Analyzing In-Memory Hash Joins: Granularity Matters

Jian Fang
Delft University of Technology
Delft, the Netherlands
j.fang-1@tudelft.nl

Jinho Lee, Peter Hofstee*
IBM Research
*and Delft Univ. of Technology
Austin, TX 78758, USA
leejinho,hofstee@us.ibm.com

Jan Hidders
Vrije Universiteit Brussel
1050 Brussels, Belgium
jan.hidders@vub.be

ABSTRACT

Predicting the performance of join algorithms on modern hardware is challenging. In this work, we focus on main-memory no-partitioning and partitioning hash join algorithms executing on multi-core platforms. We discuss the main parameters impacting performance, and present an effective performance model. This model can be used to select the most appropriate algorithm for different input data-sets for current and future hardware configurations. We find that for modern systems an optimized no-partition hash join often outperforms an optimized radix partitioning hash join.

1. INTRODUCTION

In the past decades, as new computer architectures emerged, the competition between different main-memory hash join algorithms has become more interesting. Significant research [1, 16, 12, 2, 3, 10, 9] has gone into developing algorithms that efficiently utilize the underlying hardware and this has provided a guidance to choosing better performing algorithms on modern hardware. However, the hardware develops so fast that the best algorithm today is probably not the best in the future. One obvious change that may lead to this is increased memory bandwidth.

In the past few years, development of hash join algorithms has been focused on partitioning the data to the size that fits within the last-level cache. However in modern systems, contrary to prior findings, we find that partitioning hash join is not always better than no-partitioning hash join. In this paper, we analyze the performance of hash join algorithms by a bandwidth-driven model and provide a guideline for choosing the right algorithm according to the dataset and architecture characteristics.

The contributions of the paper are as follows:

- We analyze the impact factors for the hash join algorithms. We discuss the importance of granularity.
- We build a performance model that considers both computation and memory accesses.
- Based on the proposed model, we study a no-partitioning hash join and a radix partitioning hash join.
- We predict the performance changes along with the changes of the data set and the hardware.
- We validate the model with hardware-based experiments on two processor architectures.

2. BACKGROUND

See [6] for early work on in-memory databases and hash-join.

2.1 Hash Join Algorithms

Classical hash join The classical hash join contains a build phase and a probe phase. During the build phase, tuples in the smaller relation R are scanned to build a hash table, and assigned to the corresponding hash table bucket. After the hash table is built, the probe phase scans the other relation S. For each tuple in S, it probes the hash table and validates the matches.

No-partitioning hash join No-partitioning hash join [4] is a parallel version of classical hash join. Each relation is equally divided into shares. In the build phase, each share in the smaller relation R is scanned by a thread. Together the threads build a shared hash table. Similarly, in the probe phase, each thread gets a piece of data from relation S and does the probe in parallel.

Partitioning hash join To limit cache misses, [16] introduces the partitioning hash join. Before building the hash table, both relations are divided into small partitions. Hash tables are built separately for each partition of relation R. After the hash table is built for one partition, the corresponding partition in relation S is scanned to do the probe. If the hash table for each partition is small enough to fit in the cache, this reduces cache misses, at the cost of partitioning overhead. The partitioning hash join can be further optimized as the radix partitioning hash join [12] which uses multiple partitioning passes to reduce the translation lookaside buffer (TLB) misses.

2.2 Related Work

The discussion between no-partitioning hash join and partitioning hash join never stops since the underlying hardware develops at a fast pace. Much research effort has been made on tuning the algorithms to leverage the underlying hardware platforms. Prior work [16, 9, 1, 5] shows how to use cache in a more efficient way for partitioning hash

join. Shatdal et al. [16] point out that if the hash table for each partition is small enough to store in the last level cache (LLC), the number of cache misses is reduced. Chen et al. [5] use prefetch to reduce the cache misses. Based on the cache optimization, Kim et al. [9] and Manegold et al. [12] further focus on the TLB problem. They show that using multiple passes partitioning can reduce the TLB misses caused by accessing too many pages in a one pass. Lang et al. [10] propose a NUMA-aware hash join algorithm based on their finding on NUMA effect that unreasonably distributed data in multiple NUMA nodes decreases performance.

Earlier research [2, 3, 4, 15] also makes comparisons between no-partition hash join and partitioning hash join. Blanas et al. [4] evaluate both classes of algorithms on multi-core platforms. They claim that no-partitioning hash join is competitive because partitioning hash join introduces extra cost such as computation for partitioning and synchronization which is higher than the penalty of cache misses in no-partitioning hash join. In [2] and [3], partitioning hash join, categorized as hardware-conscious algorithm, is well configured to compare with no-partitioning hash join which is hardware-oblivious. The authors argue that the partition hash join runs fast in most of the cases, and the hardware-oblivious hash join only performs well when the ratio between the size of two relations is significantly different. However, it is interesting to compare a well-configured partitioning hash join with a optimized no-partitioning hash join such as using prefetch. [15] compares 13 different join algorithms including hash join and sort merge join. Even though partitioning hash join performs better than no partitioning hash join, they point out that there is not a 100% best algorithm.

