs

case for Flow-Join: It ensures fastest processing, exposing any overhead incurred by skew detection.

Figure 4.14b compares the three distributed join algorithms using all 32 servers of the cluster. The experiment reveals two important findings: First, Flow-Join imposes only an insignificant overhead of 1.5 % for non-skewed workloads, joining 8.6 billion tuples/s (408 ms) compared to 8.8 billion tuples/s (402 ms) for the hash join and the Skew-List algorithm. Second, Flow-Join performs 6.8× better than the standard hash join for a skewed input with Zipf factor $z = 1.25$, joining 11.5 billion tuples/s (305 ms) compared to the 1.7 billion tuples/s of the standard hash join (2.1 s). There is no measurable performance overhead compared to the omniscient Skew-List algorithm that knows the heavy hitter values beforehand. Flow-Join's performance will further improve with the degree of skew as an increasing number of heavy hitters can be kept local and thus are neither materialized nor sent over the network.

Figure 4.14c evaluates the scalability of the three join algorithms for the skewed workload by increasing the number of servers in the cluster. The standard hash join does not scale well as it assigns all tuples of a heavy hitter join key to a single server and thus causes a severe load imbalance during data redistribution. Adding more servers improves performance only minimally. In contrast, Flow-Join scales near-linearly up to 28 servers. At this point, the switch becomes the bottleneck and throughput drops. Uncoordinated all-to-all communication reduces the aggregate bandwidth due to negative effects such as credit starvation for InfiniBand. Network scheduling [70] is required to scale beyond 32 servers.

4.5.3 Real Workload

The second scenario evaluates Flow-Join for a real workload from a large commercial vendor. The query plan of the use case is shown in Figure 4.15. The query plan follows the convention that the left input is used as build input for the local hash join while the right input is used to probe the hash table. The highlighted join operator is subject to skew on the probe side. It is difficult to anticipate the skew from table statistics alone in this and similar cases as the probe input is not a base relation but another join. The input consists of 53 000 build and 3.7 billion probe tuples, with a combined size of 55 GiB.

Figure 4.16a depicts the join key distribution for the probe input. The five most frequent join values occur in 9.4 % of all tuples. This moderately skewed input already leads to a huge imbalance during hash repartitioning as illustrated in Figure 4.16b for 32 servers: The largest partition has 2.6× more tuples than the expected 1/32 of the input. This limits the scalability of a standard hash join as this one large partition has to be assigned to a single server, which consequently slows down the entire join. Flow-Join instead scales much better due to

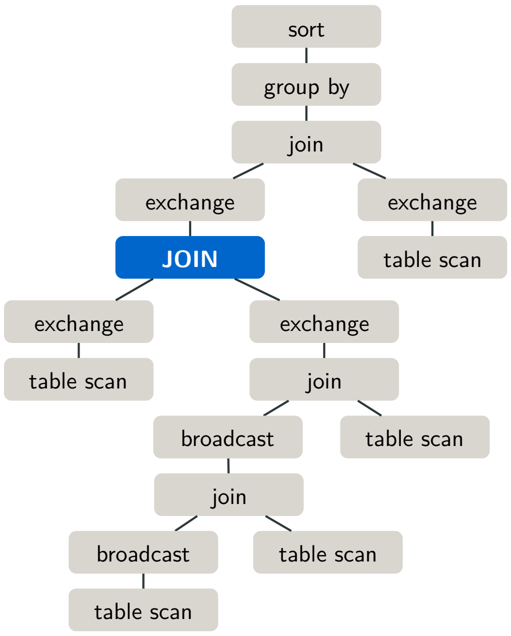
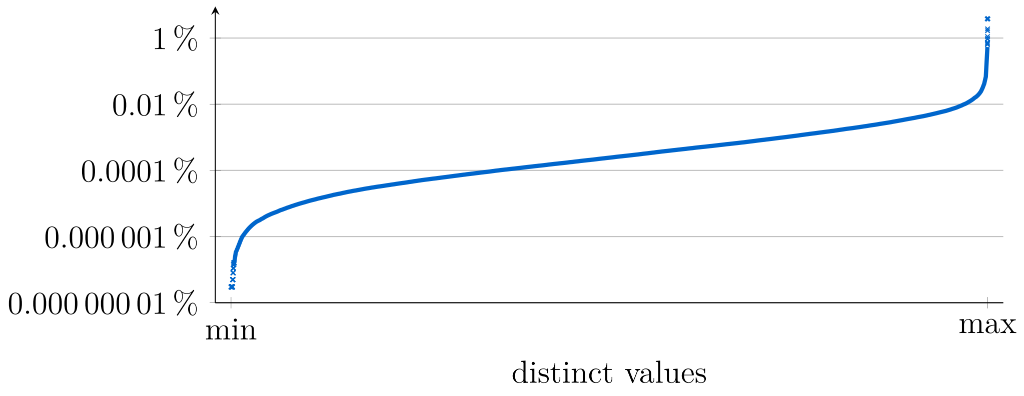


Figure 4.15: Query plan for the case study

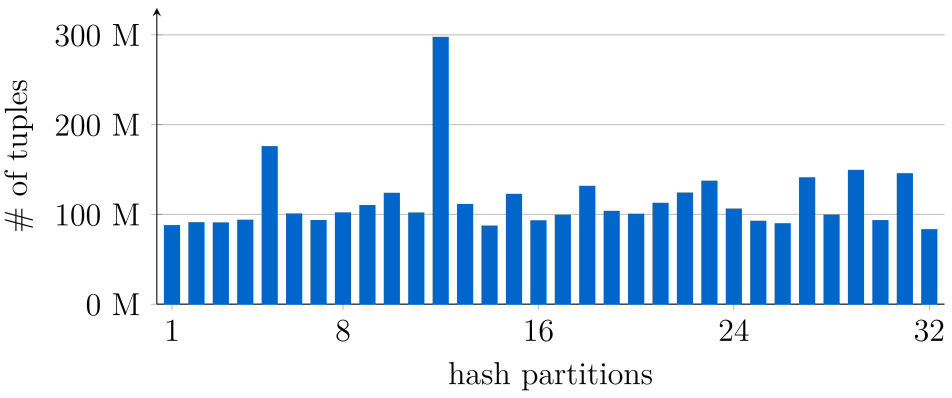
its low-overhead skew handling scheme as shown in Figure 4.16c. It processes the 55 GiB input in only 350 ms using 32 servers. However, in this experiment network congestion already kicks in at 28 servers due to the lower degree of skew, which causes a larger amount of data to be shuffled across the network. Again, network scheduling would be necessary to keep up near-linear scalability. The outlier for the hash join at 28 servers is due to the hash function that in this case assigns heavy hitters more evenly across partitions.

4.5.4 Impact of Workload Characteristics and Parameters

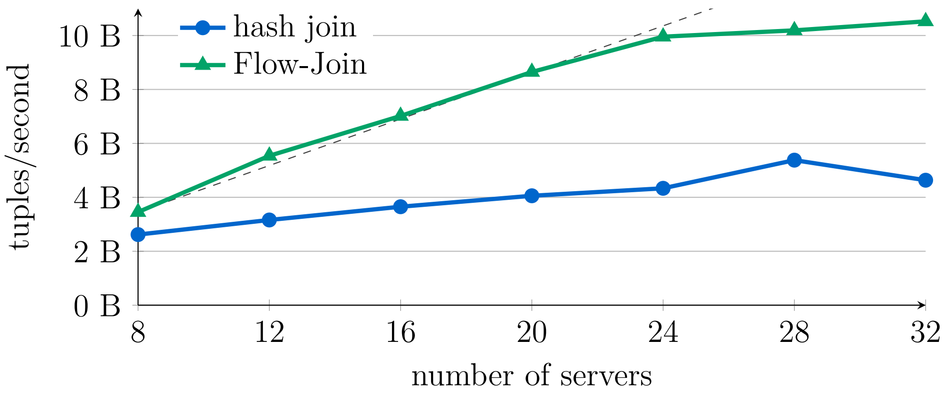
The experiments in this chapter use a build input that fits into cache as this represents the worst case for Flow-Join, revealing any overheads caused by skew detection and handling. We also conducted experiments that vary the size of the build and probe input. Our experiments revealed that the join duration increases proportionally with the size of the probe input as expected, when the size of the build input is fixed. Using a larger build input of 64 MiB, which exceeds the L3 cache, increases the cost for probing the hash table, which increases the join runtime by 2.5×—independent of whether skew detection is performed or not. However, as a consequence the relative overhead for skew detection and handling in fact decreases: The actual join processing becomes slower, but the cost for Flow-Join's heavy hitter detection stays the same. When the build input exceeds the



(a) Data distribution of the case study: the most common key occurs in 3.8 % of the tuples



(b) Largest probe partition has 2.6× more tuples than the expected 116 million



(c) Flow-Join scales near-linearly up to 24 servers (outlier due to hash function)

Figure 4.16: Performance results for Flow-Join on a real workload

cache, a partitioning join such as [54, 77] could be used. Flow-Join's techniques to detect and handle skew still apply.

The skewness of the data is another important workload characteristic. We varied the Zip factor z between 0 and 2. The runtime of the standard hash join increases proportionally with the size of the largest probe partition, which is determined by the most frequent heavy hitter. The effect is even higher when several heavy hitters are assigned to a single partition.

