Rack-Scale In-Memory Join Processing using RDMA

Claude Barthels, Simon Loesing, Gustavo Alonso, Donald Kossmann*

Systems Group
Department of Computer Science
ETH Zurich, Switzerland

{firstname.lastname}@inf.ethz.ch

ABSTRACT

Database systems running on a cluster of machines, i.e. rack-scale databases, are a common architecture for many large databases and data appliances. As the data movement across machines is often a significant bottleneck, these systems typically use a low-latency, high-throughput network such as InfiniBand. To achieve the necessary performance, parallel join algorithms must take advantage of the primitives provided by the network to speed up data transfer.

In this paper we focus on implementing parallel in-memory joins using Remote Direct Memory Access (RDMA), a communication mechanism to transfer data directly into the memory of a remote machine. The results of this paper are, to our knowledge, the first detailed analysis of parallel hash joins using RDMA. To capture their behavior independently of the network characteristics, we develop an analytical model and test our implementation on two different types of networks. The experimental results show that the model is accurate and the resulting distributed join exhibits good performance.

1. INTRODUCTION

The ability to efficiently process complex queries over large datasets is a basic requirement in real-time analytics. Given the increase of data volume, a platform of choice for data processing are rack-scale clusters composed of several multi-core machines connected by a high-throughput, low-latency network such as InfiniBand [17]. The adoption of rack-scale architectures has been further accelerated through the introduction of several data appliances such as Oracle Exadata [26], IBM Netezza [18] and SAP HANA [29], which follow a similar architecture.

In these systems, efficient inter-machine data movement is critical, forcing join algorithms to be aware of machine boundaries and to employ communication patterns suited for the underlying network technology. A natural question to ask is how join algorithms can be optimized to operate in such environments, how well they can scale with the number of machines, and which bottlenecks arise when scaling out in these architectures. However, little work is available on adapting join algorithms to make efficient use of modern high-speed networks.

Several low-latency networks provide Remote Direct Memory Access (RDMA) as a light-weight communication mechanism to transfer data. RDMA is essential for high performance applications because the data is immediately written or read by the network card, thus eliminating the need to copy the data across intermediate buffers inside the operating system (kernel bypass). This, in turn, reduces the overall CPU costs of large data transfers. However, these performance advantages can only be leveraged through thoughtful design of the distributed algorithm, in particular through careful management of the RDMA-enabled buffers used for sending and receiving data and through interleaving computation and network communication [11].

In this paper we analyze the behaviour of a radix hash join using RDMA. Building upon recent work on main-memory multi-core join algorithms [19, 2, 6, 4], we analyze how a hash join needs to be adapted in order to run on a rack-scale database cluster. In the description of the algorithm we place special emphasis on the registration, de-registration and management of RDMA-enabled buffers as these are critical components. To generalize our findings, we develop a theoretical model allowing us to predict the performance of the algorithms based on the system configuration and input data size. Last but not least, we evaluate our prototype implementation on two database clusters: a ten node cluster composed of multi-core machines connected by a Quad Data Rate (QDR) InfiniBand network and a four node cluster connected by a Fourteen Data Rate (FDR) InfiniBand network. The experimental results validate the accuracy of the analytical model and provide insights on the importance of interleaving computation and communication, the role of the network protocol, the effects of skew, and of different relation and tuple sizes.

To the best of our knowledge, this is the first detailed description, analytical model, and experimental evaluation of a state-of-the-art distributed join operator using RDMA.

The paper is structured as follows: Sections 2 and 3 present related work and discuss the necessary background on the radix hash join and InfiniBand networks. Section 4 describes the RDMA-based hash join. Section 5 provides a detailed analytical model. In Section 6 we evaluate the proposed algorithm experimentally. We discuss our findings in Section 7 and present our conclusions in Section 8.

*The author is currently on leave at Microsoft Research

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2750547.

2. RELATED WORK

2.1 Parallel and Distributed Joins

In the Gamma database machine [7, 8] tuples are routed to processing nodes using hash-based split tables. Identical split tables are applied to both input relations, thus sending matching tuples to the same processing node. This method reduces a join of two large relations to a set of separate joins which can be executed in parallel.

Schneider et al. [30] compared hash and sort-merge joins on the Gamma database machine. They conclude that with a sufficient amount of main-memory, hash-based join algorithms have superior performance to sort-merge joins.

Most modern hash join algorithms build upon the idea of the Grace hash join [20], where both input relations are first scanned and partitioned according to the join attribute before a hash table is created for each partition of the inner relation and probed with the tuples from the corresponding partition of the outer relation.

The findings of Shatdal et al. [31] and Manegold et al. [23] showed that a Grace hash join which partitions the data such that the resulting hash tables fit into the processor cache can deliver higher performance because it reduces the number of cache misses while probing the hash tables. To avoid excessive Translation Lookaside Buffer (TLB) misses during the partitioning phase caused by random memory access to a large number of partitions, Manegold et al. [23] proposed a partitioning strategy based on radix-clustering. When the amount of partitions exceeds the number of TLB entries or cache lines, the partitioning is performed in multiple passes.

2.2 Join Algorithms on Modern Hardware

Kim et al. [19] have compared hash and sort-merge joins to determine which type of algorithm is better suited to run on modern multi-core machines. In addition to their experiments, the authors also developed a model in order to predict the performance of the algorithms on future hardware. Although modern hardware currently favours hash join algorithms, they estimated that future hardware with wider single instruction over multiple data (SIMD) instructions would significantly speed up sort-merge joins.

Blanas et al. [6] reexamined several hash join variants, namely the no partitioning join, the shared partitioning join, the independent partitioning join and the radix join. The authors argue that the no partitioning join, which skips the partitioning stage, can still outperform other algorithms because modern machines are very good in hiding latencies caused by cache and TLB misses. Their results indicate that the additional cost of partitioning can be higher than the benefit of having a reduced number of cache and TLB misses, thus favouring the no partitioning join.

Albutiu et al. [2] looked at parallel sort-merge join algorithms. The authors report that their implementation of the massively parallel sort-merge (MPSM) join is significantly faster than hash joins, even without SIMD instructions.

Balkesen et al. [4] implemented efficient versions of two hash join algorithms – the no partitioning join and the radix join – in order to compare their implementations with the ones from [6]. They show that a carefully tuned hardware-conscious radix join algorithm outperforms a no partitioning join. Furthermore, the authors argue that the number of hardware-dependent parameters is low enough, such that hardware-conscious join algorithms are as portable as their hardware-oblivious counterparts. In [3], the authors further show that the radix hash join is still superior to sort-merge approaches for current SIMD/AVX sizes.

Lang et al. [21] show the importance of NUMA-awareness for hash join algorithms on multi-cores. Their implementation of a NUMA-aware join claims an improvement over [4] by a factor of more than two.

2.3 Distributed Join Algorithms

Goncalves et al. [16, 15] and Frey et al. [12, 13] have developed a join algorithm, called cyclo-join, suited for ring network topology networks. In the setup phase of the cyclo-join, both relations are fragmented and distributed over all n machines. During the execution, data belonging to one relation is kept stationary while the second relation are passed on from one machine to the next. Similar to our approach, the idea is that the data is too large to fit in one machine, but can fit in the distributed memory of the machines connected on the ring [11]. The cyclo-join uses RDMA as a transport mechanism. The cyclo-join differs from our work in that the cyclo-join is an experimental system that explores how to use the network as a form of storage. The hot set data is kept rotating in the ring and several mechanism are proposed to identify which data should be put on the storage ring [16]. In DaCyDB the authors use RDMA to connect several instances of MonetDB in a ring architecture [15].

Polychroniou at al. [27] propose three variants of a distributed join algorithm which minimize the communication costs. The authors tested their implementation of the proposed join algorithms on a Gigabit Ethernet network. They show that the 3-phase and 4-phase track join algorithms can significantly reduce the overall network traffic.

Rödiger et al. [28] propose locality-sensitive data shuffling, a set of techniques, including optimal assignment of partitions, network communication scheduling, adaptive radix partitioning, and selective broadcast indented to reduce the amount of communication of distributed operators.

Recent work around distributed joins [1, 25] in map-reduce environments focuses on carefully mapping the join operator to the relevant data in order to minimizing network traffic.

These contributions show that the network is the main bottleneck for join processing.