Another way to compare the algorithms is to build a performance model and use it for prediction. In [14, 13, 8, 7], performance models are set up describing the main cost of different hash join algorithms. These models consider both processing and disk I/O cost. It is increasingly feasible to store the entire database [6] in memory. For this organization, I/O cost is not the dominating part, but memory accesses are. Prior work[11] presents a cost model based on the cache lines transferred, but this work only covers the no-partitioning hash join algorithms. In this paper, we focus on in-memory processing and build a model to estimate the performance of in-memory hash join algorithms with special attention to granularity effects.

3. PERFORMANCE MODEL

In this section, we present a performance model to describe the cost of hash join algorithms. We first analyze the factors that should be considered to estimate the performance and how they influence the algorithms. Based on these impact factors, we propose a general performance model, followed by the study of two specific hash join algorithms: the parallel no-partitioning hash join and radix partitioning hash join.

3.1 Impact factors

Previous work has shown that there are many factors affecting the performance of hash join algorithms [3, 10, 15]. We divide them into two categories, application characteristics and hardware features. Some algorithms' parameters can be tuned according to these factors to gain better performance.

3.1.1 Application Characteristics

Relation size Relation size depends on the tuple size and the number of tuples. However, we can minimize the size with some pre-processing operations such as filtering and abstracting the payload of each tuple to a fixed size pointer that points to the original tuple. In our performance model, we assume that the relations have been pre-processed by filtering and each tuple contains a key and a pointer-style payload. When we compare hash-join algorithms we do so for same-size inputs, but we need to be aware that algorithms may scale differently with respect of each of the input relations.

Distribution of dataset. Within a dataset the distribution of values within a certain column can be even or skewed. Building a hash table for skewed data causes more hash collisions, while probing a hash table for skewed data increases cache hits. In this paper, although we know the data skew is important and interesting, we assume that the dataset is evenly distributed, since it is easier to explain the principle of our performance model. In the future, we plan to work out a refinement of the model to describe the cost of hash joins for skewed data.

3.1.2 Hardware Features

Granularity. Typical modern server processors have cache line sizes of 64 or 128 bytes, and (cached) accesses to main memory occur at this level of granularity. Randomly accessing elements smaller than a cache line, or quantities that cross a cache line, introduces granularity effect, incurring a granularity overhead, also referred to as read- or write-amplification.

Cache size. Probing hash tables requires randomly accessing the hash table. If the cache is not large enough to contain the whole hash table, cache misses occur, decreasing the performance of the hash join. To reduce the number of cache misses, we can partition the data before building the hash table, in order to fit the hash table of each partition in the cache.

TLB entries and virtual page size. The TLB caches virtual memory page address to physical page address translations. If the translation table does not cache the requested translation entry, a TLB miss occurs. The requested data cannot be loaded until the translation entry is updated. More misses are likely if more pages are accessed. Main memory hash join algorithms, especially the partitioning hash join, are sensitive to the number of TLB entries [12]. Using a large page size can reduce the TLB entries required [2, 3], increasing the TLB hit rate.

Memory bandwidth. Memory bandwidth determines the rate at which data can be transferred between main memory and the CPU. For memory-bound algorithms, the overhead of transferring data dominates.

CPU processing rate. CPU processing rate indicates how fast the processors process data. For hash join on modern hardware we believe optimized algorithms are usually memory bandwidth bound. However, memory bandwidth may increase in the near future, which means algorithms may become compute bound, and we need to explore trade-offs between computation and memory accesses.

Other features. There are many other potential impact factors such as NUMA topologies, simultaneous multithreading (SMT), synchronization, and load balancing. These factors have been proved to be influential in [3, 10].

In this paper, to simplify the model, we limit the algorithms to running on a single-node machine, and assume that the SMT, synchronization and load balancing factors contribute only a few percent of the overall cost. While we do not include these factors in the proposed performance model, we show that despite this the model can provide a reliable prediction of the performance.

3.2 Model

3.2.1 Platform Assumptions

Our model assumes a basic multi-core machine with a memory size that is large enough to store all the data, and we assume no intermediate results need to be written back to the disk. As we want to study the comparison between no-partitioning hash join and partitioning hash join, to make them competitive, we assume that neither of the two relations can fit in the last level cache (LLC). Each data transfer between the LLC and main memory should be an integer multiple of smallest main memory transfer size, namely granularity, normally the same as the cache line size. Although NUMA architectures play an important role in the hash join performance [10], we focus on a single NUMA node system for now, and defer the analysis of NUMA effects and multi-node systems to future work.

