Optimizing Hash Join with MapReduce on Multi-Core CPUs*

Tong YUAN†, Member, Zhijing LIU†, and Hui LIU†, Nonmembers

SUMMARY In this paper, we exploit MapReduce framework and other optimizations to improve the performance of hash join algorithms on multi-core CPUs, including No partition hash join and partition hash join. We first implement hash join algorithms with a shared-memory MapReduce model on multi-core CPUs, including partition phase, build phase, and probe phase. Then we design an improved cuckoo hash table for our hash join, which consists of a cuckoo hash table and a chained hash table. Based on our implementation, we also propose two optimizations, one for the usage of SIMD instructions, and the other for partition phase. Through experimental result and analysis, we finally find that the partition hash join often outperforms the No partition hash join, and our hash join algorithm is faster than previous work by an average of 30%.

key words: hash join, database system, MapReduce, multi-core CPU, cuckoo hashing

1. Introduction

In the last decade, multi-core CPUs have experienced tremendous development. Today, modern CPUs often have four or more cores, and each core has two or more threads. Recently, IBM introduced its new generation processor—power 8, which has 12 cores with 8 hardware threads per core for a total of 96 threads of parallel execution. Modern CPUs also provide parallelism at different levels: Instruction Level Parallelism via super scalar execution [13]; Data Level Parallelism by Single Instruction Multiple Data (SIMD) instructions [24]; and Thread Level Parallelism (Simultaneous Multi-Threading, SMT) that have been applied on some basic database operations [3], [5], [6], [20]. Another important change of modern CPUs is that the cache sizes of each core and shared cache sizes at lower levels of the cache hierarchy both increase significantly [9]. In addition, memory capacity is rising rapidly while memory price is continuing to drop, so that many current database management systems (DBMSs) are able to put their whole working set in main memory. In the future, this trend on modern CPUs will continue. The number of cores in a system will continue to increase, the cache sizes of each core and the shared cache sizes will continue to increase, and the memory capacity will also continue to rise. Therefore, facing such changes and the future trend of modern CPUs, it is necessary to adjust basic database operations.

Join is an important operation in relational database systems, which combines two relations on a common key. Basic join algorithms include non-indexed join, indexed nested-loop join, sort-merge join, and hash join, of which the last two are widely used in the current DBMSs. In the last few years, many hash join variants have been proposed [3], [15], [17]. However, some variants do not make full use of thread-level parallelism and data-level parallelism, some do not optimize the hash table, and some do not achieve good load imbalance. Balkesen et al. [2] concluded that hardware-conscious join algorithms can be made faster than hardware-oblivious algorithms and more robust to a wider set of parameters.

Since its introduction about one decade ago, the MapReduce programming model [7] has proved its strong computing ability of processing large data in a cluster of computers environment. The MapReduce model was initially designed for a cluster of computers, but its performance is limited by disk I/O and network I/O. Due to avoiding these disadvantages, shared-memory MapReduce [21] is more appropriate for in-memory computation, and outperforms cluster-based MapReduce. Facing the change and future trend of modern CPUs, in this paper we use the shared-memory MapReduce and some other optimizations to improve hash join algorithms on modern multi-core CPUs. Here, we improve the performance in one single node of a distributed computing system. In the future, we can improve the overall performance in distributed computing system.

There are four main contributions in this paper. First, we implement the hash join with MapReduce model on the multi-core CPUs, including No partition hash join and partition hash join. Each of them can be run as two MapReduce jobs respectively.

Second, we also design an improved cuckoo hash table for our hash join which consists of a cuckoo hash table and a chained hash table. It reduces the number of replace operations and utilizes the SIMD instructions to improve its performance.

Third, we propose two optimizations, one for the usage of SIMD instructions, and the other for partition phase. The usage of SIMD instructions can improve the performance of hash table, and the multi-pass partition optimization improves the performance of partition phase.

Finally, we compare the performances of No partition hash join and partition hash join, and make a conclusion.
that the partition hash join is superior in performance with appropriate parameters, and we identified which parameters play the most important part in hash join through many experiments.

The rest of the paper is organized as follows: Sect. 2 covers related work and background information. The hash join implementation with MapReduce is presented in Sect. 3. Section 4 describes the experimental results. Finally, Sect. 5 contains conclusions and future work.

2. Backgrounds and Related Work

2.1 Shared-Memory MapReduce

MapReduce [7] is a programming model proposed by Google Inc. This programming model was created for processing large data with thousands of computing nodes in 2004. It is being used widely on big data, cloud computing, data mining and so on.