Flow-Join has two important tuning parameters: The maximum number of heavy hitters h reported per node during global consensus (by default h is unlimited) and the skew threshold (defaults to 0.01 %). Together they determine how many heavy hitters are reported during skew detection. A node reports up to h heavy hitters that exceed the skew threshold. Lowering the skew threshold respectively increasing h will report more heavy hitters and thus broadcast more build tuples across the cluster. We conducted an experiment with 8 servers, Zipf factor 1.25, and unlimited h that varies the skew threshold from 0.01 % to 100 % in steps of 10×. For 0.01 % 121 heavy hitters are reported and the join executes in 252 ms. Using a threshold of 0.1 % detects 49 heavy hitters, increasing the runtime by 1.2 % to 255 ms. A threshold of 1 % reveals 12 heavy hitters, resulting in a 12 % higher execution time of 282 ms. A threshold of 10 % discovers only one heavy hitter, increasing the runtime by 38 % to 349 ms. A threshold of 100 % does not detect any heavy hitters, giving a 134 % longer runtime of 592 ms. We observed similar results when the skew threshold is fixed to 0.01 % and we instead vary the maximum number of heavy hitters h reported per node: Choosing $h = 128$ results in a runtime of 252 ms, $h = 64$ increases this by 2 % to 257 ms, $h = 16$ leads to a 15 % higher execution time of 289 ms, $h = 1$ gives a 40 % longer runtime of 353 ms, and $h = 0$ results in a 135 % increased runtime of 593 ms.

4.6 Related Work

Making distributed joins resilient to attribute value skew is a popular topic in the database literature [49, 41, 22, 85, 73, 87, 71, 68]. Yet, most approaches add significant overheads for non-skewed workloads—especially for high-speed interconnects that do not allow to hide the extra computation and additional phases. Often the inputs are materialized and scanned completely [49, 41, 85, 71]. Other approaches require detailed statistics [22, 87] that might be overly inaccurate or unavailable for intermediate results. Flow-Join instead detects heavy hitters during partitioning and does not rely on statistics.

DeWitt et al. [22] first introduced the idea to replicate tuples that join with heavy hitters for range partitioning called Subset-Replicate. Xu et al. [87] applied the idea to hash partitioning and called it Partial Redistribution & Partial Duplication (PRPD). PRPD requires a predefined list of skewed values in contrast to Flow-Join, which detects heavy hitters at runtime. PRPD uses every hardware context as a separate parallel unit, i.e., there is no work stealing inside a single server. Skew effects are thus much higher as heavy hitter values are sent to a single parallel unit. PRPD joins at only 20 000 tuples/s on a 10 server cluster with 8 hardware contexts per server using an unspecified "high-speed interconnect" [87].

Eddy [80] and Flux [73] are dedicated operators for adaptive stream processing. The distributed Eddy routes tuples between operators—which are considered to reside on a server of their own—to address imbalances between operators. Flux is a modified exchange operator that creates many more partitions than servers to shift partitions from overloaded to underutilized servers. However, it cannot split a large partition that consists only of a single heavy hitter value and thus does not solve the scalability problem caused by skew.

The comparatively low bandwidth of standard network interconnects such as Gigabit Ethernet creates a bottleneck for distributed query processing. Consequently, recent research focused on the network cost: Neo-Join [71] computes an optimal assignment of partitions to servers that minimizes the *network duration*. It handles skew using Selective Broadcast on a partition level. However, it materializes and scans both inputs to generate histograms and has to solve a compute-intensive linear program. Track-Join [65] redistributes join keys in a dedicated track phase to decide on the join location for each join key value separately and by this achieves minimal *network traffic*—excluding the track phase. However, the track phase itself is a separate distributed join that is sensitive to skew and materializes both inputs.

Modern interconnects increase network bandwidth significantly, which is necessary for a cluster to outperform a single server and scale the query performance when servers are added to the cluster [7, 70]. Frey et al. [29] designed the Cyclo Join using RDMA over 10 Gigabit Ethernet for join processing within a ring topology. Goncalves and Kersten [35] extended MonetDB with a novel distributed query processing scheme based on continuously rotating data over a modern high-speed network with a ring topology. However, ring topologies, by design, use only a fraction of the available network bandwidth in a fully-connected network. Mühleisen et al. [60] use RDMA to utilize remote memory for temporary database files in MonetDB. Costea and Ionescu [17] extended Vectorwise, which originated from the MonetDB/X100 project [88], to a distributed system using MPI over InfiniBand. Barthels et al. [5] implemented a distributed radix join using RDMA over InfiniBand, providing a detailed analysis that includes experiments with skewed workloads. They measured a 3.3× slow down for Zipf factor 1.2 on an 8-server cluster, which is in line with our results and shows the need for low-overhead skew handling.

Flow-Join detects heavy hitters using the SpaceSaving algorithm [56] that solves the approximate frequent items problem and was shown to outperform alternatives [16]. Roy et al. [72] designed a pre-filtering stage that speeds up Space-Saving by a factor of 10 for skewed inputs. Teubner et al. [79] employed FPGAs to solve the frequent items problem in hardware, processing 110 million items per second. Both pre-filter and hardware acceleration can be applied to Flow-Join to reduce the overhead of skew detection even further.

Elseidy et al. [23] propose a mechanism to dynamically change the grid of the Symmetric Fragment Replicate redistribution scheme [74] according to running cardinality estimates. However, an additional random redistribution of the input tuples doubles the network traffic and the symmetric join algorithm [83] materializes both inputs in memory and also processes every tuple twice (once for build and once for probe).

4.7 Concluding Remarks

The scalability of distributed joins is threatened by unexpected data characteristics: Skew can cause a severe load imbalance so that one server in the cluster has to process a much larger part of the input than its fair share—slowing down the entire query. Previous approaches often require expensive analysis phases that slow down join processing for non-skewed workloads—especially when using high-speed interconnects such as InfiniBand that are too fast to hide extra computation. Other approaches depend on detailed statistics that are often not available or overly inaccurate for intermediate results.

Flow-Join is a novel join algorithm that detects and adapts to heavy hitter skew alongside partitioning with minimal overhead and without relying on existing statistics. It uses approximate histograms to detect skew and adapts the redistribution scheme at runtime for the subset of the keys that were identified as heavy hitters. The combination of decoupled exchange operators connected via an RDMA-based communication multiplexer for distributed processing and a work-stealing-based approach for local processing enables Flow-Join to scale near-linearly with number of servers in the cluster. Our evaluation with Zipf-generated data sets as well as a real workload from a large commercial vendor has further shown that Flow-Join performs as good as an optimal omniscient approach for skewed workloads and at the same time does not compromise join performance when skew is absent from the workload. The overhead of Flow-Join's adaptive skew handling mechanism is indeed small enough to process skewed and non-skewed inputs at the full network speed of InfiniBand 4×FDR at more than 6 GB/s per link, joining a skewed workload at 11.5 billion tuples/s with 32 servers—6.8× faster than a standard hash join. Flow-Join overlaps computation completely with the network communication, so that the join finishes shortly after the last network transfer. Detecting and handling skew at runtime enables the pipelined execution of joins and thus avoids the materialization of the probe side, reducing main-memory consumption significantly.

Finally, we generalized Flow-Join beyond key/foreign-key equi joins. The generalized Flow-Join applies the Symmetric Fragment Replicate redistribution scheme for heavy hitters that occur in both inputs, reducing the query execution time by up to a factor of $(\sqrt{n}+1)/2$ for correlated skew in a cluster with n servers.

Chapter 5

Transactional Data

Parts of this chapter were previously published in [58] and [59].

Harizopoulos et al. [39] investigated transaction processing in traditional disk-based database systems. In particular, they classified and counted the instructions that the Shore database system [11] issued during transaction processing. They found that buffer management, concurrency control, and logging are responsible for a large fraction of the instructions—even when the disk-based database system operated fully in memory. These "management" instructions by far outweighed those that actually performed transaction logic. The Shore database system was found to spend 35 % of its instructions on buffer management, 31 % on latching and locking, and 12 % on logging [39]. In fact, only 7 % of the instructions were categorized as useful [39]. Modern in-memory database systems completely rewrite the traditional database system architecture to remove now unnecessary abstractions and by this achieve a much higher transaction throughput.

An example is the HyPer [45] database system that processes more than 100,000 TPC-C transactions per second in a single thread. A single core already suffices for transaction processing of most human-generated workloads. The better part of the server's compute resources are therefore available for on-line analytical processing (OLAP). HyPer is a hybrid database system that uses hardware-assisted virtual memory snapshots to isolate OLAP and OLTP processing. This allows analytical queries to operate on the latest transactional state without interfering with the mission-critical transaction processing—enabling real-time business analytics.

While HyPer's single-server performance is sufficient for most transactional workloads, a scale out is still necessary to meet the growing need for analytical query processing. The challenge is therefore to sustain HyPer's superior OLTP throughput while elastically scaling the main-memory capacity and OLAP performance with the cluster size.

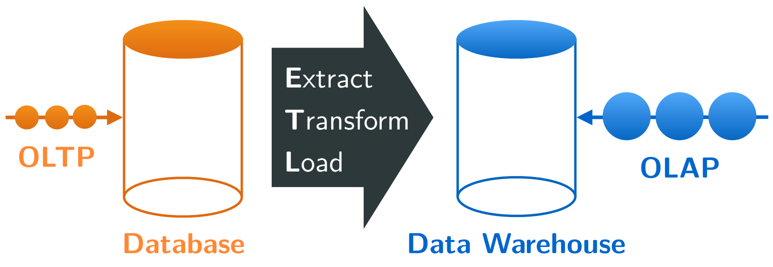


Figure 5.1: Traditional data warehouse architecture[1]

5.1 Motivation

Database systems face two distinctly different types of workloads: On-line analytical processing (OLAP) and on-line transaction processing (OLTP). The traditional approach employs two separate specialized systems, each processing one of the workloads in isolation. A production database system is employed to execute the mission-critical transactions, while a dedicated data warehouse is used for analytical queries. The data warehouse is then periodically (e.g., once every night) updated to the current state of the transactional database system. This separation prevents the long-running OLAP queries from stalling the business-critical, short-running OLTP transactions or otherwise interfering with their performance. However, the data warehouse inevitably suffers from data staleness due to the necessary extract/transform/load (ETL) process depicted in Figure 5.1 that only periodically refreshes the data. SAP's Hasso Plattner [64] argues that this does not meet the requirements of today's businesses and instead calls for real-time business analytics that enables the analysis of *fresh transactional data*.