3.2.2 The formulas and their explanation

Table 1 shows the notation of the parameters used in our performance model. Please note that to predict relative performance of algorithms, our model does not use any empirically fitted parameters. To describe different hash join algorithms and their variants, we adopt a uniform description. The total running time T of a hash join algorithm consists of the running time of each phase:

$$T = \sum_{i=1}^{n} T_i \tag{1}$$

For the running time of each phase T_i, we consider both the cost of memory accesses and that of computation. The cost of each phase should be the sum of the time the system was computing and the time it was accessing memory, minus the time it was doing both.

$$T_i = T_{com} + T_{mem} - T_{overlap} \tag{2}$$

Suppose $\beta = \frac{min\{T_{com}, T_{mem}\} - T_{overlap}}{max\{T_{com}, T_{mem}\}}$, then

$$T_i = (1+\beta) max\{T_{com}, T_{mem}\} \tag{3}$$

For a well-optimized algorithm we assume maximum overlap, i.e., $\beta = 0$. So, formula 2 reduces to:

$$T_i = max\{T_{com}, T_{mem}\} \tag{4}$$

The computation cost can be represented as:

$$T_{com} = \frac{D}{P} + C \tag{5}$$

where D denotes the processing data amount, and P indicates the base processing rate. We use C to represent the other costs such as cache miss penalty, TLB miss penalty, synchronization cost, etc. The memory accesses in each phase consist of different passes of read and write. For a multi-pass memory access we have:

Table 1: Model Parameters

Parameters	Description
T	total running time
T_i	running time of each phase
n	total number of phases
T_{com}	computation time
T_{mem}	memory accesses time
$T_{overlap}$	the overlap between T_{com} and T_{mem}
C	penalty cost during computing
m	total passes of memory access in each phase
D	required data amount, equal to relation size
D_r	data amount for read
D_w	data amount for write
P	processing rate
B	memory bandwidth
B_r	read bandwidth
B_w	write bandwidth
W	tuple size
G	granularity(size of cache line)
α	the number of cache lines the data span across
R, S	relation R, S
$\|S\|$, $\|R\|$	tuple number of relation R, S

$$T_{mem} = \sum f(D_r, B_r, D_w, B_w) \tag{6}$$

The function $f(D_r, B_r, D_w, B_w)$ indicates the data transfer time of each data pass. (D_r/D_w) is the read/write data amount and (B_r/B_w) the read/write bandwidth. We consider both a memory channel shared between read and write and a (buffered) architecture with separate read and write channels. Thus,

$$f(D_r, B_r, D_w, B_w) = \begin{cases} \frac{D_r + D_w}{B} & \text{Shared channel} \\ max\left\{\frac{D_r}{B_r}, \frac{D_w}{B_w}\right\} & \text{Non-shared channel} \end{cases}$$

When calculating D_r and D_w access granularity must be taken into account. This means that instead of calculating these as the size of the elements transferred times the number of elements transferred, D_r or D_w become αG times the number of elements where α accounts for the number of granules transferred per element because of size and alignment, and G is the granule size.

3.3 Study of Hash Join Algorithms

We analyze two prevalent parallel hash join algorithms mentioned in [2, 4], a parallel no-partitioning hash join and a histogram-based 2-pass partitioning hash join. For this case study, we assume, that relation R is not larger than relation S, or $|R| \leq |S|$. The hash table is built based on relation R. In section 3.3.1 and 3.3.2, we analyze both algorithms with a shared memory channel machine, and extend this to the non-shared channel machine in section 3.3.3.

3.3.1 No Partitioning Hash Join

The parallel no partitioning hash consists of two phases: the build phase and the probe phase.

$$T_{NP} = T_{build} + T_{probe}$$

During the build phase, the CPUs scan all the tuples in relation R, and a hash function is applied to each tuple's key to build the hash table. After that, the hash table is written back to main memory. Thus, the memory accesses in the build phase contain a sequential read of relation R, and for each tuple in R, it reads and writes the corresponding hash table buckets with granularity effect. As we can reduce

hash collision by techniques such as using larger hash tables or a better hash function, we assume that there are no or few hash collisions. So, for the build phase or the probe phase, only one hash table bucket is accessed for each tuple. Hence the actual data amount for read is:

$$W|R| + \alpha G|R|$$

The actual data amount for write is:

$$\alpha G|R|$$

The total data transfer time for the build phase is:

$$T_{mem1} = \frac{W|R| + \alpha G|R| + \alpha G|R|}{B} = \frac{(W + 2\alpha G)|R|}{B} \quad (7)$$

The cost of computation contains the pure processing part and the other cost C_1. We use P_i to indicate the processing rate of each phase. According to previous research based on the modern multi-cores architecture [2], the cache miss penalty caused by randomly accessing the hash table takes up most part of C_1, making the computation dominating. Therefore,

$$T_{com1} = \frac{W|R|}{P_1} + C_1 \quad (8)$$

In the probe phase, tuples in relation S are read, and after a same hash function being run on each tuple's key, the corresponding hash table buckets are read from the hash table for probing. Therefore, only a read is needed in this phase and the memory access cost is:

$$T_{mem2} = \frac{W|S| + \alpha G|S|}{B}$$

Similar with 8, the cache miss penalty is significant in the probe phase. The computation cost in this phase is:

$$T_{com2} = \frac{W|S|}{P_2} + C_2$$

The total cost for parallel no partitioning hash join is:

$$\begin{aligned} T_{NP} = & max\left\{\frac{(W + 2\alpha G)|R|}{B}, \frac{W|R|}{P_1} + C_1\right\} \\ & + max\left\{\frac{(W + \alpha G)|S|}{B}, \frac{W|S|}{P_2} + C_2\right\} \end{aligned} \quad (9)$$

