Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs

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

ABSTRACT

The focus of this paper is on investigating efficient hash join algorithms for modern multi-core processors in main memory environments. This paper dissects each internal phase of a typical hash join algorithm and considers different alternatives for implementing each phase, producing a family of hash join algorithms. Then, we implement these main memory algorithms on two radically different modern multiprocessor systems, and carefully examine the factors that impact the performance of each method.

Our analysis reveals some interesting results – a very simple hash join algorithm is very competitive to the other more complex methods. This simple join algorithm builds a shared hash table and does not partition the input relations. Its simplicity implies that it requires fewer parameter settings, thereby making it far easier for query optimizers and execution engines to use it in practice. Furthermore, the performance of this simple algorithm improves dramatically as the skew in the input data increases, and it quickly starts to outperform all other algorithms. Based on our results, we propose that database implementers consider adding this simple join algorithm to their repertoire of main memory join algorithms, or adapt their methods to mimic the strategy employed by this algorithm, especially when joining inputs with skewed data distributions.

Categories and Subject Descriptors

H.2.4. [**Database Management**]: Systems—*Query processing, Relational databases*

General Terms

Algorithms, Design, Performance

Keywords

hash join, multi-core, main memory

SIGMOD'11, June 12–16, 2011, Athens, Greece.

1. INTRODUCTION

Large scale multi-core processors are imminent. Modern processors today already have four or more cores, and for the past few years Intel has been introducing two more cores per processor roughly every 15 months. At this rate, it is not hard to imagine running database management systems (DBMSs) on processors with hundreds of cores in the near future. In addition, memory prices are continuing to drop. Today 1TB of memory costs as little as $25,000. Consequently, many databases now either fit entirely in main memory, or their working set is main memory resident. As a result, many DBMSs are becoming CPU bound.

In this evolving architectural landscape, DBMSs have the unique opportunity to leverage the inherent parallelism that is provided by the relational data model. Data is exposed by declarative query languages to user applications and the DBMS is free to choose its execution strategy. Coupled with the trend towards impending very large multi-cores, this implies that DBMSs must carefully rethink how they can exploit the parallelism that is provided by the modern multi-core processors, or DBMS performance will stall.

A natural question to ask then is whether there is anything new here. Beginning about three decades ago, at the inception of the field of parallel DBMSs, the database community thoroughly examined how a DBMS can use various forms of parallelism. These forms of parallelism include pure shared-nothing, shared-memory, and shared disk architectures [17]. If the modern multi-core architectures resemble any of these architectural templates, then we can simply adopt the methods that have already been designed.

In fact, to a large extent this is the approach that DBMSs have haven taken towards dealing with multi-core machines. Many commercial DBMSs simply treat a multi-core processor as a symmetric multi-processor (SMP) machine, leveraging previous work that was done by the DBMS vendors in reaction to the increasing popularity of SMP machines decades ago. These methods break up the task of a single operation, such as an equijoin, into disjoint parts and allow each processor (in an SMP box) to work on each part independently. At a high-level, these methods resemble variations of query processing techniques that were developed for parallel shared-nothing architectures [6], but adapted for SMP machines. In most commercial DBMSs, this approach is reflected across the entire design process, ranging from system internals (join processing, for example) to their pricing model, which is frequently done by scaling the SMP pricing model. On the other hand, open-source DBMSs have

largely ignored multi-core processing and generally dedicate a single thread/process to each query.

The design space for modern high performance main memory join algorithms has two extremes. One extreme of this design space focuses on minimizing the number of processor cache misses. The radix-based hash join algorithm [2] is an example of a method in this design class. The other extreme is to focus on minimizing processor synchronization costs. In this paper we propose a "no partitioning" hash join algorithm that does not partition the input relations to embody an example of a method in this later design space.

A crucial question that we ask and answer in this paper is *what is the impact of these two extreme design points in modern multi-core processors for main memory hash join algorithms*. A perhaps surprising answer is that for modern multi-core architectures, in many cases the right approach is to focus on reducing the computation and synchronization costs, as modern processors are very effective in hiding cache miss latencies via simultaneous multi-threading. For example, in our experiments, the "no partitioning" hash join algorithm far outperforms the radix join algorithm when there is skew in the data (which is often the case in practice), even while it incurs many more processor cache and TLB misses. Even with uniform data, the radix join algorithm only outperforms the "no partitioning" algorithm on a modern Intel Xeon when the parameters for the radix join algorithm are set at or near their optimal setting. In contrast, the non-partitioned algorithm is "parameter-free", which is another important practical advantage.

Reflecting on the previous work in this area, one can observe that the database community has focused on optimizing query processing methods to reduce the number of processor cache and TLB misses. We hope that this paper opens up a new discussion on the entire design space for multi-core query processing techniques, and incites a similar examination of other aspects of query processing beyond the single hash join operation that we discuss in this paper.

This paper makes three main contributions. First, we systematically examine the design choices available for each internal phase of a canonical main memory hash join algorithm – namely, the partition, build, and probe phases – and enumerate a number of possible multi-core hash join algorithms based on different choices made in each of these phases. We then evaluate these join algorithms on two radically different architectures and show how the architectural differences can affect performance. Unlike previous work that has often focused on just one architecture, our use of two radically different architectures lets us gain deeper insights about hash join processing on multi-core processors. To the best of our knowledge, this is the first systematic exploration of multiple hash join techniques that spans multi-core architectures.

Second, we show that an algorithm that does not do any partitioning, but simply constructs a single shared hash table on the build relation often outperforms more complex algorithms. This simple "no-partitioning" hash join algorithm is robust to sub-optimal parameter choices by the optimizer, and does not require any knowledge of the characteristics of the input to work well. To the best of our knowledge, this simple hash join technique differs from what is currently implemented in existing DBMSs for multi-core hash join processing, and offers a tantalizingly simple, efficient, and robust technique for implementing the hash join operation.

Finally, we show that the simple "no-partitioning" hash join algorithm takes advantage of intrinsic hardware optimizations to handle skew. As a result, this simple hash join technique often benefits from skew and its relative performance increases as the skew increases! This property is a big advancement over the state-of-the-art methods, as it is important to have methods that can gracefully handle skew in practice [8].

The remainder of this paper is organized as follows: The next section covers background information. The hash join variants are presented in Section 3. Experimental results are described in Section 4, and related work is discussed in Section 5. Finally, Section 6 contains our concluding remarks.

2. THE MULTI-CORE LANDSCAPE

In the last few years alone, more than a dozen different multi-core CPU families have been introduced by CPU vendors. These new CPUs have ranged from powerful dual-CPU systems on the same die to prototype systems of hundreds of simple RISC cores.

This new level of integration has lead to architectural changes with deep impact on algorithm design. Although the first multi-core CPUs had dedicated caches for each core, we now see a shift towards more sharing at the lower levels of the cache hierarchy and consequently the need for access arbitration to shared caches within the chip. A shared cache means better single-threaded performance, as one core can utilize the whole cache, and more opportunities for sharing among cores. However, shared caches also increase conflict cache misses due to false sharing, and may increase capacity cache misses, if the cache sizes don't increase proportionally to the number of cores.

One idea that is employed to combat the diminishing returns of instruction-level parallelism is simultaneous multi-threading (SMT). Multi-threading attempts to find independent instructions across different threads of execution, instead of detecting independent instructions in the same thread. This way, the CPU will schedule instructions from each thread and achieve better overall utilization, increasing throughput at the expense of per-thread latency.