3. BACKGROUND

3.1 Radix Hash Join

The radix hash join proposed by Manegold et al. [23] is a hardware-conscious main-memory hash join. The algorithm operates in two stages. First, both input relations R and S are divided into disjoint partitions according to the join attributes. The goal of the partitioning stage is to ensure that the resulting partitions fit into the private cache of CPU cores. Second, a hash table is built over each partition of the inner relation and is probed using the data of the corresponding partition of the outer relation. Having partitions and hash tables which fit into the processor cache has a major impact on performance compared to accessing large hash tables, which results in a higher cache miss rate [31].

The partitioning algorithm of the radix hash join determines the position of a tuple based on the key's b lower bits, thus creating 2^b partitions in total. The creation of the partitions is performed in p passes (multi-pass partitioning), each pass $i \in \{1 \dots p\}$ operating on a different non-overlapping

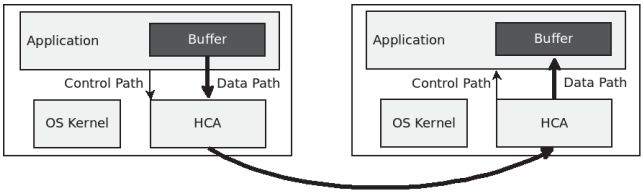


Figure 1: Example of an RDMA transfer from one machine to another. Control and data path are separated. The operating system is completely bypassed.

subset b_i of the b bits such that the number of simultaneously created partitions 2^{b_i} does not exceed the number of TLB entries or cache lines. The multi-pass partitioning scheme employed by the radix hash join avoids excessive TLB misses and cache trashing, and therefore allows to generate a large number of partitions without compromising performance.

The radix join can be parallelized by dividing both input relations into non-overlapping parts and assigning the individual parts to different worker threads. Each thread partitions the data to which it was assigned and afterwards adds the result to a task queue. Finally, each task is consumed during the build-probe phase [4, 6].

3.2 RDMA and InfiniBand

3.2.1 Benefits and Challenges of RDMA

Remote Direct Memory Access (RDMA) is a mechanism allowing direct access and placement of data in the main-memory of a remote machine. RDMA is offered by InfiniBand [17] and other recent hardware implementations, for example, RDMA over Ethernet (iWARP and RoCE).

For a memory region to be accessible by the network card, it needs to be registered. During the memory registration process, the memory is pinned to avoid those pages being swapped out while being accessed by the network card. A registered part of memory is referred to as a memory region (MR). As shown by Frey et al. [11], the memory region registration cost increase with the number of registered pages. To reduce the overall registration cost and to avoid pinning large parts of main-memory, efficient buffer management is crucial for high performance. An algorithm should reuse existing RDMA-enabled buffers as often as possible and avoid registering new memory regions on the fly.

Main-memory can directly be accessed by an InfiniBand Host Channel Adapter (HCA) or an RDMA-enabled network interface card (RNIC). This mechanism allows to bypass the network stack, avoids context switches and makes large data transfers more efficient as it eliminates the need to copy the data across intermediate buffers inside the operating system. A message which has not been copied into any temporary buffer during its transmission is called a zero-copy message (see Figure 1).

RDMA separates control and data path. Requests to read or write remote memory are added to a queue and executed asynchronously by the network card. The data transfer therefore does not involve any CPU operation, meaning that the processor remains available for processing while a network operation is taking place. In order to prevent processor cores from becoming idle, an algorithm needs to be able to interleave computation and communication.

Dragojevic et al. [9] have developed FaRM, a distributed computing platform that makes use of RDMA. FaRM combines the memory of multiple machines into one shared address space. In their evaluation on a 40 Gbit/s RoCE network, the authors show that accessing remote memory is slower than local memory accesses, even when using RDMA. This is an important observation as it highlights the importance of hiding the network latency by interleaving computation and communication when using RDMA.

3.2.2 Programming Abstractions

RDMA provides one-sided and two-sided operations. When using one-sided read and write operations (memory semantics), data is directly written into or read from a specified RDMA-enabled buffer without any interaction from the remote host. Two-sided calls on the other hand implement send/receive message-passing operations (channel semantics). The receiver registers memory regions into which incoming messages will be written and will receive a notification when a write has occurred. No significant performance difference between one-sided and two-sided operations has been observed in previous work [10].

Multiple message-passing systems, like MPICH2 [22] have been implemented over InfiniBand. Furthermore, InfiniBand also provides upper-layer protocol support such as IP-over-InfiniBand (IPoIB), which provides a transparent interface to any IP-based application.

4. ALGORITHMIC DETAILS

In this section we present a distributed variant of the radix hash join. In a distributed system with many multi-core and multi-processor machines, the ability to generate a sufficiently large number of partitions without being limited by the TLB or cache capacity is important in order to be able to assign partitions to every available core and preventing cores from becoming idle. Because there is no data dependency between the individual partitions, these can be processed with a high degree of parallelism. The radix hash join with its multi-pass partitioning strategy prevents excessive TLB and cache misses while partitioning the data into a large number of partitions, thus making it a good candidate to be transformed into a distributed algorithm. Furthermore, the authors of [4, 3] clearly show that a carefully tuned radix hash join is superior to other join algorithms on multi-core machines.

For the distributed variant of the radix hash join, we propose a couple of modifications to the partitioning, build and probe phases of existing parallel hash join solutions.

4.1 Histogram Computation Phase

As a first step, all threads compute a histogram over the input data. By assigning the threads to sections of the input relations of equal size, we can ensure an even load distribution among the worker threads.

Next, all the threads within the same machine exchange their histograms and combine them into one machine-level histogram providing an overview of the data residing on a particular machine. Computing the machine-level histograms is identical to the histogram computation of the join algorithm described in [4].

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

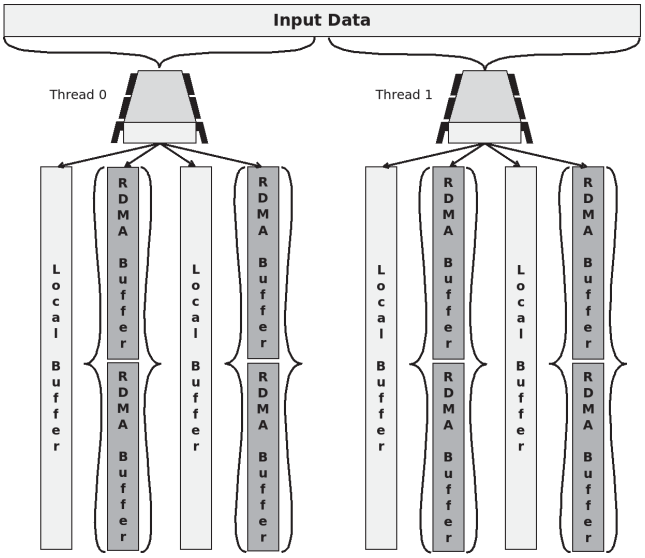


Figure 2: Two threads partitioning the input data into a set of local and RDMA-enabled buffers.

necessary size of the buffers which need to be allocated to store the data received over the network.

From the machine-level and global histograms the join algorithm can compute a machine-partition assignment for every node in the cluster. This assignment can be dynamic or static. The algorithm computing the machine-partition assignment is independent of the rest of the join algorithm. For the experiments we implemented a static round-robin assignment and, for skewed workloads, a dynamic algorithm which first sorts the partitions based on their element count and then assigns them evenly over all machines.

4.2 Partitioning Phase

The purpose of the partitioning phase of the radix hash join is to ensure that the partitions and hash tables fit into the processor cache. For the distributed radix join, we additionally want to ensure maximum resource utilization, in particular we need to be able to assign at least one partition to each processor core. Therefore, the number of partitions needs to be at least equal to the total number of CPU cores in order to prevent cores from becoming idle.

In the multi-pass partitioning phase of the algorithm we distinguish between two different types of partitioning passes: (i) a network partitioning pass which interleaves the computation of the partitions with the network transfer and (ii) local partitioning passes which partition the data locally in order to ensure that the partitions fit into the processor cache. The latter does not involve any network transfer.

4.2.1 Network Partitioning Pass

To efficiently use the asynchronous nature of RDMA, the data needs be transmitted over the network in parallel with the computation. When designing the algorithm, we need to avoid having a separate network transmission phase during which the processor cores are idle. To achieve these goals, we introduce the concept of a network-partitioning pass in which the data is partitioned and distributed in parallel.