New hybrid main-memory databases such as SAP's HANA [25] or HyPer [45] address this issue on a single machine. OLTP and OLAP are separated using a delta-mechanism in the case of HANA respectively very efficient hardware-supported virtual memory snapshots in HyPer. HyPer achieves best-of-breed OLTP throughput and OLAP query response times in one system in parallel on the same database state. While the single-core on-line transaction processing (OLTP) performance of a single server is sufficient for almost all settings, a scale-out is still necessary to meet the growing need for analytical query processing.

This chapter thus presents ScyPer, a distributed architecture for the hybrid in-memory database system HyPer based on *full replication*. ScyPer scales HyPer to a cluster of machines to achieve elastic analytical query processing on fresh transactional data. Our approach scales HyPer's analytical query throughput linearly with the number of servers in the cluster and still sustains its excellent single-node transaction throughput of more than 100 000 TPC-C transactions/s. We employ

[1]Adaptation of Figure 1 in [46].

efficient redo log multicasting to keep secondary servers up-to-date with the most recent transactional state of the primary HyPer server. ScyPer further uses global transaction-consistent snapshots to guarantee the correctness of the query results. Secondaries act as fast fail-over servers for the primary to provide high availability in the case that it does not respond due to hardware malfunction or other failures.

While ScyPer with full replication allows to add servers on demand to dynamically increase the query throughput, it is limited by the main-memory capacity of the smallest server in the cluster. Data partitioning is required to process data sets that exceed the capacity of a single server. However, fragmenting relations across servers introduces the need for distributed transactions. Distributed transactions are expensive due to the distributed commit protocol, global locking, as well as deadlock detection that are necessary for consistency and liveness. These overheads would likely reduce HyPer's transaction throughput by several orders of magnitude. We therefore sketch a more refined ScyPer architecture based on *fragmented relations* that replicates the transactional working set across servers. This avoids the huge performance penalty of distributed transactions and still partitions most data across servers to take advantage of the combined main-memory capacity of the cluster. Apart from sustaining HyPer's high single-node transaction throughput, the refined ScyPer architecture also scales the main-memory capacity and analytical query performance with the number of servers in the cluster.

In particular, this chapter makes the following contributions:

- We show how the primary server can efficiently propagate the redo log to keep the transactional state of secondary servers up-to-date. We use a reliable multicasting protocol so that servers can be added to the parallel database system without increasing the network traffic for transferring the redo log. We compare logical and physical logging and show the feasibility of our approach for fast InfiniBand interconnects as well as Gigabit Ethernet.

- We describe how to create global transaction-consistent snapshots in the parallel database system. These snapshots are necessary for certain consistency guarantees in the full replication approach and crucial for the correctness of distributed query processing in the fragmented relations approach.

- We discuss how secondary servers can be used as high-availability fail-overs for the primary server in the full replication approach. We introduce a replication factor x in the fragmented relations approach to allow $x - 1$ server failures before the parallel database system becomes unavailable.

- We use the TPC-CH [14] benchmark that combines the transactional TPC-C and analytical TPC-H workloads to evaluate the performance of a parallel hybrid database system prototype based on HyPer.

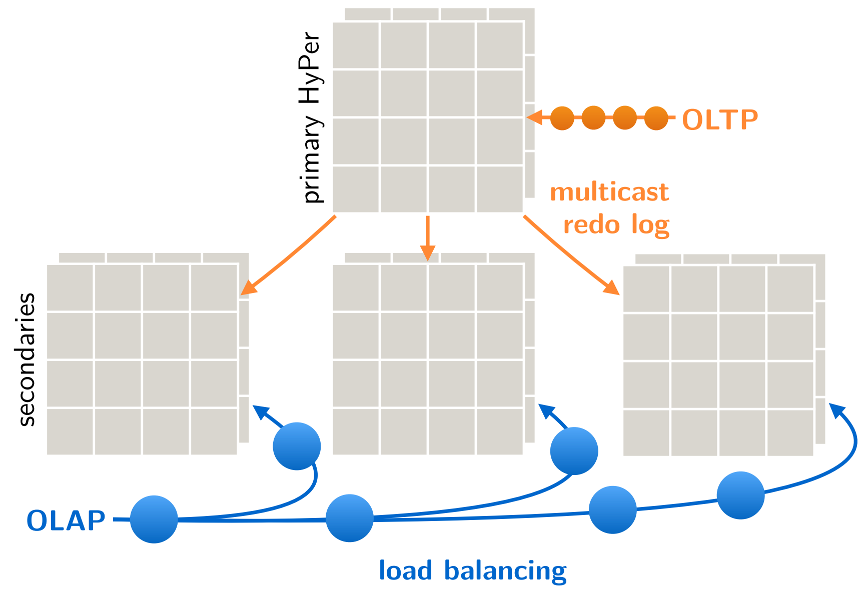


Figure 5.2: Full replication for elastic OLAP throughput

5.2 Full Replication

This chapter is based on joint work with Tobias Mühlbauer that was previously published in [58] and [59]. Sections 5.2.3, 5.2.4, and 5.2.5 are mainly due to Tobias Mühlbauer, while the author of this thesis is primarily responsible for Sections 5.2.1 and 5.2.2. Section 5.3 gives an outlook on potential future work and has not been published before.

In the following, we describe an architecture for a hybrid parallel database system that sustains the excellent OLTP performance of a single server while scaling the OLAP throughput linearly with the cluster size. It differentiates between two types of servers as shown in Figure 5.2: The primary server processes all incoming transactions and multicasts its redo log to an arbitrary number of secondary servers. Redo log propagation uses multicasting so that a log entry needs only be sent once over the network instead of once per secondary server and can still be received by every interested subscriber. This keeps the network traffic incurred by redo log propagation independent of the number of secondary servers, improving the scalability of our approach. The secondary servers replay the redo log and use their additional resources to execute OLAP queries according to a round-robin load balancing scheme. This allows us to easily scale the OLAP throughput of the parallel database system by provisioning secondary servers on-demand as shown in Figure 5.3. When a secondary instance is started, it first fetches the latest full database backup from durable storage and then replays the redo log

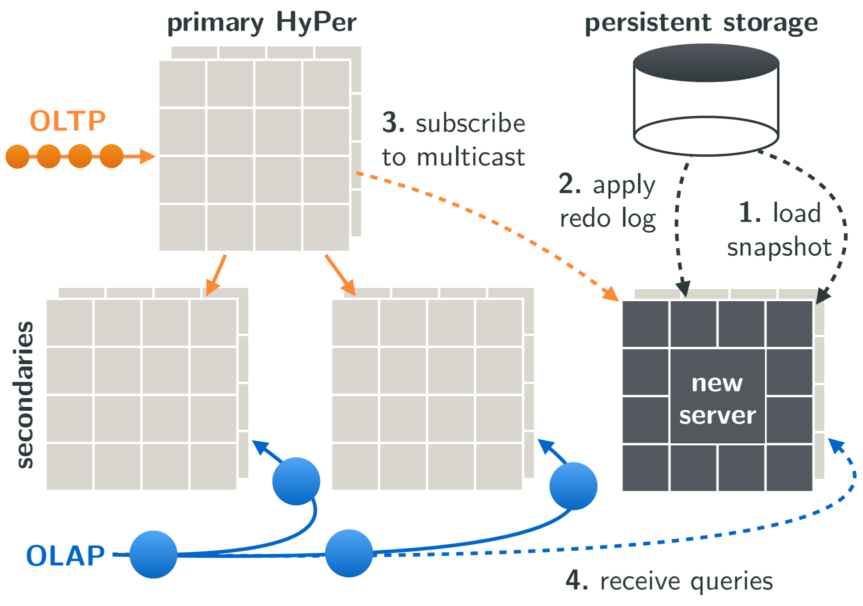


Figure 5.3: Elastically adding new servers to the cluster

until it catches up with the primary. Secondaries can always catch up as redo log replaying is about 2× faster than processing the original OLTP workload (see Section 5.2.1). Further, it is unlikely that the system experiences full load at all times. Afterwards, the new secondary subscribes to the redo log multicast and starts processing queries.

The primary server usually uses a row-store data layout, which is better suited for OLTP processing, and it maintains indexes that support efficient transaction processing. When processing the OLTP workload, the primary node multicasts the redo log of committed transactions to a specific multicast address. The address encodes the database partition such that secondaries can subscribe to specific partitions. This allows the provisioning of secondaries for specific partitions and enables a more flexible multi-tenancy model as described in Section 5.2.5. Besides being multicast, the log is further sent to a durable log. Each redo log entry for a transaction comes with a log sequence number (LSN). The parallel database system uses these LSNs to define a logical time in the distributed setting. A secondary that last replayed the entry with LSN x has logical time x. It next replays the entry with LSN $x+1$ and advances its logical time to $x+1$.