3.3.2 Radix Partitioning Hash Join

The radix partitioning hash join we mention in this section consists of three phases, the first partitioning phase, the second partitioning phase, and the build-probe phase. To reduce the potential contention between different threads in the first partitioning phase, an extra histogram phase is added to build a global histogram for each thread. The second partitioning phase is similar with the first pass. One difference is that in the second partitioning phase, the histogram is built locally for each thread. The build-probe phase can be divided into a couple of sub-phases since it repeats the build phase and the probe phase for each partition. However, we can combine the build phase for all partitions into a build phase during the cost calculation, as well as the probe phase. As not all the partitions need to do the build-probe phase, the data needed to process and access in this phase is some percentage of the relation size. To simplify the calculation in this case study, we assume that it needs to operate the build-probe phase on all the partitions, namely, the whole relations. The total cost of radix partitioning hash join is:

$$T_{RP} = T_{histogram} + T_{partition1} + T_{partition2} + T_{build} + T_{probe}$$

Following reasoning similar to the no-partition case the total cost of radix partitioning hash join is:

$$\begin{aligned} T_{RP} = & max\left\{\frac{W(|R| + |S|)}{B}, \frac{W(|R| + |S|)}{P_1} + C_1\right\} \\ & + max\left\{\frac{3W(|R| + |S|)}{B}, \frac{W(|R| + |S|)}{P_2} + C_2\right\} \\ & + max\left\{\frac{4W(|R| + |S|)}{B}, \frac{W(|R| + |S|)}{P_3} + C_3\right\} \\ & + max\left\{\frac{W|R|}{B}, \frac{W|R|}{P_4} + C_4\right\} \\ & + max\left\{\frac{W|S|}{B}, \frac{W|S|}{P_5} + C_5\right\} \end{aligned} \quad (10)$$

We briefly describe the phases to explain each of the components of this equation.

The histogram phase (phase 1) is done separately on both relations. For each relation, the processors read all the tuples and builds the histogram. The histogram is small enough to store in the cache and there is no need to write back to the memory. So, only a sequential read is done on both tables. While included in the formula we expect the compute cost to be dominated by the transfer cost.

During the first partitioning phase (phase 2), all tuples in both relations are scanned and assigned to the corresponding partition, and then the partitions are written back to the memory. Even though the access to each partition is random, the cache is large enough to maintain a small bucket for each partition. When the small bucket is full, it is written back to the main memory and the another empty bucket will be read for other tuples. So, the granularity-size data block is fully used in this phase. Assuming the size of the partitioned relation is equal to the original relation size, there is a full read and a full write for both relations. Also, the data block should be read before it is written back. If the number of partitions is too large, it may cause many TLB misses, the overhead of which can cause the computation cost dominating. However, multi-passes approaches can be used to limit the fanout of each pass partitioning, minimizing the number of TLB misses. Therefore, in the later parts of this paper, we assume that the memory access costs dominate the total running time.

The second pass partitioning phase (phase 3) is similar with the the first pass partitioning, but here we include the second histogram build with the partitioning.

For the build-probe phase (phase 4 & 5), we assume that in this case the hash table for each partition is small enough to fit in the cache and there are few cache misses when accessing the hash table. Otherwise, the cache misses penalty may take up a significant part of the cost, turning it back the the case of no partitioning hash join for each partition. The build sub-phase (phase 4) and probe sub-phase (phase 5) are modeled separately.

3.3.3 Non-shared Channels

If the two algorithms analyzed in section 3.3.1 and 3.3.2 run on a non-shared memory channel machine, the overlap

between read and write should be considered. In the no-partitioning hash join, the write only happens during the build phase. The cost for transferring data depends on the dominating part between reading both the tuples and the hash table and writing the hash table. So, formula 7 is changed to:

$$T_{mem1} = max\left\{\frac{W|R| + \alpha G|R|}{B_r}, \frac{\alpha G|R|}{B_w}\right\}$$

Correspondingly, the overall cost of no-partitioning hash join is modified to:

$$\begin{aligned} T_{NP} = & max\left\{\frac{W|R| + \alpha G|R|}{B_r}, \frac{\alpha G|R|}{B_w}, \frac{W|R|}{P_1} + C_1\right\} \\ & + max\left\{\frac{(W + \alpha G)|S|}{B_r}, \frac{W|S|}{P_2} + C_2\right\} \end{aligned} \tag{11}$$