Crucial for high performance processing is the management of the partitioning buffers, in particular the ability to reuse existing RDMA-enabled buffers. For each partition which will be processed locally, a thread receives a local buffer for writing the output. Based on the histogram computation, the required size of the local buffers can be determined such that local buffers do not overflow. Remote partitions need to be transmitted over the network. For processing remote partitions, a thread receives multiple fixed-sized RDMA-enabled buffers. Data belonging to a remote partition is partitioned directly into these buffers. When a remote buffer is full, it will be transmitted over the network to the target machine. In order to be able to continue processing while a network operation is taking place, at least two RDMA-enabled buffers are assigned to each thread for a given partition. The buffers assigned to one partition can be used in turn and reused once the preceding network operation completes. To hide the buffer registration costs, the RDMA-enabled buffers are drawn from a pool containing preallocated and preregistered buffers. All buffers, both local buffers and RDMA-enabled buffers, are private to each thread, such that no synchronization is required while partitioning the input relations (see Figure 2).

4.2.2 Receiving Incoming Data

Depending on the available amount of main memory on the receiving machine, we can chose between one-sided and two-sided operations. If the amount of main memory is large enough to hold all the data, one-sided operations can be used. In such a setup, the receiver needs to allocate one large RDMA-enabled buffer for each partition and each remote machine. The necessary size of these buffers is known from the histogram phase. No interaction from the receiving machine is required during the partitioning as the incoming data is directly written into the destination buffers by the network card. On the other hand, if only a small amount of memory is at our disposal, we want to avoid that large parts of main-memory are registered for RDMA transfer. Otherwise pages cannot be swapped out, which would significantly impact the performance of other applications and concurrent queries. In such a case, we use two-sided RDMA operations and only register a predefined number of small RDMA-enabled buffers in order to receive incoming data. In addition to these receive buffers, we allocate larger non-RDMA buffers for each partition into which the data from the receive buffers will be copied. The receive buffers can be reused once the copy operation terminated successfully.

At the end of the network partitioning pass, the partitions are assembled for further processing by combining the buffers containing the local data with the buffers holding the data received over the network.

4.2.3 Local Partitioning Passes

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

4.3 Build & Probe Phases

In the build-probe phase a hash table is built over the data of each partition of the inner relation. Data from the corresponding partition of the outer relation is used to probe the hash table. Because there is no data dependency between two partitions, they can be processed in parallel.

The result containing the matching tuples can either be output to a local buffer or written to RDMA-enabled buffers,

Symbol	Description
$\|R\|$ $\|S\|$	Size of the inner/outer relation [MB]
N_{M}	Number of machines
$N_{\mathrm{C/M}}$	Number of cores per machine
$ps_{\mathrm{Part.}}$	Partitioning speed of a thread [MB/s]
net_{max}	Network bandwidth per host [MB/s]
hb_{Thread}	Hash table build speed of a thread [MB/s]
hp_{Thread}	Hash table probe speed of a thread [MB/s]

Table 1: List of basic symbols.

depending on the location where the result will be further processed. Similar to the partitioning phase, we transmit an RDMA-enabled buffer over the network once it is full. To be able to continue processing, each thread receives multiple output buffer for transmitting data. The buffers can be reused when the proceeding network operation completed.

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

5. ANALYTICAL MODEL

In this section, we present a model describing the individual phases of the distributed hash join algorithm. In addition we want to find (i) an optimal number of processor cores per machine as well as (ii) an optimal number of machines for a given input size.

Table 1 provides an overview of the symbols used in the analytical model.

5.1 Partitioning Phase

The partitioning phase is composed of multiple partitioning passes which can be of two types: (i) network partitioning passes involving the partitioning and transfer of the data over the network and (ii) subsequent local partitioning passes which ensure that the partitions fit into the processor caches but do not involving any network operations.

In this model, we assume that every thread runs on the same identical processor core and that each individual thread can read a tuple from the input, determine its partition and write the tuple to the corresponding buffer at a rate $ps_{\mathrm{Part.}}$.

5.1.1 Network Partitioning Pass

The partitioning speed ps_{Thread} at which a thread can partition its input data is composed of two parts: (i) the speed at which tuples are written to the respective buffers $ps_{\mathrm{Part.}}$ and (ii) the speed at which tuples belonging to remote partitions can be transmitted over the network ps_{Network}.

The total network speed net_{Max} is shared equally among all partitioning threads on the same machine. When using two-sided RDMA calls, one thread is responsible for processing incoming partitions and has the full incoming bandwidth at its disposal, while the remaining $N_{\mathrm{C/M}} - 1$ partitioning threads share the outgoing network bandwidth.

$$ps_{\mathrm{Network}} = \frac{net_{\mathrm{Max}}}{N_{\mathrm{C/M}} - 1} \tag{1}$$

Assuming uniform distribution of the data over all N_{M} machines, we can estimate that $(|R|+|S|) \cdot \frac{1}{N_{\mathrm{M}}}$ tuples belong to local partitions, the rest is send to remote machines.

At this point, the system can either be limited by the partitioning speed of the threads (CPU-bound) or by the available network bandwidth on each host (network-bound). A system is network-bound if the tuples belonging to remote partitions are output at a faster rate than the network is able to transmit.

$$\frac{N_{\mathrm{M}} - 1}{N_{\mathrm{M}}} \cdot ps_{\mathrm{Part.}} > ps_{\mathrm{Network}} \tag{2}$$

In systems which are CPU-bound, the overall processing rate is fully determined by the partitioning speed of each thread $ps_{\mathrm{Part.}}$. The entire system is composed of N_{M} machines, each of which contains $N_{\mathrm{C/M}}$ processor cores. Equation 3 gives us the global partitioning speed of the network partitioning pass ps_1 for CPU-bound systems.

$$ps_1 = N_{\mathrm{M}} \cdot \left(N_{\mathrm{C/M}} - 1\right) \cdot ps_{\mathrm{Part.}} \tag{3}$$

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

$$\begin{aligned} ps_{\mathrm{Thread}} &= \frac{1}{\frac{\frac{1}{N_{\mathrm{M}}}}{ps_{\mathrm{Part.}}} + \frac{\frac{N_{\mathrm{M}}-1}{N_{\mathrm{M}}}}{ps_{\mathrm{Network}}}} \\ &= \frac{N_{\mathrm{M}} \cdot ps_{\mathrm{Part.}} \cdot ps_{\mathrm{Network}}}{(N_{\mathrm{M}} - 1) \cdot ps_{\mathrm{Part.}} + ps_{\mathrm{Network}}} \end{aligned} \tag{4}$$

From Equations 1 and 4 we can determine the overall partitioning speed of network-bound systems.

$$\begin{aligned} ps_1 &= N_{\mathrm{M}} \cdot \left(N_{\mathrm{C/M}} - 1\right) \cdot ps_{\mathrm{Thread}} \\ &= \frac{{N_{\mathrm{M}}}^2 \cdot (N_{\mathrm{C/M}} - 1) \cdot ps_{\mathrm{Part.}} \cdot net_{\mathrm{Max}}}{(N_{\mathrm{C/M}} - 1) \cdot (N_{\mathrm{M}} - 1) \cdot ps_{\mathrm{Part.}} + net_{\mathrm{Max}}} \end{aligned} \tag{5}$$

5.1.2 Local Partitioning Passes

Local partitioning passes do not involve any network transfer and all threads in the system partition the data at their maximum partitioning rate $ps_{\mathrm{Part.}}$. Therefore, the global processing speed of this phase (ps_2) increases with the total number of available CPU cores.

$$ps_2 = N_{\mathrm{M}} \cdot N_{\mathrm{C/M}} \cdot ps_{\mathrm{Part.}} \tag{6}$$

5.1.3 Combining Partitioning Passes

The partitioning phase is composed of p passes, one of them involving the transfer of the data over the network, the other $p - 1$ passes operate on local data only. We can derive an expression for the time required to partition both input relations of size $|R|$ and $|S|$.

$$t_{\mathrm{partitioning}} = (|R| + |S|) \cdot \left(\frac{1}{ps_1} + \frac{(p-1)}{ps_2}\right) \tag{7}$$

5.2 Build & Probe Phases

In the build phase, each thread is creating a cache-sized hash table over the inner relation of size $|R|$ at an average build speed hb_{Thread}. Building hash tables over the partitions can be done in parallel as there is no data dependency between partitions.

