Memory Footprint Matters: Efficient Equi-Join Algorithms for Main Memory Data Processing

Spyros Blanas and Jignesh M. Patel
University of Wisconsin–Madison
{sblanas,jignesh}@cs.wisc.edu

Abstract

High-performance analytical data processing systems often run on servers with large amounts of main memory. A common operation in such environments is combining data from two or more sources using some "join" algorithm. The focus of this paper is on studying hash-based and sort-based equi-join algorithms when the data sets being joined fully reside in main memory. We only consider a single node setting, which is an important building block for larger high-performance distributed data processing systems. A critical contribution of this work is in pointing out that in addition to query response time, one must also consider the memory footprint of each join algorithm, as it impacts the number of concurrent queries that can be serviced. Memory footprint becomes an important deployment consideration when running analytical data processing services on hardware that is shared by other concurrent services. We also consider the impact of particular physical properties of the input and the output of each join algorithm. This information is essential for optimizing complex query pipelines with multiple joins. Our key contribution is in characterizing the properties of hash-based and sort-based equi-join algorithms, thereby allowing system implementers and query optimizers to make a more informed choice about which join algorithm to use.

SoCC'13, 1–3 Oct. 2013, Santa Clara, California, USA.
ACM 978-1-4503-2428-1. http://dx.doi.org/**10.1145/2523616.2523626**

1 Introduction

Fueled by the economies of scale, businesses and governments around the world are consolidating their data processing infrastructure to public or private clouds. In order to meet the stringent response time constraints of interactive analytical applications, analytic data processing systems running on these clouds aim to keep the working data set cached in main memory. In fact, many database vendors now offer dedicated main memory database appliances, like SAP HANA [13] and Oracle Exalytics [27], or have incorporated main-memory technology, such as IBM Blink [5], to further accelerate data processing.

While memory densities have been increasing, at the other end of the motherboard, another big change has been underway. Driven by power and heat dissipation considerations, about seven years ago, processors started moving from a single core to multiple cores (i.e. multicore). The processing component has moved even further now into multi-sockets, with each socket having multiple cores. The servers powering the public and private clouds today commonly have multi-socket, multicore processors and hundreds of gigabytes of main memory.

This paper focuses on the workhorse operation of an analytical query processing engine, namely the equi-join operation. As is readily recognized, an equi-join operation is a common and expensive operation in analytics and decision support applications, and efficient join algorithms are critical in improving the overall performance of main-memory data processing for such workloads.

Naturally, there has been a lot of recent interest in designing and evaluating main memory equi-join algorithms. The current literature, however, provides a confusing set of answers about the relative merits of different join algorithms. For example, Blanas et al. [6] showed how the hash join can be adapted to single-socket multicore main memory configurations. Then, Albutiu et al. [2] showed that on multi-socket configurations, a new sort-merge join algorithm (MPSM) outperformed the hash join algorithm proposed in [6], but their study did not evaluate hash-based algorithms that are multi-socket friendly. At the same time, Balkesen et al. [4] found that a hardware-conscious implementation of the radix partitioned join algorithm can greatly improve performance over the join algorithm proposed in [6]. All of these studies are set against the backdrop of an earlier study by Kim et al. [22] that described how sort-based algorithms are likely to outperform hash-based algorithms in future processors. Thus, there is no clear answer regarding what equi-join algorithm works well, and under what circumstances, in main memory, multi-socket, multicore environments.

Previous work in this space has also largely ignored the memory footprint of each join algorithm. Main memory, however, is a precious resource in these "in-memory" settings. The peak working memory that each query requires is a key factor in determining how many tenants can be consolidated into a single system. Algorithms that use working memory judiciously allow more tenants to simultaneously use a single system, and thus lower the total infrastructure operating cost.

In this paper, we build on previous work as we adapt the hash join algorithm by Balkesen et al. [4] for a multi-socket setting. We consider the recently proposed massively parallel sort-merge (MPSM) join algorithm by Albutiu et al. [2], in addition to two more traditional sort-merge join variants. Finally, we also experiment with a sort-merge algorithm that implements a bitonic merge network using SIMD instructions to exploit data-level parallelism, as it was proposed by Kim et al. [22].

This paper makes three key contributions. First, we characterize the memory access pattern and the memory footprint of each join algorithm. Second, we identify two common physical properties of the join input and output that greatly impact the overall performance. The two physical properties are (1) data being hash partitioned on the join key, and (2) data being pre-sorted on the join key. This information is important for query optimization when considering queries with multiple joins. Finally, we conduct a comprehensive experimental study for each join algorithm in a main memory setting.

Our results show that the hash join algorithm in many cases produces results faster than the sort-based algorithms, and the memory footprint of the hash join algorithm is generally smaller than that of the sort-based algorithms. Sort-based algorithms become competitive only when one of the join inputs is already sorted. Our study concludes that to achieve the best response time and consolidation for main memory equi-join processing, it is necessary to take into account the physical properties of the inputs and the output, and it is crucial that both hash-based and sort-based join algorithms are implemented.

The remainder of this paper is organized as follows. We introduce the join algorithms in Section 2, and provide implementation details in Section 3. We continue with the experimental evaluation in Section 4. Related work is described in Section 5, and Section 6 contains our conclusions.

2 Parallel equi-join algorithms

In this section, we describe key main memory join algorithms that have been recently proposed. We first briefly present each algorithm, and then analyze how each algorithm fares with respect to two factors: (1) memory traffic and access pattern, and (2) working memory footprint.

We describe each algorithm assuming that the query engine is built using a parallel, operator-centric, pull-based model [17]. In this model, a query is a tree of operators, and T worker threads are available to execute the query. The leaves of the tree read data from the base tables or indexes directly. To execute a query, each operator recursively requests ("pulls") data from one or more children, processes them, and writes the results in an output buffer local to each worker thread. The output buffer is then returned to the parent operator. Multiple worker threads may concurrently execute (part of) the same operator.

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

2.1 Partitioning

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

Partitioning plays a key role in intra-operator parallelism in various settings, since once the input is partitioned, different worker threads can operate on different partitions [10, 12, 28] in parallel. The partitioning algorithm described here is the parallel radix partitioning described by Kim et al. [22], which is in turn based on the original radix partitioning algorithm by Boncz et al. [7]. The main idea behind this parallel partitioning algorithm is that one can eliminate expensive latch-based synchronization during repartitioning, if all the threads know how many tuples will be stored in each partition in advance.