We briefly consider two modern architectures that we subsequently use for evaluation. At one end of the spectrum, the Intel Nehalem family is an instance of Intel's latest microarchitecture that offers high single-threaded performance because of its out-of-order execution and on-demand frequency scaling (TurboBoost). Multi-threaded performance is increased by using simultaneous multi-threading (Hyper-Threading). At the other end of the spectrum, the Sun UltraSPARC T2 has 8 simple cores that all share a single cache. This CPU can execute instructions from up to 8 threads per core, or a total of 64 threads for the entire chip, and extensively relies on simultaneous multi-threading to achieve maximum throughput.

3. HASH JOIN IMPLEMENTATION

In this section, we consider the anatomy of a canonical hash join algorithm, and carefully consider the design choices that are available in each internal phase of a hash join algorithm. Then using these design choices, we categorize various previous proposals for multi-core hash join processing. In the following discussion we also present information about some of the implementation details, as they often have a significant impact on the performance of the technique that is described.

A hash join operator works on two input relations, R and S. We assume that $|R| < |S|$. A typical hash join algorithm has three phases: partition, build, and probe. The partition phase is optional and divides tuples into distinct sets using a hash function on the join key attribute. The build phase scans the relation R and creates an in-memory hash table on the join key attribute. The probe phase scans the relation S, looks up the join key of each tuple in the hash table, and in the case of a match creates the output tuple(s).

Before we discuss the alternative techniques that are available in each phase of the join algorithm, we briefly digress to discuss the impact of the latch implementation on the join techniques. As a general comment, we have found that the latch implementation has a crucial impact on the overall join performance. In particular, when using the pthreads mutex implementation, several instructions are required to acquire and release an uncontended latch. If there are millions of buckets in a hash table, then the hash collision rate is small, and one can optimize for the expected case: latches being free. Furthermore, pthread mutexes have significant memory footprint as each requires approximately 40 bytes. If each bucket stores a few <key, record-id> pairs, then the size of the latch array may be greater than the size of the hash table itself. These characteristics make mutexes a prohibitively expensive synchronization primitive for buckets in a hash table. Hence, we implemented our own 1-byte latch for both the Intel and the Sun architectures, using the atomic primitives *xchgb* and *ldstub*, respectively. Protecting multiple hash buckets with a single latch to avoid cache thrashing did not result in significant performance improvements even when the number of partitions was high.

3.1 Partition phase

The partition phase is an optional step of a hash join algorithm, if the hash table for the relation R fits in main memory. If one partitions both the R and S relations such that each partition fits in the CPU cache, then the cache misses that are otherwise incurred during the subsequent build and probe phases are almost eliminated. The cost for partitioning both input relations is incurring additional memory writes for each tuple. Work by Shatdal et al. [16] has shown that the runtime cost of the additional memory writes during partitioning phase is less than the cost of missing in the cache – as a consequence partitioning improves overall performance. Recent work by Cieslewicz and Ross [4] has explored partitioning performance in detail. They introduce two algorithms that process the input once in a serial fashion and do not require any kind of global knowledge about the characteristics of the input. Another recent paper [11] describes a parallel implementation of radix partitioning [2] which gives impressive performance improvements on a modern multi-core system. This implementation requires that the entire input is available upfront and will not produce any output until the last input tuple has been seen. We experiment with all of these three partitioning algorithms, and we briefly summarize each implementation in Sections 3.1.1 and 3.1.2.

In our implementation, a partition is a linked list of output buffers. An output buffer is fully described by four elements: an integer specifying the size of the data block, a pointer to the start of the data block, a pointer to the free space inside the data block and a pointer to the next output buffer that is initially set to zero. If a buffer overflows, then we add an empty output buffer at the start of the list, and we make its next pointer point to the buffer that overflowed. Locating free space is a matter of checking the first buffer in the list.

Let p denote the desired number of partitions and n denote the number of threads that are processing the hash join operation. During the partitioning phase, all threads start reading tuples from the relation R, via a cursor. Each thread works on a large batch of tuples at a time, so as to minimize synchronization overheads on the input scan cursor. Each thread examines a tuple, then extracts the key k, and finally computes the partitioning hash function $h_p(k)$. Next, it then writes the tuple to partition $R_{h_p(k)}$ using one of the algorithms we describe below. When the R cursor runs out of tuples, the partitioning operation proceeds to process the tuples from the S relation. Again, each tuple is examined, the join key k is extracted and the tuple is written to the partition $S_{h_p(k)}$. The partitioning phase ends when all the S tuples have been partitioned.

Note that we classify the partitioning algorithms as "non-blocking" if they produce results on-the-fly and scan the input once, in contrast to a "blocking" algorithm that produces results after buffering the entire input and scanning it more than once. We acknowledge that the join operator overall is never truly non-blocking, as it will block during the build phase. The distinction is that the non-blocking algorithms only block for the time that is needed to scan and process the smaller input, and, as we will see in Section 4.3, this a very small fraction of the overall join time.

3.1.1 Non-blocking algorithms

The first partitioning algorithm creates p shared partitions among all the threads. The threads need to synchronize via a latch to make sure that the writes to a shared partition are isolated from each other.

The second partitioning algorithm creates $p * n$ partitions in total and each thread is assigned a private set of p partitions. Each thread then writes to its local partitions without any synchronization overhead. When the input relation is depleted, all threads synchronize at a barrier to consolidate the $p * n$ partitions into p partitions.

The benefit of creating private partitions is that there is no synchronization overhead on each access. The drawbacks, however, are (a) many partitions are created, possibly so many that the working set of the algorithm no longer fits in the data cache and the TLB; (b) at the end of the partition phase some thread has to chain n private partitions together to form a single partition, but this operation is quick and can be parallelized.

3.1.2 Blocking algorithm

Another partitioning technique is the parallel multi-pass radix partitioning algorithm described by Kim et al. [11]. The algorithm begins by having the entire input available in a contiguous block of memory. Each thread is responsible for a specific memory region in that contiguous block. A histogram with $p * n$ bins is allocated and the input is then scanned twice. During the first scan, each thread scans all the tuples in the memory region assigned to it, extracts the key k and then computes the exact histogram of the hash values $h_p(k)$ for this region. Thread $i \in [0, n-1]$ stores the number of tuples it encountered that will hash to partition $j \in [0, p-1]$ in histogram bin $j*n+i$. At the end of the scan, all the n threads compute the prefix sum on the histogram

in parallel. The prefix sum can now be used to point to the beginning of each output partition for each thread in the single shared output buffer. Finally, each thread performs a second scan of its input region, and uses h_p to determine the output partition. This algorithm is recursively applied to each output partition for as many passes as requested.

The benefit of radix partitioning is that it makes few cache and TLB misses, as it bounds the number of output destinations in each pass. This particular implementation has the benefit that, by scanning the input twice for each pass, it computes exactly how much output space will be required for each partition, and hence avoids the synchronization overhead that is associated with sharing an output buffer. Apart from the drawbacks that are associated with any blocking algorithm when compared to a non-blocking counterpart, this implementation also places a burden on the previous operator in a query tree to produce the compact and contiguous output format that the radix partitioning requires as input. Efficiently producing a single shared output buffer is a problem that has been studied before [5].

3.2 Build phase

The build phase proceeds as follows: If the partition phase was omitted, then all the threads are assigned to work on the relation R. If partitioning was done, then each thread i is assigned to work on partitions $R_{i+0*n}, R_{i+1*n}, R_{i+2*n}$, etc. For example, a machine with four cores has $n = 4$, and thread 0 would work on partitions $R_0, R_4, R_8, ...$, thread 1 on $R_1, R_5, R_9, ...$, etc.

Next, an empty hash table is constructed for each partition of the input relation R. To reduce the number of cache misses that are incurred during the next (probe) phase, each bucket of this hash table is sized so that it fits on a few cache lines. Each thread scans every tuple t in its partition, extracts the join key k, and then hashes this key using a hash function $h(\cdot)$. Then, the tuple t is appended to the end of the hash bucket $h(k)$, creating a new hash bucket if necessary. If the partition phase was omitted, then all the threads share the hash table, and writes to each hash bucket have to be protected by a latch. The build phase is over when all the n threads have processed all the assigned partitions.