$$hb = N_{\text{M}} \cdot N_{\text{C/M}} \cdot hb_{\text{Thread}} \tag{8}$$

$$t_{\text{build}} = \frac{|R|}{hb} \tag{9}$$

During the probe phase, tuples from the outer relation of size $|S|$ are used to probe the hash tables at a rate hp_{Thread}. Partitions can be processed in parallel during the probe phase. The global speed at which hash tables can be probed increases with the number of available cores.

$$hp = N_{\text{M}} \cdot N_{\text{C/M}} \cdot hp_{\text{Thread}} \tag{10}$$

$$t_{\text{probe}} = \frac{|S|}{hp} \tag{11}$$

5.3 Optimal Number of Cores and Machines

Maximum utilization of the available resources is achieved in the network partitioning phase if the processing speed at which each thread can partition the data is equal to the maximum thread partitioning speed (maximum CPU utilization) and the data belonging to remote partitions is transmitted over the network at the maximum network transmission speed per host (maximum network utilization).

$$\begin{aligned} &\frac{N_{\text{M}} - 1}{N_{\text{M}}} \cdot ps_{\text{Part.}} = \frac{net_{\text{Max}}}{N_{\text{C/M}} - 1} \\ &\Leftrightarrow \left(N_{\text{C/M}} - 1\right) = \frac{N_{\text{M}}}{N_{\text{M}} - 1} \cdot \frac{net_{\text{Max}}}{ps_{\text{Part.}}} \end{aligned} \tag{12}$$

From the above we can determine that the optimal number of processor cores is such that it can exactly saturate the available network bandwidth. The optimal number of cores should therefore be equal to the ratio of the network bandwidth and the partitioning rate of a thread.

In the network partitioning pass, data is partitioned into N_{P1} partitions. Within a machine each of the $N_{\text{C/M}} - 1$ partitioning threads has N_{P1} partitioning buffers. These buffers can either be local buffers or RDMA-enabled buffers. RDMA-enabled buffers are of a fixed predetermined size $S_{\text{RDMA-Buffer}}$. If the smaller inner relation R is spread across too many machines, these RDMA-enabled buffers will no longer be fully filled before being transmitted over the network, thus resulting in an inefficient usage of the network.

$$\begin{aligned} &\frac{|R|}{N_{\text{M}} \cdot N_{\text{P1}} \cdot \left(N_{\text{C/M}} - 1\right) \cdot S_{\text{RDMA-Buffer}}} \geq 1 \\ &\Leftrightarrow N_{\text{M}} \leq \frac{|R|}{N_{\text{P1}} \cdot \left(N_{\text{C/M}} - 1\right) \cdot S_{\text{RDMA-Buffer}}} \end{aligned} \tag{13}$$

Equation 13 determines an upper-bound of the number of machines given a specific workload (size of the inner relation) and system configuration (size of the RDMA-buffers). Scaling above this number will lead to loss of bandwidth in the network partitioning pass.

In addition, we need to ensure that every core is assigned to at least one partition for further processing. Therefore the total number of processor cores should not exceed N_{P1}.

$$N_{\text{C/M}} \cdot N_{\text{M}} \leq N_{\text{P1}} \tag{14}$$

6. EXPERIMENTAL EVALUATION

6.1 Experimental Setup

We evaluated our implementation of the distributed join on a cluster of ten machines connected by an QDR Infiniband network as well as a four-machine cluster featuring an FDR InfiniBand network.

The goal of this evaluation is to understand how to use RDMA in the context of distributed rack-scale databases, rather than to compare the performance of a distributed join to that of a single-machine algorithm. Because of a lower overhead in terms of coordination and communication, a single-machine algorithm is expected to always be faster. In this paper, we use the algorithm of [4] as it makes the results comparable to a public baseline.

Like [21], we noticed that the algorithm in [4] did not run beyond certain amounts of data. We have extended the algorithm such that it can process large data sizes. In order to have a more realistic baseline, we have also modified the algorithm in [4] to make it more NUMA-aware. In particular we created multiple task queues, one for each NUMA region. If a buffer is located in region i, it is added to the i-th queue. A thread first checks the task queue belonging to the local NUMA-region and only when there is no local work to be done, will it check other queues. Furthermore, we have implemented both the first and second partition passes with SIMD/AVX vector instructions. With these modifications, the single-machine algorithm of [4] reaches a throughput of 700 million join argument tuples-per-second, similar to that of [21] (see Figure 5a).

In addition, we implemented a network component using TCP/IP and evaluated our algorithm using IPoIB, which gives a bandwidth slightly larger to a 10Gbit Ethernet network. To better compare both versions, we evaluated the RDMA implementation which uses channel semantics.

Our C++ implementation of a distributed join is partially based on the source code made available by [4][1] and [6][2].

Details on the hardware can be found in Table 2.

6.1.1 Workloads

In order to compare our results to the results of [4] and [6], we have selected a workload composed of narrow tuples, each consisting of a join key and a record id (e.g., <key, rid>). The tuples are 16 byte wide. In the data loading phase the input data is distributed evenly across all available machines. The Rids are range-partitioned at load time and each machine is assigned a particular range of Rids.

Similar to [4] and [6], we focus on highly distinct value joins. For each tuple in the inner relation, there is at least one matching tuple in the outer relation. The ratio of the inner and outer relation sizes which are used throughout the experiments are either 1:1, 1:2, 1:4, 1:8 or 1:16. To analyze the impact of data skew, we generated two skewed datasets, with different values of the Zipf distribution: a low skew

[1] http://www.systems.ethz.ch/projects/paralleljoins
[2] http://pages.cs.wisc.edu/~jignesh/

	FDR Cluster	QDR Cluster	Multi-Core Server
CPUs	Intel Xeon E5-4650 v2 2.40 GHz	Intel Xeon E5-2609 2.40 GHz	Intel Xeon E5-4650 v2 2.40 GHz
Cores/Threads	40/80	8/8	40/80
Memory	512 GB	128 GB	512 GB
Cache Sizes	64 KB 256 KB 25 MB	64 KB 256 KB 10 MB	64 KB 256 KB 25 MB
InfiniBand	Mellanox FDR HCA	Mellanox QDR HCA	-

Table 2: Hardware used in our evaluation.

dataset with a value of 1.05 and high skew with a skew factor of 1.20. To gain insights on the behaviour of the join not only for column stores but for row stores as well, we also use a workload with a variable payload. These tuples can either be 16, 32 or 64 bytes wide.

6.2 Size of RDMA Buffers

As explained in Section 4, the primary communication abstraction is the sending and receiving of RDMA-enabled buffers. Therefore, a natural question to ask is how much buffer space should be allocated and which impact the buffer size has on the network throughput.

InfiniBand networks can either be bound by the maximum package rate which can be processed by the HCA or by the available network bandwidth. Figure 3 shows the observed bandwidth on both the QDR and FDR network between two machines for message sizes ranging from 2B to 512KB. One can observe that both systems can reach and maintain full bandwidth for buffers larger than 8KB.

Unless otherwise stated, the size of the RDMA-enabled buffers is fixed to 64KB for the rest of the experiments.

6.3 Joins on Rack-Scale Systems

One of the first questions to ask is how the hash join algorithm behaves on the different hardware configurations described in Table 2. In order to be able to compare the distributed join with the implementation from [4], we selected a high-end multi-processor server containing four sockets using eight out of the ten CPU cores on each socket and compared it against four nodes from the FDR and QDR cluster. On each of the cluster machines we used eight cores. Thus, the total number of processor cores for each of the hardware configurations is 32 physical cores.

Inside the high-end server the CPUs are connected via QuickPath (QPI). Each processor is attached to two neighbours. Using the STREAM benchmark [24], we measured the bandwidth with which one core can write to a remote NUMA region. The total bandwidth offered by QPI is not fully available to a single core. On different hardware configurations, we measured different values for the per-core write bandwidth, even within the same processor family. In this paper, we show the results for the configuration which offered us the highest inter-socket bandwidth, which peaked at 8.4 GB/s. The distributed system is composed of individual machines connected to each other via a single InfiniBand switch. The measured bandwidth on the QDR network is around 3.4 GB/s. The FDR network offers a higher bandwidth with a peak performance close to 6.0 GB/s. The architecture of both systems is illustrated in Figure 4.