As a large portion of a usual OLTP workload is read-only (i.e., no redo is necessary), replaying the redo log on secondary nodes is usually cheaper than processing the original workload on the primary node. Further, read operations of writer transactions do not need to be evaluated when physical logging is employed. The available resources on the secondaries are used to process incoming OLAP queries

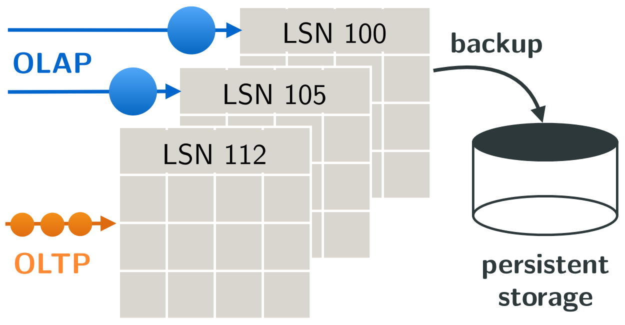


Figure 5.4: Long-running snapshots for OLAP and backups

on transaction-consistent snapshots. HyPer's efficient snapshotting mechanism allows to process several OLAP queries in parallel on multiple snapshots as shown in Figure 5.4. A snapshot can also be written to persistent storage so that it can be used as a transaction-consistent starting point for recovery. Furthermore, the faster OLTP processing allows to create additional indexes for efficient analytical query processing. Secondary nodes can either store data in a row-, column-, or a hybrid row- and column-store data format. Additionally, these nodes can include non-transactional data in OLAP analyses, which need not necessarily be kept in the main memory but could instead reside on disk or a distributed file system such as the Hadoop Distributed File System (HDFS).

In the following we describe our redo log propagation and distributed snapshotting approaches. We further show how our architecture provides scalable OLAP throughput while sustaining the OLTP throughput of a single server and how secondary nodes can act as high availability fail-overs.

5.2.1 Redo Log Propagation

When processing a transaction, HyPer creates a memory-resident undo log which is used to roll back aborted transactions. Additionally, a redo log is created. For committed transactions, this redo log has to be persisted and written to durable storage so that it can be replayed. The undo log however can be discarded when a transaction commits.

We use multicasting to propagate the redo log of committed transaction from the primary to secondary nodes to keep them up-to-date with the most recent transactional state. Multicasting allows to add secondaries on-demand without increasing the network bandwidth usage. A log entry is sent over the network only once and can still be received by any number of interested subscriber.

UDP vs. PGM multicast. Standard UDP multicasting is not a feasible solution for redo log multicasting as it may drop messages, deliver them multiple

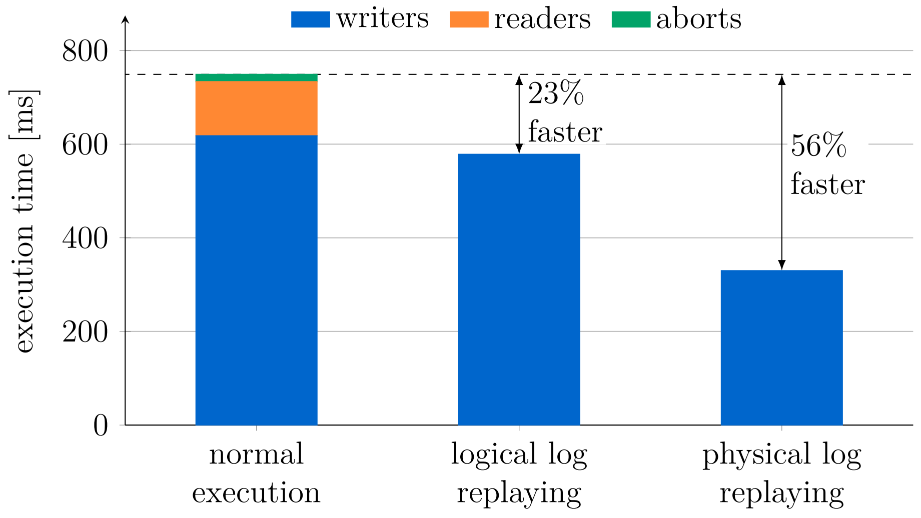


Figure 5.5: Logical vs. physical redo log replaying for TPC-C

times, or transfer them out of order. Instead, we use OpenPGM for multicasting, an open source implementation of the Pragmatic General Multicast (PGM) protocol[2], which is designed for reliable and ordered transmission of messages from a single source to multiple receivers. Receivers detect message loss and recover by requesting a retransmission from the sender.

Logical vs. physical logging. Our design in principal supports both, the use of logical and physical redo logs for redo log propagation. These two alternatives differ in the size of the resulting log and the time needed to replay it. While in a logical redo log only the transaction identifier and invocation parameters are logged, the physical redo log logs the individual insert, update, and delete statements, which modified the database during the transaction. Physical redo logging results in a larger log but replaying it is often much faster compared to logical logging, especially when the logged transaction executed costly logic or many read operations. In any case, transactions that use operations where the outcome can not be determined solely by the transactional state, e.g., random operations or current time information, have to be logged using physical redo logging. It is of note that logical redo logging is restricted to pre-canned stored procedures. However, stored procedures can be added at any time by a low-overhead system-internal transaction.

As mentioned before, secondaries do not need to replay all transactions. Only committed transactions that modified data are logged. Figure 5.5 shows that replaying the logical log of 100 000 TPC-C transactions saves 17 % in execution

[2]PGM is specified in RFC 3208: `https://tools.ietf.org/html/rfc3208`

	Gigabit Ethernet		**InfiniBand 4×QDR**	
	UDP	PGM	UDP	PGM
Bandwidth [Mbit/s]	906	675	14 060	1832
Throughput [1000/s]	81	43	1252	112
Latency [µs]	100.4		13.5	

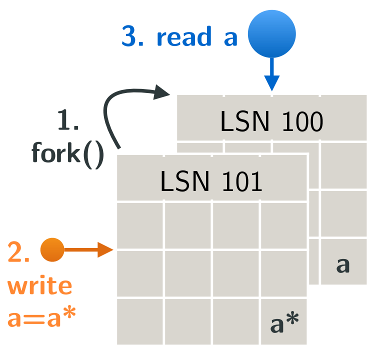
Table 5.1: Comparison of UDP and PGM

time compared to the original processing of the transactions by not having to re-execute reader and aborted transactions and an additional 6 % for not having to log again (undo and redo log)—together this adds up to savings of 23 %. Physical logging is even able to save 56 % of execution time as it further does not re-execute read operations of writers and only replays basic inserts, updates, and deletes.

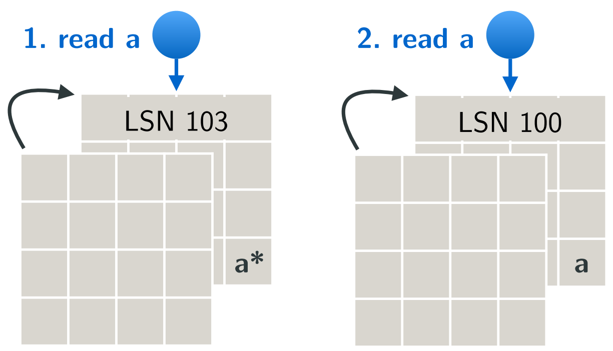
The physical log for 100 000 TPC-C transactions has a size of 85 MiB and is therefore about 5× larger than the logical log, which needs only 14 MiB. An individual physical log entry has an average size of ≈ 1500 B, whereas a logical log entry has ≈ 250 B. Group commits allow to bundle and compress log entries for improved network usage. Compression is not feasible on a per-transaction basis as the individual log entries are simply too small. Compressing the log for 100 000 TPC-C transactions using LZ4 compression reduces the size by 48 % in the case of physical and by 54 % for logical logging.

Ethernet vs. InfiniBand. Table 5.1 compares the single-threaded performance of UDP and PGM multicast in a 1 Gigabit Ethernet (1 GbE) and a 4×QDR IPoIB InfiniBand infrastructure. Our setup consists of four machines each equipped with an on-board Intel 82579V 1 GbE adapter and a Mellanox ConnectX-3 InfiniBand adapter (PCIe 3.0 ×8). We used a standard 1 GbE switch and a Mellanox 8-port 40 Gbit/s QSFP switch. UDP was measured with 1.5 KiB datagrams; PGM messages had a size of 2 KiB. The UDP bandwidth and throughput increases by a factor of 15 from 1 GbE to InfiniBand; PGM still profits by a factor of 2.7. The latency is, in both cases, reduced by a factor of 7.

With a processing speed of around 110 000 TPC-C transactions per second, HyPer creates ≈ 60 000 redo log entries per second per OLTP thread. 1 GbE allows the multicasting of the 60 000 logical log entries but offers not enough performance for physical logging due to its low PGM multicast performance. Only when group commits with log compression are used, physical redo log entries can be multicast over 1 GbE. Our InfiniBand setup can handle physical redo logging without compression and even has free capacities to support multiple outgoing



(a) Local order violation



(b) Diverging distributed reads

Figure 5.6: The need for global transaction-consistent snapshots

multicast streams. These could be used for the simultaneous propagation of the redo logs of all transaction-processing threads in a partitioned execution setting.

5.2.2 Distributed Snapshots

We adapt HyPer's efficient virtual memory snapshotting mechanism [45] to the distributed setting. In the following, we describe how we designed the global transaction-consistent snapshotting mechanism to solve two potential problems that affect query processing on transactional data: *local order violations* and *diverging distributed reads*.

Local order violations. Figure 5.6a shows a schedule which exhibits a local order violation: First, the snapshot is created. Then a transaction modifies a data item that is afterwards read by an OLAP query. In this example, the query reads the data item's old value `a` because the snapshot was created before the transaction changed it to `a*`. A single client who issued both, the transaction and the query, would get an unexpected result—even though the schedule satisfies serializability. Order-preserving serializability (OPS) avoids such order violations as it "requires that transactions that do not overlap in time appear in the same order in a conflict-equivalent schedule" [82]. In the example the transaction finished before the query, i.e., both did not overlap, therefore OPS requires that the query reads the new state `a*` of data item `a`.