The programmers need to complete two main functions to use this model, map and reduce. The map function is applied on a unit of input data which is obtained after split phase and produces a set of intermediate <key, value> pairs. The reduce function is applied on the intermediate pairs produced by the map phase. It merges the intermediate pairs with the same key and produces the final output pairs in merge phase.

The MapReduce model was firstly designed for the cluster of computers where one of the computers is seen as a processing node, but its performance is limited by disk I/O and network communications. Shared-memory MapReduce [21] in which one thread is seen as a processing node is often applied on Chip Multi-processors (CMPs) or Symmetric Multi-Processors (SMPs) and not limited by disk I/O and network communications, so it is more appropriate for in-memory computation. The success of Shared-memory MapReduce stems from hiding the details of parallelization, load balance, and fault tolerance. Besides these features, it also provides an efficient management of intermediate data. Based on above advantages, we adopt Shared-memory MapReduce framework in our implementation. Figure 1 shows the overview of shared-memory MapReduce runtime system.

2.2 Intermediate Buffer in MapReduce

The intermediate buffer is one of the most important factors in MapReduce model. It stores the intermediate pairs outputted by the map phase and forms reduce tasks with the same keys. In the intermediate buffer, the pairs with the same key are stored. Map threads write their intermediate pairs to the intermediate buffer concurrently, so it may lead to write collision with concurrent threads. To avoid this collision, there are three universal schemes for writing to the intermediate buffer.

1. **Lock scheme**: All worker threads share a mutual buffer which all worker threads write intermediate pairs to. The threads need to synchronize via a latch to avoid the write collision.

2. **Lock-free scheme**: Assume that p threads work concurrently, this scheme creates p buffers in total and each thread is assigned to a private buffer. Each thread writes its private intermediate pairs to its private buffer without any synchronization overhead. When the map phase finishes, all threads need to merge the p buffers into one buffer.

3. **Count and Move scheme**: Another method to solve the write collision is the count and move scheme described by Fang et al. [12]. In this method, each thread needs to scan its task twice. Firstly, each task outputs three counts, the number of intermediate pairs, the size of keys (bytes) and the size of values (bytes). Each task computes the prefix sum on the keys size and the value size, and get its start position in the intermediate buffer. Then each task computes the prefix sum on the number of intermediate pairs, and get the start positions of its intermediate pairs in the intermediate buffer. Secondly, according to the deterministic start position, each task outputs the intermediate pairs to the output array, so the write collisions are avoided.

The three schemes described above have their respective merits and demerits. The Lock scheme needs a latch to avoid the write collision, but it does not need any overhead to consolidate all individual buffers into a one final buffer; The Lock-free scheme does not need any synchronization overhead when writing intermediate pairs, but it needs to merge the buffers; The count and move scheme needs to scan the input twice, but it can produce contiguous intermediate buffer.

2.3 Cuckoo Hashing

Cuckoo hashing [19] firstly proposed by Pagh and Rodler uses two hash tables and two hash functions rather than one hash table and one hash function. Cuckoo hashing has several good properties of high occupancy rate, constant time for lookup operation, and efficient memory usage. When inserting a key, each key has two objective entries calculated by two hash functions. If one of the two entries is empty, the key can be inserted into it. If two entries are occupied, which means a collision occurs, one of the keys in the two entries is kicked out to its other entry using other hash function, and then the objective key is stored in this entry. This process is repeatedly executed for finding a vacant entry for each key in two hash tables. If we can not find a replacement entry for the key to be inserted, insertion fails (likely to oc-
cur when n/2 elements have been inserted). To overcome the drawback, the previous work [10] showed that increasing the number of hash functions used and using bins of size more than 1 can improve the performance of cuckoo hashing.

2.4 Related Works

There is a long history of improving join performance for main memory database system. DeWitt et al. [8] first studied hash join performance for main memory database system. A decade later, Shatdal et al. [23] studied cache-conscious algorithms and discovered that partition phase in which two relations are partitioned so as to reduce cache miss and TLB miss can improve hash join performance.

Manegold et al. [18] and Ailamaki et al. [1] demonstrated the importance of memory access and cache miss on DBMSs in main memory processing. The following paper [17] presented the radix join algorithm on modern processor.

In the new century, there were many new works for improving join performance. Cieslewicz and Ross [6] presented an efficient way to partition data on chip multiprocessors. Ross [22] presented an efficient way to improve hash join performance by cuckoo hashing [19]. He et al. [11] presented GPU-based implementations for many join algorithms, including hash join and sort-merge join.