3.3 Probe phase

The probe phase schedules work to the n threads in a manner similar to the scheduling during the build phase, described above. Namely, if no partitioning has been done, then all the threads are assigned to S, and they synchronize before accessing the read cursor for S. Otherwise, the thread i is assigned to partitions S_{i+0*n}, S_{i+1*n}, S_{i+2*n}, etc.

During the probe phase, each thread reads every tuple s from its assigned partition and extracts the key k. It then checks if the key of each tuple r stored in hash bucket $h(k)$ matches k. This check is necessary to filter out possible hash collisions. If the keys match, then the tuples r and s are joined to form the output tuple. If the output is materialized, it is written to an output buffer that is private to the thread.

Notice that there is parallelism even inside the probe phase: looking up the key for each tuple r in a hash bucket and comparing it to k can be parallelized with the construction of the output tuple, which primarily involves shuffling bytes from tuples r and s. (See Section 4.10 for an experiment that explores this further.)

3.4 Hash Join Variants

The algorithms presented above outline an interesting design space for hash join algorithms. In this paper, we focus on the following four hash join variations:

1. **No partitioning join**: An implementation where partitioning is omitted. This implementation creates a shared hash table in the build phase.
2. **Shared partitioning join**: The first non-blocking partitioning algorithm of Section 3.1.1, where all the threads partition both input sources into shared partitions. Synchronization through a latch is necessary before writing to the shared partitions.
3. **Independent partitioning join**: The second non-blocking partitioning algorithm of Section 3.1.1, where all the threads partition both sources and create private partitions.
4. **Radix partitioning join**: An implementation where each input relation is stored in a single, contiguous memory region. Then, each thread participates in the radix partitioning, as described in Section 3.1.2.

4. EXPERIMENTAL EVALUATION

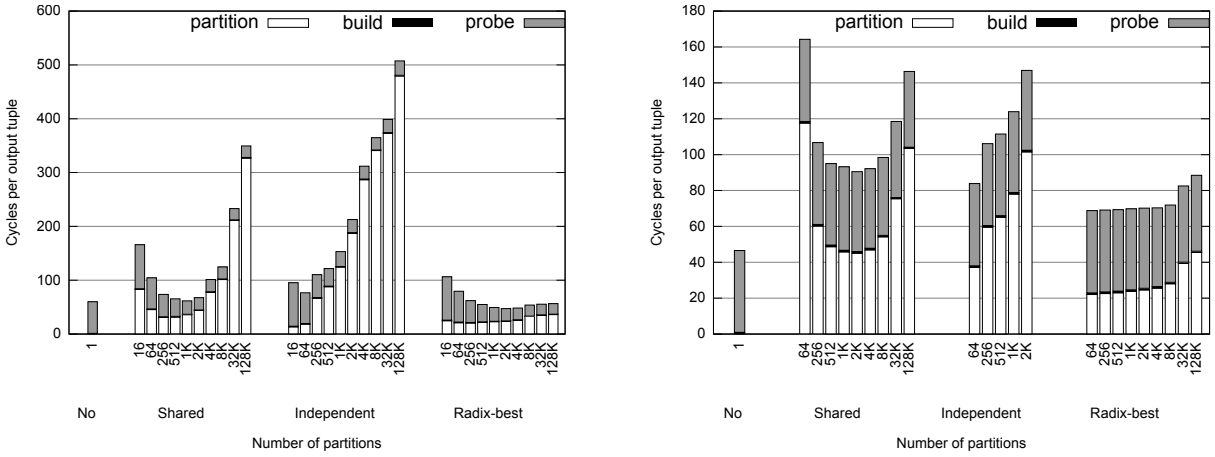
We have implemented the hash join algorithms described in Section 3.4 in a stand-alone C++ program. The program first loads data from the disk into main memory. Data is organized in memory using traditional slotted pages. The join algorithms are run after the data is loaded in memory. Since the focus of this work in on memory-resident datasets, we do not consider the time to load the data into main memory and only report join completion times.

For our workload, we wanted to simulate common and expensive join operations in decision support environments. The execution of a decision support query in a data warehouse typically involves multiple phases. First, one or more dimension relations are reduced based on the selection constraints. Then, these dimension relations are combined into an intermediate one, which is then joined with a much larger fact relation. Finally, aggregate statistics on the join output are computed and returned to the user. For example, in the TPC-H decision support benchmark, this execution pattern is encountered in at least 15 of the 22 queries.

We try to capture the essence of this operation by focusing on the most expensive component, namely the join operation between the intermediate relation R (the outcome of various operations on the dimension relations) with a much larger fact relation S. To allow us to focus on the core join performance, we initially do not consider the cost of materializing

	Intel Nehalem
CPU	Xeon X5650 @ 2.67GHz
Cores	6
Contexts per core	2
Cache size, sharing	12MB L3, shared
Memory	3x 4GB DDR3
	Sun UltraSPARC T2
CPU	UltraSPARC T2 @ 1.2GHz
Cores	8
Contexts per core	8
Cache size, sharing	4MB L2, shared
Memory	8x 2GB DDR2

Table 1: Platform characteristics.



(a) Intel Nehalem (b) Sun UltraSPARC T2

Figure 1: Cycles per output tuple for the uniform dataset.

the output in memory, adopting a similar method as previous work [7, 11]. In later experiments (see Section 4.8), we consider the effect of materializing the join result – in these cases, the join result is created in main memory and not flushed to disk.

We describe the synthetic datasets that we used in the next section (Section 4.1). In Section 4.2 we give details about the hardware that we used for our experiments. We continue with a presentation of the results in Sections 4.3 and 4.4. We analyze the results further in Sections 4.5 through 4.7. We present results investigating the effect of output materialization, and the sensitivity to input sizes and selectivities in Sections 4.8 through 4.10.

4.1 Dataset

We experimented with three different datasets, which we denote as *uniform*, *low skew* and *high skew*, respectively. We assume that the relation R contains the primary key and the relation S contains a foreign key referencing tuples in R. In all the datasets we fix the cardinalities of R to 16M tuples and S to 256M tuples[1]. We picked the ratio of R to S to be 1:16 to mimic the common decision support settings. We experiment with different ratios in Section 4.9.

In our experiments both keys and payloads are eight bytes each. Each tuple is simply a <key, payload> pair, so tuples are 16 bytes long. Keys can either be the values themselves, if the key is numeric, or an 8-byte hash of the value in the case of strings. We chose to represent payloads as 8 bytes for two reasons: (a) Given that columnar storage is commonly used in data warehouses, we want to simulate storing <key, value> or <key, record-id> pairs in the hash table, and (b) make comparisons with existing work (i.e. [11, 4]) easier. Exploring alternative ways of constructing hash table entries is not a focus of this work, but has been explored before [15].

For the uniform dataset, we create tuples in the relation S such that each tuple matches every key in the relation R with equal probability. For the skewed datasets, we added skew to the distribution of the foreign keys in the relation S. (Adding skew to the relation R would violate the primary key constraint.) We created two skewed datasets, for two different s values of the Zipf distribution: low skew with $s = 1.05$ and high skew with $s = 1.25$. Intuitively, the most popular key appears in the low skew dataset 8% of the time, and the ten most popular keys account for 24% of the keys. In comparison, in the high skew dataset, the most popular key appears 22% of the time, and the ten most popular keys appear 52% of the time.

In all the experiments, the hash buckets that are created during the build phase have a fixed size: they always have 32 bytes of space for the payload, and 8 bytes are reserved for the pointer that points to the next hash bucket in case of overflow. These numbers were picked so that each bucket fits in a single, last-level cache line for both the architectures. We size the hash table appropriately so that no overflow occurs.

4.2 Platforms

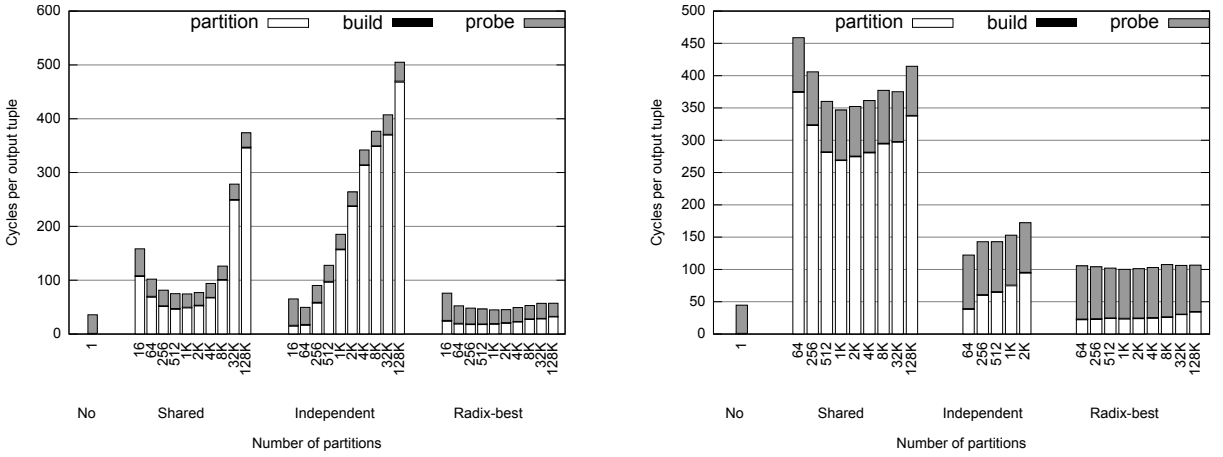
We evaluated our methods on two different architectures: the Intel Nehalem and the Sun UltraSPARC T2. We describe the characteristics of each architecture in detail below, and we summarize key parameters in Table 1.

The Intel Nehalem microarchitecture is the successor of the Intel Core microarchitecture. All Nehalem-based CPUs are superscalar processors and exploit instruction-level parallelism by using out-of-order execution. The Nehalem family supports multi-threading, and allows two contexts to execute per core.

For our experiments, we use the six-core Intel Xeon X5650 that was released in Q1 of 2010. This CPU has a unified 12MB, 16-way associative L3 cache with a line size of 64 bytes. This L3 cache is shared by all twelve contexts executing on the six cores. Each core has a private 256KB, 8-way associative L2 cache, with a line size of 64 bytes. Finally, private 32KB instruction and data L1 caches connect to each core's load/store units.

The Sun UltraSPARC T2 was introduced in 2007 and relies heavily on multi-threading to achieve maximum throughput. An UltraSPARC T2 chip has eight cores and each core has hardware support for eight contexts. UltraSPARC T2 does not feature out-of-order execution. Each core has a single instruction fetch unit, a single floating point unit, a single memory unit and two arithmetic units. At every cycle, each core executes at most two instructions, each taken from two different contexts. Each context is scheduled in a round-robin fashion every cycle, unless the context has ini-

[1]Throughout the paper, M=2^{20} and K=2^{10}.



(a) Intel Nehalem (b) Sun UltraSPARC T2

Figure 2: Cycles per output tuple for the low skew dataset.

tiated a long-latency operation, such as a memory load that caused a cache miss, and has to wait for the outcome.

At the bottom of the cache hierarchy of the UltraSPARC T2 chip lies a shared 4MB, 16-way associative write-back L2 cache, with a line size of 64 bytes. To maximize throughput, the shared cache is physically split into eight banks. Therefore, up to eight cache requests can be handled concurrently, provided that each request hits a different bank. Each core connects to this shared cache through a non-blocking, pipelined crossbar. Finally, each core has a 8KB, 4-way associative write-through L1 data cache with 16 bytes per cache line that is shared by all the eight hardware contexts. Overall, in the absence of arbitration delays, the L2 cache hit latency is 20 cycles.

4.3 Results

We start with the uniform dataset. In Figure 1, we plot the average number of CPU cycles that it takes to produce one output tuple, without actually writing the output, for a varying number of partitions. (Note that to convert the CPU cycles to wall clock time, we simply divide the CPU cycles by the corresponding clock rate shown in Table 1). The horizontal axis shows the different join algorithms (bars "No", "Shared", "Independent"), corresponding to the first three hash join variants described in Section 3.4. For the radix join algorithm, we show the best result across any number of passes (bars marked "Radix-best"). Notice that we assume that the optimizer will always be correct and pick the optimal number of passes.

Overall, the build phase takes a very small fraction of the overall time, regardless of the partitioning strategy that is being used, across all architectures (see Figure 1). The reason for this behavior is two-fold. First and foremost, the smaller cardinality of the R relation translates into less work during the build phase. (We experiment with different cardinality ratios in Section 4.9.) Second, building a hash table is a really simple operation: it merely involves copying the input data into the appropriate hash bucket, which incurs a lot less computation than the other steps, such as the output tuple reconstruction that must take place in the probe phase. The performance of the join operation is therefore mostly determined by the time spent partitioning the input relations and probing the hash table.

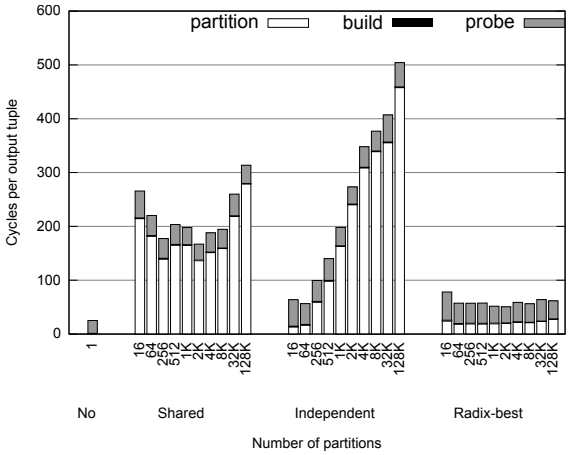
As can be observed in Figure 1(a) for the Intel Nehalem architecture, the performance of the non-partitioned join algorithm is comparable to the optimal performance achieved by the partition-based algorithms. The shared partitioning algorithm performs best when sizing partitions so that they fit in the last level cache. This figure reveals a problem with the independent partitioning algorithm. For a high number of partitions, say 128K, each thread will create its own private buffer, for a total of $128\text{K} * 12 \approx 1.5$ million output buffers. This high number of temporary buffers introduces two problems. First, it results in poor space utilization, as most of these buffers are filled with very few tuples. Second, the working set of the algorithm grows tremendously, and keeping track of 1.5 million cache lines requires a cache whose capacity is orders of magnitude larger than the 12MB L3 cache. The radix partitioning algorithm is not affected by this problem, because it operates in multiple passes and limits the number of partition output buffers in each pass.

Next, we experimented with the Sun UltraSPARC T2 architecture. In Figure 1(b) we see that doing no partitioning is at least 1.5X faster compared to all the other algorithms. The limited memory on this machine prevented us from running experiments with a high number of partitions for the independent partitioning algorithm because of the significant memory overhead discussed in the previous paragraph. As this machine supports nearly five times more hardware contexts than the Intel machine, the memory that is required for bookkeeping is five times higher as well.

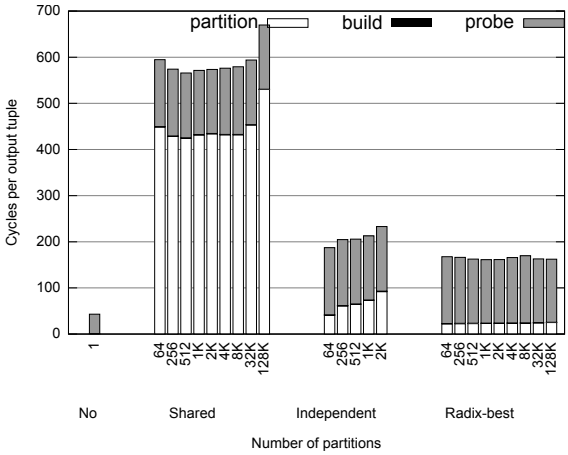
To summarize our results with the uniform dataset, we see that on the Intel architecture the performance of the no partitioning join algorithm is comparable to the performance of all the other algorithms. For the Sun UltraSPARC T2, we see that the no partitioning join algorithm outperforms the other algorithms by at least 1.5X. Additionally, the no partitioning algorithm is more robust, as the performance of the other algorithms degrades if the query optimizer does not pick the optimal value for the number of partitions.

4.4 Effect of skew

We now consider the case when the distribution of foreign keys in the relation S is skewed. We again plot the average time to produce each tuple of the join (in machine cycles) in Figure 2 for the low skew dataset, and in Figure 3 for the high skew dataset.



(a) Intel Nehalem



(b) Sun UltraSPARC T2

Figure 3: Cycles per output tuple for the high skew dataset.

	Intel Nehalem	Sun UltraSPARC T2
NO	No / 1	No / 1
SN	Indep. / 16	Indep. / 64
L2-S	Shared / 2048	Shared / 2048
L2-R	Radix / 2048	Radix / 2048

Table 2: Shorthand notation and corresponding partitioning strategy / number of partitions.

By comparing Figure 1 with Figure 2, we notice that, when using the shared hash table (bar "No" in all graphs), performance actually improves in the presence of skew! On the other hand, the performance of the shared partitioning algorithm degrades rapidly with increasing skew, while the performance of the independent partitioning and the radix partitioning algorithms shows little change on the Intel Nehalem and degrades on the Sun UltraSPARC T2. Moving to Figure 3, we see that the relative performance of the non-partitioned join algorithm increases rapidly under higher skew, compared to the other algorithms. The non-partitioned algorithm is generally 2X faster than the other algorithms on the Intel Nehalem, and more than 4X faster than the other algorithms on the Sun UltraSPARC T2.

To summarize these results, skew in the underlying join key values (data skew) manifests itself as partition size skew when using partitioning. For the shared partitioning algorithm, during the partition phase, skew causes latch contention on the partition with the most popular key(s). For all partitioning-based algorithms, during the probe phase, skew translates into a skewed work distribution per thread. Therefore, the overall join completion time is determined by the completion time of the partition with the most popular key. (We explore this behavior further in Section 4.7.1.) On the other hand, skew improves performance when sharing the hash table and not doing partitioning for two reasons. First, the no partitioning approach ensures an even work distribution per thread as all the threads are working concurrently on the single partition. This greedy scheduling strategy proves to be effective in hiding data skew. Second, performance increases because the hardware handles skew a lot more efficiently, as skewed memory access patterns cause significantly fewer cache misses.

		Cycles	L3 miss	Instruc--tions	TLB load miss	TLB store miss
NO	partition	0	0	0	0	0
	build	322	2	2,215	1	0
	probe	15,829	862	54,762	557	0
SN	partition	3,578	18	29,096	6	2
	build	328	8	2,064	0	0
	probe	21,717	866	54,761	505	0
L2-S	partition	11,778	103	31,117	167	257
	build	211	1	2,064	0	0
	probe	6,144	35	54,762	1	0
L2-R	partition	6,343	221	34,241	7	237
	build	210	1	2,064	0	0
	probe	6,152	36	54,761	1	0

Table 3: Performance counter averages for the uniform dataset (millions).

4.5 Performance counters

Due to space constraints, we focus on specific partitioning configurations from this section onward. We use "NO" to denote the no partitioning strategy where the hash table is shared by all threads, and we use "SN" to denote the case when we create as many partitions as hardware contexts (join threads), except we round the number of partitions up to the next power of two as is required for the radix partitioning algorithm. We use "L2" to denote the case when we create partitions to fit in the last level cache, appending "-S" when partitioning with shared output buffers, and "-R" for radix partitioning. We summarize this notation in Table 2. Notice that the L2 numbers correspond to the best performing configuration settings in the experiment with the uniform dataset (see Figure 1).

We now use the hardware performance counters to understand the characteristics of these join algorithms. In the interest of space, we only present our findings from a single architecture: the Intel Nehalem. We first show the results from the uniform dataset in Table 3. Each row indicates one particular partitioning algorithm and join phase, and each column shows a different architectural event. First, notice the code path length. It takes, on average, about 55 billion instructions to complete the probe phase and an additional 50% to 65% of that for partitioning, depending on the algorithm of choice. The NO algorithm pays a high cost in

		Cycles	L3 miss	Instruc--tions	TLB load miss	TLB store miss
NO	partition	0	0	0	0	0
	build	323	3	2,215	1	0
	probe	6,433	98	54,762	201	0
SN	partition	3,577	17	29,096	6	1
	build	329	8	2,064	0	0
	probe	13,241	61	54,761	80	0
L2-S	partition	36,631	79	34,941	67	106
	build	210	5	2,064	0	0
	probe	8,024	13	54,762	1	0
L2-R	partition	5,344	178	34,241	5	72
	build	209	4	2,064	0	0
	probe	8,052	13	54,761	1	0

Table 4: Performance counter averages for the high skew dataset (millions).

terms of the L3 cache misses during the probe phase. The partitioning phase of the SN algorithm is fast but fails to contain the memory reference patterns that arise during the probe phase in the cache. The L2-S algorithm manages to minimize these memory references, but incurs a high L3 and TLB miss ratio during the partition phase compared to the NO and SN algorithms. The L2-R algorithm uses multiple passes to partition the input and carefully controls the L3 and TLB misses during these phases. Once the cache-sized partitions have been created, we see that both the L2-S and L2-R algorithms avoid incurring many L3 and TLB misses during the probe phase. In general, we see fewer cache and TLB misses across all algorithms when adding skew (in Table 4).

Unfortunately, interpreting performance counters is much more challenging with modern multi-core processors and will likely get worse. Processors have become a lot more complex over the last ten years, yet the events that counters capture have hardly changed. This trend causes a growing gap between the high-level algorithmic insights the user expects and the specific causes that trigger some processor state that the performance counters can capture. In a uniprocessor, for example, a cache miss is an indication that the working set exceeds the cache's capacity. The penalty is bringing the data from memory, an operation the costs hundreds of cycles. However, in a multi-core processor, a memory load might miss in the cache because the operation touches memory that some other core has just modified. The penalty in this case is looking in some other cache for the data. Although a neighboring cache lookup can be ten or a hundred times faster than bringing the data from memory, both scenarios will simply increment the cache miss counter and not record the cause of this event.

To illustrate this point, let's turn our attention to a case in Table 3 where the performance counter results can be misleading: The probe phase of the SN algorithm has slightly fewer L3 and TLB misses than the probe phase of the NO algorithm and equal path length, so the probe phase of the SN algorithm should be comparable or faster than probe phase of the NO algorithm. However, the probe phase of the NO algorithm is almost 25% faster! Another issue is latch contention, which causes neither L3 cache misses nor TLB misses, and therefore is not reported in the performance counters. For example, when comparing the uniform and high skew numbers for the L2-S algorithm, the number of the L3 cache misses during the high skew experiment is

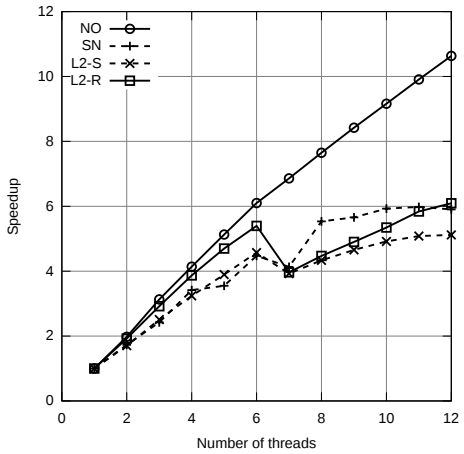


Figure 4: Speedup over single threaded execution, uniform dataset.

30% lower than the number of the cache misses observed during the uniform experiment. However, partitioning performance worsens by more than 3X when creating shared partitions under high skew!

The performance counters don't provide clean insights into why the non-partitioned algorithm exhibits similar or better performance than the other cache-efficient algorithms across all datasets. Although a cycle breakdown is still feasible at a macroscopic level where the assumption of no contention holds (for example as in Ailamaki et al. [1]), this experiment reveals that blindly assigning fixed cycle penalties to architectural events can lead to misleading conclusions.

4.6 Speedup from SMT

Modern processors improve the overall efficiency with hardware multithreading. Simultaneous multi-threading (SMT) permits multiple independent threads of execution to better utilize the resources provided by modern processor architectures. We now evaluate the impact of SMT on the hash join algorithms.

We first show a speedup experiment for the Intel Nehalem on the uniform dataset in Figure 4. We start by dedicating each thread to a core, and once we exceed the number of available physical cores (six for our Intel Nehalem), we then start assigning threads in a round-robin fashion to the available hardware contexts. We observe that the algorithms behave very differently when some cores are idle (fewer than six threads) versus in the SMT region (more than six threads). With fewer than six threads all the algorithms scale linearly, and the NO algorithm has optimal speedup. With more than six threads, the NO algorithm continues to scale, becoming almost 11X faster than the single-threaded version when using all available contexts. The partitioning-based algorithms SN, L2-S and L2-R, however, do not exhibit this behavior. The speedup curve for these three algorithms in the SMT region either flattens completely (SN algorithm), or increases at a reduced rate (L2-R algorithm) than the non-SMT region. In fact, performance drops for all partitioning algorithms for seven threads because of load imbalance: a single core has to do the work for two threads. (This imbalance can be ameliorated through load balancing, a technique that we explore in Section 4.7.1.)

	Uniform		
	6 threads	12 threads	Improvement
NO	28.23	16.15	1.75X
SN	34.04	25.62	1.33X
L2-S	19.27	18.13	1.06X
L2-R	14.46	12.71	1.14X
	High skew		
	6 threads	12 threads	Improvement
NO	9.34	6.76	1.38X
SN	19.50	17.15	1.14X
L2-S	38.37	44.87	0.86X
L2-R	15.04	13.61	1.11X

Table 5: Simultaneous multi-threading experiment on the Intel Nehalem, showing billions of cycles to join completion and relative improvement.

	Uniform		
	8 threads	64 threads	Improvement
NO	37.30	12.64	2.95X
SN	55.70	22.25	2.50X
L2-S	51.62	23.86	2.16X
L2-R	46.62	18.88	2.47X
	High skew		
	8 threads	64 threads	Improvement
NO	23.92	11.67	2.05X
SN	70.52	49.54	1.42X
L2-S	73.91	221.01	0.33X
L2-R	66.01	43.16	1.53X

Table 6: Simultaneous multi-threading experiment on the Sun UltraSPARC T2, showing billions of cycles to join completion and relative improvement.

We summarize the benefit of SMT in Table 5 for the Intel architecture, and in Table 6 for the Sun architecture. For the Intel Nehalem and the uniform dataset, the NO algorithm benefits significantly from SMT, becoming 1.75X faster. This algorithm is not optimized for cache performance, and as seen in Section 4.5, causes many cache misses. As a result, it provides more opportunities for SMT to efficiently overlap the memory accesses. On the other hand, the other three algorithms are optimized for cache performance to different degrees. Their computation is a large fraction of the total execution time, therefore they do not benefit significantly from using SMT. In addition, we notice that the NO algorithm is around 2X slower than the L2-R algorithm without SMT, but its performance increases to almost match the L2-R algorithm performance with SMT.

For the Sun UltraSPARC T2, the NO algorithm also benefits the most from SMT. In this architecture the code path length (i.e. instructions executed) has a direct impact on the join completion time, and therefore the NO algorithm performs best both with and without SMT. As the Sun machine cannot exploit instruction parallelism at all, we see increased benefits from SMT compared to the Intel architecture.

When comparing the high skew dataset with the uniform dataset across both architectures, we see that the improvement of SMT is reduced. The skewed key distribution incurs fewer cache misses, therefore SMT loses opportunities to hide processor pipeline stalls.

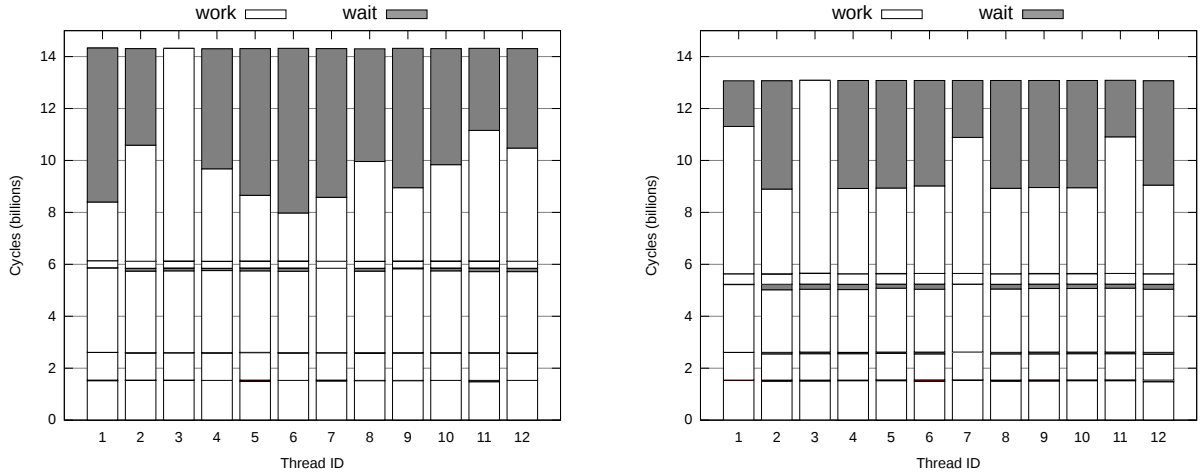
4.7 Synchronization

Synchronization is used in multithreaded programs to guarantee the consistency of shared data structures. In our join implementations, we use barrier synchronization when all the threads wait for tasks to be completed before they can proceed to the next task. (For example, at the end of each pass of the radix partition phase, each thread has to wait until all other threads complete before proceeding.) In this section, we study the effect of barrier synchronization on the performance of the hash join algorithm. In the interest of space, we only present results for the Intel Nehalem machine. Since the radix partitioning algorithm wins over the other partitioning algorithms across all datasets, our discussion only focuses on results for the non-partitioned algorithm (NO) and the radix partitioning algorithm (L2-R).

Synchronization has little impact on the non-partitioned (NO) algorithm for both the uniform and the high skew datasets, regardless of the number of threads that are running. The reason for this behavior is the simplicity of the NO algorithm. First, there is no partition phase at all, and each thread can proceed independently in the probe phase. Therefore synchronization is only necessary during the build phase, a phase that takes less than 2% of the total time (see Figure 1). Second, by dispensing with partitioning, this algorithm ensures an even distribution of work across the threads, as all the threads are working concurrently on the single shared hash table.

We now turn our attention to the radix partitioning algorithm, and break down the time spent by each thread. Unlike the non-partitioned algorithm, the radix partitioning algorithm is significantly impacted by synchronization on both the uniform and the high skew datasets. Figure 5(a) shows the time breakdown for the L2-R algorithm when running 12 threads on the Intel Nehalem machine with the high skew dataset. Each histogram in this figure represents the execution flow of a thread. The vertical axis can be viewed as a time axis (in machine cycles). White rectangles in these histograms represent tasks, the position of each rectangle indicates the beginning time of the task, and the height represents the completion time of this task for each thread. The gray rectangles represent the waiting time that is incurred by a thread that completes its task but needs to synchronize with the other threads before continuing. In the radix join algorithm, we can see five expensive operations that are synchronized through barriers: (1) computing the thread-private histogram, (2) computing the global histogram, (3) doing radix partitioning, (4) building a hash table for each partition of the relation R, and (5) probing each hash table with a partition from the relation S. The synchronization cost of the radix partitioning algorithm accounts for nearly half of the total join completion time for some threads.

The synchronization cost is so high under skew primarily because it is hard to statically divide work items into equally-sized subtasks. As a result, faster threads have to wait for slower threads. For example, if threads are statically assigned to work on partitions in the probe phase, the distribution of the work assigned to the threads will invariably also be skewed. Thus, the thread processing the partition with the most popular key becomes a bottleneck and the overall completion time is determined by the completion time of the partition with the most popular keys. In Figure 5(a), this is thread 3.



(a) High skew dataset (b) High skew dataset with work stealing

Figure 5: Time breakdown of the radix join.

4.7.1 Load balancing

If static work allocation is the problem, then how would the radix join algorithm perform under a dynamic work allocation policy and highly skewed input? To answer this question, we tweaked the join algorithm to allow the faster threads that have completed their probe phase to steal work from other slower threads. In our implementation, the unit of work is a single partition. In doing so, we slightly increase the synchronization cost because work queues need to now be protected with latches, but we balance the load better.

In Figure 5(b) we plot the breakdown of the radix partitioning algorithm (L2-R) using this work stealing policy when running on the Intel Nehalem machine with the high skew dataset. Although the work is now balanced almost perfectly for the smaller partitions, the partitions with the most popular keys are still a bottleneck. In the high skew dataset, the most popular key appears 22% of the time, and thread 3 in this case has been assigned only a single partition which happened to correspond to the most popular key. In comparison, for this particular experiment, the NO algorithm can complete the join in under 7 billion cycles (Table 4), and hence is 1.9X faster. An interesting area for future work is load balancing techniques that permit work stealing at a finer granularity than an entire partition with a reasonable synchronization cost.

To summarize, under skew, a load balancing technique improves the performance of the probe phase but does not address the inherent inefficiency of all the partitioning-based algorithms. In essence, there is a coordination cost to be paid for load balancing, as thread synchronization is necessary. Skew in this case causes contention, stressing the cache coherence protocol and increasing memory traffic. On the other hand, the no partitioning algorithm does skewed memory loads of read-only data, which is handled very efficiently by modern CPUs through caching.

4.8 Effect of output materialization

Early work in main memory join processing [7] did not take into account the cost of materialization. This decision was justified by pointing out that materialization comes at a fixed price for all algorithms and, therefore, a join algorithm will be faster regardless of the output being materialized or discarded. Recent work by Cieslewicz et al. [3] highlighted the trade-offs involved when materializing the output.

Machine	NO	SN	L2-S	L2-R
Intel Nehalem	23%	4%	7%	10%
Sun UltraSPARC T2	29%	21%	20%	23%

Table 7: Additional overhead of materialization with respect to total cycles without materialization on the uniform dataset.

	Scale 0.5	Scale 1	Scale 2
NO	7.65 (0.47X)	16.15 (1.00X)	62.27 (3.86X)
SN	11.76 (0.46X)	25.62 (1.00X)	98.82 (3.86X)
L2-S	8.47 (0.47X)	18.13 (1.00X)	68.48 (3.78X)
L2-R	5.82 (0.46X)	12.71 (1.00X)	DNF

Table 8: Join sensitivity with varying input cardinalities for the uniform dataset on Intel Nehalem. The table shows the cycles for computing the join (in billions) and the relative difference to scale 1.

In Table 7 we report the increase in the total join completion time when we materialize the output in memory for the uniform dataset and the partitioning strategies described in Table 2. If the join operator is part of a complex query plan, it is unlikely that the entire join output will ever need to be written in one big memory block, but, even in this extreme case, we see that no algorithm is being significantly impacted by materialization.

4.9 Cardinality experiments

We now explore how sensitive our findings are to variations in the cardinalities of the two input relations. Table 8 shows the results when running the join algorithms on the Intel Nehalem machine. The numbers obtained from the uniform dataset (described in detail in Section 4.1) are shown in the middle column. We first created one uniform dataset where both relations are half the size (scale 0.5). This means the relation R has 8M tuples and the relation S has 128M tuples. We also created a uniform dataset where both relations are twice the size (scale 2), i.e. the relation R has 32M tuples and the relation S has 512M tuples. The scale 2 dataset occupies 9GB out of the 12GB of memory our system has (Table 1) and leaves little working memory, but the serial

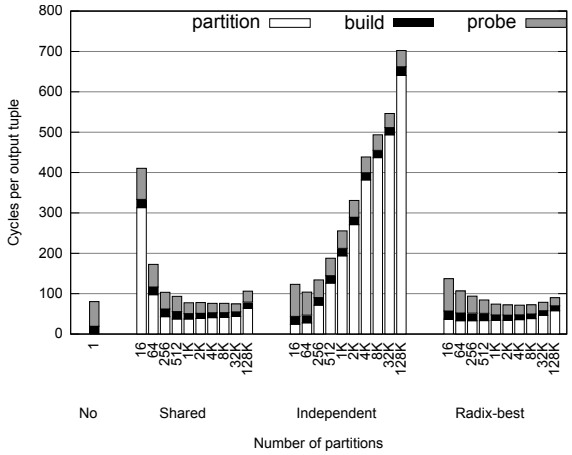


Figure 6: Experiment on Intel Nehalem with uniform dataset and |R|=|S|.

access pattern allows performance to degrade gracefully for all algorithms but the L2-R algorithm. The main memory optimizations of the L2-R algorithm cause many random accesses which hurt performance. We therefore mark the L2-R algorithm as not finished (DNF).

We now examine the impact of the relative size of the relations R and S. We fixed the cardinality of the relation S to be 16M tuples, making $|R| = |S|$, and we plot the cycles per output tuple for the uniform dataset when running on the Intel Nehalem in Figure 6. First, the partitioning time increases proportionally to $|R| + |S|$. Second, the build phase becomes significant, taking at least 25% of the total join completion time. The probe phase, however, is at most 30% slower, and less affected by the cardinality of the relation R. Overall, all the algorithms are slower when $|R| = |S|$ because they have to process more data, but the no partitioning algorithm is slightly favored because it avoids partitioning both input relations.

The results show that no join algorithm is particularly sensitive to our selection of input relation cardinalities, therefore our findings are expected to hold across a broader spectrum of cardinalities. The outcome of the experiments for the Sun UltraSPARC T2 is similar, and is omitted.

4.10 Selectivity experiment

We now turn our attention to how join selectivity affects performance. As all our original datasets are examples of joins between primary and foreign keys, all the experiments that have been presented so far have a selectivity of 100%. For this experiment we created two different S relations that have the same cardinality but only 50% and 12.5% of the tuples join with a tuple in the relation R. The key distribution is uniform.

Results for the Intel Nehalem are shown Figure 7(a). Decreasing join selectivity has a marginal benefit on the probe phase, but the other two phases are unaffected. The outcome of the same experiment on Sun UltraSPARC T2 is shown in Figure 7(b). In this architecture, the benefit of a small join selectivity on the probe phase is significant.

Inspecting the performance counters in this experiment revealed additional insights. Across all the architectures, the code path length (i.e. instructions executed) increases as join selectivity increases. The Intel Nehalem is practically insensitive to different join selectivities, because its out-of-order execution manages to overlap the data transfer with the byte shuffling that is required to assemble the output tuple. On the other hand, for the Sun UltraSPARC T2 machine, there is a strong linear correlation between the code path length and the cycles that are required for the probe phase to complete. The in-order Sun UltraSPARC T2 cannot automatically extract the instruction-level parallelism of the probe phase, unless the programmer explicitly expresses it by using multiple threads.

4.11 Implications

These results imply that DBMSs must reconsider their join algorithms for current and future multi-core processors. First, modern processors are very effective in hiding cache miss latencies through multi-threading (SMT), as it is shown in Tables 5 and 6. Second, optimizing for cache performance requires partitioning, and this has additional computation and synchronization overheads, and necessitates elaborate load balancing techniques to deal with skew. These costs of partitioning on a modern multi-core machine can be higher than the benefit of an increased cache hit rate, especially on skewed datasets (as shown in Figures 2 and 3.) To fully leverage the current and future CPUs, high performance main memory designs have to achieve good cache and TLB performance, while fully exploiting SMT, and minimizing synchronization costs.

5. RELATED WORK

There is a rich history of studying hash join performance for main memory database systems, starting with the early work of DeWitt et al. [7]. A decade later Shatdal et al. [16] studied cache-conscious algorithms for query execution and discovered that the probe phase dominates the overall hash join processing time. They also showed that hash join computation can be sped up if both the build and probe relations are partitioned so as to fit in the cache.

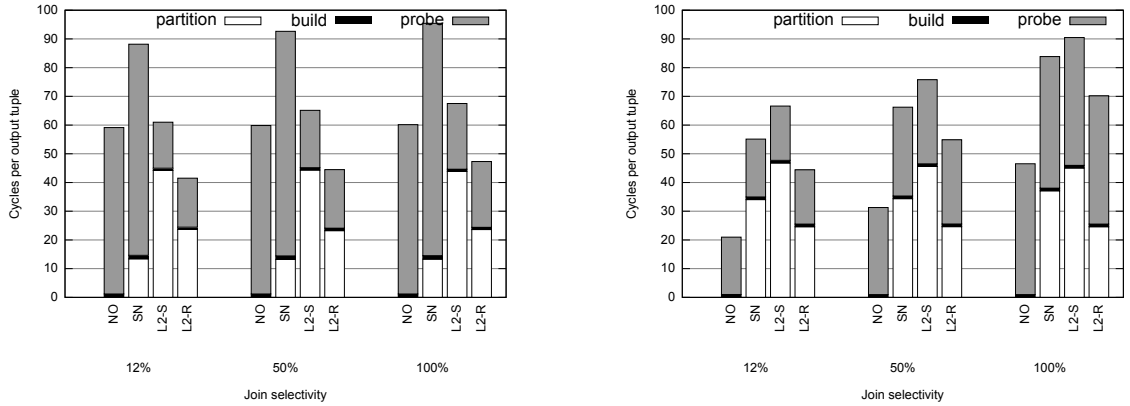
Ailamaki et al. [1] studied where DBMSs spend their time on modern processors, whereas Manegold et al. [12] inspected the time breakdown for a hash join operation. Both papers break down the query execution time by examining performance counters, and single out cache and TLB misses as the two primary culprits for suboptimal performance in main memory processing. A follow-up paper [13] presented a cost model on how to optimize the performance of the radix join algorithm on a uniprocessor [2].

Ross [15] presented a more efficient way to improve the performance of hash joins by using cuckoo hashing [14] and SIMD instructions. Garcia and Korth [9] have studied the benefits of using simultaneous multi-threading for hash join processing. Graefe et al. [10] described how hash-based algorithms can improve the performance of a commercial DBMS.

Finally, there has been prior work in handling skew during hash join processing. The experiments with a high number of partitions that we presented in Section 4.4 are an extension of an idea by DeWitt et al. [8] for a main memory, multi-core environment.

6. CONCLUSIONS AND FUTURE WORK

The rapidly evolving multi-core landscape requires that DBMSs carefully consider the interactions between query processing algorithms and the underlying hardware. In this



(a) Intel Nehalem (b) Sun UltraSPARC T2

Figure 7: Sensitivity to join selectivity. Increasing join selectivity impacts the critical path for the Sun UltraSPARC T2, while the out-of-order execution on Intel Nehalem overlaps computation with data transfer.

paper we examine these interactions when executing a hash join operation in a main memory DBMS. We implement a family of main memory hash join algorithms that vary in the way that they implement the partition, build, and probe phases of a canonical hash join algorithm.

We also evaluate our algorithms on two different multi-core processor architectures. Our results show that a simple hash join technique that does not do any partitioning of the input relations often outperforms the other more complex partitioning-based join alternatives. In addition, the relative performance of this simple hash join technique rapidly improves with increasing skew, and it outperforms every other algorithm in the presence of even small amounts of skew.

Minimizing cache misses requires additional computation, synchronization and load balancing to cope with skew. As our experiments show, these costs on a modern multi-core machine can be higher than the benefit of an increased cache hit rate. To fully leverage the current and future CPUs, high performance main memory designs have to consider how to minimize computation and synchronization costs, and fully exploit simultaneous multi-threading, in addition to maintaining good cache and TLB behavior. While a large part of the previous work in this area has mostly focused on minimizing cache and TLB misses for database query processing tasks, our work here suggests that paying attention to the computation and synchronization costs is also very important in modern processors. This work points to a rich direction for future work in exploring the design of more complex query processing techniques (beyond single joins) that consider the joint impact of computation, synchronization costs, load balancing, and cache behavior.

Acknowledgments

We thank David DeWitt for his deeply insightful comments on this paper. We also thank the reviewers of this paper and Willis Lang for their feedback on an earlier draft of this paper. David Wood and the Wisconsin Multifacet project were invaluable supporters of this project and gave us exclusive access to their hardware, and we thank them. 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 CNS-0551401.

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

[3] J. Cieslewicz, W. Mee, and K. A. Ross. Cache-conscious buffering for database operators with state. In *DaMoN*, pages 43–51, 2009.

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

[5] J. Cieslewicz, K. A. Ross, and I. Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, 2007.

[6] D. J. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad? *SIGMOD Record*, 19(4):104–112, 1990.

[7] 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 Conference*, pages 1–8, 1984.

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

[9] P. Garcia and H. F. Korth. Database hash-join algorithms on multithreaded computer architectures. In *Conf. Computing Frontiers*, pages 241–252, 2006.

[10] G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in Microsoft SQL Server. In *VLDB*, pages 86–97, 1998.

[11] 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.

[12] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *VLDB*, pages 339–350, 2000.

[13] 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.

[14] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[15] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301, 2007.

[16] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, pages 510–521, 1994.

[17] M. Stonebraker. The case for shared nothing. In *HPTS*, 1985.