To achieve OPS, a query has to be executed on a snapshot that is created after its arrival. While one might argue that if OPS is desired, the query has to be executed as a transaction, we propose a solution that does not require this. A simple solution is to create a snapshot for every single query. However, while snapshot creation is cheap, it does not come for free. Therefore, we associate queries with a logical arrival time and delay their execution until a snapshot with a greater logical creation time is available. The primary node then acts as a load balancer for OLAP queries and tags every incoming query with a new LSN as its logical arrival time. The primary also triggers the periodic creation of global transaction-consistent snapshots (e.g., every second). Together, this guarantees order-preserving serializability as transactions are executed sequentially and queries always execute on a state of the data set that is newer than their arrival time.

Diverging distributed reads. The system as described until now is further subject to a problem that we call diverging reads: Executing the same OLAP query twice can lead to two different results in which the second result is based on an older transactional state—i.e., a database state with a smaller LSN than that of the first query. Figure 5.6b shows an example for this. The diverging reads problem is caused by the load balancing mechanism, which may assign a successive query to a different node whose snapshot represents the state of the data set for an earlier point in time. This problem is not covered by order-preserving serializability but is solved by the synchronized creation of snapshots.

To create such a global snapshot, the primary node sends a system-internal transaction to the secondary nodes, which then create local snapshots using the `fork()` system call at the logical time point defined by the transaction's LSN. We use a logical time based on LSNs to avoid problems with clock skew across nodes. The creation of the global transaction-consistent snapshot is fully asynchronous on the primary node, which avoids any interference with transaction processing. Therefore, the time needed to create a global transaction-consistent snapshot only affects the OLAP response time on the secondaries. The time to create a global transaction-consistent snapshot on n secondary nodes is defined by

$$\max_{0 \leq i < n} (\mathrm{RTT}_i + T_{\mathrm{replay}_i} + T_{\mathrm{fork}_i})$$

where RTT_i is the round trip time from the primary to secondary i, T_{replay_i} is the time required to replay the outstanding log at i, and T_{fork_i} is the time to fork at i. In our high-speed InfiniBand setup, RTTs are as low as a few µs. To avoid inconsistencies, the snapshot transaction has to be processed in order, i.e., the outstanding log at the secondary has to be processed first. However, it is expected that at most one transaction has to be replayed before the snapshot can be created—as the secondaries process transactions faster than they arrive. On average a transaction takes only 10 µs. The time of the `fork` depends on the memory page size and the database size but is in general very fast, e.g., with a database size of 8 GiB a `fork` takes 1.5 ms with huge and 50 ms with small pages. All in all, the time needed to create a global transaction-consistent snapshot adds up to only a few milliseconds, which has no significant impact on the OLAP response time.

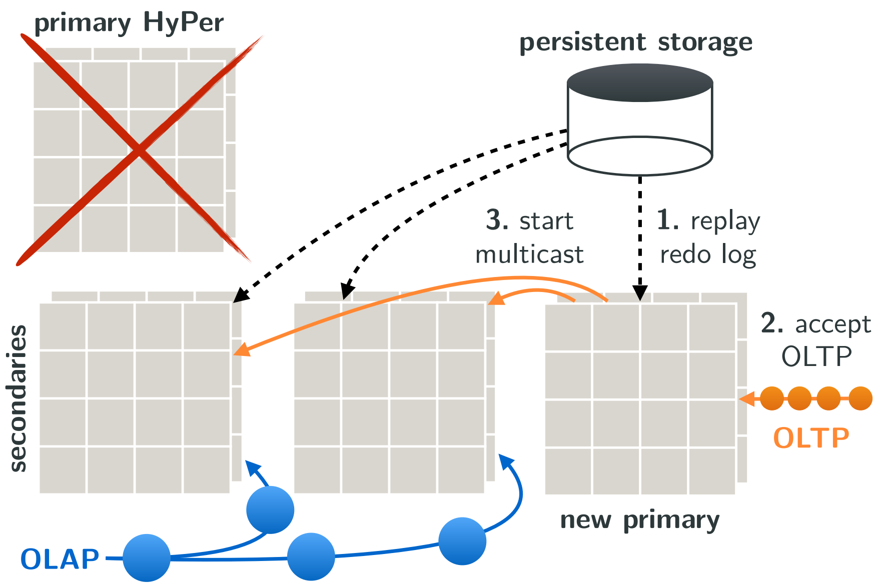
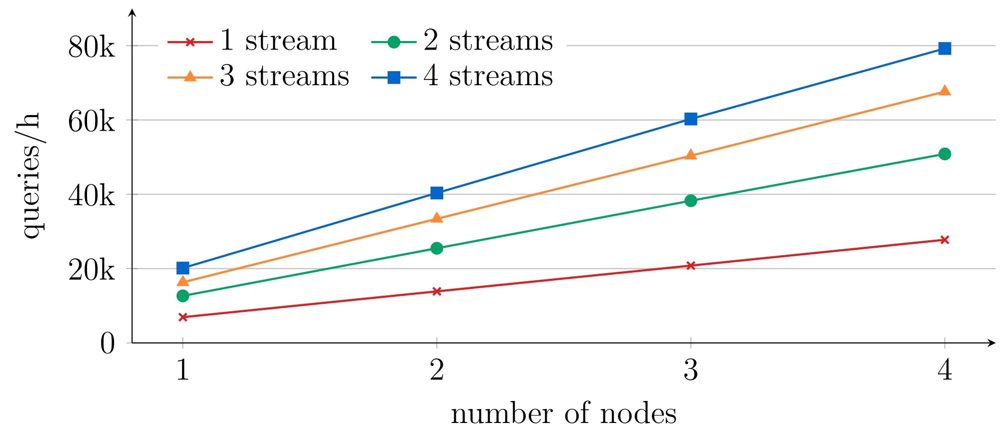


Figure 5.7: Secondaries act as high-availability fail-overs for the primary

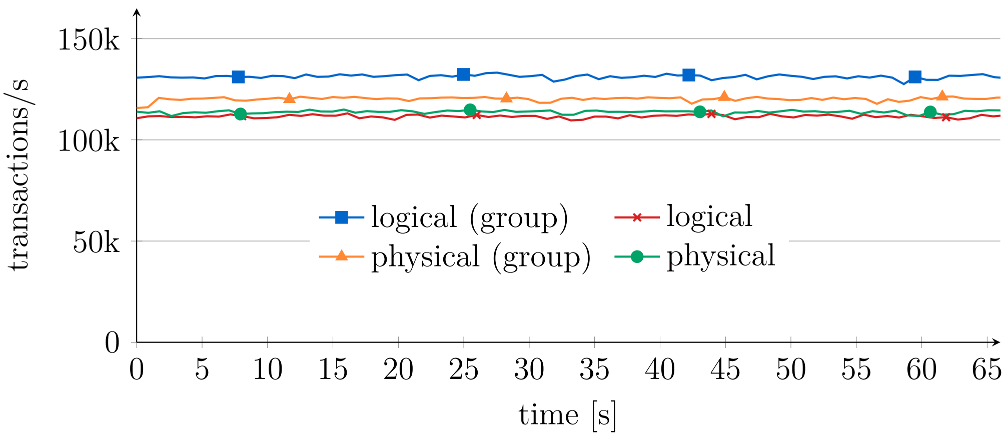
Distributed processing. As global transaction-consistent snapshots avoid inconsistencies between local snapshots, they also enable the distributed processing of a single query on multiple secondaries. The distributed processing has the potential to further reduce query response times.

5.2.3 High Availability

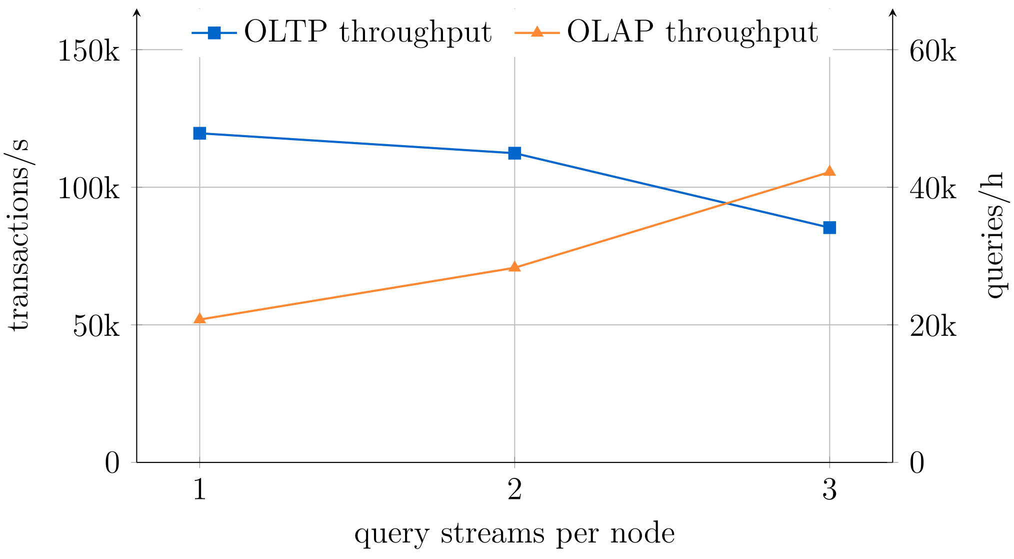
Besides providing scalable OLAP throughput on transactional data, secondary HyPer nodes can further be used as high availability fail-over nodes. This is illustrated in Figure 5.7. Secondaries detect the failure of the primary when no redo log message—or heartbeat message—is received from the primary within a given timeframe. In case of failure, the secondaries then elect a new primary using a distributed consensus protocol. The new primary and the remaining secondaries replay all transactions in the durable redo log for which they have not yet received the multicast log. Once the primary replayed these transactions, it is active and can process new transactions. It is thus recommendable to choose the new primary depending on the number of transactions it has to replay, i.e., to choose the secondary with smallest difference between its LSN and the LSN of the durable redo log. Further, if a secondary using a row-store layout exists, this node should be preferred over nodes using a column-store layout. However, for our main-memory database system HyPer, TPC-C transaction processing performance only decreases by about 10 % using a column- compared to a row-store layout. In conclusion, the design can resolve a failure of its primary node within a very short period of time.