The write happens during all partitioning phases in the radix partitioning hash join. As we analyze above, the read and write amount for the first partitioning phase are $2W(|R| + |S|)$ and $W(|R| + |S|)$, respectively, the memory access time for either pass should change to:

$$T_{mem2} = max\left\{\frac{2W(|R| + |S|)}{B_r}, \frac{W(|R| + |S|)}{B_w}\right\}$$

Similarly, cost for the second partitioning phase is:

$$T_{mem3} = max\left\{\frac{3W(|R| + |S|)}{B_r}, \frac{W(|R| + |S|)}{B_w}\right\}$$

Consequently, the overall performance for radix partitioning hash join is:

$$\begin{aligned} T_{RP} = & max\left\{\frac{W(|R| + |S|)}{B_r}, \frac{W(|R| + |S|)}{P_1} + C_1\right\} \\ & + max\left\{\frac{2W(|R| + |S|)}{B_r}, \frac{W(|R| + |S|)}{B_w}, \frac{W(|R| + |S|)}{P_2} + C_2\right\} \\ & + max\left\{\frac{3W(|R| + |S|)}{B_r}, \frac{W(|R| + |S|)}{B_w}, \frac{W(|R| + |S|)}{P_3} + C_3\right\} \\ & + max\left\{\frac{W|R|}{B_r}, \frac{W|R|}{P_4} + C_4\right\} \\ & + max\left\{\frac{W|S|}{B_r}, \frac{W|S|}{P_5} + C_5\right\} \end{aligned} \tag{12}$$

3.4 Model Analysis

In this section, we validate the proposed model and show how to make a prediction with the model. We assume we can use techniques such as prefetch and multi-pass partitioning, to reduce the cost of computation, and both algorithms are limited by the memory bandwidth. We use a 16B tuple size W and 64B cache line size G as an example. We assume the hash table, the tuples and the meta data are within the same cache line. Each access to a 16B tuple in the hash table causes transfer of a 64B block, or $\alpha = 1$. For a non-shared memory channel machine, according to formula 9 the running time of the no-partitioning hash join is:

$$T_{NP} = \frac{(W + 2\alpha G)|R|}{B} + \frac{(W + \alpha G)|S|}{B} = \frac{144|R| + 80|S|}{B}$$

In a two-pass partitioning radix hash join, assume a good partition number is chosen to reduce the cache misses and TLB misses, so that the algorithm is memory bandwidth limited. According to formula 10, the running time of the radix partitioning hash join is:

$$\begin{aligned} T_{RP} = & \frac{W(|R| + |S|)}{B} + \frac{3W(|R| + |S|)}{B} + \frac{4W(|R| + |S|}{)}B \\ & + \frac{W|R|}{B} + \frac{W|S|}{B} = \frac{144(|R| + |S|)}{B} \end{aligned}$$

We can conclude that for this case the no-partition hash join will always perform better.

There are some differences when running the algorithms in a non-shared channel machine since the read and write have overlap. We select a cache line size of 128 Bytes. Suppose the read bandwidth is twice as large as the write bandwidth. That means in one phase if the read data amount is not twice larger than the write data amount, that phase is dominated by the write cost. According to formula 11, the cost of the no-partitioning hash join in a non-shared channel machine is:

$$T_{NP} = \frac{\alpha G|R|}{B_w} + \frac{(W + \alpha G)|S|}{B_r} = \frac{128|R|}{B_w} + \frac{144|S|}{B_r}$$

Similarly, based on formula 12, the running time of the radix partitioning hash join in a non-shared channel machine is:

$$\begin{aligned} T_{RP} = & \frac{W(|R| + |S|)}{B_r} + \frac{W(|R| + |S|)}{B_w} + \frac{3W(|R| + |S|)}{B_r} \\ & + \frac{W|R|}{B_r} + \frac{W|S|}{B_r} = \frac{80(|R| + |S|)}{B_r} + \frac{16(|R| + |S|)}{B_w} \end{aligned}$$

It is also be useful to look at granularity effects . Assuming that the tuple does not exceed the size of a cache line (i.e., $\alpha = 1$), and setting $X = \frac{|S|}{|R|}$,

$$\begin{aligned} T_{NP} = & \frac{|R|}{B}((W + 2G) + (W + G)X) \\ T_{RP} = & \frac{|R|}{B}(W(1 + X) + W(3 + 3X) \\ & + W(4 + 4X) + W(1 + X)) \end{aligned}$$

solving $T_{NP} < T_{RP}$ yields

$$W > \frac{G}{8}(\frac{X + 2}{X + 1}) \tag{13}$$

especially when $X = \frac{|S|}{|R|}$ is large enough,

$$W_{break\_even} \simeq \frac{G}{8} \tag{14}$$

Thus, unlike the change in $\frac{|S|}{|R|}$, change in the tuple size plays a significant role in which algorithm performs best. For G=64B, the break-even tuple size is about 8B.

