Distributed Join Algorithms on Thousands of Cores

Claude Barthels, Ingo Müller[‡], Timo Schneider, Gustavo Alonso, Torsten Hoefler
Systems Group, Department of Computer Science, ETH Zurich
{firstname.lastname}@inf.ethz.ch

ABSTRACT

Traditional database operators such as joins are relevant not only in the context of database engines but also as a building block in many computational and machine learning algorithms. With the advent of big data, there is an increasing demand for efficient join algorithms that can scale with the input data size and the available hardware resources.

In this paper, we explore the implementation of distributed join algorithms in systems with several thousand cores connected by a low-latency network as used in high performance computing systems or data centers. We compare radix hash join to sort-merge join algorithms and discuss their implementation at this scale. In the paper, we explain how to use MPI to implement joins, show the impact and advantages of RDMA, discuss the importance of network scheduling, and study the relative performance of sorting vs. hashing. The experimental results show that the algorithms we present scale well with the number of cores, reaching a throughput of 48.7 billion input tuples per second on 4,096 cores.

1. INTRODUCTION

The ability to efficiently query large sets of data is crucial for a variety of applications, including traditional data warehouse workloads and modern machine learning applications [28]. Most of these workloads involve complex large-to-large join operations and, thus, modern data processing systems would benefit from having efficient distributed join algorithms that can operate at massive scale.

Recent work on hash and sort-merge join algorithms for multi-core machines [1, 3, 5, 9, 27] and rack-scale data processing systems [6, 33] has shown that carefully tuned distributed join implementations exhibit good performance. These algorithms have been designed for and evaluated on rack-scale systems with hundreds of CPU cores and limited inter-node network bandwidth.

‡ Work conducted while employed at Oracle Labs Zurich.

Proceedings of the VLDB Endowment, Vol. 10, No. 5
Copyright 2017 VLDB Endowment 2150-8097/17/01.

This paper addresses the challenges of running state-of-the-art, distributed radix hash and sort-merge join algorithms at scales usually reserved to massively parallel scientific applications or large map-reduce batch jobs. In the experimental evaluation, we provide a performance analysis of the distributed joins running on 4,096 processor cores with up to 4.8 terabytes of input data. We explore how join algorithms behave when high-bandwidth, low-latency networks are used and specialized communication libraries replace hand-tuned code. These two points are crucial to understand the evolution of distributed joins and to facilitate the portability of the implementation to future systems.

Operating at large scale requires careful process orchestration and efficient communication. This poses several challenges when scaling out join algorithms. For example, a join operator needs to keep track of data movement between the compute nodes in order to ensure that every tuple is transmitted to the correct destination node for processing. At large scale, the performance of the algorithm is dependent on having a good communication infrastructure that automatically selects the most appropriate method of communication between two processes.

We implemented both algorithms on top of MPI [31], a standard library interface used in high-performance computing applications and evaluated the join implementations on two large-scale systems with a high number of cores connected through a state-of-the-art low-latency network fabric. The algorithms are hardware-conscious, make use of vector instructions to speed up the processing, access remote data through fast one-sided memory operations, and use remote direct memory access (RDMA) to speed up the data transfer. For both algorithms, we provide a performance model and a detailed discussion of the implementation.

Important insights from the paper include: (i) Achieving maximum performance requires having the right balance of compute and communication capacity. Adding more cores to a compute node does not always improve, but can also worsen performance. (ii) Although both join algorithms scale well to thousands of cores, communication inefficiencies have a significant impact on performance, indicating that more work is needed to efficiently use the full capacity of large systems. (iii) Hash and sort-merge join algorithms have different communication patterns that incur different communication costs, making the scheduling of the communication between the compute nodes a crucial component. (iv) Our performance models indicate that the sort-merge join implementation achieves its maximum performance. The radix hash join on the other hand is far from its theoretical maximum, but still outperforms the sort-merge join, confirming previous studies comparing hash- and sort-based join algorithms [3].

2. BACKGROUND

This section provides the necessary background on remote direct memory access (RDMA), one-sided remote memory operations, and the technologies used in high-performance computing (HPC) systems.

2.1 RDMA & RMA

Remote Direct Memory Access (RDMA) enables direct access to the main memory of a remote host. The primary benefit of RDMA is a reduction in CPU load because the data does not need to be copied into intermediate buffers in the network stack. The operating system is not on the performance-critical path and the CPU remains available for processing while a network operation is taking place in parallel. Therefore, RDMA enables the interleaving of communication and computation, thereby hiding all or parts of the network latency.

RDMA provides two communication mechanisms: two-sided (send & receive) and one-sided (read & write) operations. Both mechanisms differ in terms of semantics and their need for inter-process synchronization.

Two-sided operations represent traditional channel semantics, where both the sender and the receiver need to be active in order to complete the transfer. Because of this, the receiver is often referred to as an *active target*. The receiver posts several RDMA-enabled buffers into which the data will be written by the network card. The sender of a message is not aware of the exact locations of the receive buffers.

One-sided operations represent the *remote memory access* (RMA) semantics. The initiator of a request can directly access parts of the remote memory and has full control where the data will be placed. Read and write operations are executed without any involvement of the target machine, therefore it is often called a *passive target*.

Modern low-latency networks such as InfiniBand [23] rely on direct access to memory in order to achieve high bandwidth utilization and low latency. However, in most implementations, memory has to be registered with the network card before it is accessible for RDMA transfers [17]. During the registration process, the memory is pinned such that it cannot be swapped out, and the necessary memory translation information are stored accessible by the network card. Memory that can be accessed through RDMA operations is referred to as a *memory region*. The application initializing an operation has to know the necessary access information before it can read from or write to a memory region.

2.2 Message-Passing Interface

The Message-Passing Interface (MPI) is a widely used interface in high-performance computing systems. In the following section, we provide an introduction to MPI and a detailed description of the one-sided MPI operations used in our join implementations.

2.2.1 Overview

MPI is a portable standard library interface for writing parallel programs in high-performance computing (HPC) applications [20]. Its operations have been defined from the ground up to support parallel large-scale systems, but the interface is used on a variety of computing infrastructure: from small clusters to high-end supercomputers. The latter often ship with a highly optimized MPI implementation.

MPI is widely used in high-performance applications because it provides a rich hardware-independent interface, thus making the application code portable, while at the same time leveraging the performance of the underlying hardware by binding to different implementations optimized for the target platform. The developer is shielded from the complexity of the distributed environment and is – to a large extend – unaware of the physical location of the different processes that make up his program. It is the responsibility of the library to select the most appropriate communication method for each pair of processes.

An example of an efficient RMA library is Fast One-sided MPI (foMPI) [19], a scalable MPI implementation optimized for the Cray XC30 and XC40 systems [11]. The library interfaces with multiple low-level interfaces and selects the most suitable communication mechanism based on the relative distance of two processes. For inter-node communication, it uses the Distributed Memory Application API (DMAPP), while for intra-node communication, it interfaces with XPMEM, a kernel module that allows to map the memory of one process into the virtual address space of another.

2.2.2 MPI One-Sided Operations

The memory that is accessible by other processes through RMA operations is referred to as a *memory window*. To create a window, MPI provides an `MPI_Win_create` operation that makes a contiguous section of main memory available to RMA operations. This call is a *collective call*, which means that it has to be executed by every process that wants to perform RMA operations, even if the process does not register memory itself.