(a) OLAP throughput on different numbers of nodes (without transaction load)



(b) Transaction throughput on the primary with redo log propagation to secondaries



(c) Combined OLTP and OLAP processing with different numbers of OLAP query streams

Figure 5.8: Isolated and combined OLAP and OLTP throughput

5.2.4 Evaluation

The evaluation was conducted on four commodity workstations, each equipped with an Intel Core i7-3770 CPU and 32 GiB dual-channel DDR3-1600 DRAM. The CPU is based on the Ivy Bridge microarchitecture, has 4 cores, 8 hardware threads, a 3.4 GHz clock rate, and 8 MiB of last-level shared L3 cache. As operating system we used Linux 3.5 in 64 bit mode. Sources were compiled using GCC 4.7 with `-O3 -march=native` optimizations.

Figure 5.8 shows the isolated and combined TPC-CH benchmark [14] OLAP and OLTP throughput that can be achieved with our system. We prioritized the OLTP process so that replaying the log is preferred over OLAP query processing to avoid that secondaries cannot keep up with redo log replaying. Figure 5.8a demonstrates that the OLAP throughput scales linearly when no transactions are processed at the same time. Multiple query streams allow the nodes to process queries in parallel using their 4 cores and therefore increase the OLAP throughput. Figure 5.8b shows the transaction throughput that was achieved on the primary node while the redo log is simultaneously broadcasted to and replayed by the secondaries. The figure shows the transaction rate for different redo log types and commit variants. While the transaction rates for the four options differ by at most 15 %, group commits clearly provide a better performance than per-transaction log propagation. The reason for this is the reduced PGM processing overhead since group commits lead to fewer and larger messages. Finally, Figure 5.8c shows the combined execution of OLTP and OLAP with logical redo log propagation and uncompressed group commits. All four nodes, including the primary, process OLAP queries. The nodes are able to handle up to two query streams each, while sustaining a OLTP throughput of over 100 000 transaction/s (normal execution on primary, replaying on secondaries). Three streams degrade the OLTP throughput noticeably and with four streams, secondaries can no longer keep up with transaction log replaying. This is reasonable, as the nodes only have 4 cores, of which in this case all are busy processing queries.

5.2.5 Cloud

"In-memory computing will play an ever-increasing role in Cloud Computing" [64]: HyPer with its elastic scale-out is particularly suitable for the deployment on Cloud infrastructures. In an infrastructure-as-a-service scenario, HyPer runs on multiple physical machines in the Cloud. Nodes for secondaries are rented on-demand, which makes this model highly cost-effective. Figure 5.9 shows a flexible database-as-a-service scenario for HyPer. The service provider aims at an optimal resource usage. Following the partitioned execution model of H-Store [44] and VoltDB, HyPer—and thus primary ScyPer instances—provides high single-server OLTP

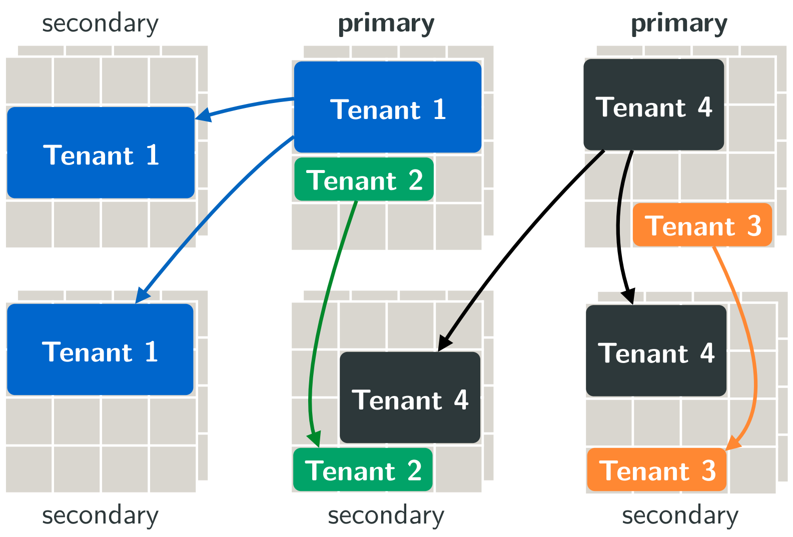


Figure 5.9: Flexible database-as-a-service deployment

throughput on multiple partitions in parallel, which allows running multiple tenants on one physical machine [57]. Redo logs for the partitions are multicast on a per-tenant basis so that OLAP secondaries can be created for specific tenants. OLAP processing of multiple tenants can again be consolidated on a single server. In case a primary node that processes the OLTP of multiple tenants faces an increased load, partitions of a tenant can migrate from one primary to another or a secondary can take over as a primary very quickly (see Section 5.2.3). In summary, HyPer in the Cloud allows great flexibility, very good resource utilization, and high cost-effectiveness. However, let us add a word of caution: many Cloud infrastructure offerings are virtualized. Our previous experiments, which are out of scope for this work, suggest that running applications tuned for modern hardware like HyPer on such instances can lead to severe performance degradations; unvirtualized instances should thus be preferred.

5.2.6 Summary

The ScyPer architecture utilizes redo log propagation and distributed transaction-consistent snapshots to scale the query throughput with the cluster size while sustaining HyPer's excellent single-server transaction throughput at the same time. By fully replicating the database, any server can take over in case the primary fails. ScyPer further allows to add new servers at runtime to scale the query throughput. A new server simply loads the latest full backup from persistent storage, replays the outstanding redo log, and finally registers for query processing. The flexibility of this architecture lends itself naturally to cloud settings.

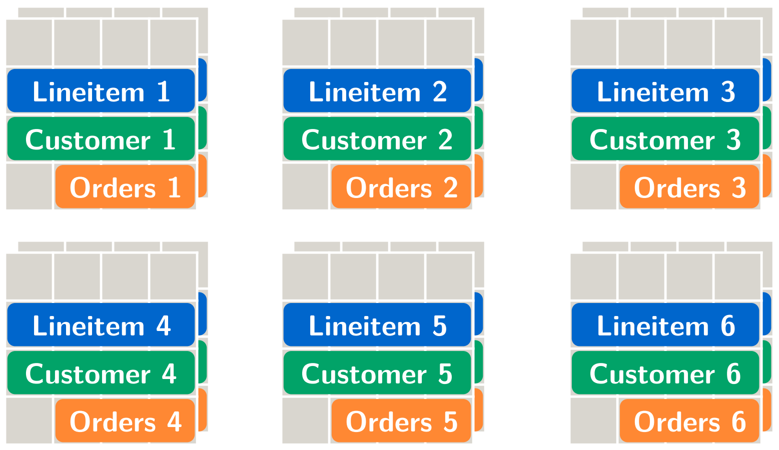


Figure 5.10: Fragmented relations utilize the cluster's combined memory capacity

5.3 Fragmented Relations

The ScyPer architecture utilizes full replication to scale the query throughput linearly with the cluster size and sustain HyPer's excellent transaction throughput. However, it is inherently limited to workloads that fit into the main memory of a single server. For larger workloads, it becomes necessary to fragment relations across servers to use the combined capacity of the cluster as illustrated in Figure 5.10. Tuples are typically assigned to servers according to the hash value of the primary key or a foreign key that is commonly used in distributed joins. This achieves a balanced load distribution across servers and reduces the amount of data shuffled for key/foreign-key joins, speeding up distributed query processing.

In the following, we sketch a design that extends the ScyPer architecture to fragmented relations. Fragmenting relations between servers allows to take advantage of the combined main-memory capacity of the cluster. However, it also introduces the need for both distributed query and transaction processing.

5.3.1 Hot/Cold Separation

We have addressed distributed query processing in Chapter 3 with a novel distributed query engine that can fully leverage the high bandwidth of modern interconnects and thereby scale query performance with the cluster size. This query engine can be used *as is* to process distributed queries on the global transaction-consistent snapshots that were described in Section 5.2.2. However, sustaining the high transaction throughput of a single server when transactions span multiple machines is an entirely different problem and might even be impossible to achieve with current hardware. Distributed transactions suffer from the high latency of the

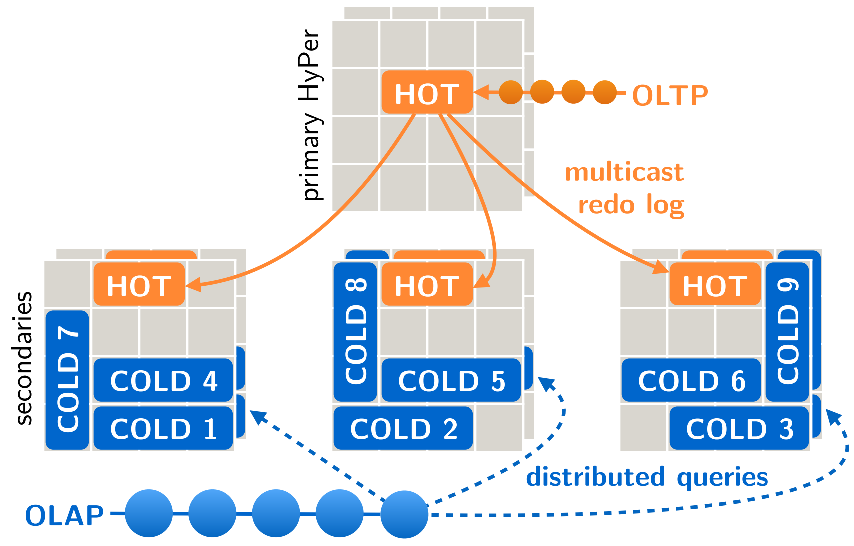


Figure 5.11: Hot/cold approach avoids distributed transactions

network hardware (compared to local memory) as they require multiple round-trips for distributed commit handling, global locking and latching, as well as distributed deadlock detection. These are necessary to guarantee correctness and the liveness of the system. As a consequence, distributed transaction performance traditionally lags far behind what is possible on a single server.