4. EXPERIMENT SETUP

4.1 Platform

We validate the proposed model on two different multi-core machines. The HP Proliant DL-360P has 10 cores per node, with SMT2 configuration. Different with HP Proliant DL-360P, the IBM POWER8 S824L has fewer cores but more threads. Each of 4 NUMA nodes in the IBM POWER8 S824L has 5 cores with up to SMT8 setting per cores. Cache and TLB configurations are summarized in table 2.

Both platforms have multiple NUMA nodes. As we want to eliminate the NUMA effect, we only use one node in both

Table 2: Hardware Platforms Feature

	HP Proliant DL-360p Gen8	IBM POWER8 S824L
CPU	Intel Xeon E5-2670 v2 2.5GHz	IBM POWER8E 3.7GHz
Cores	10/20	5/20
Threads	20/40	40/160
Cache L1	32 KiB	64 KiB
L2	256 KiB	512 KiB
L3	25 MiB	8 MiB
TLB L1	64	48(96)
L2	512	256
L3	N/A	2048
Page size	4 KiB	4 KiB
Memory	192 GB	256 GB
Mem. BW	42/84 GB/s	Read: 76.8/307.2 GB/s Write: 38.4/153.6 GB/s
Cacheline	64B	128B

platforms. The memory bandwidth in HP Proliant DL-360P is around 42 GB/s within each NUMA node. It is shared between the read and write. The IBM POWER8 S824L has seperate read channels and write channels with each node supporting 76.8 GB/s read and 38.4 GB/s write bandwidth.

4.2 Workload

For evaluating both algorithms analyzed in section 3.3, we use the workload from [4] and extend it to support various tuple sizes from 4B to 128B. We assume the workload is in a column-oriented mode with each tuple in a form of $< key, value >$. Except for the 4B tuple case, which has a 4B key and no value, we assume an 8B key. We assume the key in each relation is unique and uniformly distributed.

5. EXPERIMENTAL RESULTS

In this section, the proposed model is evaluated. We first analyze both algorithms by tuning prefetch distance, radix bits, and SMT configuration. After that, we demonstrate the granularity effect by changing the tuple size. Finally, we vary the relation size ratio and show that the relation size ratio does not change the winner in the competition between no-partitioning hash join and radix partitioning hash join.

5.1 Impact of Software Prefetch

One of the dominating costs in a no-partitioning hash join is the result of cache misses. Since the latency is high for getting the data from main memory instead of from cache, cache misses cause stalls in the processor. The processor cannot continue to work until the data is returned. Using software prefetch is a way to reduce the cache miss penalty.

Figure 1 shows the impact of using prefetch in the no-partitioning hash join. A prefetch distance i means that when processing the current tuple, the processors do the prefetch of the hash bucket for the next ith tuple. We can see from the figure that, the performance improves more than 25% in the Intel machine and 35% in the POWER8 machine, respectively. For the Intel machine, the performance remains stable when the prefetch distance reaches 4, and after 10, the performance nearly doesn't increase any more. This is because it reaches the memory bandwidth limitation. The prefetch effect is more obvious in the POWER machine. The running time drops sharply from no-prefetch to only adopting a prefetch distance of 1. In the rest of the paper, the no-partitioning hash join uses prefetch distance of 10 in the Intel machine and 6 in the POWER machine unless otherwise specified.

The performance improvement is the result of latency hiding. When randomly accessing the hash table either during the build phase or the probe phase, as the cache is not large enough to hold the whole hash table, a lot of cache misses occur. Doing prefetch can access the data before it is needed, reducing the wait cycles for the data response. If the prefetch distance is large enough, it can reduce most of the waiting time. A drawback of utilizing prefetch is the increase of the number of instructions. However, this penalty is far less than the performance increase introduced by prefetch.

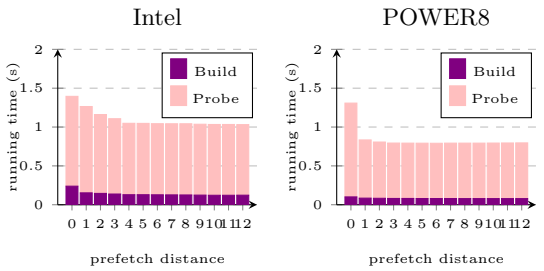

Figure 1: Running time for no-partitioning hash join with different prefetch distance (Intel, 10 cores, 20 threads, POWER8, 5 cores, 40 threads)

5.2 Number of Partitions and Partitioning Passes

Radix partitioning hash join is a hardware-sensitive algorithm. The radix bit, indicating the number of partitions ($2^{radix\ bit}$), is an important tuning parameter. It should be well configured to gain good performance. On the one hand, If the partition number is too small, the size of each partition may not fit in the cache, leading to cache misses. On the other hand, if the number of partition is too large, each partitioning pass may cause a lot of TLB misses.