In the first experiment we used three different workloads consisting of 1024 million, 2048 million and 4096 million tuples per relation. The results are shown in Figure 5a. The centralized algorithm outperforms the distributed version for all data sizes. This is expected because the algorithm has a lower coordination overhead and the bandwidth between cores is significantly higher than the inter-machine bandwidth. For large data sizes, the distribution overhead is amortized. The execution time 2048 million and 4096 million tuples per relation shows an increase less than 30%.

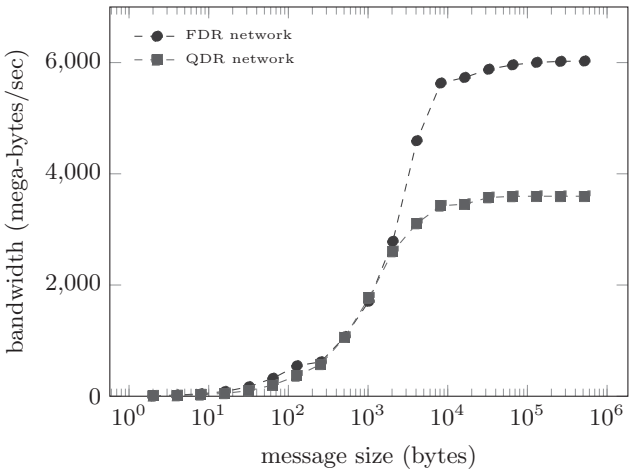


Figure 3: Point-to-point bandwidth for different message sizes on the QDR and FDR networks.

In Figure 5b we compare the TCP/IP version and two RDMA-based implementations. The first RDMA-based variant does not interleave computation and communication. After issuing an RDMA request, a thread waits for the network transfer to finish before it continues processing. Hence, partitioning and network communication are never interleaved. The second RDMA-based version is the algorithm described in Section 4. It tries to hide the network latency by interleaving computation and communication.

We can observe that the differences in execution time is caused only by differences in the network partitioning pass. The TCP/IP versions takes long to complete this phase. The reasons for this performance difference are three-fold: (i) although FDR InfiniBand provides 6.0 GB/s of network bandwidth, we cannot reach this throughput using IPoIB. We measured the IPoIB bandwidth to be only 1.8 GB/s, slightly higher than the bandwidth provided by 10Gb Ethernet; (ii) when using TCP/IP, a context switch into the kernel is required which causes additional overhead; and (iii) the message needs to be copied across intermediate buffers during the network transfer.

We can also see a small difference between the execution time of the interleaved and non-interleaved RDMA implementations. Although both can benefit from the increased bandwidth, the version which interleaves computation and communication hides parts of the network latency leading to a reduced execution time of the network partitioning pass.

From the second experiment we can conclude that we cannot rely on the upper-layer protocol support of the network in order to achieve the full performance. Instead we have to use the RDMA primitives. Furthermore, interleaving computation and communication also brings down the execution time of the network partitioning pass by an additional 35%. We expect that this benefit is more pronounced as more data needs to be transmitted over the network.

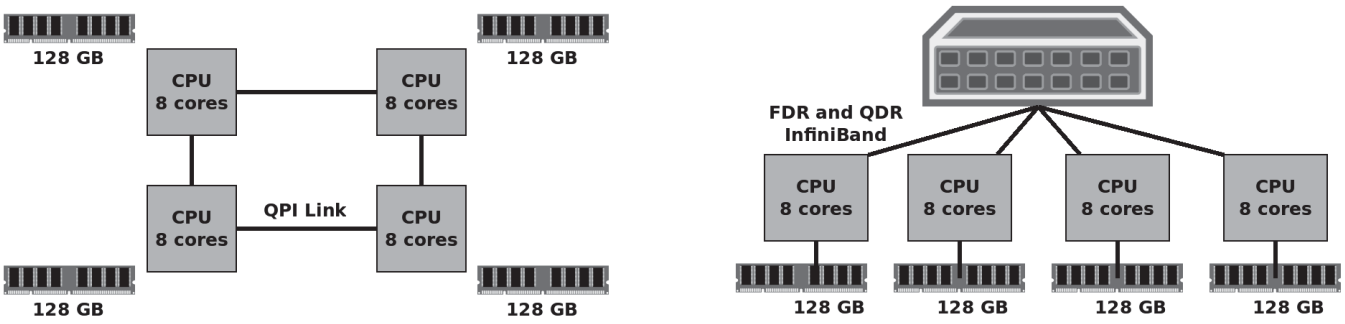
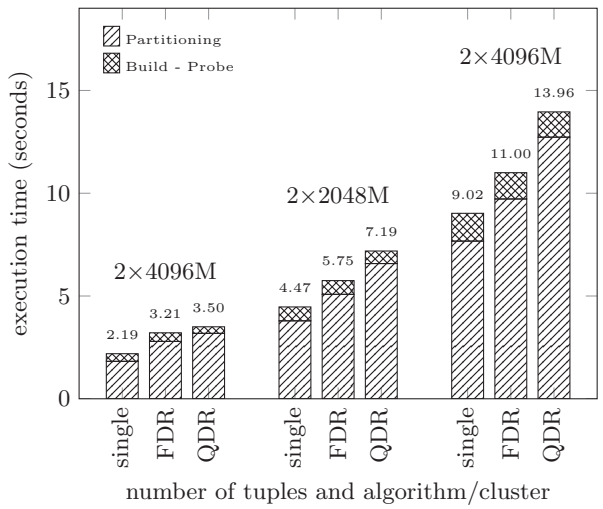
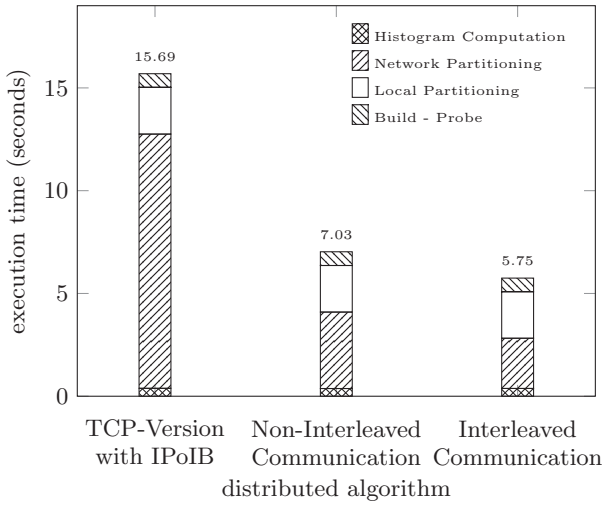


Figure 4: Comparison of the single-machine and distributed setup: The multi-processor machine has 4 CPUs connected by a QPI link. The distributed setup is composed of 4 distinct machines. The machines are connected by a QDR and FDR InfiniBand network.



(a) Comparison of the execution time on the high-end server and the distributed setup with 4 machines on the QDR and FDR network. The total number of CPU cores for all three experiments is 32 cores.



(b) Execution time of a join of 2×2048 million tuples for three variants of the distributed radix hash join. All experiments run on 4 machines with 32 CPU cores in total (FDR cluster). The TCP/IP-based version runs over IPoIB.

Figure 5: Baseline experiments. The distributed radix hash join is compared against a single-machine algorithm running on a high-end server and a TCP/IP-based implementation.

6.4 Horizontal Scale-Out Behaviour

6.4.1 Large-to-Large Table Joins

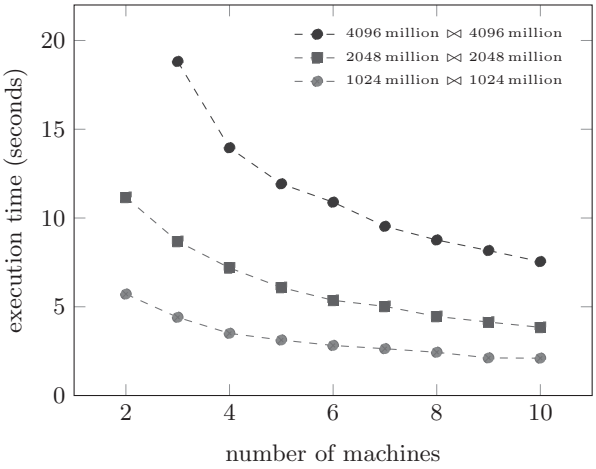
To study the impact of the input relation sizes on the performance of the distributed join, we varied the input relation sizes and the number of machines. In large-to-large table joins, both input relations are of the same size and each element of the inner relation is matched with exactly one element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

Due to the available memory, the largest workload containing 2×4096 million tuples (≈ 128 GB) cannot be executed on two machines.