H-Store [44] avoids expensive distributed transactions by tuning the schema so that partitioned execution of transactions becomes the common case. This works well for the TPC-C workload where most transactions address a single warehouse and thus only a single server. However, this is not possible for all workloads and further requires prior knowledge of the workload to tune the schema accordingly.

Our proposed architecture (shown in Figure 5.11) is designed to avoid distributed transactions as well but does not restrain itself to partitioned execution. At the same time, it still fragments data to take advantage of the combined main-memory capacity of all servers in the cluster. Our design relies on the working set hypothesis to avoid distributed transactions. The hypothesis states that transactions often target only a limited working set while most of the data is read-only [30]. We thus separate the database into two parts: The first part comprises the small amount of hot data that is accessed by transactions and can therefore be replicated across servers via redo log propagation (see Section 5.2.1) to avoid distributed transactions. The remaining and much larger part of the data is cold and fragmented between servers as it is only accessed by analytical queries. We use the distributed query engine of Chapter 3 to process distributed queries on the global transaction-consistent snapshots introduced in Secion 5.2.2.

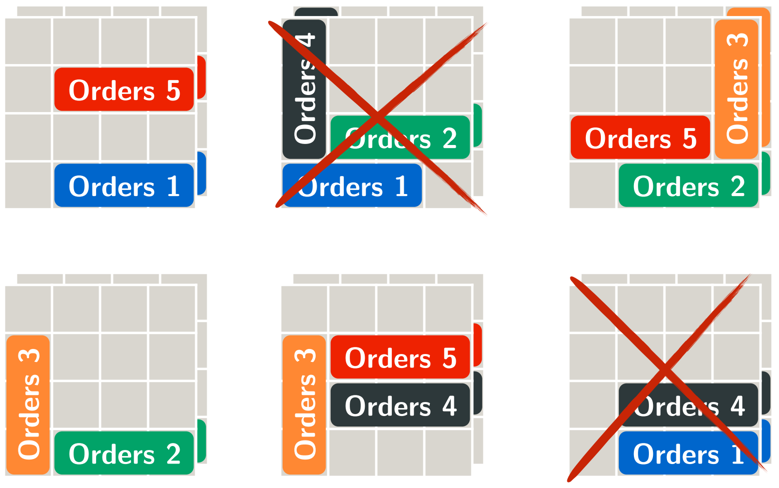


Figure 5.12: High availability via HDFS-style replication

5.3.2 High Availability

In the original ScyPer architecture any secondary server could take over for the primary to process transactions while query processing continues. The failure of a single server did not impact the availability of data items as every server had a complete copy of the database available. This is not the case when relations are fragmented across servers. Thus, we introduce a replication factor x similar to the Hadoop distributed file system (HDFS). The cold data is organized in blocks. Blocks are replicated x-times between secondaries to allow $x - 1$ servers to fail before the parallel database system becomes unavailable. An example with replication factor 3 is shown in Figure 5.12. In this scenario, two servers can fail before the parallel database system becomes unavailable.

5.3.3 Elasticity

Adding new servers to and removing servers from the cluster was straightforward for the original ScyPer architecture. The new server simply loads the most recent full-database snapshot from persistent storage and applies the outstanding redo log. Afterwards, it can start processing queries. When a server leaves the cluster there is no need for action. When relations are fragmented across servers, we need to reassign data to the new server—respectively assign its data to the other servers in case it is leaving the cluster. However, we want to avoid a complete reshuffle, which is prohibitively expensive. At the same time, we aim at a balanced assignment of data blocks that still allows to efficiently compute the server that is responsible for a specific key without scanning all servers.

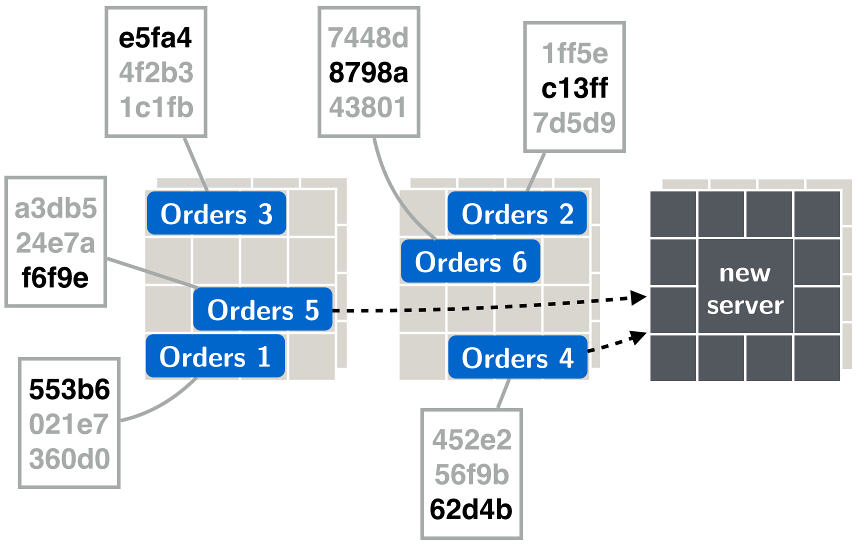


Figure 5.13: Elasticity via Highest Random Weight Hashing

Mukherjee et al. [61] proposed Highest Random Weight (HRW) hashing (also known as Rendezvous Hashing) to provide elasticity when relations are fragmented across nodes. Figure 5.13 illustrates how HRW hashing can be used for this purpose. HRW hashing assigns one hash value per server to every data block. A data block is then assigned to the server with the highest hash value for this block. E.g., consider the data block "Orders 5", it has the hash values *a3db5*, *24e7a*, and *f6f9e* for servers 1, 2, and 3, respectively. As long as there are only two servers, this block is assigned to server 1 as *a3db5* is larger than *24e7a*. However, when the third server is added to the cluster, this data block has to be moved to the new server as its hash value *f6f9e* for this block is larger than *a3db5*. Using HRW hashing achieves the minimal amount of data movement when servers are added or removed. At the same time it also leads to fair load balancing so that servers store roughly the same amount of data.

5.4 Concluding Remarks

In this chapter we have shown that a distributed version of the HyPer in-memory database system is indeed able to sustain the superior OLTP throughput of a single HyPer server while providing elastic OLAP throughput by provisioning additional servers on-demand. OLAP queries are executed on global transaction-consistent snapshots of the transactional state to guarantee correct results. We have demonstrated that our novel global transaction-consistent snapshotting mechanism guarantees order-preserving serializability and further prevents the problem of diverging reads in a distributed setting. Secondary nodes are efficiently kept up-to-date using a redo log propagation mechanism based on reliable multicasting. In case of a primary node failure, these secondaries act as high availability fail-overs.

We have further sketched an extension of the ScyPer architecture that fragments relations across nodes to leverage the combined main-memory capacity of the cluster while sustaining the high transaction throughput. In particular, we have described techniques that avoid expensive distributed transactions, achieve elasticity, and provide high availability in this scenario.

Chapter 6

Summary

Ongoing advances in the computing power and main-memory capacity of modern servers have fueled the development of a new generation of in-memory database systems. These achieve unprecedented single-node query processing performance by completely rewriting the traditional database architecture to use main memory as primary storage and exploit new features of modern hardware. At the same time, network communication slows down queries when more than one server is involved. The result is a significant performance gap between local and distributed query processing. Still, the main-memory capacity of a single server is limited and scaling out to a cluster of machines becomes inevitable once the workload exceeds the capacity of a single server.

This thesis seeks to further the state-of-the-art of distributed query processing in parallel in-memory database systems by addressing the performance barrier introduced by network communication. Thus, instead of concentrating on an isolated problem, we presented the design of a novel distributed query engine that adapts not only to the available network bandwidth but also to unexpected workload characteristics that hinder the scalability of distributed query processing. We proved the feasibility of our design with a prototypical implementation of our distributed query engine for the high-performance in-memory database system HyPer.

The limited network bandwidth of commodity interconnects in today's data centers has a detrimental effect on the throughput and response time of analytic queries as soon as data is shuffled between servers. In fact, a single server easily outperforms a cluster of machines connected via Gigabit Ethernet. However, a single server can only process data sets that fit into its limited main memory. Our novel distributed query engine allows to scale the capacity of a parallel database system with the size of the data set while avoiding expensive all-to-all data shuffling as much as possible. It adapts to slow commodity networks by exploiting locality in the data distribution via linear programming and avoids cross traffic by scheduling the network communication.

Recently, high-speed interconnects such as InfiniBand have become economically viable. InfiniBand offers a one to two orders of magnitude higher network bandwidth than Gigabit Ethernet. At the same time, however, it also reveals new bottlenecks in traditional distributed query engines that use the classic exchange operator model and TCP/IP for communication. Our new query engine instead implements a novel hybrid approach to parallelism that overcomes the inflexibility of the exchange operator model and uses remote direct memory access to avoid the high cost of TCP/IP stack processing. It uses low-latency network scheduling to avoid switch contention. In combination, this allows our engine to scale HyPer's excellent query performance with the number of servers in the cluster—something that was not possible before.