Figure 2 shows how the performance changed running against the Intel and the POWER8 machine by varying the radix bit. We can see that, in both figures, the partitioning time increases when the radix bit goes up, along with the build-probe time decreasing. The best trade off radix bit configuration for the Intel machine is 10, after which the partitioning time jump sharply due to the TLB miss penalty. The POWER8 machine is more robust with respect to this parameter. A radix bit of 14 is the best configuration, and when the radix bit is larger than 19, the partitioning time increases significantly. The build-probe time, on the other hand, declines rapidly at the beginning when radix bit increases beyond 6. The reduction of cache misses contributes to this performance enhancement. However, the build-probe cost does not keep shrinking when the partition number is too large (radix bit is 20 in both machines). This is because the computation for each partition takes up a larger percentage of the overhead and starts to dominate.

5.3 SMT effect

Figure 3 demonstrates the curve of the SMT effect in the POWER8 machine. We run the no-partitioning hash join both with and without prefetch, and radix partitioning hash join. We can see that the no-partitioning hash join benefits

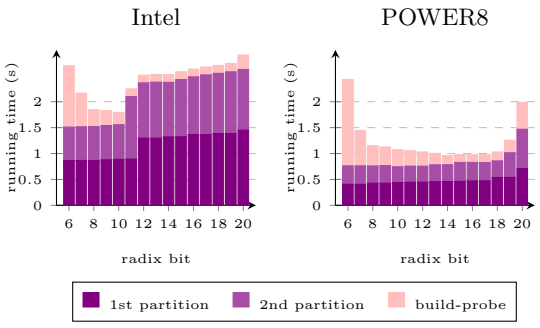

Figure 2: Running time for partitioning hash join with different radix bits (Intel, 10 cores, 10 threads, POWER, 5 cores, 15 threads). The cost for building histogram is included in the first partitioning phase.

from the SMT, especially the case without prefetch. This logical because both threading and prefetching hide memory latency. However, it keeps stable when the threads number is more than 20 for the no-partitioning hash join without prefetch and more than 10 for that with prefetch. Conversely, the radix partitioning hash join is more oblivious to the SMT configuration. The running time does not change a lot when the SMT is in different configurations. It even increases slightly when the thread number reaches 20 which means SMT4 configuration. For the rest of this paper, we use SMT8 for the no-partitioning hash join and SMT3 for the radix partitioning hash join in the POWER8 machine as they are the best selection based on this experiment.

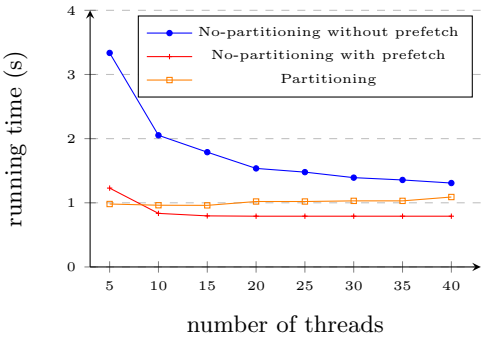

Figure 3: Running time with different numbers of threads (POWER, 5 cores)

5.4 Granularity

Because we cannot change the cache line size we vary the size of the tuples. Figure 4 shows how the running time changes as a function of the tuple size changing from 4B to 128B for measured data and our model on the Intel machine. For larger tuple sizes the no-partitioning hash-join performs better. This is expected because for larger sizes there is increased bandwidth pressure and the amplification benefit of the partitioned hash join is reduced. The measured and predicted curves have similar shape. The measured break-even point is around 8B, close to 9B provided by the model.

Figure 5 shows the performance of both hash join algorithms running on the POWER8 machine. The running time of radix partitioning hash join increases smoothly in a same pace with the tuple size. The running time of the no-partitioning hash join, showing a different curve, climbing stably at the beginning when the tuple size is small. From a 32B tuple size to 64B size, there is a jump, almost doubling the running time, which is as shown in the 32B size in the Intel machine test. This is because the tuple and the meta data in a same hash table bucket are split over two cache lines. Consequently, each access to the hash table actually transfers two cache lines instead of one. Measured radix partitioning hash join win when the tuple size is 16B or smaller, while the no-partitioning hash join win when the tuple size is 32B or larger. The modeled break even is around 21B which matches the prediction analyzed in section 3.4

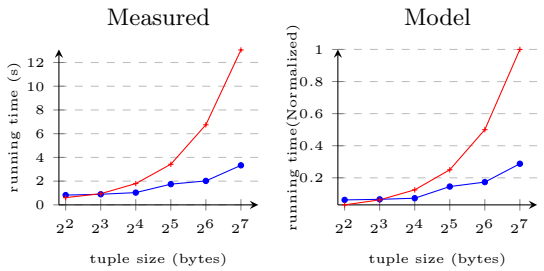

Figure 4: Running time with different tuple size (Intel, 10 cores, radix (red) and no-partition (blue))

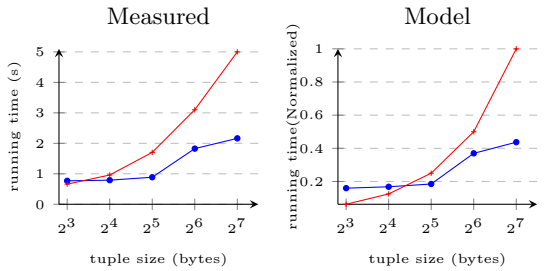

Figure 5: Running time with different tuple size (POWER, 5 cores, radix (red) and no-partition (blue))