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

2.2 In-place sorting

Work on sorting algorithms can be traced back to the earliest days of computing, and textbooks on data structures, databases and algorithms commonly have a chapter dedicated to sorting. Graefe has written a survey [19] of sorting techniques used by commercial DBMSs, and fast parallel sort algorithms for main memory DBMSs are an active research area [22, 29].

We originally experimented with a sort-merge algorithm that implements a bitonic merge network using SIMD instructions to exploit data-level parallelism [22]. (We show results from this algorithm in Section 4.4 for the dataset where all tuples are eight bytes wide.) We ultimately decided to use introspective sort [25], which is a popular in-place sorting algorithm, for three reasons. First, many highly-optimized sorting algorithms make strong assumptions about the physical placement of the data and the width of the tuple (for example, only eight bytes in [22]). The second reason is that introspection sort has been used in recently published results [2]. Using the same algorithm promotes uniformity with prior work. Finally, introspection sort is implemented in many widely-used and heavily optimized libraries, like the C++ standard template library. Our findings are therefore less likely to be affected by suboptimal configuration settings or poorly optimized code.

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

2.3 Hash join

The algorithm described here is the main-memory hash join proposed in [6], adapted to ensure that the hash table is striped across the local memory of all threads that participate in the join [4]. But, exactly as the original algorithm, it is otherwise oblivious to any NUMA effects.

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

During the probe phase, each thread reads a tuple $s \in S$, and hashes on the join key of s using the same function $h(\cdot)$. For each $r_{h(s)}$ tuple in the hash bucket $h(s)$, the join keys of $r_{h(s)}$ and s are compared and the tuples are joined together, if the join keys match. These memory reads may either target local or remote memory depending on the physical location of the hash bucket $h(s)$. Because the hash table is only read during the probe phase, there is no need to acquire latches. When all the S tuples have been processed, the probe phase is complete and the memory holding the hash table can be reclaimed.

If the R tuples processed by a thread are pre-sorted or pre-partitioned on the join key, neither property will be reflected in the output. However, if the S tuples that are processed by a thread are pre-sorted or pre-partitioned on any S key, the tuples produced by this thread will also be sorted or partitioned on the same key.

2.4 Streaming merge join (STRSM)

This algorithm is a parallel version of the standard merge join algorithm [11]. It is assumed that both R and S are already partitioned on the join key with some partitioning function that has produced as many partitions as the number of threads that participate in the join. If T is the number of threads, we use R_i to refer to the partition of R that will be processed by thread $i \in [1,T]$, and similarly we use S_i to denote the partition of S that will be processed by the same thread i. Furthermore, this algorithm requires that both R_i and S_i are already sorted on the join key.

Each thread i reads the first tuple r from R_i and the first tuple s from S_i. If the r tuple has a smaller join key value than the s tuple, then the r tuple is discarded and the next tuple from R_i is read. Otherwise, if the r tuple has a larger join key value than the s tuple, the s tuple is discarded and the next S_i tuple is read. If the join keys of r and s are equal, then the algorithm will buffer all R_i and S_i tuples whose join key is equal, produce the join output, and then discard these temporary buffers. This process is repeated until either R_i or S_i are depleted. The output of each thread is sorted and partitioned on the join key.

2.5 Range-partitioned MPSM merge join

This algorithm is the merge phase of the range-partitioned P-MPSM algorithm [2]. The algorithm requires R to be partitioned in T partitions on the join key. We keep the notation introduced in Section 2.4, and use R_i to refer to the partition of R that will be processed by thread i, where $i \in [1,T]$. The algorithm requires that each R_i partition has already been sorted on the join key.

Each thread i first allocates a private buffer B_i in NUMA-local memory, and starts copying tuples from S into B_i. (B_i is not a partition of S as two tuples with the same key might be processed by different threads, and thus be stored in different buffers.) When S is depleted, each thread i sorts its B_i buffer in-place, using the algorithm described in Section 2.2. Let B_i^j be the region in the B_i buffer that corresponds to the tuples that join with tuples in partition R_j. Thread i then computes the offsets of every B_i^j region in the B_i buffer, for each $j \in [1,T]$. Because B_i is sorted on the join key, one can compute the partition boundary offsets efficiently using either interpolation or binary search. Once all the threads have computed these offsets, thread i proceeds to merge R_i and B_1^i using the streaming merge join algorithm described in Section 2.4. Thread i continues with merging R_i with B_2^i, then R_i with B_3^i, and so on, until R_i has been fully merged with B_T^i. The memory for the B_i buffer can only be reclaimed by the memory manager after all the threads have completed.

The output of the MPSM join is partitioned on the join key with the same partitioning function as R per thread. If S is range-partitioned on the join key, even with a different partitioning function than that for R, each thread i can produce output that is sorted on the join key by processing each B_j^i fragment in order.

2.6 Parallel merge join (PARSM)

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

Each thread i that participates in the parallel merge join algorithm starts by reading and storing tuples from S in the private buffer B_i. Then, thread i sorts B_i on the join key, and the B_i^j regions are computed exactly as in the MPSM algorithm. The difference in the parallel merge join algorithm lies in how thread i merges all the B_j^i regions with R_i. The original MPSM algorithm performs T passes, and then merges two inputs at a time. In the parallel merge join algorithm, thread i merges all B_j^i buffers with the R_i partition in one pass, where $j \in [1,T]$. This is a parallel merge between T inputs for the S side, and one input for the R side, which results in $T+1$ input tuples being candidates for merging at any given time.

The output of the parallel merge join is partitioned on the join key per thread, with the same partitioning function as R, and each thread's output is always sorted on the join key.

2.7 Analysis

We now analyze how each algorithm fares with respect to two factors: (1) memory traffic and access pattern, and (2) working memory footprint. We summarize the analysis in Table 1. Paying attention to the first factor, memory accesses, is important because the CPU may stall while waiting for a particular datum to arrive. Such stalls have been shown to have a significant impact on performance [1]. With modern CPUs packing more than eight cores per die, the available memory bandwidth per hardware thread has been shrinking and access to memory is becoming relatively more expensive than in the past. The second factor, working memory size, is important because memory is a precious storage medium, and using it judiciously can allow a database service to admit more concurrent queries and achieve higher throughput.

The partitioning algorithm first scans R twice, once from the input and once from each thread's local buffer, for a total of $2 \times \|R\|$ memory read operations and $\|R\|$ memory write operations. These reads and writes are sequential and to the local NUMA memory, making this phase extremely efficient. Producing the output parti-

Algorithm	Input	Number of memory accesses				Peak memory footprint
		Sequential		Random		
		Reads	Writes	Reads	Writes	
Partitioning	R	$2 \times \|R\|$	$\|R\|$	—	$\|R\|$	$2 \times \|R\|$
In-place sort	R	—	—	$\Theta(\|R\|log\|R\|)$	$\Theta(\|R\|log\|R\|)$	$\|R\|$
Hash join	R,S	$\|R\| + \|S\|$	$\|R\| \times \|S\| \times \sigma$	$\|S\|$	$\|R\|$	$\|R\|$
Streaming merge	R,S	$\|R\| + \|S\|$	$\|R\| \times \|S\| \times \sigma$	—	—	$\Theta(1)$
MPSM merge	R,S	$\|R\| \times T + \|S\|$	$\|R\| \times \|S\| \times \sigma$	—	—	$\|R\| + \|S\|$
Parallel merge	R,S	$\|R\|$	$\|R\| \times \|S\| \times \sigma$	$\|S\|$	—	$\|S\|$

Table 1: Number of memory accesses and peak memory footprint (in tuples) for each algorithm described in Section 2. $||R||$ is the cardinality of R, the number of worker threads is denoted by T, and σ is the join selectivity.

tions requires $||R||$ additional memory writes, assuming the partitioning is completed in one pass. This is a more costly operation as the writes may target remote NUMA memory. The partitioning algorithm needs to maintain an input and an output buffer, a total of $2 \times ||R||$ tuples. Once the repartitioning is completed, the input buffer can be deallocated, bringing the total space requirement down to $||R||$ tuples.

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

The hash join algorithm exhibits a memory access pattern that is hard to predict, and represents a demanding workload for the memory subsystem. Building a hash table requires writing $||R||$ tuples at the locations pointed to by the hash function. A good hash function would pick any bucket with the same probability for each distinct join key, causing writes that are randomly scattered across different hash buckets. Similarly, probing the hash table causes $||S||$ reads, and these reads are randomly distributed across all the hash buckets, causing remote memory reads from other NUMA nodes. Each probe lookup reads the entire chain of tuples associated with a hash bucket, but, in the absence of skew, the chain length is low in correctly-sized hash tables [16]. Overall, the hash join algorithm performs $||R||$ writes and $||S||$ reads with a random memory access pattern, potentially to remote NUMA nodes. When it comes to memory footprint, the hash join algorithm needs to buffer all the tuples in the build table R. In addition, the memory needed to store bucket metadata can be significant, and can even be greater than the size of R if the size of each R tuple is small.

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

The MPSM merge join algorithm can cause a lot of memory traffic. The number of tuples that MPSM reads grows linearly with the number of threads participating in the join; this algorithm reads a total of $||R|| \times T + ||S||$ tuples, where T is the number of threads. The access pattern of the MPSM algorithm is sequential and only two inputs are merged at any given time per thread. This memory access pattern is highly amenable to hardware prefetching. Regarding memory footprint, we find that the MPSM join needs $||R|| + ||S||$ tuples of space to perform well. At minimum, the MPSM join needs to store the entire S input, so the memory capacity requirement is at least $||S||$ tuples. Although it is not necessary for correctness, we have observed that buffering R significantly improves performance: As the join is commonly an operator that is closer to the root than the leaves of a query tree, it is preferable to execute the subquery that produces R once, buffer the output and reuse this buffer $T-1$ times, than it is to execute the entire R subquery T times.

The parallel merge join (PARSM) causes significantly lower memory traffic compared to the MPSM join. The biggest gain comes from reading the inputs once, so the number of tuples that the parallel merge join reads remains constant regardless of the number of threads participating in the join. The memory access pattern, however, is very different, as $T+1$ locations need to be inspected. As we will show in Section 4.2, keeping track of $T+1$ locations stresses the TLB, causing frequent page walks which waste many cycles, and also makes hardware prefetching less effective. When it comes to memory space, the parallel merge join algorithm only needs sufficient memory to buffer the entire S input. Unlike the MPSM algorithm, each R_i partition is scanned once by each thread i so there is no performance benefit in buffering the R input.

3 Evaluation methodology

3.1 Hardware

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

We refer to a number of performance counters when presenting our experimental results in Section 4.2. We have written our own utility to tap into the Uncore counters [21]. When we refer to memory reads, we count the L3 cache lines filled as reported by the LLC_S_FILLS event. When we refer to memory writes, we count the L3 cache lines victimized in the M state, or the LLC_VICTIMS_M event. We obtain the QPI utilization by comparing the NULL_IDLE flits sent across all QPI links with the number of NULL_IDLE flits sent when the system is idle. We then report the utilization of the most heavily utilized QPI link (ie. the link most likely to be a bottleneck) averaged over the measurement interval. Finally, we obtain timings for the TLB page miss handler by measuring the DTLB_LOAD_MISSES.WALK_CYCLES event.

3.2 Query engine prototype

We have implemented all algorithms described in Section 2 in a query engine prototype written in C++. The engine is parallel and NUMA-aware, and can execute simple queries on main memory data. The engine expects queries in the form of a physical plan, which is provided by the user in a text file. The plan carries all the information necessary to compute the query result. All operators in our query engine have a standard interface consisting of start(), stop() and next() calls. This is a pull-based model similar to the Volcano system [17]. We amortize the cost of function calls; each next() call returns a 64KB array of tuples.

Our query engine prototype implements all the algorithms described in Section 2. We base the hash join implementation on the code written by Blanas et al. [6], which is publicly available. Each hash bucket in our implementation has 16 bytes of metadata, and these are used for: (1) a latch, (2) a counter holding the number of unused bytes in this hash bucket, and (3) an overflow pointer to the next bucket, or `NULL` if there is no overflow. We have extended the original implementation to be NUMA-aware by allocating the entire hash bucket i from NUMA node $i \bmod N$, where N is the number of NUMA nodes in the system. This means that tuples in the same hash bucket always reside in the same NUMA node, and reading hash buckets sequentially accesses all NUMA nodes in a round-robin fashion. We size each hash bucket so that two tuples can fit in a bucket before there is an overflow. The number of hash buckets is always a power of two, and we size the hash table such that the load factor is greater than one, but less than two. The original code preallocated the first hash bucket, and we do likewise.

The hash join algorithm needs to compute the hash function for $\|R\|$ tuples in the build phase, and $\|S\|$ tuples in the probe phase, for a total of $\|R\|+\|S\|$ hash function evaluations. An ideal hash function has low computational overhead and low probability of hash collisions. Database management systems typically choose the hash function dynamically based on the data type and the domain of the key when the hash table is constructed. The number of cycles spent on computing the hash value per tuple depends on the chosen hash function and the size of the join key. In general, this number ranges from one or two cycles for hash functions based on bit masking and shifting, to hundreds of cycles for hash functions that involve one multiplication per byte of input, like FNV-1a [14]. For our experimental evaluation, if the hash table has 2^b hash buckets, we use the hash function $h(x) = (x * 2654435761) \bmod 2^b$. This multiplicative hash function [23] performs one multiplication and one modulo operation per evaluation; it is more expensive to evaluate than bit shifting, but much cheaper than FNV-1a.

We implemented the MPSM merge join algorithm from scratch based on the description of the P-MPSM variant in [2], as the original implementation is not available.

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

Table 2: List of query plans that we evaluate: "random" means tuples are in random order and not partitioned; "sorted" means sorted on the join key and range-partitioned; "hash" means stored in a hash table and hashed on the join key.

3.3 Metrics

The two metrics we report are response time and memory footprint. Short query response times are, obviously, important for decision support queries. Memory footprint is also an important metric for two reasons. The first is that query plans that use less memory allow for higher tenant consolidation in a cloud-based database service. This permits the concurrent execution of more queries, increasing the throughput per node. Second, main memory is used as working memory for active queries, and also as storage for user data. A main-memory database engine that selects plans that use working memory frugally can keep a larger database in memory, improving the overall utility of the system.

4 Experimental results

4.1 Workload

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

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

1. **Random**, which corresponds to the case where no pre-processing has been done and data is in random order. Random input is processed by worker threads in any order.
2. **Sorted**, which corresponds to the case where the input is already sorted on the join key. Sorted input can easily be partitioned among worker threads in distinct ranges, as the range partition boundaries can be computed from existing statistics on the input.
3. **Hash**, which reflects the case where a table is stored in a hash table, indexed on a particular key. This type of pre-processing is common for dimension tables that are small and are updated less frequently. Storing the large fact table in a hash table may be pro-

hibitive in terms of space overhead, and maintaining the hash table is costly in the presence of updates. We therefore only consider this physical property for dimension tables.

4.2 Results from a uniform dataset with narrow tuples

A common pattern in decision support queries is the equi-join between a dimension table and a fact table, followed by an aggregation. We use a synthetic dataset to evaluate an ad-hoc equi-join between the dimension table *R*, and the fact table *S*. The dimension table *R* contains the primary key and the fact table *S* contains the foreign key. *R* has 800×2^{20} tuples and each tuple is sixteen bytes wide. Each *R* tuple consists of an eight-byte unique primary key in the range of $\left[1, 800 \times 2^{20}\right]$, and an eight-byte random integer. The fact table *S* has four times as many tuples as *R*, namely 3200×2^{20} tuples, and each tuple is also sixteen bytes wide.[2] The first eight-byte attribute of an *S* tuple is the foreign key of *R*, and the second eight-byte attribute is a random integer. The cardinality of the output of the primary-key foreign-key join is the cardinality of *S*, or 3200×2^{20} tuples. This dataset does not have any skew, and every primary key in *R* occurs in *S* exactly four times.

Assuming that the dimension table *R* has two integer attributes `R.a` and `R.b`, and the fact table *S* has two integer attributes `S.c` and `S.d`, for all the experiments described here, we produce plans corresponding to the following SQL query:

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

Depending on the physical properties of *R* and *S*, this logical query may be translated into a number of physical execution plans. For our experimental evaluation, we consider candidate plans that feature each join algorithm presented in Section 2 for all combinations of the physical properties of *R* and *S*. The query plan space we explore is shown in Table 2. For all our experiments, we execute each query in isolation, and use 80 worker threads, which is the number of hardware contexts our system supports.

As described in Section 3.3, we measure cost both in terms of response time, and memory space. Figures 1 and 2 show the response time in seconds on the x-axis, and the memory consumption in gigabytes (2^{30} bytes) on the y-axis. We use "HASH" for the hash-based plan, and the "SM" suffix for the three sort-based query plans: "STRSM" for the plan with the streaming merge join operator, "MPSM" for the plan containing the MPSM merge join operator, and "PARSM" for the plan with the parallel merge join operator. Table 2 shows the actual operators for each plan, and Table 3 breaks down time per operator.

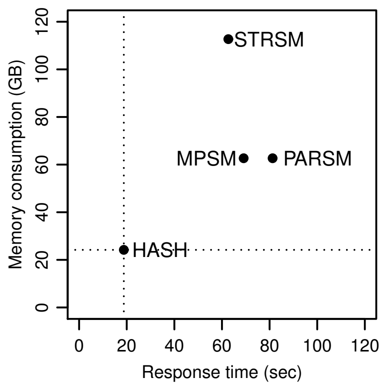
[2] We picked the one-to-four ratio to match the cardinality ratios of a primary-key foreign-key join between tables `Orders` and `LineItem`, or `Part` and `PartSupp` of the TPC-H decision support benchmark.

In Figure	1(a)	1(b)	1(c)	2(a)	2(b)	2(c)
HASH	*S* in random order			*S* in join key order		
	R rnd	*R* ord	*R* ht	*R* rnd	*R* ord	*R* ht
Build HT on *R*	4.2	5.0	—	5.1	5.6	—
Probe HT with *S*	13.5	13.7	13.6	6.0	5.3	6.0
Query	18.8	18.8	13.9	11.1	11.1	6.1
STRSM	*S* in random order			*S* in join key order		
	R rnd	*R* ord	*R* ht	*R* rnd	*R* ord	*R* ht
Partition *R*	1.5	—	1.5	2.8	—	2.8
Sort *R*	7.2	—	7.2	7.3	—	7.3
Partition *S*	6.4	8.8	6.4	—	—	—
Sort *S*	36.3	36.7	36.3	—	—	—
Stream merge	5.9	2.0	5.9	1.8	1.9	1.8
Query	62.7	52.3	62.7	11.9	2.0	12.0
MPSM	*S* in random order			*S* in join key order		
	R rnd	*R* ord	*R* ht	*R* rnd	*R* ord	*R* ht
Partition *R*	3.4	—	3.4	2.7	—	2.7
Sort *R*	5.9	—	5.9	6.7	—	6.7
Sort *S*	29.2	29.1	29.2	—	—	—
MPSM merge	24.5	16.2	24.5	1.7	1.3	1.7
Query	69.1	51.2	69.1	16.2	7.5	16.2
PARSM	*S* in random order			*S* in join key order		
	R rnd	*R* ord	*R* ht	*R* rnd	*R* ord	*R* ht
Partition *R*	3.1	—	3.1	2.7	—	2.7
Sort *R*	5.8	—	5.8	6.7	—	6.7
Sort *S*	28.9	29.0	28.9	—	—	—
Parallel merge	35.6	28.2	35.6	4.8	4.4	4.8
Query	81.3	63.4	81.3	19.5	10.6	19.5

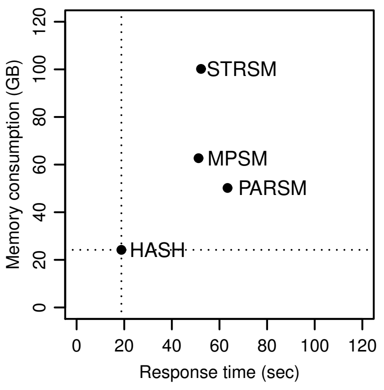
Table 3: Time breakdown per operator for the uniform dataset. "*R* rnd" means that *R* is in random order, "*R* ord" means that *R* is sorted in join key order, and "*R* ht" means that *R* is in a hash table on the join key. "Query" reflects the query response time as shown in Figures 1 and 2, and is not the sum due to synchronization and buffering overheads that are not reflected above.

4.2.1 S in random order

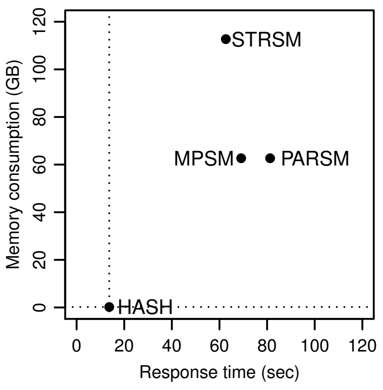
We start with the case where the larger fact table *S* contains tuples with keys in random join key order. This requires each algorithm to pay the full cost of transforming the input *S* to have the physical property necessary to compute the join, like being sorted or partitioned on the join key. In Figure 1 we plot the response time and the memory consumption of each plan, for all three physical



(a) *R* is in random order



(b) *R* is sorted in ascending join key order



(c) *R* is in hash table on join key

Figure 1: Response time (in seconds) and memory consumption (in GB) when *S* is in random order, uniform dataset.

properties of *R* we explore: random, sorted, and hash. We describe each in turn.

R in random order

We start with the general case where both the *R* and *S* tables contain tuples with join keys in random order. The response time and memory demand for all the four query plans is shown in Figure 1(a), and a breakdown per operator is shown in Table 3, column 1(a).

The hash join query plan computes the query result in 18.8 seconds, the fastest among the all four query plans. The hash join query plan also uses the least space, needing 24.2 GB of scratch memory space, nearly all of which (99%) is used to store the hash table. The majority of the time is spent in the probe phase, which does about 1.90 memory reads per *S* tuple. Because the *S* tuples are processed in random join key order, these reads are randomly scattered across the shared hash table. This random memory access pattern causes a significant number of TLB misses, with 10% of the probe time, or 1.31 seconds, being spent on handling page misses on memory loads. As the hash table is striped across all the NUMA nodes, the cross-socket traffic is the highest among all the query plans: we have observed an average QPI utilization of 34%, which means that remote memory access requests might get queued at the QPI links during bursts of higher memory activity.

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

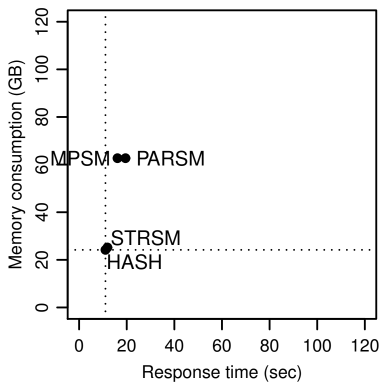
The MPSM sort-merge join query plan takes 69.1 seconds and 62.7 GB of space to run. Compared to the streaming sort-merge query plan, MPSM is only 10% slower, and, because the MPSM merge-join algorithm avoids repartitioning the larger *S* table, it needs about half the space. The MPSM merge join algorithm performs T scans of *R* during the merge phase (see Section 2.5), which means reading an additional 0.96 TB, for this particular dataset. This causes 4.32 memory reads per *S* tuple, during the merge phase, which accounts for 35% of the total time.

The parallel sort-merge query plan (PARSM) is identical to the MPSM sort-merge join query plan, described above, for the partitioning and sort phases. During the merge phase, however, the parallel merge join algorithm does not partition *S* and does not scan *R* multiple times. This cuts the memory traffic significantly, resulting in only 0.38 memory reads per *S* tuple for the merge phase. Merging from 81 ($T+1$) locations in parallel, however, overflows the 64-entry TLB (see Section 2.6 for a description of the algorithm). We observed that page miss handling on memory load accounts for at least 37% of the merge time (13.2 seconds). The high TLB miss rate causes the merge phase to take 11 more seconds than the MPSM merge phase, which brings the end-to-end response time to a total of 81.3 seconds, making the parallel merge-join plan the slowest of the four alternatives.

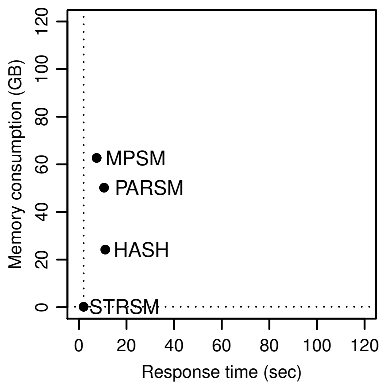
Overall, when *R* and *S* are processed in random order, the hash join query plan produces results 3.34$\times$ faster and uses 2.59$\times$ less memory than the best sort-merge based query plan, despite the many random accesses to remote NUMA nodes that result in increased QPI traffic.

R in ascending join key order

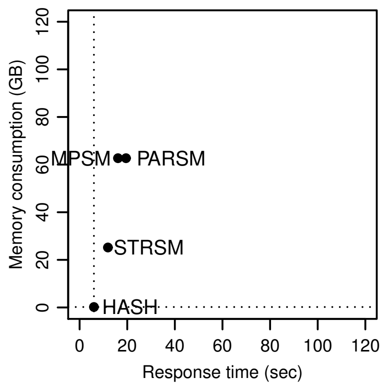
We now consider the case where the smaller dimen-



(a) *R* is in random order



(b) *R* is sorted in ascending join key order



(c) *R* is in hash table on join key

Figure 2: Response time (in seconds) and memory consumption (in GB) when *S* is sorted in ascending join key order, uniform dataset.

sion table *R* is sorted, and the larger table *S* is in random order. We plot the response time and memory demand for the four query plans in Figure 1(b), and a breakdown per operator is shown in Table 3, column 1(b).

The response time and memory consumption of the hash-join query plan are 18.8 seconds and 24.2 GB respectively. Comparing with the case where *R* is in random order, Figure 1(a), we see that performance is not affected by whether the build side *R* is sorted or not. Also, the number of memory operations and the QPI link utilization are the same as when processing *R* with tuples in random physical order. This happens because the hash function we use, described in Section 3.2, scatters the sorted *R* input among distant buckets. This results in memory accesses that are not amenable to hardware caching or prefetching, exactly as when processing *R* in random order.

Moving to the stream-merge join query plan (STRSM), we find that it completes the join in 52.3 seconds and has a memory footprint of about 100 GB, with nearly all of this space being used for repartitioning *S*. There is no need to repartition and sort *R*, as it is already in join key order. Repartitioning and sorting *S* causes 5.25 memory reads and 3.82 memory writes, on average, per *S* tuple.

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

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

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

R in hash table, on join key

We now move to the case where *R* is stored in a hash table created on the join key, and the *S* tuples are in random order. We plot the results in Figure 1(c). We observe that the three sort-based queries have similar response time and memory consumption as when the *R* tuples are in random order (cf. Figure 1(a)). Due to the pre-allocated space in the hashtable buckets, no pointer chasing is needed to read the first bucket of each hash chain, allowing for sequential reading across all NUMA nodes. *R* is partitioned in buckets based on hash value, so all sort-based plans first repartition *R* to create range partitions. These partitions subsequently need to be sorted, before being processed by each merge-join algorithm. Overall, these steps result in the same data movement operations as when *R* is in random order.

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

4.2.2 S in ascending join key order

We now consider the case when the S table is sorted in ascending join key order. In this case, we also assume that S is partitioned, as each thread i can discover the boundaries of the S_i partition by performing interpolation or binary search in the sorted table S. In the results that follow, we have discounted the cost of computing the partition boundaries in this fashion. We plot the response time and the memory consumption of each query, for all three physical properties of R we explore. These results are shown in Figure 2.

R in random order

First we consider the case when the larger table S is sorted on the join key, but the smaller table R is not. We plot the results in Figure 2(a), and a breakdown per operator is shown in Table 3, column 2(a).

The hash join query computes the result in 11.1 seconds and needs 24.2 GB of memory space. If we compare with the case where S is in random order, in Figure 1(a), we see a 1.70× improvement in response time, if S is sorted on the join key. This happens because now that S is sorted, the four S tuples that match a given R tuple occur in sequence. This allows the R tuple to be read only once and then stay in the local cache, reducing the total number of memory operations. Looking at the performance counters, we find that 0.36 memory reads occur per S tuple, which is 5.31× less than the number of memory operations that happen when S is randomly ordered. Probing the hash table with a sorted S input also results in fewer cycles spent on TLB miss handling (0.6 seconds, or 10% of the probe time, a 2.22× improvement), as well as lower QPI link utilization during the probe phase (14.6% utilization on average, a 2.35× improvement) when compared with processing a randomly ordered S table.

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

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

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

R in ascending join key order

This is the case where both R and S are presorted on the join key. The results for this experiment are shown in Figure 2(b), and a breakdown per operator is shown in Table 3, column 2(b). The hash join plan has a response time of 11.1 seconds, and uses 24.2 GB to store the hash table for R. The hash join plan cannot take advantage of the sorted R side and needs to construct a hash table on R. Join keys appear sequentially in S, and this translates into less memory traffic when probing the hash table due to caching. The streaming merge join plan (STRSM) is the fastest, as it only needs to scan the pre-sorted and pre-partitioned inputs in parallel to produce the join output. The response time of the STRSM plan is 1.98 seconds, and because of its streaming nature the memory size is fixed, regardless of the input size. We measured memory consumption to be 0.01 GB, primarily for output buffer space. As before, the MPSM and the PARSM plans are degenerate two-way stream merge join queries.

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

R in hash table, on join key

Finally, we consider the case where R is stored in a hash table, on the join key, and S is sorted in ascending join key order. We plot the response time and memory demand of each query plan in Figure 2(c), and a breakdown per operator is shown in Table 3, column 2(c).

The hash join plan omits the build phase, as R is already in the hash table, and proceeds directly to the probe phase. The hash join plan completes the query in 6.1 seconds, and needs only 0.01 GB of space for storing metadata, the input state and the output buffers of each thread. None of the sort-based plans can take advantage of the hash-partitioned R. As a consequence, the first step in all plans is to repartition R and produce range partitions. The streaming merge join plan (STRSM) takes 12.0 seconds, and needs 25 GB for the repartitioning, and the remaining two sort-based plans are degenerate cases of the streaming merge join query.

To summarize, if R is already stored in a hash table,

	1	2	3	4	5	6
Compare to	1(a)	1(b)	1(c)	2(a)	2(b)	2(c)
	S in random order			S in join key order		
Query plan	R rnd	R ord	R ht	R rnd	R ord	R ht
HASH	**12.72**	**12.77**	**7.63**	**9.27**	9.29	**4.19**
STRSM	84.72	66.13	84.72	15.64	**5.45**	15.64
MPSM	75.84	51.39	75.84	19.70	11.05	19.70
PARSM	84.82	61.08	84.82	23.00	13.98	23.00

Table 4: Response time (in seconds) for the skewed dataset. "R rnd" means that R is in random order, "R ord" means that R is sorted in join key order, and "R ht" means that R is in a hash table on the join key. The query plan with the fastest response time is highlighted in bold.

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

4.3 Effect of data skew

Many real workloads exhibit data skew. As a consequence, data processing systems are rarely able to distribute data as uniformly as in the experiments of the previous section. In this section, we show how response time and memory consumption is affected by the presence of data skew.

We generated a new skewed dataset where the foreign keys in S are generated based on the Zipf distribution with parameter $s = 1.05$. In this skewed dataset, the top 1000 keys from R appear in nearly half (48%) the tuples of S. All other parameters of the skewed dataset are the same as for the uniform dataset (described in Section 4.2). The query plans we experiment with are identical to the ones used in Section 4.2, and are described in Table 2.

The memory consumption of all query plans was not meaningfully influenced by data skew, because the two datasets are of equal size and the relational operators in our implementation dynamically allocate memory as needed. Although the aggregate memory demand does not change with skew, the memory needs of individual threads differ and depend on the size of the partitions they are assigned to.

Data skew, however, has a significant effect on the query response time of certain query plans. Table 4 shows the response time for each query (in seconds). For each physical property of R of interest, we report the response time of all query plans in columns 1–3 when S is in random order, and in columns 4–6 when S is presorted on the join key. We highlight the query plan with the fastest response time in bold.

Starting from the hash join query plan, we see that the response time of the hash join query plans actually improves with skew, akin to what has been observed for the single-socket case [6]. Comparing with the uniform dataset, we find that response time improves with skew for all hash-based query plans. The improvement ranges from 1.2× (columns 4–5) to 1.8× (column 3). Breaking this down further by operation, we see that the build phase is nearly unchanged, and all the response time gains come from the probe phase. This improvement is because of CPU caching, as now the most popular items are accessed frequently enough to remain cached. This significantly reduces the number of memory operations needed to complete the hash join: For example, for the case where both R and S tuples are in random physical order (column 1), the number of memory reads is now 0.68 per S tuple, a 2.8× improvement from the 1.90 memory reads per S tuple for the uniform dataset (shown in Figure 1(a)).

Turning our attention to the sort-merge based query plans, we find that they are all negatively impacted by data skew. The MPSM and parallel merge plans (PARSM) are more resilient, but the streaming merge plan (STRSM) is affected to a larger degree. For instance, when both R and S are in random physical order (column 1), the response time of the STRSM query plan is 84.7 seconds, which is 1.35× slower than the 62.7 seconds with the uniform dataset (shown in Figure 1(a)).

The culprit here is the partitioning step that all sort-merge based query plans rely on. Data skew results in skewed partition sizes, and as partitions are assigned to specific threads, this results in a skewed work distribution. Although this effect can be ameliorated by smart partition boundary selection [2, 12] and work stealing, it cannot be eliminated: When a popular data item is in a single partition by itself, it cannot be further subdivided into smaller partitions. In our 80-thread system, some partition will be imbalanced if any key occurs more frequently than $\frac{1}{80} = 1.25\%$ of the time. As the number of threads in a system increases, the probability of having an imbalanced work distribution for a given skewed dataset increases.

To summarize, the hash join is the preferred query plan when there is data skew in a primary key-foreign key join for two reasons. First, frequent accesses to the hottest items of the hash table effectively "pin" them in the cache, significantly reducing memory traffic. This improves the query response time as skew increases. Second, the hash join algorithm results in a balanced work distribution for all threads during the probe phase, regardless of the skew in S and the number of threads. Partitioning-based algorithms, in comparison, cannot guarantee a uniform work distribution, and their response time increases with higher skew when a few threads must handle disproportionate amount of work.

4.4 Impact of tuple size

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

We generate five different datasets to explore how different tuple sizes affect the equi-join query response time. Across all five datasets, we fix the size of the dimension table *R* to be 10GB, and the fact table *S* to be 40GB, and we only vary the tuple size. Tuples in both the dimension table *R* and the fact table *S* have *N* integer attributes, which are r1, r2, ... , r*N*, and s1, s2, ... , s*N*, respectively. The join key is always fixed to eight bytes, and we experiment with tuples as thin as eight bytes (*N*=1) and as wide as 128 bytes (*N*=16). Larger tuple sizes, such as 128 bytes, capture the case when tuples are stored in a row-oriented fashion, as is the case when data is retrieved from a transaction processing engine. A tuple size of eight bytes enables additional performance optimizations by cleverly using SIMD registers to form a bitonic merge network for sort-merge join [22]. All datasets are uniform, and every primary key in *R* always occurs in *S* exactly four times.

For the experiments in this section, we produce plans corresponding to the following SQL query:

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

We only present the case were both the *R* and *S* inputs are in random order, and we plot the results in Figure 3. Each tick on the horizontal axis corresponds to a dataset with a different tuple width. The vertical axis shows the response time, in seconds, of each join query. The query plans that are produced for this query consist of the same operators as the query plans discussed in Section 4.2 and described in Table 2. For the dataset where the tuple width is 8 bytes, all sort-based query plans use an implementation of the SIMD-optimized bitonic sort-merge join algorithm that is described by Kim et al. [22].

We find that the response time for all query plans drops as tuples get wider, but the hash-based query plan retains its advantage across all tuple sizes. The performance gain is primarily due to reduced memory activity that results from better spatial locality. Intuitively, as tuples get wider, the hash join query plan based processes more bytes per hash table lookup, and the sort-based plans process more bytes per tuple copy when sorting in-place. This spatial locality benefit is especially pronounced for the hash join plan: 16.0 billion memory operations take place when the dataset consists of eight-byte tuples, compared with only 2.9 billion memory operations when the same hash join query plan is executed on 128-byte wide tuples.

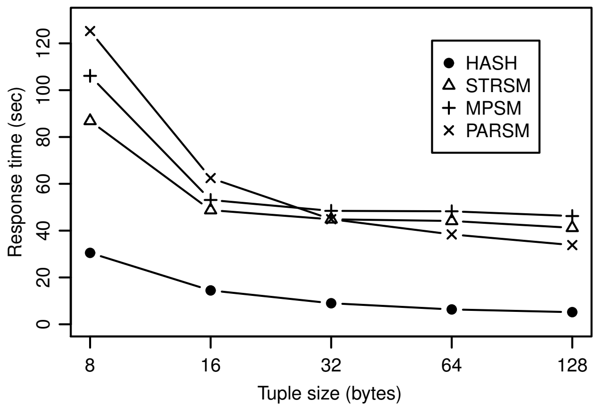


Figure 3: Response time (in seconds) as dataset size remains fixed and tuple size varies between 8 and 128 bytes. *S* and *R* are in random order. When the tuple size is 8 bytes, all sort-based plans ("SM" suffix) use the SIMD-optimized bitonic sort-merge join algorithm by Kim et al. [22] for sorting.

To summarize, regardless of the size of the input tuples, when *R* and *S* are in random join key order, the hash join plan has the fastest response time. The sort-based query plans cannot close the performance gap even when using a SIMD-optimized bitonic sort-merge sorting algorithm [22] that is designed for eight-byte tuples.

4.5 Summary of our findings

The hash join is the best join strategy when tuples in the fact table *S* are randomly ordered. In this case, the hash join outperforms all other algorithms both in terms of response time, and memory consumption. For the uniform dataset, the hash-based join plans have from 2.72× to 4.57× lower response time compared to their fastest sort-based counterpart, while using at least 2.07× less memory. The response time margin widens even further, in favor of the hash join, when the larger table *S* has data skew, as the popular items get cached and memory traffic is reduced. The hash join remains the preferred join algorithm for tuples as wide as 128 bytes.

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

5 Related work

There has been a lot of work on analyzing and comparing the performance of sort-based and hash-based join methods, especially in the context of traditional databases, e.g. [11, 18]. As the hardware landscape evolved, join algorithms became more sophisticated to better take advantage of caching in uniprocessors. Radix join [7, 24] is a cache-friendly join algorithm that performs multiple passes over the data until they can fit in the cache. Chen et al. [9] optimize hash joins by inspecting the data to choose the most efficient cache-friendly algorithm for the probe phase [8].

With the advent of multi-core CPUs, there was a renewed interest in parallel join algorithms. Kim et al. [22] compare a parallel sort-merge join with a parallel radix join. Their study relies on an analytical model and raises interesting issues about the impact of SIMD register lengths on the relative performance of the sort-merge join versus the hash-join in the future. As the SIMD registers have doubled in size with the introduction of the AVX extension, revisiting the use of SIMD for join processing is a promising direction that complements our work. Blanas et al. [6] argue for a hash join algorithm that does no partitioning in a single-socket setting. Balkesen et al. [4] demonstrate that hardware-conscious implementations of the non-partitioned and radix partitioned join algorithms can greatly improve performance. Albutiu et al. [2] introduce a NUMA-aware sort-merge algorithm that is designed for a modern multi-socket server and they show that it outperforms the hash join that was proposed in [6].

These has also been work that looks at query execution more broadly. Graefe introduced the widely-used pull-based model for encapsulating parallelism in the Volcano system [17]. More recently, Harizopoulos et al. [20] explore how one can exploit sharing opportunities among multiple queries in the context of a disk-based system. Arumugam et al. [3] experiment with the DataPath engine, which is built around a push-based model. Giannikis et al. [15] created the SharedDB system and demonstrate that sharing opportunities can be exploited for performance in non-OLAP workloads that also do updates. Finally, Neumann [26] leverages the LLVM infrastructure to compile entire queries into a single highly optimized operator to improve the performance of a main-memory engine.

6 Conclusions and future work

In this paper we have shown that it is too early to write off the hash join algorithm for main memory equi-join processing. We have carefully characterized the impact of various input physical properties on the simple hash-based join and three flavors of sort-based equi-join algorithms, and shown that the hash-based join algorithm often answers queries faster than the current state-of-the-art sort-based algorithms. We also characterize the memory footprint that is required to run each join algorithm. When neither input is sorted, we find that the memory footprint of the hash join algorithm is smaller than that of the sort-based algorithms, as the hash join algorithm needs to buffer only one of the two join inputs. Overall, our results find that there is a place for both hash-based and sort-based algorithms in a high performance main-memory data processing engine.

One promising direction for future work is expanding this study to more complex queries with multiple joins. Database management systems traditionally optimize multi-join queries by identifying query plans that maintain "interesting orders" during query execution. An interesting order, for example, is an operator sequence that keeps the input sorted or partitioned on a particular key. The optimizer then estimates the disk access cost of all query plans and picks the best plan for execution. Whether such an optimization process produces efficient query plans for memory-resident input is an open question. Other interesting directions for future work include considering non-equality joins, and further investigating methods to improve the basic join algorithms.

Acknowledgments

We thank the anonymous SoCC reviewers and our shepherd, Zack Ives, who helped us improve this paper with their thorough reviews and suggestions. We also thank Venkatraman Govindaraju for providing the SIMD-optimized implementation of the bitonic merge algorithm for the experiment in Section 4.4. This work was supported in part by a grant from the Microsoft Jim Gray Systems Lab, and in part by the National Science Foundation under grants IIS-0963993 and IIS-1250886.

References

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

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

[3] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. L. Perez. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD*, pages 519–530, 2010.

[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*, 2013.

[5] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, T. T. Li, G. M. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Business analytics in (a) blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.

[6] 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.

[7] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[8] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Inspector joins. In *VLDB*, pages 817–828, 2005.

[9] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3):17, 2007.

[10] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In *DaMoN*, pages 25–34, 2008.

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

[12] 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.

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

[14] G. Fowler, L. C. Noll, and P. Vo. FNV hash. http://www.isthe.com/chongo/tech/comp/fnv/.

[15] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.

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

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

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

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

[20] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

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

[22] 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.

[23] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*, chapter 6.4. Addison-Wesley, 1998.

[24] 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.

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

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

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

[28] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[29] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, 2010.