At this point skew hinders the scalability of distributed query processing. Heavy hitter skew can cause a load imbalance in the cluster so that one server has to process a much larger part of the input than its fair share. Consequently, all the other servers have to wait for this one straggler, drastically increasing the query response time. Traditional skew handling schemes add a significant overhead for non-skewed workloads, especially when high-speed networks such as InfiniBand are used that are too fast to hide any extra computational work. We therefore designed a novel lightweight skew handling approach that allows our distributed query engine to detect heavy hitter values at runtime with minimal overhead. The new always-on skew detection approach avoids the load imbalance caused by heavy hitter skew and at the same time incurs only minimal additional work for non-skewed workloads.

HyPer enables real-time business analytics by processing analytical queries and business-critical transactions at the same time on the same data. This distinguishes it from most other systems that are instead specialized for just one of the two workloads. Our goal was to bring HyPer's hybrid processing capabilities over to the distributed setting in a way that neither compromises query nor transaction performance. We presented a novel approach that propagates the current transactional state via low-overhead redo log multicasting and separates query from transaction processing using global transaction-consistent snapshots. It replicates relations to scale the query throughput linearly with the cluster size. While full replication offers excellent fault-tolerance, it inherently limits the usable main-memory capacity of the parallel database system to that of a single server in the cluster. We therefore gave an outlook on how to fragment relations instead to utilize the combined main-memory capacity of the cluster. The proposed hybrid architecture is based on a novel hot/cold approach that replicates the transactional working set to avoid the high cost of distributed transactions. The cold data is fragmented across secondaries and available for query processing using our scalable distributed query engine on global transaction-consistent snapshots.

Our new distributed query processing engine for the HyPer in-memory database system adapts to commodity and high-speed networks as well as unexpected workload characteristics to close the performance gap between local and distributed query processing. An extensive evaluation with the renowned TPC-H ad-hoc analytical benchmark demonstrated that HyPer with our novel distributed query engine not only outperforms competing parallel database systems but also scales its query performance with the number of servers in the cluster.

Bibliography

[1] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *TKDE*, 23(9):1282–1298, 2011.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, pages 281–296, 2010.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[4] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, and A. N. Wilschut. PRISMA/DB: A parallel main memory relational DBMS. *TKDE*, 4(6):541–554, 1992.

[5] C. Barthels, S. Loesing, D. Kossmann, and G. Alonso. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, pages 1463–1475, 2015.

[6] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *PVLDB*, 5(11):1627–1637, 2012.

[7] C. Binnig, U. Çetintemel, A. Crotty, A. Galakatos, T. Kraska, E. Zamanian, and S. B. Zdonik. The end of slow networks: It's time for a redesign. *CoRR*, abs/1504.01048, 2015.

[8] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD*, pages 975–986, 2010.

[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[10] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *TKDE*, 2(1):4–24, 1990.

[11] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *SIGMOD*, pages 383–394, 1994.

[12] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. Robust and skew-resistant parallel joins in shared-nothing systems. In *CIKM*, pages 1399–1408, 2014.

[13] D. D. Clark, J. Romkey, and H. C. Salwe. An analysis of TCP processing overhead. In *LCN*, pages 284–291, 1988.

[14] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. A. Kuno, R. O. Nambiar, T. Neumann, M. Poess, K. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload CH-benCHmark. In *DBTest*, 2011.

[15] G. Copelande, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. *SIGMOD Record*, 17(3):99–108, 1988.

[16] G. Cormode and M. Hadjieleftheriou. Methods for finding frequent items in data streams. *VLDB J.*, 19(1):3–20, 2010.

[17] A. Costea and A. Ionescu. Query optimization and execution in Vectorwise MPP. Master's thesis, Vrije Universiteit, Amsterdam, Netherlands, 2012.

[18] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6):85–98, 1992.

[19] D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In *VLDB*, pages 151–164, 1985.

[20] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *TKDE*, 2(1):44–62, 1990.

[21] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, pages 1–8, 1984.

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

[23] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. *PVLDB*, 7(6):441–452, 2014.

[24] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *SIGMOD*, pages 169–180, 1978.

[25] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[26] A. Floratou, U. F. Minhas, and F. Özcan. SQL-on- Hadoop: Full circle back to shared-nothing database architectures. *PVLDB*, 7(12):1295–1306, 2014.

[27] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier. TCP performance re-visited. In *ISPAS*, pages 70–79, 2003.

[28] P. W. Frey. *Zero-copy network communication.* PhD thesis, ETH Zürich, Zurich, Switzerland, 2010.

[29] P. W. Frey, R. Goncalves, M. Kersten, and J. Teubner. Spinning relations: high-speed networks for distributed join processing. In *DaMoN*, pages 27–33, 2009.

[30] F. Funke, A. Kemper, and T. Neumann. Compacting transactional data in hybrid OLTP&OLAP databases. *PVLDB*, 5(11):1424–1435, 2012.

[31] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine GRACE. In *VLDB*, pages 209–219, 1986.

[32] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1990.

[33] G. Gligor and S. Teodoru. Oracle Exalytics: Engineered for speed-of-thought analytics. *Database Systems Journal*, 2(4):3–8, 2011.

[34] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengießer, C. Mathis, T. Bodner, and W. Lehner. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *PVLDB*, 8(12):1716–1727, 2015.

[35] R. Goncalves and M. Kersten. The Data Cyclotron query processing scheme. *TODS*, 36(4), 2011.

[36] T. Gonzalez and S. Sahni. Open shop scheduling to minimize finish time. *Journal of the ACM*, 23(4):665–679, October 1976.

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

[38] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. *SIGMOD Record*, 23(2):243–252, 1994.

[39] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[40] J. L. Hennessy and D. A. Patterson. *Computer Architecture*. Morgan Kaufmann, 5th edition, September 2011.

[41] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, pages 525–535, 1991.

[42] IEEE study group on Next Generation 802.3 BASE-T. Next generation BASE-T call for interest. `http://www.ieee802.org/3/NGBASET`, July 2012.

[43] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. *SIGCOMM*, 44(4):295–306, 2014.

[44] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[45] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[46] A. Kemper and T. Neumann. HyPer: Hybrid OLTP&OLAP high performance database system. Technical Report TUM-I1010, Insititut für Informatik der Technischen Universität München, 2011.

[47] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.

[48] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. CloudRAM-Sort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster. In *SIGMOD*, pages 841–850, 2012.

[49] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the Super Database Computer (SDC). In *VLDB*, pages 210–221, 1990.

[50] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *NGC*, 1(1):63–74, 1983.

[51] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, and D. Hecht. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[52] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[53] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. Lohman. NUMA-aware algorithms: The case of data shuffling. In *CIDR*, 2013.

[54] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *TKDE*, 14(4):709–730, 2002.

[55] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.

[56] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, pages 398–412, 2005.

[57] H. Mühe, A. Kemper, and T. Neumann. The mainframe strikes back: Elastic multi-tenancy using main memory database systems on a many-core server. In *EDBT*, pages 578–581, 2012.

[58] 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 *BTW*, pages 499–502, 2013.

[59] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. ScyPer: Elastic OLAP throughput on transactional data. In *DanaC*, 2013.

[60] H. Mühleisen, R. Gonçalves, and M. Kersten. Peak performance: Remote memory revisited. In *DaMoN*, 2013.

[61] N. Mukherjee, S. Chavan, M. Colgan, D. Das, M. Gleeson, S. Hase, A. Holloway, H. Jin, J. Kamp, K. Kulkarni, T. Lahiri, J. Loaiza, N. MacNaughton, V. Marwah, A. Mullick, A. Witkowski, J. Yan, and M. Zaït. Distributed architecture of Oracle Database In-memory. *PVLDB*, 8(12):1630–1641, 2015.

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

[63] H. Plattner. The impact of in-memory databases on applications. Talk, July 7, 2014.

[64] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer, 2011.

[65] O. Polychroniou, R. Sen, and K. A. Ross. Track join: Distributed joins with minimal network traffic. In *SIGMOD*, pages 1483–1494, 2014.

[66] P. Raghavan and C. D. Tompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, December 1987.

[67] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[68] S. Ray, B. Simion, A. D. Brown, and R. Johnson. Skew-resistant parallel in-memory spatial join. In *SSDBM*, 2014.

[69] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In *ICDE*, 2016.

[70] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4):228–239, 2015.

[71] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*, pages 592–603, 2014.

[72] P. Roy, J. Teubner, and G. Alonso. Efficient frequent item counting in multi-core hardware. In *KDD*, pages 1451–1459, 2012.

[73] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.

[74] J. W. Stamos and H. C. Young. A symmetric fragment and replicate algorithm for distributed joins. *TPDS*, 4(12):1345–1354, 1993.

[75] M. Stonebraker. The case for shared nothing. *DEBU*, 9(1):4–9, 1986.

[76] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[77] J. Teubner, G. Alonso, C. Balkesen, and M. T. Ozsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[78] J. Teubner and R. Mueller. How soccer players would do stream joins. In *SIGMOD*, pages 625–636, 2011.

[79] J. Teubner, R. Müller, and G. Alonso. Frequent item computation on a chip. *TKDE*, 23(8):1169–1181, 2011.

[80] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.

[81] T. Urhan and M. J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *DEBU*, 23(2):27–33, 2000.

[82] G. Weikum and G. Vossen. *Transactional Information Systems.* Morgan Kaufmann, 2002.

[83] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, pages 68–77, 1991.

[84] J. L. Wolf, D. M. Dias, and P. S. Yu. A parallel sort merge join algorithm for managing data skew. *TPDS*, 4(1):70–86, 1993.

[85] J. L. Wolf, P. S. Yu, J. Turek, and D. M. Dias. A parallel hash join algorithm for managing data skew. *TPDS*, 4(12):1355–1371, 1993.

[86] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

[87] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD*, pages 1043–1052, 2008.

[88] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical DBMS. In *ICDE*, pages 1349–1350, 2012.