Recently, researchers have explored new ways to improve join performance. Blanas et al. [3] proposed No partition join, which leverages hardware SMT to hide memory access latencies. Kim et al. [15] compared sort-merge join and hash join on multi-cores CPUs, and concluded that currently the hash join outperformed sort-merge join in many cases. Kaldewey et al. [14] pushed the idea of leveraging hardware to hide memory access latencies to GPU.

3. Hash Join Implementation with MapReduce

In this section, we first present a detailed hash join implementation with MapReduce on multi-core CPUs. Then we introduce our improved hash table designed for the hash join. Finally we present some optimizations that are available in our hash join processing.

3.1 Hash Join Implementation

A hash join algorithm works on two input relations, R and S. We often assume |R| < |S|. The basic hash join algorithm has two phases: build and probe. In the build phase, the smaller relation R is scanned, and all R tuples are inserted to an in-memory hash table. The probe phase scans the larger relation S and probes the join key in the in-memory hash table for each S tuple to look for the matching R tuples. Existing hash join can be classified into two classes: No partition hash join and partition hash join. The partition phase is an optional phase in a hash join algorithm. Shatdal et al. [23] showed that the cost of partition phase is often less than the cost of cache miss and TLB miss in the both build phase and probe phase.

3.1.1 No Partition Hash Join with MapReduce

No partition hash join can be run as two MapReduce jobs; one is build phase and the other is probe phase. In our experiments, each tuple is a simple <id, value> pair, so the result of a hash join contains the <id, value> combinations of R and the matched S.

- Build MapReduce job

  In the split phase, R tuples are split into some units, and each unit is organized as a map task, which would be inserted into a task queue. So the task queue only needs to know the start address of each unit and the length of each unit.

  In the map phase, each map thread fetches tasks from the task queue and inserts each tuple into the proper position of the hash table in parallel. In this MapReduce job, Lock scheme mentioned in above section is applied when all map threads share the same hash table, and all map threads put R tuples into the hash table in parallel with lock/unlock operations.

- Probe MapReduce job

  The Probe MapReduce job is also run as a map-only job. In the split phase, S tuples are also split in the same way as the build MapReduce job.

  In the map phase, each map thread reads every S tuple <id, value> from its assigned units and find whether there are matching R tuples in the hash table, where the value of tuple R <id, value> equals to the value of tuple S <id, value>. If the values are matched, the tuple R and the tuple S are joined to form the output pair <R.id, S.id>, which would be written into a final result buffer.

3.1.2 Partition Hash Join with MapReduce

Partition hash join has three phases: partition, build and probe. In partition phase, relations R and S are first divided into some distinct partitions. In this way, every partition has its own hash table after build phase. The size of individual hash table fits in the CPU cache so that the number of cache misses can be reduced in the probe phase. Partition hash join can be run as two MapReduce jobs, one is applied on relation R and the other on relation S.

- Partition-Build MapReduce job

  In the map phase, the map function reads each R tuple <id, value> from its assigned units iteratively and inserts each tuple into the correct partition according to its value. The relation R is divided into P partitions. There, we can adopt the multi-pass partition if necessary, which is summarized in detail in the Sect. 3.3.2. In this MapReduce job, we adopt the count and move scheme as described in Sect. 2.2 to avoid the write collisions with threads.
In the reduce phase of this MapReduce job, a reduce task is created for each \( R_i \) partition \((0 \leq i < P)\), and each reduce thread creates a separate hash table for each \( R_i \) partition in parallel. The number of working threads is \( T \), and the reduce thread \( t \) \((0 \leq t < T)\) works on partitions \( R_{t+0T}, R_{t+1T}, R_{t+2T}, \ldots \) etc. Now hash tables of all partitions have been created for the next MapReduce job.

**Partition-Probe MapReduce job**

The map phase of this MapReduce is to divide relation \( S \) into \( P \) partitions in parallel. The partition function, the number of partitions and the structure of the intermediate buffer in this job should be the same as these in the partition-build MapReduce job. In the reduce phase, a reduce thread reads tuples from a \( S_j \) partition and finds matched \( R \) tuples in the corresponding hash table of the \( S_j \) partition. The reduce thread \( t \) \((0 \leq t < T)\) is also assigned to partitions \( S_{t+0T}, S_{t+1T}, S_{t+2T}, \ldots \) etc. Finally, matched pairs \(<R.id, S.id>\) are put into the final result buffer.

### 3.2 Improved Cuckoo Hash Table for Hash Join

Based on the previous work on d-ary cuckoo hashing [10] and shared-memory MapReduce, we design an improved cuckoo hash table for our hash join algorithm on multi-core CPU in Sect. 3.1. The cuckoo hash guarantees \( O(1) \) lookup performance, but the overhead of the insert operation is not negligible, due to the replacement operation. Existing cuckoo hash table has these disadvantages: (1) it has large overhead for insert operations, including search the cuckoo path overhead (we call the sequence of replaced keys in an insert operation as a cuckoo path) and replace operation overhead, (2) it often sets a upper threshold for the length of cuckoo path to improve space utilization and avoid insertion fails, however the higher threshold affects the insert performance, (3) it can not take full advantage of the CMP’s parallel execution resources. The insert procedure can be divided into search cuckoo path procedure and replace procedure. To overcome these disadvantages, we choose the chained hash table as auxiliary table and set a very low threshold for the length of cuckoo path. Thus, if we can not find a proper cuckoo path in the limited length for the inserted tuple, we can insert the tuple into the chained hash table. In this way, we can avoid a large number of replacement operations and the insert fails. In addition, we adopt the Lock After Discovering a Cuckoo Path and the Breadth-first Search proposed in the previous work [16]. The Lock After Discovering a Cuckoo Path makes threads search the cuckoo path in parallel without any lock operation, and the Breadth-first Search can search the cuckoo path efficiently.

Figure 2 presents the structure of our improved cuckoo hash table. The improved cuckoo hash table consists of two hash tables - one is the cuckoo hash table and the other is the chained hash table. For a key to be inserted, we check the entries at \( h_1(k) \) and \( h_2(k) \) in the cuckoo hash table firstly. If one of these entries is empty, the key \( k \) is stored in the entry. If none of these entries is empty, the replacement operations happen as the cuckoo hashing shown in the paper [16]. When we can not find a proper cuckoo path in our threshold, the key \( k \) is inserted into the chained hash table via \( h_3(k) \) without any replacement operation. We use the same hash function for \( h_1(k) \) and \( h_2(k) \) so as to reduce the overhead of the calculation.

In the cuckoo hash table, each bucket stores several \(<key, value>\) tuples, and the size of bucket equals to the CPU cache line. In addition, we also need a lock to avoid write collision with concurrent threads and a count to record the number of tuples currently in the bucket. Setting the size of bucket to the size of cache line can reduce the number of cache misses in the probe phase, and the design of keys sequentially stored in memory makes us use SIMD operation in the probe phase. In the chained hash table, each node has a similar structure as the bucket described above. The only difference is that the node includes a pointer to the next node rather than a lock as in the bucket. We adopt parallel buffers [5] to avoid write collision in the chained hash table. Each thread atomically obtains a private node, so each thread does not need lock/unlock operations on every write to its private node. It only needs an interthread coordination when it obtains a new node.

We implemented the fast concurrent hash table with lock later and breadth-first search, which is shown in [16], as close to the description as possible. Figure 3 shows the insert performance of our improved cuckoo hash table in comparison to the concurrent hash table in [16], and Fig. 4 shows the lookup performance comparison. Here, we use uniformly distributed pairs, and each pair is made up of 8-
byte key and 8-byte value. We choose the dataset of $2^{23}$ pairs as 8M. In the previous cuckoo hash table, we set 200 as the threshold for the length of cuckoo path, where the insert operation fail happens, while we just set 50 as the threshold in our cuckoo hash table.

Our improved cuckoo hash table has better insert performance, due to the lower threshold. Compared with previous cuckoo hash table, the insert performance of our cuckoo hash table improves by about 20.8%, 21.6%, 20.1%, 23.2% and 27.0%, respectively, when the number of working threads is 1, 2, 4, 8, and 16.

Meanwhile, the lookup operation of our cuckoo hash table is a little slower, because we need to look up in the chained hash table sometimes. However, the gap in lookup performance between two cuckoo hash tables is small. The lookup operation of our cuckoo hash table is only 6.0%, 3.3%, 3.1%, 2.7% and 1.3% slower than that of previous cuckoo hash table, respectively, when the number of working threads is 1, 2, 4, 8, and 16. So our improved hash table has better overall performance.

3.3 Optimizations

3.3.1 Usage of SIMD Instructions

We can accelerate the probe phase by using SIMD instructions to process multiple keys in a bucket/node in parallel. Our hash table described above may need to be adjusted to different hash join implementations. In the No partition hash join, all threads share a same hash table. So each bucket of cuckoo hash table needs a lock of size 8 bytes and a count of size 8 bytes, and each node of chained hash table needs a count of size 8 bytes and a pointer to the next node of size 8 bytes. So, each bucket and each node both have 48 bytes space for $<key, value>$ tuples. If the size of $<key, value>$ tuple is 8/8 bytes (as shown in the Dataset A), each bucket and each node can store 3 tuples. If the size of $<key, value>$ tuple is 4/4 bytes (as shown in the Dataset B), each bucket and each node can store 6 tuples.

In the partition hash join, every partition has a hash table, and each bucket does not need a Lock. So each bucket has 56 bytes space for $<key, value>$ tuples, and each node still has 48 bytes space for $<key, value>$ tuples. If the size of $<key, value>$ tuple is 8/8 bytes, each bucket still store 3
tuples at most. If the size of <key, value> tuple is 4/4 bytes, each bucket can store 7 tuples.

3.3.2 Multi-Pass Partition Optimization with MapReduce

Radix-cluster Algorithm proposed by Manegold et al. [17] is the basic principle of many multi-pass partition implementations. We can also implement the multi-pass partition with MapReduce model. We can implement the first partition in the map phase, and the second in the reduce phase as shown in Fig. 5.

In the map phase, the input data is split into some units first, and then each worker thread executes the first partition on each unit, yielding $p_1$ partitions. The write collision may occur in this process, so we can choose one scheme described in the Sect. 2.2 to avoid the collision. Because the input data is split into equal-size units, the load imbalance does not happen there. In the reduce phase, each reduce thread executes the second partition on each partition created by the first pass, yielding $p_1 * p_2$ final partitions. Note that: (1) $p_1$ partitions are not equal-size, so the load imbalance may occur in the reduce phase. However, with the help of MapReduce framework and our optimizations, our implementation can solve this problem well. (2) There is not write collision in the reduce phase, because every tuple in the one of final partitions only comes from the one partition created by the first pass. Our Multi-pass partition Optimization obtains a good performance. The detailed experimental results will be given in the next section.

4. Experimental Evaluations

In this section, we show the experimental results of both No partition hash join implementation and partition hash join implementation. We have implemented the hash join in C++ program. Because we focus the performance of memory-resident data, we do not count the time to load the data into main memory and only consider the hash join operation time. We run our experiments on Intel multi-core platform with Sandy Bridge architecture. The processor runs at 2.6GHz and has eight processor cores that support SMT form with Sandy Bridge architecture. The processor runs Intel Sandy Bridge.

Table 1 Platform used in our evaluation.

|                | Intel Sandy Bridge |
|----------------|--------------------|
| CPU            | Xeon E5-2670 @ 2.6GHz |
| Cores          | 8/16               |
| L1 Cache Size  | 32KB per core      |
| L2 Cache Size  | 256KB per core     |
| L3 Cache Size  | 20MB Shared        |
| Cache line     | 64Bytes            |
| Memory         | 4+8GB DDR3 1600MHz |

Table 2 Datasets used in our evaluation.

| Dataset A | Dataset B |
|-----------|-----------|
| Size of id: 8/8 bytes | 4/4 bytes |
| Size of R: $16 * 2^{20}$ tuples (256MB) | $128 * 10^6$ tuples (977MB) |
| Size of S: $256 * 2^{20}$ tuples (4.09GB) | $128 * 10^6$ tuples (977MB) |

Fig. 6 Cycles per output tuple in No partition hash join (Dataset A, the number of hash buckets: 2M).

Each column is represented as an array of fixed size pair <id, value> where id is used to combine columns and value represents the true value of the tuple in this column. To simulate the in-memory hash join in column-oriented database, we choose <id, value> tuple configuration. We pick two datasets, dataset A and dataset B shown in Table 2. In dataset A (from [3]), id and value are eight Bytes, the size of R is 16M tuples and the size of S is 256M tuples. In dataset B (from [15]), id and value are four Bytes, the size of R and S are all 128M tuples. Thus, we can compare our results with existing work [3, 15].

Besides these two datasets with uniform distribution, we also add skew to the distribution of id in the relation S of dataset A (adding skew to relation R can violate the primary key constraint). We use three different values of Zipf distribution in the relation S of dataset A (factor $z$ of Zipf distribution: $z=0.75, z=1.05, z=1.25$). With these three skewed datasets, we can study the effect of skewed dataset for the performance of hash join.

4.2 Results of No Partition Hash Join

The split time is so short that we do not need to show it in the following figures. The cost of No partition hash join includes two parts, build MapReduce cost and probe MapReduce cost. We choose the dataset A in this experiment.

The overall execution times of the build phase and the
_probe phase (as cycles per output tuple) are shown in Fig. 6. The build phase takes a small portion of the overall time, because the main operation in the build phase is copying the data into the appropriate hash bucket. So the performance of No partition hash join is mostly determined by the probe phase. In Fig. 6, as the number of threads increases, the build time and probe time decrease dramatically at first, and then decrease a little after 8 threads. There are two reasons for this behavior. Firstly, as the number of threads increases, the TLB miss and L3 cache miss also increase. Secondly, the memory access becomes the performance bottleneck when the number of threads is large. The Fig. 7 confirms that SIMD optimization can improve the performance of probe phase significantly, and our code with optimizations described above is faster than the code of [4].

Table 3 shows the numbers of cache misses and TLB misses for No partition hash join with a varying number of threads. From the table, we can see L3 misses and TLB misses increase a bit from 4 threads to 8 threads, and then increase apparently after 8 threads. This is because these resources are shared by all threads, and more working threads lead to more competitions for hardware resources with each other, especially with the other thread in the same core. We also can see that L2 misses reach bottom when the number of threads is equal to the number of physical cores. This is because every core has a separate L2 cache, and the number of L2 caches increases from 4 threads to 8 threads. After 8 threads, the L2 misses increase due to the competition for hardware resources with the other thread in the same core.

Here is the Table 3 shown in the text:

| Number of threads | Build | Probe | Build | Probe | Build | Probe |
|-------------------|-------|-------|-------|-------|-------|-------|
|                   |       |       |       |       |       |       |
| L2 Misses         | 61    | 535   | 48    | 426   | 72    | 514   |
| L3 Misses         | 23    | 303   | 25    | 318   | 32    | 381   |
| TLB Misses        | 40    | 341   | 44    | 353   | 56    | 406   |

4.3 Results of Partition Hash Join

The cost of partition hash join also includes two parts, partition-build MapReduce cost and partition-probe MapReduce cost. We start this experiment with the dataset A.

Figure 8 shows the number of CPU cycles spent to produce one output tuple for a varying number of threads. In this experiment, we adopt the two-pass partition with 4096 partitions. The chart confirms that the partition phase takes the large portion of the overall time, even though we adopt the two-pass partition. Moving from left to right, we observe that as the number of threads increases the number of CPU cycles in the build phase decreases at first, and then increases a bit after 8 threads, meanwhile, the partition phase and the probe phase performs better. There are two reasons for this phenomenon. First, the partition phase and the build phase do not need so much CPU computation resource, so the decisive factor is memory access rather than CPU resource when the number of threads increases. Second, if there are two threads working in one core, it possibly causes cache miss sometimes, especially in the build phase. Figure 9 confirms that the SIMD optimization can also improve the performance of probe phase in partition hash join. The
Table 4  CPU performance count for partition hash join (in Millions).

| Number of threads | 4   | 8   | 16  |
|-------------------|-----|-----|-----|
|                   | Part. | Build | Probe | Part. | Build | Probe | Part. | Build | Probe |
| L2 Misses         | 20   | 1.8  | 14   | 17   | 1.2  | 12   | 25   | 2.4   | 18   |
| L3 Misses         | 8    | 0.8  | 5    | 8    | 0.8  | 7    | 10   | 1.4   | 9    |
| TLB Misses        | 204  | 0.2  | 1    | 223  | 0.2  | 1.1  | 240  | 0.2   | 1.4  |

Fig. 10  Elapsed time for different numbers of partitions in one-pass partition and two-pass partition (Dataset: 256M tuples, the number of threads: 16).

probe phase does not take a large portion of the overall time, so the improvement of SIMD optimization is not so significant as that in No partition hash join. Figure 9 also confirms that our code has better performance than the previous work [4], due to the use of MapReduce for hash join and our optimizations described above.

Table 4 shows the numbers of cache misses and TLB misses for partition hash join with a varying number of threads. Compared with No partition hash table, every partition has a separate hash table, so the cache misses decrease significantly in build phase. Because partitions fit into L2 cache, the cache misses also decrease significantly in probe phase. The reason for the change of cache miss and TLB miss with a varying number of threads is similar to that in the No partition hash join.

4.4 Effect of Partitions in Partition Hash Join

In this experiment, we use the relation S of dataset A as input data, and compare the performance between the one-pass partition and the two-pass partition. The execution times with various numbers of partitions and passes are shown in Fig. 10. With the increase in the number of partitions, the one-pass partition time increases, and the two-pass partition performs best when the number of partitions is 256. The reason is that memory and TLB costs are very low with small numbers of partitions, and increase significantly with a growing number of partitions, especially in the one-pass partition. With more than 256 partitions, the two-pass partition exceeds the one-pass partition, due to the reduction of memory and TLB costs.

Figure 11 illustrates the costs of partition phase and probe phase in cycles per output tuple in the partition-probe MapReduce job for different numbers of partitions. In this experiment, we also use the relation S of dataset A, and adopt the two-pass partition in the partition-probe MapReduce job. The figure confirms the expected behavior that the partition cost increases and the probe cost reduces with an increasing number of partitions. In practice, there is a sweet spot for the probe phase, because the high number of partitions (e.g. 256K) leads to poor space utilization and memory pressure. When the size of each partition fits in the L2 cache (i.e. 16384 partitions), the probe phase performs best. But taking into account the partition phase, the partition-probe MapReduce job performs best when the number of partitions reaches 4096.

4.5 Effect of Different Datasets

The results of experiments on different datasets are summarized in following figures. Figure 12 shows the performance of No partition hash join on different datasets, and Fig. 13 show the performance of partition hash join on dif-
different datasets. We can see that the cost of probe phase on dataset A is a little more than the cost on dataset B, even though the number of S tuples in dataset A is twice than that in dataset B. It is because the cost of build phase on dataset A is minimized. We can also see that there is a smaller gap between No partition hash join and partition hash join, only when the sizes of R tuples and S tuples are very different. This is because the smaller hash table in No partition hash join phase leads to good performance of cache and main memory.

4.6 Effect of Skewed Data

In the experiment, we study the effect of skewed dataset for the performance of hash joins, including No partition hash join and partition hash join. We use a uniform dataset (dataset A) and three skewed datasets (factor z of Zipf distribution: z=0.75, z=1.05, z=1.25). The experimental results of hash joins on different skewed datasets are shown in Fig. 14. In the same factor z, the left column shows the result of No partition hash join, and the right column shows the result of partition hash join. From Fig. 14, we can see that: (1) skewed data can improve the performance of No partition hash join, and the performance of partition hash join remains almost stable, (2) partition hash join performs better than No partition hash join in low skew case, and No partition hash join perform better in high skew case.

For No partition hash join, we can see that the overall performance of No partition hash join improves with increasing skew. There are two reasons for improved performance of probe phase. First, No partition hash join can ensure an even workload distribution for all threads, because all threads work concurrently on the relation which is not partitioned. This does not lead to load imbalance, which often happens in partition hash join. Second, skewed data can reduce number of cache misses when they are in the probe phase, because their matching tuples are often stored in cache.

For partition hash join, we can see that the performance remains almost stable. During the partition phase, skewed datasets may produce partitions with different size. Therefore, the overall time is determined by the largest partition. Although skewed data may increase the probe cost, with the help of MapReduce framework and optimizations above our implementation can solve the load imbalance problem, and keep the performance stable in high skew case.

5. Conclusion and Future Work

Facing the changes and future trend of modern CPUs, the paper is helpful for future work. In this paper, we show that: (1) how to solve the collision between threads, which may be a big problem when the number of cores in CPU continues to increase; (2) how to reduce memory access cost, and how to improve cache and TLB hit rates; (3) how to utilize parallelism at different levels efficiently. These points will have good prospect in the new hardware environment.

We also implement hash join algorithms with a shared-MapReduce framework, including No partition hash join and partition hash join. Then we design an improved cuckoo hash table, which consists of a cuckoo hash table and a chained hash table. In addition, we propose some optimizations for our hash join implementations. We finally evaluate our implementations on Intel multi-core platform with Sandy Bridge architecture. The results show that: (1) our two hash join implementations achieve better results than the preview works, (2) the partition hash join often outperforms the No partition hash join, and the memory access becomes performance bottleneck when the number of threads increases, (3) the number of buckets, the number of partitions, the number of threads and dataset can also affect join performance.

There are some directions for future work. We can further improve our hash table to get better performance in both write and read. We will further consider how to reduce memory access and synchronization costs, and how to improve cache and TLB hit rates. Another important future direction is to implement other join algorithms with MapReduce model on multi-core CPUs, such as sort-merge join and non-indexed join.

References

[1] A. Ailamaki, D.J. DeWitt, M.D. Hill, and D.A. Wood, “DBMSs on a modern processor: Where does time go?,” Proceedings of the 25th
YUAN et al.: OPTIMIZING HASH JOIN WITH MAPREDUCE ON MULTI-CORE CPUs

International Conference, VLDB’99, Edinburgh, pp.266–277, Sept. 1999.

[2] C. Balkesen, J. Teubner, G. Alonso, and M.T. Ozsu, “Main-memory hash join on multi-core CPUs: Tuning to the underlying hardware,” Proceedings of the 29th International Conference on Data Engineering, ICDE’2013, Brisbane, pp.362–373, April 2013.

[3] S. Blanas, Y. Li, and J.M. Patel, “Design and evaluation of main memory hash join algorithms for multi-core CPUs,” Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD’2011, Athens, pp.37–48, June 2011.

[4] S. Blanas, Y. Li, and J.M. Patel, “Source code of main-memory hash join algorithms for multi-core CPUs,” Available on: http://pages.cs.wisc.edu/~sblanas/files/multijoin.tar.bz2.

[5] J. Cieslewicz, K.A. Ross, and I. Giannakakis, “Parallel Buffers for Chip Multiprocessors,” Proceedings of the 3rd International Workshop on Data Management on New Hardware, DaMoN’2007, Beijing, pp.1–10, June 2007.

[6] J. Cieslewicz and K.A. Ross, “Data partitioning on chip multiprocessors,” Proceedings of the 4th International Workshop on Data Management on New Hardware, DaMoN’2008, Vancouver, pp.25–34, June 2008.

[7] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” Proceedings of the 6th Symposium on Operating Systems Design and Implementation, OSDI’2004, San Francisco, pp.137–149, Dec. 2004.

[8] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker, and D. Wood, “Implementation techniques for main memory database systems,” Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD’1984, Boston, pp.1–8, June 1984.

[9] U. Drepper, What every programmer should know about memory, 2007.

[10] U. Erlingsson, M. Manasse, and F. Mcsherry, “A cool and practical alternative to traditional hash table,” Proceedings of the seventh Workshop on Distributed Data and Structures, WDAS’2006, Santa Clara, pp.31–36, Jan. 2006.

[11] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, “Relational joins on graphics processors,” Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD’2008, Vancouver, pp.511–524, June 2008.

[12] W. Fang, B. He, Q. Luo, and N.K. Govindaraju, “Marx: Accelerating MapReduce with Graphics Processors,” IEEE Transactions on Parallel and Distributed Systems, vol.22, no.4, pp.608–620, 2011.

[13] J.L. Hennessy and D.A. Patterson, Computer Architecture - A Quantitative Approach. Morgan Kaufmann, 5th ed., 2012.

[14] T. Kaldewaij, G. Lohman, R. Mueller, and P. Volk, “GPU join processing revisited,” Proceedings of the 8th International Workshop on Data Management on New Hardware, DaMoN’2012, Scottsdale, pp.55–62, May 2012.

[15] C. Kim, T. Kaldewaij, V.W. Lee, E. Seidlar, A.D. Nguyen, N. Satish, J. Chhugani, A.D. Blas, and P. Dubey, “Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs,” Proceedings of the VLDB Endowment, vol.2, no.2, pp.1378–1389, 2009.

[16] X. Li, D.G. Andersen, M. Kaminsky, and M.J. Freedman, “Algorithmic improvements for fast concurrent cuckoo hashing,” Proceedings of the 9th European Conference on Computer Systems, EuroSys’2014, Amsterdam, pp.374–386, April 2014.

[17] S. Manegold, P. Boncz, and M. Kersten, “Optimizing main-memory join on modern hardware,” IEEE Transactions on Knowledge and Data Engineering, vol.14, no.4, pp.709–730, 2002.

[18] S. Manegold, P.A. Boncz, and M.L. Kersten, “What happens during a join? Dissecting CPU and memory optimization effects,” Proceedings of the 26th International Conference on Very Large Data Bases, VLDB’2000, Cairo, pp.339–350, Sept. 2000.

[19] R. Pagh and F.F. Rodler, “Cuckoo hashing,” Journal of Algorithms, vol.51, no.2, pp.122–144, 2004.

[20] H. Pirk, E. Petrali, S. Ideos, S. Manegold, and M. Kersten, “Database Cracking: Fancy Scan, not Poor Man’s Sort!,” Proceedings of the 10th International Workshop on Data Management on New Hardware, DaMoN’2014, Snowbird, June 2014.

[21] C. Ranger, R. Raghuraman, A. Pennetsa, G. Bradski, and C. Kozyrakis, “Evaluating MapReduce for multi-core and multiprocessor systems,” Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture, HPCA’2007, Scottsdale, pp.13–24, Feb. 2007.

[22] K.A. Ross, “Efficient hash probes on modern processors,” Proceedings of the 23rd International Conference on Data Engineering, ICDE’2007, Istanbul, pp.1297–1301, April 2007.

[23] A. Shatdal, C. Kant, and J.F. Naughton, “Cache conscious algorithms for relational query processing,” Proceedings of the 20th International Conference on Very Large Data Bases, VLDB’1994, Santiago, pp.510–521, Sept. 1994.

[24] J. Zhou and K.A. Ross, “Implementing Database Operations Using SIMD Instructions,” Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD’2002, Madison, pp.145–156, June 2002.