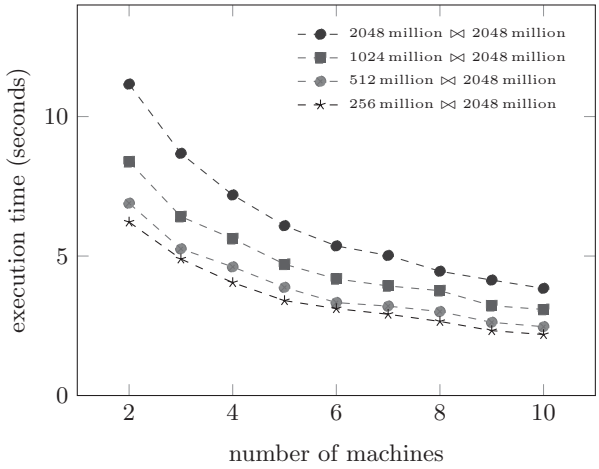
Figure 6a presents the average execution time for each of the three workloads using different numbers of machines. We can observe that the execution time doubles when doubling the amount of input data. The relative difference in execution time between the first two workloads is on average a factor of 1.98. The difference between the second and third workload is a factor of 1.92.

The experiment shows that the execution time for a large-to-large join increases linearly with the size of both input relations: doubling the relation sizes results in a doubling of the total execution time of the join algorithm.

The execution time for all three workloads goes down as we increase the number of machines. However, we can also observe a sub-linear speed-up when comparing the configuration with two and ten nodes. The optimal speed-up in such a setup should lead to a five times improvement in the execution time, which cannot be observed in this experiment. The reason for this sub-linear scale-out behavior on the QDR cluster will be studied in greater detail in Section 6.4.3 and Section 6.6.



(a) Execution time of the distributed hash join for different large-to-large joins using a variable number of machines. The experiment was conducted on the QDR cluster.



(b) Execution time of the distributed hash join for small-to-large joins with different relative relation sizes. The experiment was conducted on the QDR cluster.

Figure 6: Execution time for large-to-large and small-to-large table joins.

6.4.2 Small-to-Large Table Joins

To explore the impact of the relative sizes of the inner and outer relations, we measured the performance of the distributed join using an outer relation of fixed size, composed of 2048 million tuples, and a variable number of tuples for the inner relation ranging from 2048 million tuples (1-to-1 workload) to 256 million tuples (1-to-8 workload). All measurements were taken on the QDR cluster.

From Figure 6b we can see that the execution time of the join decreases when reducing the size of the inner relation. The execution time of the radix hash join is dominated by the time to partition the data. These partitioning costs decrease linearly with the size of both input relations. Therefore, when keeping the size of the outer relation fixed at 2048 million tuples and decreasing the number of tuples in the inner relation, we can see a reduction in the execution time by almost half when comparing the 1-to-1 workload to the 1-to-8 workload.

6.4.3 Execution Time Break Down

In the previous experiments we see a sub-linear reduction in the execution when increasing the number of machines. To understand the cause of this behaviour, we take a closer look at the 2048 million ⋈ 2048 million tuple join on the QDR cluster.

Figure 7a visualizes the execution time of the different phases of the join and illustrates the effects of scale-out in more detail. The partitioning phase is composed of two passes, each creating 2^{10} partitions. The resulting 2^{20} partitions are $\sim$ 32KB in size and fit into the processor caches.

During the first partitioning pass the data is distributed over the network. This phase is completed once all the data has been send out and acknowledged by the receiving hosts. When increasing the number of machines from two to ten machines we expect – in an ideal scenario – a speed-up factor of 5. However, when examining the execution time of the individual phases (Figure 7a), one can observe a near-linear speed-up for the second partitioning pass (speed-up by 4.73) and for the build/probe phase (speed-up by 5.00). The speed-up of the first partitioning pass on the other hand is limited because the network transmission speed of 3.4 GB/s is significantly lower than the partitioning speed of a multi-core machine. As a consequence, the network presents a major performance bottleneck and limits the speed-up.

With an increasing number of machines, a larger percentage of the input data needs to be transmitted over the network, which puts additional pressure on the network component and does not allow us to fully leverage the performance gains of the increased parallelism. Furthermore, adding machines to the network is likely to increase overall network congestion during the network partitioning pass if communication is not scheduled carefully. The overall speed-up when scaling from two to ten machines is 2.91.

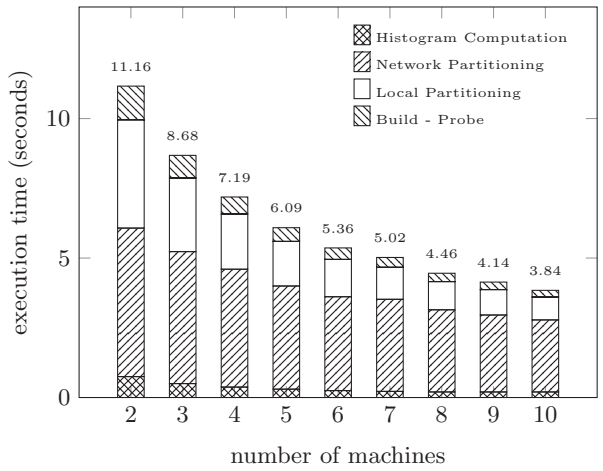
6.4.4 Scale-Out with Increasing Workload

In order to deal with ever increasing workload sizes, a common approach is to add more resources to an existing system to maintain a constant execution time despite the increase in data volumes.

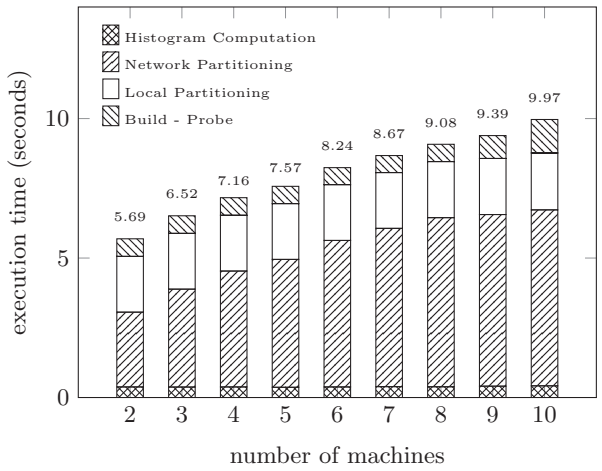
In the experiment, we vary the workload size from 2×1024 million ($\approx$ 60 GB) to 2×5120 million ($\approx$ 300 GB) tuples. For each increase in the data size by 512 million tuples per relation, we add another machine to the system.

Figure 7b shows the execution time of each phase. One can observe that the algorithm maintains a constant performance for the second partitioning pass as well as the build-probe phase. On the other hand, we see a significant increase in the execution time of the network partitioning pass as we add more machines.

When increasing the input sizes along with the number of machines, the amount of data which needs to be processed per machine remains identical. Thus all local partitioning passes and the build-probe phase show constant performance. However, increasing the number of machines, leads to a higher percentage of the data that needs to be exchanged over the network. Because the QDR network bandwidth is significantly lower than the combined partitioning speed of all threads, the network will become a significant performance bottleneck, thus leading to a notable increase in the execution time of the network partitioning phase.



(a) Execution time of each phase of the distributed hash join for a workload of 2048 million ⋈ 2048 million tuples on a variable number of machines. The experiment was conducted on the QDR cluster.



(b) Execution time of each phase of the distributed hash join for an increasing number of tuples and machines. The relation size increases by 2×512 million tuples for each machine that is added. The experiment was conducted on the QDR cluster.

Figure 7: Scale-out behaviour for constant and increasing relation sizes.

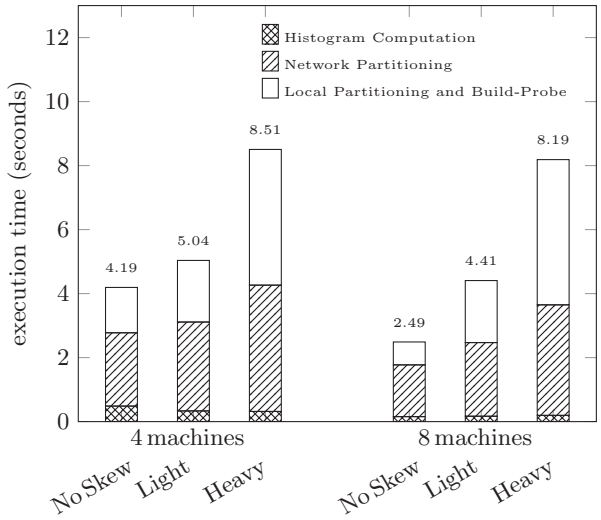


Figure 8: Effect of data skew for two skew factors and hardware configurations on the QDR cluster.

6.5 Impact of Data Skew

In this section, we study the effects of data skew. Similar to the authors of [6] we populate the foreign key column of the outer relation with two data sets. The first one with a low data skew which follows a Zipf distribution law with a skew factor of 1.05 and a highly skewed data set with a factor of 1.2. The relation sizes are 128 million tuples for the inner relation and 2048 million tuples for the outer relation.

In order to ensure that two skewed partitions are not assigned to the same machine, we use a dynamic partition-machine assignment. In this dynamic assignment the partitions are first sorted in decreasing order according to their element count before being distributed in a round-robin manner, thus preventing that the largest partitions are assigned to the same machine. In the build-probe phase, partitions are split according to the description in Section 4.3 when they contain more than twice the average number of tuples.

In Figure 8 we see an increase in the execution time for both workloads and configurations. We notice an increase in execution time for the network partitioning pass and local processing part, i.e. local partitioning and build-probe phase. The network phase is dominated by the time it takes to send all the data to the machine responsible for processing the largest partition. Similarly, the execution time of the local processing part is also dominated by that same machine. This effect is more pronounced for higher skew factors.

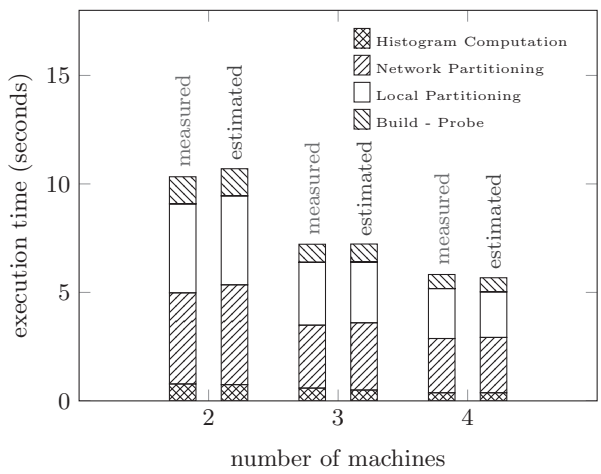
This result highlights the need to share tasks between machines. Although heavily skewed partitions can be split and distributed among threads in order to allow for a higher degree of parallel processing, the current implementation only allows work sharing among threads within the same machine and not across multiple machines, thus not fully exploiting the parallelism of the entire system. Nevertheless, we are confident that this issue can be addressed by extending the algorithm to allow work sharing between machines.

6.6 Comparing Joins on QDR and FDR

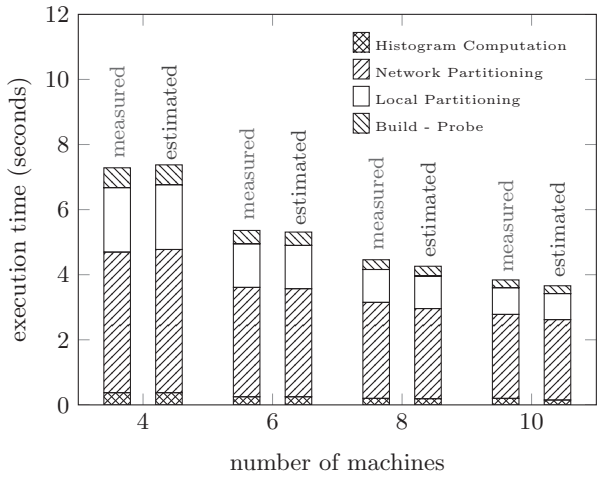
The experiments on the QDR cluster showed limited scalability of the network partitioning pass because of the limit network bandwidth. To address this issue we deployed the algorithm on a second cluster composed of four machines connected by an FDR InfiniBand network, offering close to twice the available network bandwidth.

For a 2048 million ⋈ 2048 million tuple join conducted on two, three and four machines on the FDR cluster, we can observe identical execution times for the phases not involving any network operation, i.e. the local partitioning pass as well as the build-probe phase. The increase in performance compared to the QDR cluster is due to the shorter network partitioning pass which benefits from the extra bandwidth.

The available network bandwidth of 6.0 GB/s cannot be over-saturated by the seven partitioning threads when running on two and three machines. In this configuration, the system is fully CPU-bound. The partitioning threads reach the maximum network bandwidth on four nodes, in which case $^3/_4$ of the data will be exchanged over the network.

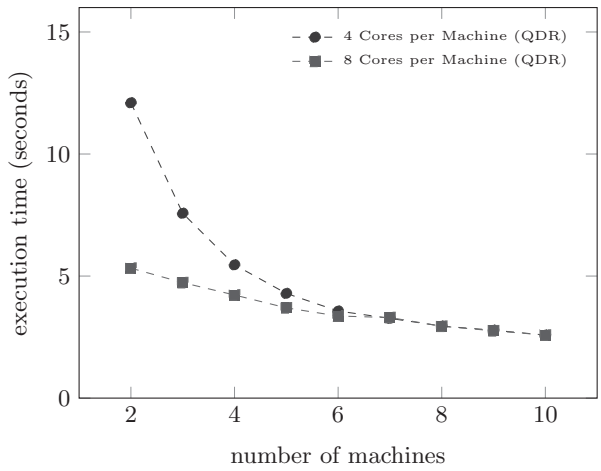


(a) Measured and estimated execution times on the FDR cluster for up to 4 machines.

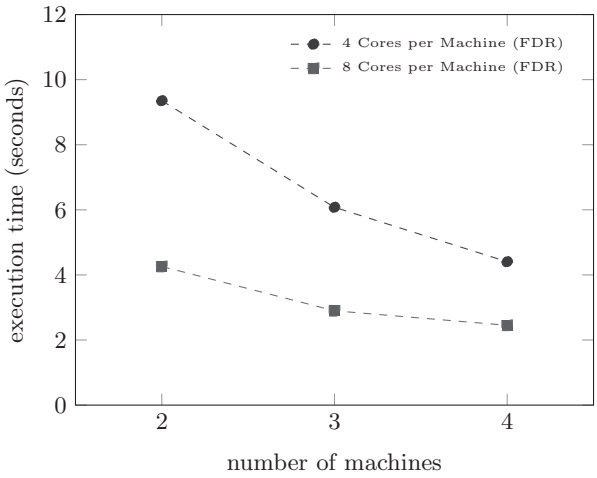


(b) Measured and estimated execution times on the QDR cluster for 4, 6, 8 and 10 machines.

Figure 9: Model verification for a 2048M ⋈ 2048M join on the FDR and QDR cluster.



(a) Execution time of the network partitioning phase with 4 and 8 threads per machine on the QDR cluster.



(b) Execution time of the network partitioning phase with 4 and 8 threads per machine on the FDR cluster.

Figure 10: Execution time of the network partitioning pass for a 2048 million ⋈ 2048 million join on both clusters for 4 and 8 threads.

The speed-up factor from two to four nodes of the network partitioning pass alone is 1.7 on the FDR cluster compared to only 1.3 on the QDR cluster.

From this experiment we can conclude that the network partitioning phase can scale to a large number of machines, provided that a sufficient amount of inter-machine bandwidth is available.

6.7 Impact of Wide Tuples

In previous experiments we focused on narrow tuples composed of 8-byte keys with and 8-byte record id in order to evaluate the performance of the join for column-store systems. To get additional insides into the behavior of the join for row-stores, we use tuples with variable payload size.

To keep the overall data size constant, we use relations composed of 2048 million 16-byte tuples, 1024 million 32-byte tuples and 512 million 64-byte tuples. We observed that the execution time of the join, as well as the execution time of each phase, is identical for all three workloads.

This results highlights that data movement is the primary performance factor of distributed join processing. The execution time is determined by the data size, independent whether the workload is composed of a large number of small tuples or a small number of large tuples.

6.8 Model Verification

In this section we validate the accuracy of the analytical model described in Section 5 by comparing its predictions to the experimental results gathered on both clusters.

The measured network throughput is 6.0 GB/s on the FDR network, respectively 3.4 GB/s on the QDR network. In addition, we observed a small performance degradation when increasing the number of machines on the QDR cluster. This decrease is due to the fact that adding machines

increases the overall network congestion. On each machine we use eight cores. Each thread is able to reach a local partitioning speed of 955 MB/s.

$$ps_{\text{FDR}}(N_M) = \frac{6000}{8-1} \quad \text{[MB/s]}$$

$$ps_{\text{QDR}}(N_M) = \frac{3400 - (N_M - 1) * 110}{8-1} \quad \text{[MB/s]} \qquad (15)$$

$$ps_{\text{Part.}} = 955 \quad \text{[MB/s]}$$

Using Equation 2 we know that the join is CPU bound on the FDR network for two and three machines and is close to being network-bound on four machines. Thus, for two and three machines we can assume that all threads partition the data at their full capacity $ps_{\text{Part.}}$ (CPU-bound). In all the other cases the join is network-bound. Using Equation 4 we can compute the partitioning speed of a thread for the network partitioning pass. The second local partitioning pass is always executed at the local partitioning rate $ps_{\text{Part.}}$.

Figure 9a shows the predicted and measured performance of a 2×2048 million tuple join on the FDR cluster, while Figure 9b compares the model to the results gathered on the QDR machines. One can clearly see that the predictions closely match the experimental results, varying on average by only 0.17 seconds.

6.8.1 Finding an Optimal Number of Threads

The analytical model allows us to find the optimal number of threads for a given hardware specification. Given Equation 12, we know that in order to achieve maximum utilization of the network and processing resources, the number of partitioning threads should be such that it can saturate the network without being fully network-bound.

Given the network speed and partitioning rate from Equation 15, we can determine the required number of processor cores for each of the two networks, which is four cores per machine on the QDR and seven cores per machine on the FDR cluster. To verify this result, we conducted two runs of experiments: the first run was performed with four and the second run with eight threads.

In Figure 10a we compare the execution times of the network partitioning pass on the QDR cluster. When increasing the number of machines, the percentage of data which needs to be exchanged over the network increases. We can observe that from five machines onwards, three partitioning threads are sufficient to fully saturate the QDR network. Adding additional cores (i.e. eight threads) will not speed up the execution as threads need to wait for network operations to complete before being able to reuse the RDMA-buffers.

Figure 10b shows the same experiment on the FDR cluster. Given that four threads are not able to fully saturate the available network bandwidth, increasing the number of cores will speed up the network partitioning pass.

7. DISCUSSION

In this paper we have developed a distributed version of the parallel radix hash join using RDMA. However, the ideas described in this work, i.e. RDMA buffer pooling, reuse of RDMA buffers, and interleaving computation and communication are general techniques which can be used to create distributed versions of many database operators like sort-merge joins or aggregation.

In this work we treated the join operation as part of an operator pipeline in which the result of the join is materialized at a later point in the query execution. We are aware that distributed result materialization involves moving large amounts of data over the network and will therefore be an expensive operation. We leave studying the combination of join computation and result materialization to future work.

Our experimental evaluation focuses on running one join operator at a time. Scheduling concurrent database operators in a distributed setup remains an open research area. However, we are confident that recent work on query plan deployment for multi-core systems [14] can also be applied to rack-scale databases.

The experiments clearly show that distributed joins are at a similar level of performance than parallel join algorithms. In fact, our results indicate that modern multi-core hardware should be treated more and more as a distributed system as it has been suggested for operating systems [5].

Although it is not the goal of this paper to compare distributed to centralized algorithms, our findings suggest that the answer to the question whether join performance can be improved by scaling up or scaling out is dependent on the bandwidth provided by the NUMA interconnect and the network. For instance, faster CPU interconnects and a higher number of cores per processor favor vertical scale-up, whereas a higher inter-machine bandwidth would favor horizontal scale-out. In the experimental evaluation we could show that our implementation of a distributed join exhibits good performance, despite the network being a major bottleneck. Current technical road-maps project that InfiniBand will be able to offer a bandwidth of 25 GB/s (HDR) by 2017 [17], which suggests that the network bottleneck will be reduced, which would increase the performance of the proposed algorithm.

8. CONCLUSIONS

In this paper we presented a distributed hash join algorithm which makes use of RDMA as a light-weight communication mechanism. We described how RDMA-enabled buffers can be used to partition and distribute the data efficiently. We were able to show that that the scalability of the distributed join algorithm is highly dependent on the right combination of processing power and network bandwidth. Although the algorithm in its current form is susceptible to data skew, we believe that this can be addressed by introducing inter-machine workload sharing.

In addition to the prototype implementation, we presented an analytical model of the algorithm and were able to show that it can be used to predict the performance of the algorithm with very high accuracy. We performed an experimental evaluation of the algorithm on multiple hardware platforms using two different low-latency networks.

To the best of our knowledge this is the first paper to combine a detailed analysis, analytical model, and experimental evaluation of a distributed join using RDMA.

9. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work has been funded in part by a grant from Oracle Labs. The authors would like to thank Oracle Labs, in particular Cagri Balkesen, Vikas Aggarwal and Michael Duller.

10. REFERENCES

[1] F. N. Afrati and J. D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT'10*, pages 99–110, 2010.

[2] M. Albutiu, A. Kemper, and T. Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB*, 5(10):1064–1075, 2012.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB*, 7(1):85–96, 2013.

[4] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *ICDE'13*, pages 362–373, 2013.

[5] A. Baumann, P. Barham, P. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS Architecture for Scalable Multicore Systems. In *SOSP'09*, pages 29–44, 2009.

[6] S. Blanas, Y. Li, and J. M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *SIGMOD'11*, pages 37–48, 2011.

[7] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB'86*, pages 228–237, 1986.

[8] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[9] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *NSDI'14*, pages 401–414, 2014.

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

[11] P. W. Frey and G. Alonso. Minimizing the Hidden Cost of RDMA. In *ICDCS'09*, pages 553–560, 2009.

[12] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. Spinning Relations: High-Speed Networks for Distributed Join Processing. In *DaMoN'09*, pages 27–33, 2009.

[13] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. A Spinning Join That Does Not Get Dizzy. In *ICDCS'10*, pages 283–292, 2010.

[14] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of Query Plans on Multicores. *PVLDB*, 8(3):233–244, 2014.

[15] R. Goncalves. *The Data Cyclotron - Juggling Data and Queries for a Data Warehouse Audience*. PhD thesis, CWI, 2013.

[16] R. Goncalves and M. L. Kersten. The Data Cyclotron query processing scheme. In *EDBT'10*, pages 75–86, 2010.

[17] InfiniBand Trade Association. InfiniBand Architecture Specification. http://www.infinibandta.org/. Nov. 2014.

[18] International Business Machines Corporation. IBM Netezza. http://www.netezza.com/. Nov. 2014.

[19] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[20] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Data Base Machine and Its Architecture. *New Generation Comput.*, 1(1):63–74, 1983.

[21] H. Lang, V. Leis, M. Albutiu, T. Neumann, and A. Kemper. Massively Parallel NUMA-Aware Hash Joins. In *IMDM'13*, pages 3–14, 2013.

[22] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. D. Gropp, and B. R. Toonen. Design and implementation of MPICH2 over infiniband with RDMA support. In *IPDPS'04*, 2004.

[23] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[24] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee on Computer Architecture Newsletter*, pages 19–25, 1995.

[25] A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. In *SIGMOD'11*, pages 949–960, 2011.

[26] Oracle Corporation. Oracle Exadata. http://www.oracle.com/us/products/database/exadata-database-machine/. Nov. 2014.

[27] O. Polychroniou, R. Sen, and K. A. Ross. Track Join: Distributed Joins with Minimal Network Traffic. In *SIGMOD'14*, pages 1483–1494, 2014.

[28] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-Sensitive Operators for Parallel Main-Memory Database Clusters. In *ICDE'14*, pages 592–603, 2014.

[29] SAP AG. SAP HANA. http://www.saphana.com/. Nov. 2014.

[30] D. A. Schneider and D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *SIGMOD'89*, pages 110–121, 1989.

[31] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *VLDB'94*, pages 510–521, 1994.