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

To allow for applications with more complex communication patterns, MPI provides *passive target synchronization* mechanisms through the `MPI_Win_lock` and `MPI_Win_unlock` operations. Before an RMA operation on a specific window can be executed, it needs to be locked. The lock provides either exclusive (`MPI_LOCK_EXCLUSIVE`) or concurrent (`MPI_LOCK_SHARED`) access. When releasing a lock, the library ensures that all pending RMA operations have completed both at the origin and at the target before the call returns. To amortize the costs of synchronization, the user should initiate multiple data transfers per epoch.

To read from and write to a remote window, MPI offers the `MPI_Get` and `MPI_Put` calls respectively. When using passive synchronization, `MPI_Win_flush` is used to ensure that all outstanding RMA operations initiated by the calling process have been executed without the need to release the lock. After the flush call, the buffers provided to previous `MPI_Put` and `MPI_Get` operations can be reused or read [21].

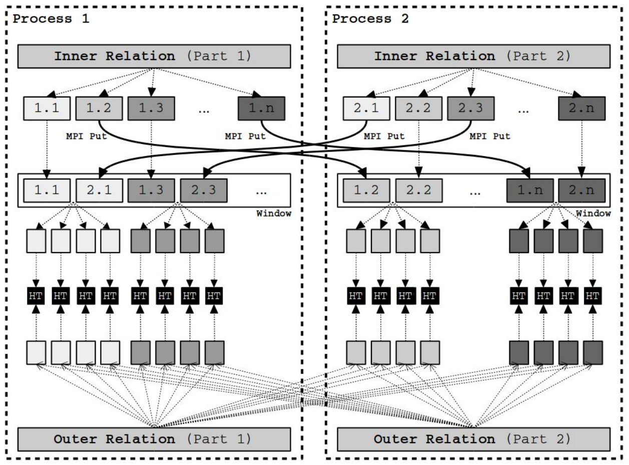


Figure 1: Distributed radix hash join with two processes.

3. JOIN ALGORITHMS

In this section, we provide a high-level overview and a performance model of the distributed radix hash join and the distributed sort-merge join.

3.1 Radix Hash Join

In the radix hash join, the input data is first partitioned into cache-sized partitions. Next, hash tables are build for each partition of the inner relation, and probed with the data from the corresponding partition of the outer relation. Figure 1 illustrates the execution of the radix hash join with two processes.

3.1.1 Histogram and Assignment Computation

The distributed radix hash join computes a global histogram in order to determine the size of the communication buffers and memory windows. The time required to compute the histograms T_{hist} depends on the size of both input relations (R and S) and the rate P_{scan} at which each of the p processes can scan over the data.

$$T_{\text{hist}} = \frac{|R| + |S|}{p \cdot P_{\text{scan}}} \tag{1}$$

Using this histogram, the algorithm can determine (i) an assignment of partitions to nodes and (ii) a set of offsets within the memory windows into which each process can write exclusively, thus reducing the amount of synchronization required during the join operation.

3.1.2 Multi-Pass Partitioning

The goal of the partitioning phase is to create small cache-sized partitions of the inner relation. This significantly speeds up the build and probe phases [30]. In order to create a large number of small partitions, a large partitioning fan-out is required. However, random access to a large number of locations leads to a significant increase in TLB misses if the number of partitions is larger than the TLB size and to cache trashing if the number of partitions is larger than the number of available cache lines. A multi-pass partitioning strategy has been adopted to address this issue [30]. In each pass, the partitions of the previous pass are refined such that the partitioning fan-out F_P in each step does not exceed the number of TLB and cache entries. This is achieved by using a different hash function in each pass. The number of passes d depends on the size of the inner relation R.

$$d = \lceil \log_{F_P} (|R| / \text{cache size}) \rceil \tag{2}$$

Previous work on distributed radix joins has shown that the partitioning phase can be interleaved with the data redistribution over the network [6]. A relation is partitioned into (i) a local buffer, if the partition will be processed locally, or (ii) into an RDMA buffer, if it has been assigned to a remote node. Remote write operations are executed at regular intervals in order to interleave the computation and communication. The partitioning rate P_{net} of each process is either limited by the partitioning speed of the process P_{part} (compute-bound) or by the available network bandwidth BW_{node}, which is shared among all t processes on the same node (bandwidth-bound).

$$P_{\text{net}} = \min\left(P_{\text{part}}, \frac{BW_{\text{node}}}{t}\right) \tag{3}$$

Subsequent partitioning passes are executed locally without any network transfer. Each process can therefore partition the data at the partitioning rate P_{part}. The total time required to partition the data is equal to the time required to iterate over the data d times.

$$T_{\text{part}} = \left(\frac{1}{p \cdot P_{\text{net}}} + \frac{d-1}{p \cdot P_{\text{part}}}\right) \cdot (|R| + |S|) \tag{4}$$

At the end of this phase, the data has been partitioned and distributed among all the processes. Matching tuples have been assigned to the same partition.

3.1.3 Build & Probe

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

$$T_{\text{build}} = (F_P)^d \cdot \frac{|R_p|}{p \cdot P_{\text{build}}} = \frac{|R|}{p \cdot P_{\text{build}}} \tag{5}$$

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

$$T_{\text{probe}} = (F_P)^d \cdot \frac{|S_p|}{p \cdot P_{\text{probe}}} = \frac{|S|}{p \cdot P_{\text{probe}}} \tag{6}$$

Equation 6 does not include the time required to materialize the output of the join. The cost of fetching additional payload data over the network depends on the selectivity of the join and the size of the payload fields.

3.1.4 Performance Model

The hash join executes the partitioning, build, and probe phases sequentially. Assuming no interference between the phases, we can determine a lower bound for the execution time of the radix hash join algorithm.

$$T_{\text{rdx}} = T_{\text{hist}} + T_{\text{part}} + T_{\text{build}} + T_{\text{probe}} \tag{7}$$

A comparison between the predicted and measured execution time is provided in Section 5.5.

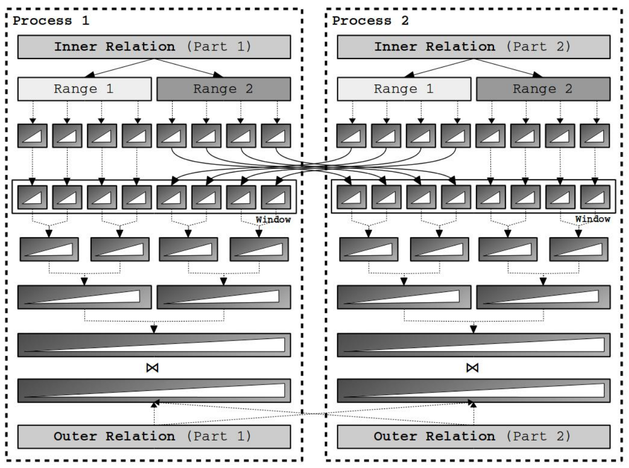


Figure 2: Distributed sort-merge join with two processes.

3.2 Sort-Merge Join

A second variant of join algorithms is based on sorting both input relations. In the sorting phase, we make use of a highly parallel sort-merge algorithm, which allows to interleave the sorting and the data transfer. Figure 2 illustrates the execution of the sort-merge join for two processes.

3.2.1 Sorting

In the first phase of the sorting operation, each thread partitions the input data. We use range-partitioning to ensure that matching elements in both relations will be assigned to the same machine for processing. Because we use a continuous key space, we can split the input relations into ranges of identical size. The time required to range-partition the data depends on the size of the input relations, the number of processes p, and the partitioning rate P_{part}.

$$T_{\text{part}} = \frac{|R| + |S|}{p \cdot P_{\text{part}}} \tag{8}$$

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

$$N_R = \frac{|R|}{l} \quad \text{and} \quad N_S = \frac{|S|}{l} \tag{9}$$

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

$$P_{\text{sort}} = \min\left(P_{\text{run}}(l), \frac{BW_{\text{node}}}{t}\right) \tag{10}$$

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

$$T_{\text{sort}} = (N_R + N_S) \cdot \frac{l}{p \cdot P_{\text{sort}}} = \frac{|R| + |S|}{p \cdot P_{\text{sort}}} \tag{11}$$

After a process has sorted its input data, it waits until it has received all the sorted runs of its range from the other nodes. Once all the data has been received, the algorithm starts merging the sorted runs using *m-way* merging, which combines multiple input runs into one sorted output. Several iterations over the data might be required until both relations are fully sorted. The number of iterations $d_{\{R,S\}}$ needed to merge the data depends on the number of runs $N_{\{R,S\}}$ and the merge fan-in F_M.

$$d_R = \lceil \log_{F_M}(N_R/p) \rceil \quad \text{and} \quad d_S = \lceil \log_{F_M}(N_S/p) \rceil \tag{12}$$

From the depth of both merge trees, we can determine the time required to merge the runs of both relations.

$$T_{\text{merge}} = d_R \cdot \frac{|R|}{p \cdot P_{merge}} + d_S \cdot \frac{|S|}{p \cdot P_{merge}} \tag{13}$$

After the sorting phase, both relations are partitioned among all the nodes. Within each partition the elements are fully sorted.

3.2.2 Joining Sorted Relations

To compute the join result, each process merges the sorted partition of the inner relation with the corresponding partition of the outer relation.

$$T_{\text{match}} = \frac{|R| + |S|}{p \cdot P_{\text{scan}}} \tag{14}$$

The tuples in the output of the join are also sorted. The costs associated with materializing the output of the sort-merge join depends on the selectivity of the input and the payload size that needs to be accessed over the network.

3.2.3 Performance Model

The hash join executes the partitioning, sorting, merging, and matching phases sequentially. Assuming no interference between the phases, we can determine a lower bound for the execution time of the sort-merge join algorithm.

$$T_{\text{sm}} = T_{\text{part}} + T_{\text{sort}} + T_{\text{merge}} + T_{\text{match}} \tag{15}$$

The predicted and measured execution times of the sort-merge join are discussed in the experimental evaluation in Section 5.5.

4. IMPLEMENTATION

We have implemented the distributed hash and sort-merge joins on top of foMPI [19] with 7,000 lines of C++ code[1].

4.1 Radix Hash Join

The implementation of the radix hash join is hardware-conscious and uses a cache-conscious partitioning algorithm.

4.1.1 Histogram & Assignment Computation

In the beginning of the algorithm, each process scans its part of the input data and computes two process-level histograms, one for each input relation. These local histograms are combined into a global histogram through an `MPI_Allreduce` call. We use the `MPI_SUM` operator as an argument to the call. This operation combines the values from all processes – in our case it computes the sum – and distributes the result back, such that each process receives a copy of the global histogram.

[1]`http://www.systems.ethz.ch/projects/paralleljoins`

The join supports arbitrary partition-process assignments. In our implementation, we use a round-robin scheme to assign partitions to processes.

To compute the window size, each process masks the assignment vector with its process number such that the entries of the assigned partitions are one, and zero otherwise. This mask is applied to the global histogram. The sum of all remaining entries is equal to the required window size.

Computing the private offsets for each process and each partition is performed in three steps. First, the base offsets of each partition are computed. The base offsets are the starting offsets of each partition in relation to the starting address of the window. Next, the relative offsets within a partition need to be computed from the local histograms using a prefix sum computation. To perform this prefix computation across all processes, MPI provides an `MPI_Scan` functionality. This function returns for the i-th process the reduction (calculated according to a user-defined function) of the input values of processes 0 to i. In our case, the prefix sum is implemented by combining the `MPI_Scan` function with the `MPI_SUM` operator. Third, the private offsets of a process within a window can be determined by adding the starting offset of a partition and the relative private offset.

At the end of this computation, each process is aware of (i) the assignment of partitions to processes, (ii) the amount of incoming data, and (iii) the exact location to which the process has exclusive access when partitioning its input.

4.1.2 Network Partitioning

From the computation described in Section 4.1.1, we know the exact window size of each process for both input relations. Two windows are allocated: one for the inner and one for the outer relation. Because `MPI_Win_create` is a collective routine, this phase requires global synchronization.

After the window allocation phase, each process acquires an `MPI_LOCK_SHARED` lock on all the windows. We allow concurrent access because the histogram computation provides us with the necessary information to determine ranges of exclusive access for each partition and process.

Next, each process allocates a set of communication buffers for each partition into which the process will partition the data. These buffers are of fixed size (64 kilobytes).

After the setup phase, the algorithm starts with the actual partitioning and redistribution of the input. Data is partitioned using AVX instructions into the local output buffers. When an output buffer is full, the process will issue an `MPI_Put` into its private offset in the target window. Interleaving computation and communication is essential to reach good performance. Therefore, we allocate multiple (at least two) output buffers for each remote partition. When all the buffers of a specific partition have been used once, the process needs to ensure that is can safely reuse them. This is achieved by executing an `MPI_Win_flush`.

During the partitioning, the 16-byte ⟨key, record id⟩ tuples are compressed into 64-bit values using prefix compression. Radix partitioning groups keys with $\log(F_P)$ (fan-out) identical bits. The partitioning bits can be removed from the key once the tuple has been assigned to a partition. If an input relation contains less than 274 billion tuples (4 TB per relation), the key and the record id can be represented with 38 bits each. On 4,096 cores, the minimum fan-out is 2^{12}. Hence, a tuple can be compressed into $2 \cdot 38 - 12 = 64$ bits. This reduces the total amount of data by a factor of 2.

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

4.1.3 Local Processing

The local processing is composed of (i) subsequent partitioning passes and (ii) the build-probe phase.

If the partitions from the first partitioning pass are larger than the cache size, additional partitioning passes are required. For these passes we use a similar AVX partitioning code as in the first partitioning pass with the following modifications: (i) the output buffers are no longer of fixed size but are sized such that they can hold all the data and (ii) no MPI calls are being generated.

For the build-probe phase, we use the hash table implementation of Balkesen et al. [4], which consists of a contiguous array of buckets and enables cache-friendly access.

4.2 Sort-Merge Join

The sort-merge join is hardware-conscious and uses vector instructions in its sorting phase. To be able to compare our results with previous work on sort-merge join algorithms [3] and as it is done in most modern column stores [15], we use compression to optimize the performance of the join.

4.2.1 Sorting

During the partitioning operation, every process tracks how many elements are assigned to each of the p partition, thus creating a histogram. The same prefix compression is used as in the hash join. The compression is order-preserving and the resulting elements are 64 bit wide.

To compute the window size, a process must know how much data has been assigned to it. The histogram from the partitioning phase, together with the `MPI_SUM` operator, is given as an input to the `MPI_Reduce_scatter_block` call. This call performs an element-wise reduction (in this case it computes a sum) of all the histograms and scatters the result to the nodes. This means that node i will receive the sum of the i-th element of the histograms. The result of the reduction is equal to the required window size. This value is passed as an argument to the `MPI_Win_create` call.

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

Afterwards, each thread creates runs of fixed size (128 kilobytes), which are sorted locally. For sorting, we use an in-cache sorting algorithm with AVX instructions [3]. When a run has been sorted, it is immediately transmitted to the target window by executing an `MPI_Put` operation. To avoid contention on the receiving node, not every process starts sorting the first partition. Instead, process i starts processing partition $i + 1$. Each process writes to its private range within the window. Individual runs are appended one after the other. Because the amount of data in a partition is not necessarily a multiple of the run size, the last run might contain fewer elements.

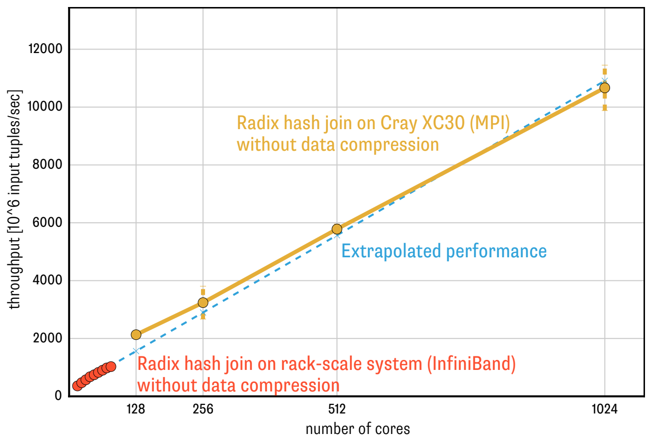


Figure 3: Comparison between the performance of the radix hash join on a rack-scale system and the Cray XC30 system. Error bars represent 95% confidence intervals.

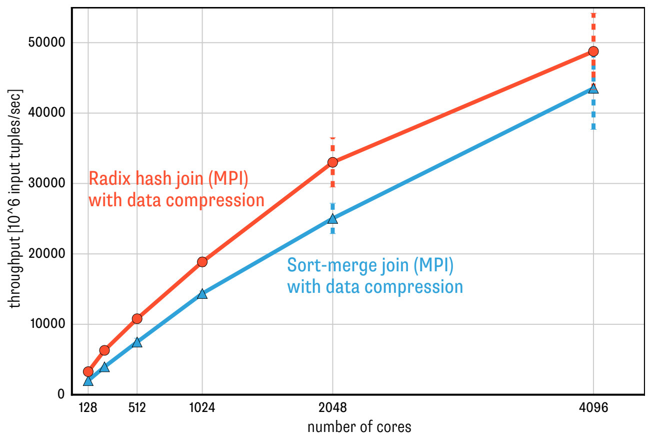


Figure 4: Scale-out experiment of the radix hash join and sort-merge join algorithm on the Cray XC30 system. Error bars represent 95% confidence intervals.

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

Next, the algorithm merges the sorted runs into one single relation. Multiple runs are merged simultaneously using an in-cache merge tree. The merge process is accelerated through the use of AVX instructions [3].

4.2.2 *Joining Sorted Relations*

After the distributed sort-merge operation, the relations are partitioned into p ranges (where p is the number of processes) and all elements within a range have been sorted. Range-partitioning ensures that matching elements from both relations have been assigned to the same process.

At this stage, every process can start joining its part of the data. No further communication or synchronization between processes is necessary. Scanning the relations is a linear operation through both relations, and modern hardware allows for very fast sequential access.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate our algorithms experimentally on two Cray supercomputers, which provide us with a highly-tuned distributed setup and a large number of cores.

5.1 Experimental Setup

In order to make our results comparable to previous work on join algorithms, we use the same workloads as the authors of [3, 5, 6, 9, 27]. The experiments focus on large-to-large joins with highly distinct key values. The data is composed of narrow 16-byte tuples, containing an 8-byte key and an 8-byte record id (RID). The record identifiers are range partitioned among the compute nodes. The key values can occur in arbitrary order. Each core is assigned to the same amount of input data. In our experiments, one process serves up to 40 million tuples per relation, which results in 4.8 TB of input data on 4,096 cores. The relative size of the inner and outer relation ranges between 1-to-1 and 1-to-8. The impact of different selectivities is also studied in this section.

To create the input relations, each node is assigned to generate a range of keys. The nodes ensure that the generated keys are in random order. Next, the input is divided into chunks. These chunks are exchanged between each pair of processes. After the shuffling operation, every node is in possession of keys from the entire value range. A final pass over the data puts the elements into a random order.

5.1.1 *Cray Supercomputers*

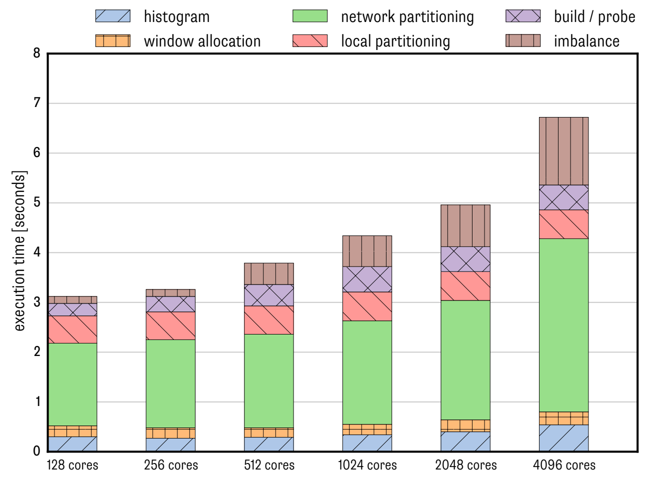
The Cray XC30 [11] used in the experimental evaluation has 28 compute cabinets implementing a hierarchical architecture: each *cabinet* can be fitted with up to three *chassis*. A chassis can hold up to sixteen compute *blades*, which in turn are composed of four compute *nodes*. The overall system can offer up to 5,272 usable compute nodes [12].

Compute nodes are single-socket 8-core (Intel Xeon E5-2670) machines with 32 GB of main memory. They are connected through an Aries routing and communications ASIC, and a Dragonfly network topology with a peak network bisection bandwidth of 33 TB/s. The Aries ASIC is a system-on-a-chip device comprising four NICs and an Aries router. The NICs provide network connectivity to all four nodes of the same blade. Each NIC is connected to the compute node by a 16x PCI Express 3 interface. The router is connected to the chassis back plane and through it to the network fabric.

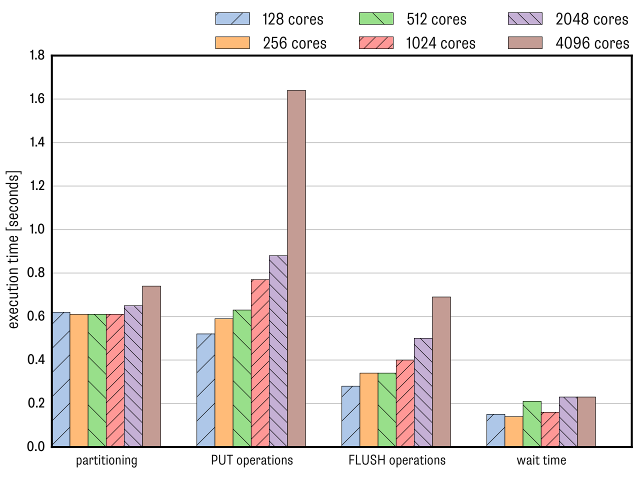
The second machine used for the experiments is a Cray XC40 machine. It has the same architecture as the XC30 but differs in the node design: each compute node has two 18-core (Intel Xeon E5-2695 v4) processors and 64 GB of main memory per node.

Comparison with commodity hardware: Despite the large amount of compute power packaged as one large installation, the individual compute nodes used in Cray supercomputers resemble commodity hardware, i.e., x86 processors, a network card, and 32 GB of main memory.

The Cray Aries network provides higher throughput and lower latency than commodity hardware. However, we ex-



(a) Total join execution



(b) Network partitioning pass

Figure 5: Breakdown of the execution time of the radix hash join for 40 million tuples/relation/core.

pect that high-speed networks such as silicon photonics or InfiniBand 4xEDR [23], which provides 100 Gb/s bandwidth and a latency in the single digit microsecond range, will close this gap in the near future, making it possible to obtain similar results to ours on a conventional cluster.

5.2 Baseline Experiments

Previous work on distributed radix hash joins has shown that the hash join can achieve good performance on RDMA-enabled high-speed networks [6]. The experiments have been conducted on two InfiniBand networks (QDR and FDR) with up to 10 machines and 8 threads per machine. Because the nodes of the supercomputer are also composed of multi-core Intel Xeon CPUs and are connected through low-latency network, we use the performance results gathered on rack-scale systems with InfiniBand as a baseline.

We extrapolate the performance of the radix hash join algorithm on a larger number of cores using linear regression and compare this estimate with the measured performance of the join on the Cray XC30 system. Both algorithms share a common code base. This version of the radix hash join does not use compression to reduce the amount of data that needs to be transmitted over the network.

The comparison between the estimated and measured performance is shown in Figure 3. We can observe that the measured performance follows the extrapolated line very closely. From this experiment we can conclude that the algorithm proposed in this paper achieves similar performance than the baseline. Furthermore, we can observe that the algorithm maintains this performance when scaling to a thousand processor cores.

5.3 Scale-Out Experiments

Figure 4 shows the overall throughput of the radix hash join and the sort-merge join using data compression. We assign 40 million tuples to each relation and core. Every tuple of the inner relation matches with exactly one element of the outer relation.

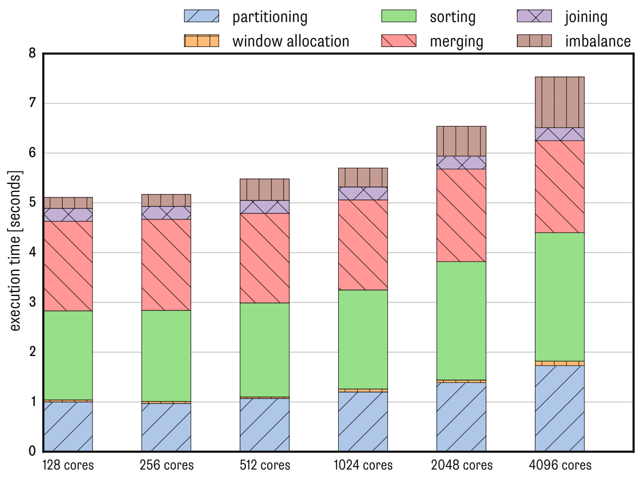
The results show that both algorithms are able to increase the throughput as more cores are added to the system. The radix hash join can process 48.7 billion tuples per second. The sort-merge join reaches a maximum throughput of 43.5 billion tuples per second on 4,096 cores. The scale-out behaviour of both algorithms is sub-linear. On systems with 4,096 cores, hashing outperforms the sort-merge approach by 12%, which is in line with previous work on hash and sort-merge joins [3].

A comparison with the baseline experiments highlights the importance of data compression. During partitioning, the 16-byte tuples are compressed into 8-byte values, which results in a significant performance increase as less data needs to be transmitted over the network. This result emphasizes that the primary cost of joins is data movement.

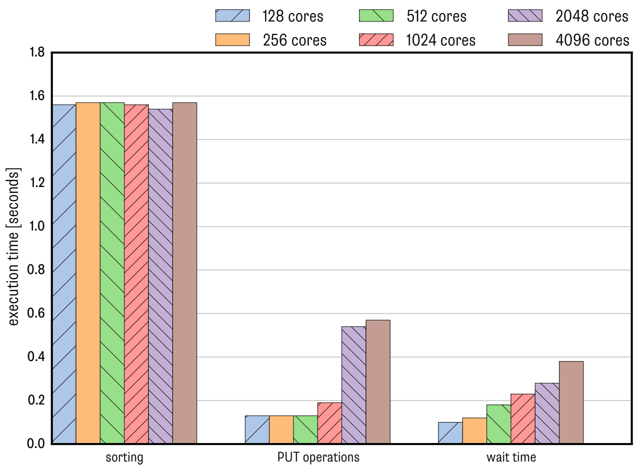
5.3.1 Radix Hash Join Analysis

Figure 5a shows the execution time of the different phases of the radix hash join and illustrates the effects of scale-out in more detail. We break down the execution of the join as follows: (i) the histogram computation, which involves computing the local histogram, the exchange of the histograms over the network, and the computation of the partition offsets, (ii) the time required to allocate the RMA windows, (iii) the network partitioning phase, which includes the partitioning of the data, the asynchronous transfer to the target process, and the flushing of transmission buffers, (iv) the local partitioning pass, which ensures that the partitions fit into the processor cache, and (v) the build and probe phase, in which a hash table is created over each partition of the inner relation and probed using the data from the corresponding partition of the outer relation. All times are averaged over all the processes. Because we consider the join only to be finished when the last process terminates, we report the difference between the maximum execution time and the sum of the averaged execution times. This value gives an indication of how evenly the computation has been balanced across all processes.

Given that we scale out the system resources and the input size simultaneously, one would expect constant execution time of all phases. However, we observe an increase in execution time as we add more cores, which explains the sub-linear increase in throughput shown in Figure 4.



(a) Total join execution



(b) Sorting phase

Figure 6: Breakdown of the execution time of the sort-merge join for 40 million tuples/relation/core.

We observe that the execution time of the histogram computation and the window allocation phase remains constant.

The network partitioning phase on the other hand increases significantly. Figure 5b shows a detailed breakdown of this phase. One can observe that the time required to partition the data remains constant up to 1,024 cores. Starting from 1,024 cores, the partitioning fan-out has to be increased beyond its optimal setting, which incurs a minor performance penalty. Most of the additional time is spent in the `MPI_Put` and `MPI_Flush` operations which generate the requests to transmit the data, respectively ensure that the data transfers have completed. This increase is caused by the additional overhead of managing a larger amount of buffers and the lack of any network scheduling. Further details on the costs of communication at large scale are provided in Section 5.3.3.

The local partitioning phase exhibits constant execution time because the per-core amount of data is kept constant throughout the experiment. The build-probe operation on the other hand shows a minor increase in execution time because the generated partitions get larger as we add more cores and process more data overall.

For the compute imbalance, i.e., the time difference between the average and maximum execution time, we observe a clear increase as we add cores to the system. This is expected as the supercomputer is shared by multiple organizations and complete performance isolation cannot be guaranteed for large deployments. Furthermore, the nodes involved in a large experiment cannot always be physically colocated, resulting in a higher remote memory access latency for some nodes. We observe that the performance of the hash join is influenced by a small number of stragglers.

5.3.2 *Sort-Merge Join Analysis*

Figure 6a shows the breakdown of the execution time of the sort-merge join. The individual phases are (i) the range partitioning phase, which includes the histogram and offset computation, (ii) the window allocation time, (iii) the time needed to sort and transmit the tuples, (iv) the time required to merge the sorted runs, and (v) the time required to join both relations. Similar to the hash join, the execution times shown in Figure 6a are averaged over all processes and the difference between the average and total execution time is reported as the compute imbalance.

For the sort-merge join, we can observe an increase in the time required to partition and sort the data. For 2,048 and 4,096 cores, the partitioning fan-out has to be pushed beyond its optimal configuration, which leads to a small increase in execution time. The sort-merge join uses one single partitioning pass over the data. However, given that the increase is small, a second partitioning pass does not pay off at these scales.

In Figure 6b, we see that the sorting phase is dominated by the time required to sort the tuples. The `MPI_PUT` operation time remains constant up to 1,024 cores, followed by a sudden increase in its execution time. This effect can be explained by the fact that sorting is more compute intensive than hashing, which allows for better interleaving of computation and communication. Furthermore, the communication pattern of the sort-merge join is better suited for the underlying network hardware. A detailed discussion is provided in Section 5.3.3.

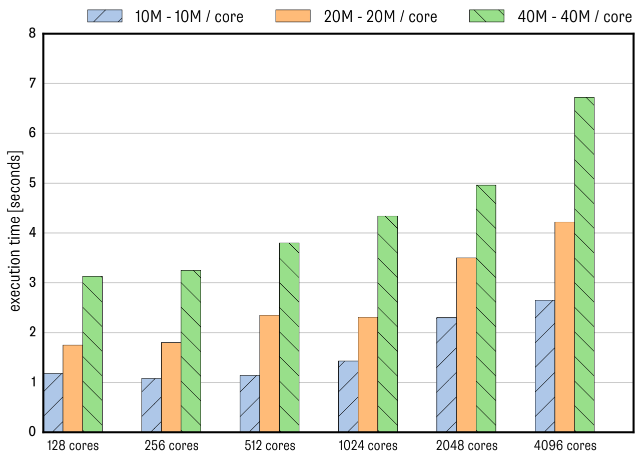
Given that the per-core data size remains constant, the time required to merge and match the data does not change.

5.3.3 *Network Communication Analysis*

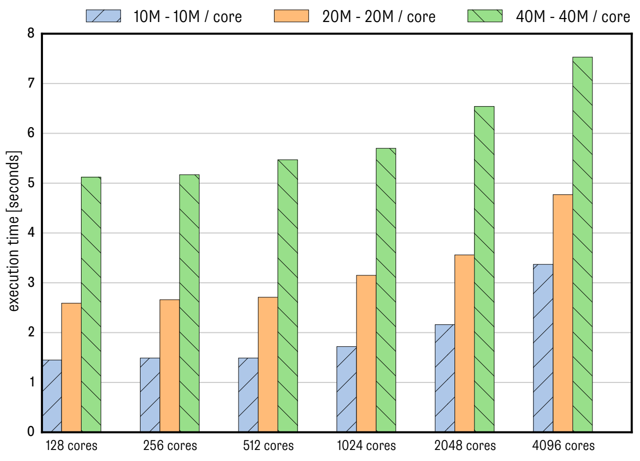
A key performance factor for both algorithms is the cost of communication. In Sections 5.3.1 and 5.3.2 we made the following observations: (i) The time required to execute all `MPI_Put` calls is significantly higher for the hash join than for the sort-merge join. (ii) The cost of enqueuing an `MPI_Put` request steadily increases for the hash join as the number of cores is increased. (iii) The `MPI_Put` cost remains constant for the sort-merge join up to 1,024 cores, followed by a sudden increase in execution time.

These observations can be explained by the fact that both algorithms have different communication patterns.

The hash join interleaves the partitioning and the network communication. To that end, it allocates a temporary buffer space into which data is written. Once a buffer is full, an `MPI_Put` request is generated and a new buffer is used to continue processing. Because the amount of buffer



(a) Radix hash join



(b) Sort-merge join

Figure 7: Execution time of the radix hash join and the sort-merge join algorithms for different input sizes.

space is the same for every partition and uniform data is used, the partition buffers will be scheduled for transmission at similar points in time, causing temporal hotspots on the network. This is aggravated by having more processes per machine. Because the hardware has a limited request queue, the processes will be blocked while trying to enqueue their request, causing a significant increase in the time spend in the `MPI_Put` call. This problem is further enhanced as the partitioning fan-out increases. During the network partitioning phase, every process communicates with every other process in the system simultaneously. Having more active communication channels incurs a significant overhead.

The sort-merge join partitions the data into individual ranges before it interleaves the sorting operation and the network transfer. A process only sorts one run at a time. After the run is sorted, it is immediately enqueued for transfer. Alternating between sorting and executing an `MPI_Put` calls creates an even transmission rate on the sender side. To avoid over-saturation at the receiver, each thread starts processing a different range, i.e. the i-th process starts sorting range $i + 1$. Since the data is distributed uniformly and the processes are synchronized at the start of the sorting phase, for small deployments, they remain synchronized throughout the phase. During any point in time, a process i is transmitting data to exactly one process j, which in turn receives data only from the i-th process. Without synchronization, this pair-wise communication pattern can only be maintained for small deployments. In large deployments, nodes cannot be guaranteed to be physically colocated and variable network latencies disrupt this pattern, causing the increase in `MPI_Put` costs for 2,048 and 4,096 cores.

5.3.4 Small and Large Relations

To study the effect of different input data sizes and the ratio of the inner and outer relation, we use several workloads: (i) A 1-to-1 input where each tuple of the inner relation matches with exactly one element in the outer relation. We use 10, 20, and 40 million tuples per relation and core; (ii) 1-to-N workloads, where each element in the inner relation finds exactly N matches in the outer relation.

In Figure 7a, we see the performance of the hash join for different input sizes. We observe that a reduction of the input size by half does not lead to a 2× reduction in execution time. The execution time of both partitioning passes as well as the build-probe phase is directly proportional to the input size. However, the histogram computation, window allocation, and the compute imbalance are not solely dependent on the input but have additional fixed costs.

For the sort-merge join (Figure 7b), the time for sorting, merging, and matching the tuples is reduced by half. Window allocation and compute imbalance are not directly affected by the input size, resulting in a sub-linear speed-up.

Table 1 (lines 1-3) shows the execution time of both algorithms on 1,024 cores for different relation sizes. One can observe that the execution time depends primarily on the input size and is therefore dominated by the larger relation.

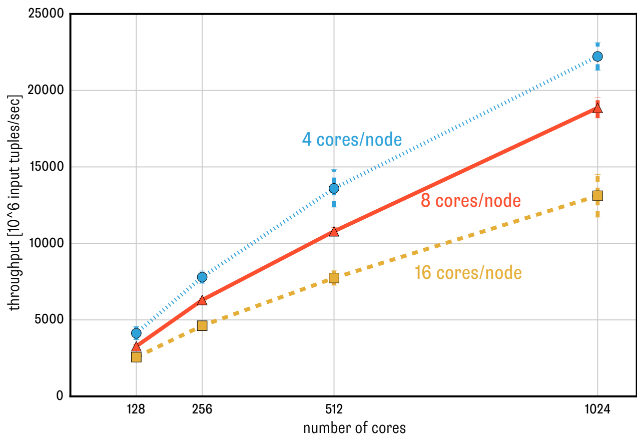
Table 1: Execution time for workloads with different relation sizes and selectivities for 1,024 processes.

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

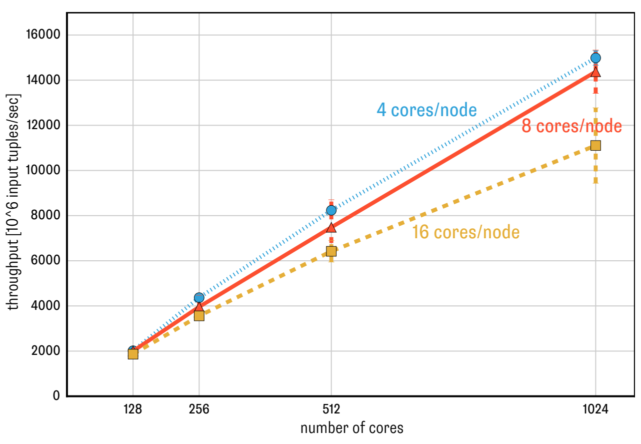
5.3.5 Input Selectivity

To study the impact of selectivity on the join algorithms, we use 1-to-4 workloads with 10 million and 40 million tuples per core. For each of the workloads, a different number of output tuples is produced. In Table 1 (lines 3-5), we show that the performance of the join remains constant for all three workloads. This is due to the fact that the execution time of the join is dominated by the costs of processing the input. The actual matching of the tuples accounts for a small percentage of the execution time.

As previous work [27, 9, 4, 5, 3], we investigate the join operation in isolation and do not materialize the output, i.e., we do not fetch additional data over the network after the join result has been computed.



(a) Radix hash join



(b) Sort-merge join

Figure 8: Scale-out experiments with different number of cores per compute node for 40 million tuples/relation/core.

5.4 Scale-Up Experiments

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

For both algorithms, we observe that the configuration with 4 cores per machine yields the highest throughput. As seen in Figure 8, the radix hash join benefits from the reduced interference as it is more memory intensive in its partitioning phase than the sorting operation of sort-merge join.

The performance of both algorithms suffers when increasing the number of processes to 16 cores per node. We measured that considerably more time is spent executing the `MPI_Put` and `MPI_Flush` operations. More processes per machine put more load on each individual network card, which makes it difficult to fully interleave computation and communication. In general, the more processes share the same network card, the more state the network card needs to hold (connections, memory translations, etc.). This is an important observation because this phenomenon is difficult to observe in conventional clusters.

We conclude that the performance of both joins is directly related to the performance of the network and the number of processes that share the same network card.

5.5 Comparison with the Model

Using the model of both algorithms, we can compare the estimated and measured execution time. Table 2 shows the results of the experiments along with the predictions of the model for both algorithms on 1,024 cores. To instantiate the model, we use performance numbers gathered through component-level micro benchmarks.

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

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

Table 2: Model for 1,024 cores and 40M tuples/rel./core.

Phase	Exec. Time	Model	Diff.
Radix hash join			
Histogram Comp.	0.34s	0.36s	−0.02s
Window Allocation	0.21s	—	+0.21s
Network Partitioning	2.08s	0.67s	+1.41s
Local Partitioning	0.58s	0.67s	−0.09s
Build-Probe	0.51s	0.51s	+0.00s
Imbalance	0.62s	—	+0.62s
Total	4.34s	2.21s	+2.13s
Sort-merge join			
Partitoning	1.20s	1.02s	+0.18s
Window Allocation	0.06s	—	+0.06s
Sorting	1.99s	1.45s	+0.54s
Merging	1.81s	1.78s	+0.03s
Matching	0.26s	0.36s	−0.10s
Imbalance	0.38s	—	+0.38s
Total	5.70s	4.61s	+1.09s

Parameters [million tuples per second]

RHJ: $P_{\text{scan}} = 225$, $P_{\text{Part}} = 120$, $P_{\text{net}} = 1024$, $P_{\text{build}} = 120$, $P_{\text{probe}} = 225$

SMJ: $P_{\text{part}} = 78$, $P_{\text{sort}} = 75$, $P_{\text{net}} = 1024$, $P_{\text{merge}} = 45$, $P_{\text{scan}} = 225$

6. DISCUSSION

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

6.1 Hashing vs. Sorting at Large Scale

In this work, we look at the behaviour of sort-based and hash-based join algorithms on large scale-out architectures. Our findings show that the hash join is still the algorithm of choice in terms of raw throughput. However, our results reveal that several shortcomings prevent it from reaching an even higher throughput. One significant disadvantage lies in the uncoordinated all-to-all communication pattern during the first partitioning pass. Addressing these issues requires significant changes to the structure of the algorithm, potentially resulting in a new type of algorithm.

Although the raw throughput is lower, the sort-merge join has several inherent advantages over the hash join. The interleaving of sorting and network communication creates a more steady load on the network. The fact that at each point in time every node has exactly one communication partner allows for more efficient processing on the network. This implicit scheduling can be maintained up to thousand cores, after which more sophisticated methods are required. In addition, the sort-merge join outputs sorted tuples, which might be advantageous later on in a query pipeline.

6.2 Network Scheduling

Issuing `MPI_Put` requests to the hardware is significantly more costly for the radix hash than for the sort-merge join. This is caused by the fact that the underlying hardware can only handle a limited number of simultaneous requests. To improve performance, these operations need to be coordinated. The results show that the hash join suffers from not having an effective scheduling technique. The problem is aggravated as more processes share the same network card.

The sort-merge join overcomes this problem at small scale. Each process starts sorting a different range of the input. Despite this implicit network schedule, we observe that significantly more time is spend in the network calls as the number of CPU cores increases.

In essence, light-weight but effective network scheduling techniques are needed for both algorithms in order to maintain good performance while scaling out.

6.3 Data Skew

In the experimental evaluation, we use uniform data, that is distributed evenly among all the processor cores. The goal of this study is to investigate the maximum achievable performance of the most popular algorithms on large scale-out architectures. To be able to process skewed data, good load-balancing needs to be achieved. Several techniques have been introduced for hash and sort-merge algorithms. These techniques are orthogonal to our evaluation and both join implementations could be enhanced to effectively mitigate workload imbalances caused by data skew.

Rödiger et al. [33] propose to detect skewed elements in the input with approximate histograms. The performance impact of these heavy hitters is reduced through redistribution and replication of the skewed elements. The authors show that their join implementation achieves good performance and is able to scale well on a rack-scale system. This process can be integrated into the histogram computation and the network partitioning pass of our radix hash join.

In HPC applications, sorting is a commonly used operation. By default, sorting algorithms can work with skewed data. Most distributed sorting algorithms can be put in one of two categories: merge-based and splitter-based approaches. Merge-based sorting algorithms combine data from two or more processes [7]. Splitter-based approaches try to subdivide the input into chunks of roughly equal size [13, 16, 22, 25, 36]. The latter category utilize minimal data movement because the data only moves during the split operation. In our implementation of the sort-merge join, we use a splitter-based approach. When processing skewed data, techniques for finding the optimal pivot elements can be used [36]. We expect that a histogram-based technique for finding the optimal splitter values can be integrated into the partitioning phase of our sort-merge join.

7. RELATED WORK

Comparing sort-merge and hash join algorithms has been the topic of recent work for joins on multi-core systems and efficient algorithms for both strategies have been proposed [1, 3, 4, 5, 9, 27, 30]. Kim et al. [27] concluded that although modern hardware currently favors hash join algorithms, future processors with wider Single-Instruction-Multiple-Data (SIMD) instructions would significantly speed up sort-merge joins. Albutiu et al. [1] presented a sort-merge join optimized for multi-socket NUMA machines. Their implementation reaches a throughput of 160M tuples/second on 64 cores. Based on the ideas proposed by Manegold et al. [30], Balkesen et al. [4, 5] developed and evaluated an efficient implementation of the radix hash join and report a maximum throughput of 750M tuples/second on 64 cores. The same authors compared sort-merge and hash joins [3], and concluded that despite wider SIMD vectors, the radix hash join still outperforms sort-merge based algorithms.

RDMA has been used in key-value stores, and these systems exhibit good performance and scalability [14, 24, 26]. On 20 machines, FaRM [14] can serve 150M lookups/second.

Recent work on distributed database algorithms has investigated the use of new network primitives and RDMA in the context of joins. Frey et al. [18] proposed the idea of the data-cyclotron, a system with a ring-shaped network topology in which fragments of one relation are constantly transmitted from one machine to the next. Barthels et al. [6] proposed to use RDMA in order to reduce the communication costs of large data transfers and hide the remote memory access latencies by interleaving communication and computation. The reported throughput of 1B tuples/second on 10 machines is used as a baseline in this paper. Binnig et al. [8] argued that given the trend towards high-speed interconnects with RDMA capabilities, database systems need to be fundamentally redesigned and proposed a new architectural design in which the memory of each storage node can directly be accessed through one-sided RMA operations. On 4 machines, their join implementation can process up to 130M tuples/second. Li et al. [29] investigated how a relational database whose memory demands exceed the available amount of main memory can make use of remote memory through RDMA technology.

Rödiger et al. [34, 35] looked at distributed query execution with high speed networks. FlowJoin [33] is a distributed hash join algorithm which can mitigate negative effects on performance caused by data skew.

Polychroniou et al. [32] focused on minimizing the network traffic during a join. This approach is critical to improve performance of joins on bandwidth-limited systems.

In recent work, the use of MPI has spread to other areas of computer science primarily because of the high-level interface which allows to write portable distributed applications. Several distributed databases and data processing systems, have adopted MPI as their communication layer [2, 10].

8. CONCLUSIONS

In this paper, we propose distributed hash and sort-merge join algorithms that use MPI as their communication abstraction. The algorithms are optimized to use one-sided memory operations in order to take full advantage of modern high-speed networks. Their design addresses several challenges arising from large-scale distribution, primarily the automatic selection of the underlying communication method and the management of communication buffers.

We evaluated both join implementations on two different distributed environments and showed that having the right balance of compute and communication resources is crucial to reach maximum performance. The proposed models show that the sort-merge join reaches its peak throughput. Reducing the network overhead would significantly speed up the radix hash join. Despite this fact, the performance of the hash join is superior to that of the sort-merge join.

Executing joins over large data sets in real-time has many applications in analytical data processing, machine learning, and data sciences. This paper analyses the behavior of distributed joins at large scale and shows that the radix hash and sort-merge join algorithms scale to 4,096 cores, achieving a throughput of 48.7 billion input tuples per second.

9. ACKNOWLEDGMENTS

The work presented in this paper has been funded in part by a grant from Oracle Labs.

10. REFERENCES

[1] M. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, pages 1064–1075, 2012.

[2] K. Anikiej. Multi-core parallelization of vectorized query execution. Master's thesis, VU University, 2010.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, pages 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*, pages 362–373, 2013.

[5] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on modern processor architectures. *IEEE TKDE*, pages 1754–1766, 2015.

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

[7] K. E. Batcher. Sorting networks and their applications. In *AFIPS*, pages 307–314, 1968.

[8] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, pages 528–539, 2016.

[9] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, pages 37–48, 2011.

[10] A. Costea, A. Ionescu, B. Raducanu, M. Switakowski, C. Barca, J. Sompolski, A. Luszczak, M. Szafranski, G. D. Nijs, and P. Boncz. VectorH: taking SQL-on-Hadoop to the next level. In *SIGMOD*, pages 1105–1117, 2016.

[11] Cray XC Series. http://www.cray.com/products/computing/xc-series/.

[12] CSCS Piz Daint Supercomputer. http://user.cscs.ch/computing_systems/piz_daint/index.html.

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

[14] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[15] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 2012.

[16] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, pages 496–507, 1970.

[17] P. W. Frey and G. Alonso. Minimizing the hidden cost of RDMA. In *ICDCS*, pages 553–560, 2009.

[18] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. A spinning join that does not get dizzy. In *ICDCS*, pages 283–292, 2010.

[19] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling highly-scalable remote memory access programming with MPI-3 one sided. In *SC*, pages 53:1–53:12, 2013.

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

[21] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood. Remote Memory Access Programming in MPI-3. *ACM TOPC*, page 9, 2015.

[22] J. Huang and Y.C.Chow. Parallel sorting and data partitioning by sampling. In *COMPSAC*, 1983.

[23] InfiniBand Trade Association. Architecture specification 1.3, 2015.

[24] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached design on high performance RDMA capable interconnects. In *ICPP*, pages 743–752, 2011.

[25] L. V. Kalé and S. Krishnan. A comparison based parallel sorting algorithm. In *ICPP*, pages 196–200, 1993.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, pages 295–306, 2014.

[27] 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*, pages 1378–1389, 2009.

[28] A. Kumar, J. Naughton, J. M. Patel, and X. Zhu. To join or not to join? Thinking twice about joins before feature selection. In *SIGMOD*, pages 19–34, 2016.

[29] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In *SIGMOD*, pages 355–370, 2016.

[30] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE TKDE*, pages 709–730, 2002.

[31] Message Passing Interface Forum. MPI: a message-passing interface standard, version 3.0, 2012.

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

[33] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-Join: adaptive skew handling for distributed joins over high-speed networks. In *ICDE*, pages 1194–1205, 2016.

[34] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, pages 228–239, 2015.

[35] 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.

[36] E. Solomonik and L. V. Kalé. Highly scalable parallel sorting. In *IPDPS*, pages 1–12, 2010.