5.5 Relation Size Ratio

In this section, we explore how the relation size ratio impacts the performance. We run both algorithms in both platforms with various $|R|$ from $16 * 2^{20}$ to $256 * 2^{20}$ and keep the size of S at $256 * 2^{20}$. Figure 6 and figure 7 summarize the prediction trend of both algorithms' performance with different R size according to the proposed model. We can see from figure 6, consistent with the model, that when the tuple size is 16B, the no-partitioning hash join is always better than the radix partitioning hash join. Conversely, when the tuple size is 8B, the radix partitioning hash join outperforms the no-partitioning hash join. In figure 7, the radix partitioning hash join runs faster than no-partitioning hash join both when the tuple size is in 8B and in 16B.

Figure 6 and figure 7 illustrate the experiment results of the relation size ratio effect. Both figures have similar shapes with the proposed model prediction. The curve of no-partitioning hash join in 8B tuple size is close to that

in 16B tuple size due to the granularity effect, while the running time doubles for the radix partitioning hash join when the tuple size changes from 8B to 16B. The radix partitioning hash join runs slower than the estimation by the model, likely because the overlap between computation and memory accesses cannot be ignored. So all the curves for radix partitioning hash join in the model should be shifted up. Consequentially, as shown in figure 6 and figure 7, the no-partitioning hash join can win in some cases when the difference between the size of both relations are large.

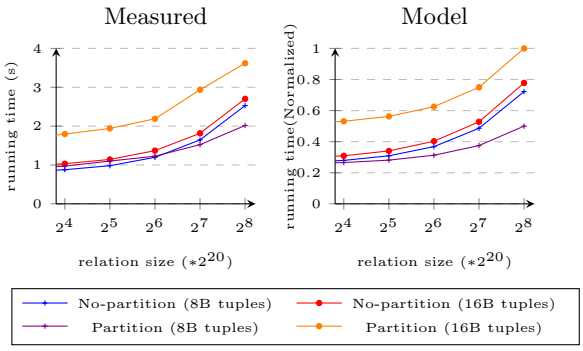

Figure 6: Running time with different relation size ratio (Intel, 10 cores)

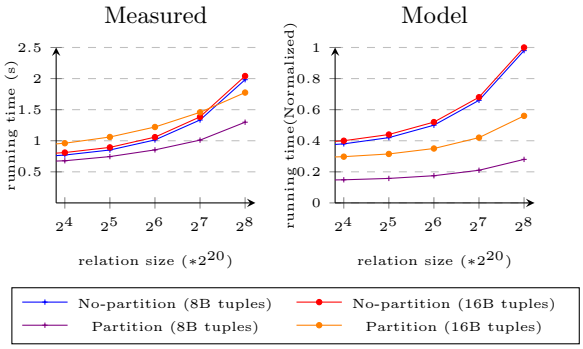

Figure 7: Running time with different relation size ratio (POWER, 5 cores.)

6. CONCLUSIONS AND FUTURE WORK

In this paper, we analyze the performance of in-memory hash join algorithms on multi-core platforms. We discuss the factors that impact the performance and find that the granularity is one of the main impact factors. Based on this finding, we propose a performance model considering both computation and memory accesses. According to the model, no-partitioning hash join should be more competitive than the partitioning hash join when the tuple size is large and the granularity is small. The results show that our model can accurate predict the winner between no-partitioning hash join and partitioning hash join. In the future, we expect to extend the proposed model to account for NUMA effects and skewed data distributions.

7. ACKNOWLEDGMENTS

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC, visualization, or storage resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu

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

[3] Ç. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on modern processor architectures. *IEEE TKDE*, 27(7):1754–1766, 2015.

[4] 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. ACM, 2011.

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

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

[7] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about ad hoc join costs. *VLDB*, 6(3):241–256, 1997.

[8] K. A. Hua, W. Tavanapong, and Y.-L. Lo. Performance of load balancing techniques for join operations in shared-noting database management systems. *Elsevier Journal of Parallel and Distributed Computing*, 56(1):17–46, 1999.

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

[10] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. Massively parallel NUMA-aware hash joins. In *IMDM*, pages 3–14. Springer, 2015.

[11] F. Liu and S. Blanas. Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In *SoCC*, pages 153–166. ACM, 2015.

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

[13] E. Omiecinski. Performance analysis of a load balancing hash-join algorithm for a shared memory multiprocessor. In *VLDB*, pages 375–385, 1991.

[14] J. M. Patel, M. J. Carey, and M. K. Vernon. Accurate modeling of the hybrid hash join algorithm. In *SIGMETRICS*, pages 56–66. ACM, 1994.

[15] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*, pages 1961–1976. ACM, 2016.

[16] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing.