Massively Parallel NUMA-aware Hash Joins

Harald Lang, Viktor Leis, Martina-Cezara Albutiu,
Thomas Neumann, and Alfons Kemper

Technische Universität München
firstname.lastname@in.tum.de

Abstract. Driven by the two main hardware trends increasing main memory and massively parallel multi-core processing in the past few years, there has been much research effort in parallelizing well-known join algorithms. However, the non-uniform memory access (NUMA) of these architectures to main memory has only gained limited attention in the design of these algorithms. We study recent proposals of main memory hash join implementations and identify their major performance problems on NUMA architectures. We then develop a NUMA-aware hash join for massively parallel environments, and show how the specific implementation details affect the performance on a NUMA system. Our experimental evaluation shows that a carefully engineered hash join implementation outperforms previous high performance hash joins by a factor of more than two, resulting in an unprecedented throughput of 3/4 billion join argument tuples per second.

1 Introduction

The recent developments of hardware providing huge main memory capacities and a large number of cores led to the emergence of main memory database systems and a high research effort in the context of parallel database operators. In particular, the probably most important operator, the equi-join, has been investigated. Blanas et al. [1] and Kim et al. [2] presented very high performing variants of hash join outperforming prior implementations by factors.

So far, those algorithms only considered hardware environments with uniform access latency and bandwidth over the complete main memory. With the advent of architectures which scale main memory via non-uniform memory access, the need for NUMA-aware algorithms arises. While in [3] we redesigned the classic sort/merge join for multi-core NUMA machines, we now concentrate on redesigning the other classic join method, the hash join.

In this paper we present our approach of a NUMA-aware hash join. The core improvement concerns the design of the build input's hash table. We optimized its parallelism via a lock-free synchronization mechanism based on optimistic validation instead of a costly pessimistic locking/latching, as illustrated in Figure 1. Also, we devised a NUMA-optimized storage layout for the hash table in order to effectively utilize the aggregated memory bandwidth of all NUMA nodes. In addition, we engineered the hash table such that (unavoidable) collisions are locally

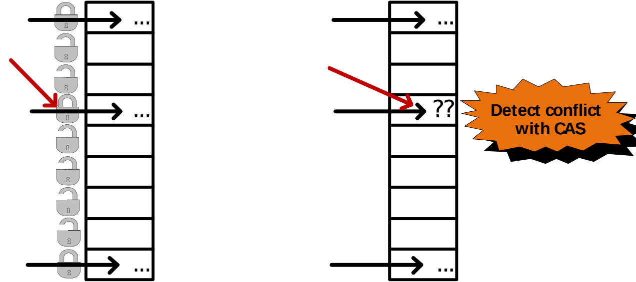

Fig. 1: Pessimistic vs. optimistic write access to a hash table

consolidated, i.e., within the same cache line. These improvements resulted in a performance gain of an order of magnitude compared to the recently published multi-core hash join of Blanas et al. [1]. Meanwhile Balkesen et al. [4] also studied the results of [1] and published hardware optimized re-implementations of those algorithms [5] which also far outperform the previous ones. They focused their research on multi-core CPU architectures with uniform memory access, albeit the source code contains rudimentary NUMA support which improves performance by a factor of 4 on our NUMA machine.

Throughout the paper we refer to the hash join algorithms as described in [1]:

1. **No** partitioning join: A simple algorithm without partitioning phase that creates a single shared hash table during the build phase.
2. **Shared** partitioning join: Both input relations are partitioned. Thereby, the target partitions' write buffers are shared among all threads.
3. **Independent** partitioning join: All threads perform the partitioning phase independently from each other. They first locally create parts of the target partitions which are linked together after all threads have finished their (independent) work.
4. **Radix** partitioning join: Both input relations are radix-partitioned in parallel. Thereby, the partitioning is done in multiple passes by applying the algorithm recursively. The algorithm was originally proposed by Manegold et al. [6] and further revised in [2].

We started to work with the original code provided by Blanas et al. on a system with uniform memory access, on which we were able to reproduce the published results. By contrast, when executing the code on our NUMA system (which is described in section 4) we noticed a decreased performance with all algorithms. We identified three major problems of the algorithms.

1. **Fine-grained locking** while building the hash table reduces parallelism, which is not just NUMA related, but becomes more critical with an increasing number of concurrently running threads.
2. **Extensive remote memory accesses** to shared data structures (e.g., the shared partitions' write buffers of the radix partitioning join) which reside within a single NUMA node. This results in link contention and thus decreased performance.

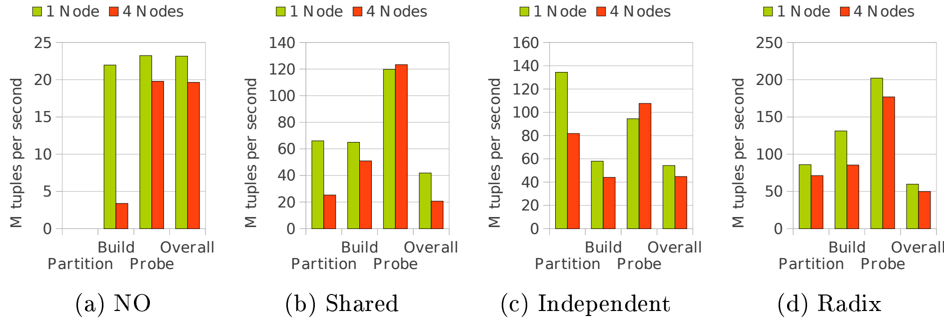

Fig. 2: Performance of the algorithms presented in [1] on a NUMA system, when 8 threads are restricted to 1 memory node, or distributed over 4 nodes

3. **Accessing multiple memory locations within a tight loop** increases latencies and creates additional overhead by the cache coherence protocol which is more costly on NUMA systems.

In the following section we examine the effects on the given implementations that are mostly caused by non-uniform memory accesses. In section 3 we focus on how to implement a hash join operator in a NUMA-aware way. Here we address the main challenges for hash join algorithms on modern architectures: Reduce synchronization costs, reduce random access patterns to memory, and optimize for limited memory bandwidth. The results of the experimental evaluations are discussed in section 4.

2 NUMA Effects

To make the NUMA effects visible (and the changes comparable) we re-ran the original experiments with the *uniform data set* in two different configurations. First we employed eight threads on eight physical cores within a single NUMA node, thereby simulating a uniform-memory-access machine. Then, we distributed the threads equally over all 4 nodes, i.e., 2 cores per node.

Figure 2 shows the performance of the individual hash join implementations. It gives an overview how the join phases are influenced by NUMA effects. The performance of all implementations decreases. Only the shared-partitioning and the independent-partitioning algorithms show a slightly better performance during the probe phase. The no-partitioning and shared-partitioning algorithms are most affected at the build and the partition phase, respectively. In both phases they extensively write to shared data structures. The build performance drops by 85% and the performance of the partitioning phase by 62%. The overall performance decreases by 25% in average in the given scenario.

In contrast to the original results we can see that the build performance is always slower than the probe performance, which we provoked by shuffling the input. However, due to synchronization overhead it is reasonable that building

a hash table is slower than probing it. Therefore, the build phase becomes more important, especially when the ratio $|R|/|S|$ becomes greater. This is the reason why we pay additional attention to the build phase in the following section.

3 NUMA-aware Hash Join

3.1 Synchronization

Synchronization in a hash join with a single shared hash table is intensively needed during the build phase where the build input is read and the tuples are copied to their corresponding hash buckets. Here it is guaranteed that the hash table will not be probed until the build phase has been finished. Additionally, it will no longer be modified after the build phase has been finished. Therefore no synchronization is necessary during the later probe phase. Another crucial part are the write buffers which are accessed concurrently. Especially the shared partitioning algorithm makes heavy use of locks during the partitioning phase where all threads write concurrently to the same buffers. This causes higher lock contention with an increasing number of threads. In this paper we only focus on the synchronization aspects of hash tables.

There are many ways to implement a thread safe hash table. One fundamental design decision is the synchronization mechanism. The implementation provided by Blanas et al. [1] uses a very concise *spin-lock* which only reserves a single byte in memory. Each lock protects a single hash bucket, whereas each bucket can store two tuples. In the given implementation, all locks are stored within a contiguous array. Unfortunately, this design decision has some drawbacks that affect the build phase. For every write access to the hash table, we have to access (at least) two different cache lines. Whereas, the one that holds the lock is accessed twice. Once for acquiring and once for releasing the lock after the bucket has been modified. This greatly increases memory latencies and has been identified as one of the three major bottlenecks (listed in section 1). We can reduce the negative effects by modifying the buckets' data structure so that each bucket additionally holds its corresponding lock. Balkesen et al. [4] also identified this as a bottleneck on systems with uniform memory access. Especially on NUMA systems, we have to deal with higher latencies and we therefore expect an even higher impact on the build performance. In the later experimental evaluation (Section 4) we show how the lock placement affects the performance of our own hash table. Thereby, we also consider the case, where a single lock is responsible for multiple hash buckets.

For our hash table we use an optimistic, lock-free approach instead of locks. The design was motivated by the observation that hash tables for a join are insert-only during the build phase, then lookup-only during the probe phase, but updates and deletions are not performed. The buckets are implemented as triples (h, k, v), where h contains the hash value of the key k and v holds the value (payload). In all our experiments we (realistically for large databases) use 8 bytes of memory for each component. We use h as a marker which signals whether a bucket is empty or already in use. During the build phase, the threads first check

if the marker is set. If the corresponding bucket is empty they exchange the value zero by the hash value within an atomic *Compare-and-Swap* operation (CAS). If meanwhile the marker has already been set by another thread, the atomic operation fails and we linearly probe, i.e., try again on the next write position. Once the CAS operation succeeds the corresponding thread implicitly has exclusive write access to the corresponding bucket and no further synchronization is needed for storing the tuple. We only have to establish a barrier between the two phases to ensure that all key-value pairs have been written before we start probing the hash table.

3.2 Memory Allocation

In this section we describe the effects of local and remote memory access as well as what programmers have to consider when allocating and initializing main memory. On NUMA systems we can directly access the whole available memory, but we (at least) have to differentiate between local and remote memory. Accessing local memory is cheaper than accessing remote memory. The costs depend on how the NUMA partitions are connected and therefore this is hardware dependent. In our system the four nodes are fully connected though we always need to pass exactly one QPI link (hop) when accessing remote memory. By default the system allocates memory within that node the requesting thread is running on. This behavior can be changed by using the `numactl` tool. Especially the command line argument `-interleave=all` tells the operating system to interleave memory allocations among all available nodes, an option which non-NUMA aware programs may benefit from. It might be an indicator for optimization potentials if a program runs faster on interleaved memory, whereas NUMA-aware programs may suffer due to loss of control over memory allocations. We show these effects in our experiments.

For data intensive algorithms we have to consider where to place the data the algorithm operates on. In C++ memory is usually allocated dynamically using the *new* operator or the *malloc* function. This simply reserves memory but as long as the newly allocated memory has not been initialized (e.g., by using *memset*) the memory is not pinned to a specific NUMA-node. The first access places the destination page within a specific node. If the size of the requested memory exceeds the page size, the memory will then only be partially pinned and does not affect the remaining untouched space. A single contiguous memory area can therefore be distributed among all nodes as long as the number of nodes is less than or equal to the number of memory pages. This can be exposed to keep the implementations simple by just loosing a reasonable amount of control and granularity with respect to data placement.

For evaluation we started with a naive implementation which we improved step-by-step. Our goal was to develop a hash join implementation that performs best when using non-interleaved memory because running a whole DBMS process in interleaved mode might not be an option in real world scenarios. We also avoided to add additional parameters to the hash join, especially we do not want to constrain our implementation to a particular hardware layout. We consider

the general case that the input is equally distributed across the nodes and the corresponding memory location in known to the "nearest" worker thread. We will show that interleaved memory increases performance of non-NUMA-aware implementations, but we will also show in the following section that our hash join performs even better when we take care about the memory allocations by ourselves than leaving it to the operating system.

3.3 Hash Table Design

Hash tables basically use one of two strategies for collision handling: *chaining* or *open addressing*. With chaining, the hash table itself contains only pointers, buckets are allocated on demand and linked to the hash table (or previous buckets). With open addressing, collisions are handled within the hash table itself. That is, when the bucket that a key hashes to is full, more buckets are checked according to a certain *probe sequence* (e.g., linear probing, quadratic probing, etc.). For open addressing we focus on linear probing as this provides higher cache locality than other probe sequences, because a collision during insert as well as during probing likely hits the same cache line. Both strategies have their strengths. While chaining provides better performance during the build phase, linear probing has higher throughput during the probe phase. For real world scenarios the build input is typically (much) smaller than the probe input. We therefore chose to employ linear probing for our hash join implementation.

3.4 Implementation Details

In listing 1.1 we sketch the insert function of our hash table. In line 2 we compute the hash value of the given key (more details on hash functions in Section 4.3) and in line 3 the bucket number is computed by masking all bits of the hash value to zero that would exceed the hash table's size. The size of the hash table is always a power of two and the number of buckets is set to at least twice the size of the build input. Thus, for n input tuples we get the number of buckets $b = 2^{\lceil log_2(n) \rceil + 1}$ and the $mask = b - 1$. The relatively generous space consumption for the hash table is more than compensated by the fact that the probe input, which is often orders of magnitude larger than the build input, can be kept in-place. The radix join, in contrast, partitions both input relations.

Listing 1.1: Atomic insert function

```
1  insertAtomic(uint64_t key, uint64_t value) {
2    uint64_t hash = hashFunction(key);
3    uint64_t pos = hash & mask;
4    while (table[pos].h != 0
5        || (! CAS(&table[pos].h, 0, hash))) {
6      pos = (pos + 1) & mask;
7    }
8    table[pos].k = key;
9    table[pos].v = value;
10 }
```

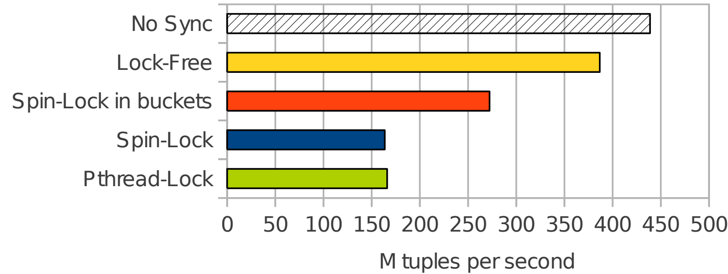

Fig. 3: Build performance using different synchronization mechanisms

Within the condition of the while loop (line 4 to 7) we first check, if the bucket is empty. If this is the case the atomic CAS function is called as described in section 3.1. If either the hash value does not equal zero [1] or the CAS function returns `false`, the bucket number (write position) is incremented and we try again. Once the control flow reaches line 8 the current thread has gained write access to the bucket at position `pos` where the key-value pair is stored.

The notable aspect here is that there is no corresponding operation to giving up an acquired lock. Usually a thread acquires a lock, modifies the bucket, and finally gives up the lock, which establishes a happened-before relationship between modification and unlocking. In our implementation the CPU is free to defer the modification or to execute them in an out of order manner because we do not have any data dependencies until the probe phase starts. Further, we optimized for sequential memory accesses in case of collisions by applying the open addressing scheme with a linear probing sequence for collision resolution. This strategy leads to a well predictable access pattern which the hardware prefetcher can exploit.

4 Evaluation

We conducted our experiments on a Linux server (kernel 3.5.0) with 1 TB main memory and 4 Intel Xeon X7560 CPUs clocked at 2.27 GHz with 8 physical cores (16 hardware contexts) each, resulting in a total of 32 cores and, due to hyperthreading, 64 hardware contexts. Unless stated otherwise we use all available hardware contexts.

4.1 Synchronization

In our first experiment we measure the effect of different synchronization mechanisms on build performance. To reduce measurement variations we increased the

[1] *hashFunction* sets the most significant bit of the hash value to 1 and thus ensures no hash value equals 0. This limits the hash domain to 2^{63}, but does not increase the number of collisions, since the least significant bits determine the hash table position.

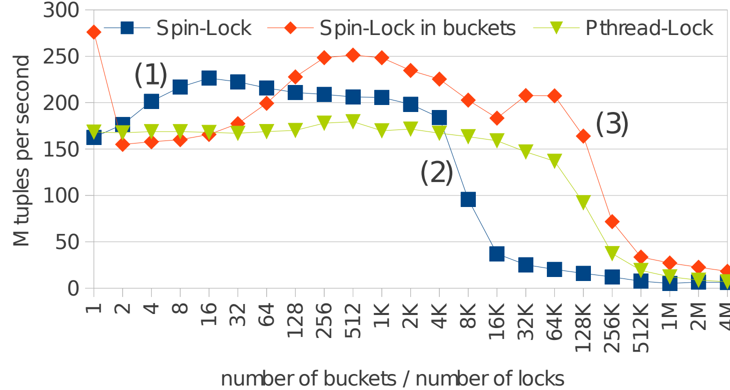

Fig. 4: Effects of lock-striping on the build phase

cardinality of the build input R to 128 M tuples. Again we used a uniform data set with unique 64 bit join keys. The results are shown in Figure 3. We compared the original spin-lock implementation with the POSIX-threads mutex and our lock-free implementation. While the spin-lock and the pthreads implementation offer almost the same performance, our lock-free implementation outperforms them by factor 2.3. We can also see a performance improvement of 1.7 x when placing the lock within the hash bucket instead of placing all locks in a separate (contiguous) memory area. The hatched bar (labeled "No Sync") represents the theoretical value for the case where synchronization costs would be zero.

In the second experiment we reduce the number of locks n that are synchronizing write accesses to the hash buckets. We start with one lock per bucket and successively halve the number of locks in every run. Therefore a lock becomes responsible for multiple hash buckets ("lock striping"). In our case these are the neighboring buckets. The ith lock is responsible for the buckets $\left[i \cdot \frac{b}{n},\ (i+1) \cdot \frac{b}{n}\right)$, where n is the number of locks, b is the total number of buckets and $b \geq 2 \cdot |R|$ $\wedge$ b is a power of two.

In Figure 4 we can see that the build performance can be significantly increased by reducing the number of locks. Especially when using the spin-lock it shows a steep growth in the beginning. Blanas' spin-lock implementation only needs a single byte and therefore a single cache line can hold up to 64 locks. Reducing the total number of locks leads to fewer cache misses when locks are acquired (marker (1) in the plot). Because we implemented linear probing it is very unlikely that a hash collision leads to a cache miss when acquiring the lock of the neighboring bucket. On the other hand, the spin-lock performance drops significantly when multiple threads access locks that reside in the same cache-line (2). Even if there is low lock contention, manipulating the same cache line causes cache coherency misses in order to keep the caches coherent. This effect cannot be observed when the locks are stored in the buckets. Here, we just can see the natural performance drop due to higher lock contention (3). In

contrast to the 1-byte spin-lock, we can not observe those caching effects with the pthreads mutex. This is due to the fact that a single lock uses 40 bytes.

Our experiment also revealed that using lock striping together with locks that are stored in the hash buckets is very cache inefficient. The more buckets a single lock is responsible for, the higher is the propability that a single hash table access incur two cache line loads (for the lock and for the bucket).

The experiments confirmed that an efficient lock implementation is crucial for the build phase. It also showed that protecting multiple buckets with a single lock indeed can have positive effects on the performance but cannot compete with a lock-free implementation. Especially the first two data points of the "Spin-Lock in buckets" curve show that on NUMA architectures writing to two different cache lines within a tight loop can cause crucial performance differences.

4.2 Memory Allocation

For the experimental evaluation of the different memory allocation strategies we consider the build and the probe phase separately. We focus on how they are affected by those strategies, but we also plot the overall performance for completeness. To get good visual results we set the cardinality of both relations to the same value (128 M). During all experiments we only count and do not materialize the output tuples. We use the following four setups:

1) **non-NUMA-aware**: Allocation of the input data and the hash table takes place within a single NUMA node.
2) **interleaved**: Same as 1) but running in `-interleave=all` mode.
3) **NUMA-aware / dynamic**: Both input relations are allocated and initialized thread-local whereas the hash tables' memory is initialized dynamically during runtime.
4) **NUMA-aware**: As in 3), both relations are placed thread-local and the memory of the hash table is (manually) initialized across all NUMA nodes in chunks of page size.

Figure 5 shows the results of all four experiments. We measured the performance of the build and probe phase as well as the overall performance in M tuples per second. The distributed memory allocation of the hash table in 4) is done as follows: We divide the size of the hash table into i equally sized chunks of size 2 MB and let them be initialized by all threads in parallel where the ith chunk is "memsetted" by thread $i \bmod \#threads$.

We can see an improvement by a factor of more than three by just running the non-NUMA-aware implementation on interleaved memory. When comparing the setup 3) with 2) a decreased performance during the build phase can be seen which is caused by the dynamic allocation of the hash tables' memory. Finally the 4th setup shows the best performance. Our own implementation, that simulates an interleaved memory only for the hash tables' memory achieves (approximately) the same build performance as in the second setup, but we can increase the performance of the probe phase by additional 188 mtps, because we are not enforced to read the probe input from interleaved memory.

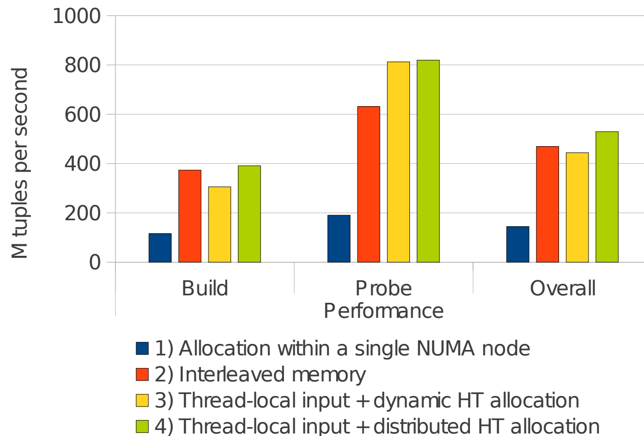

Fig. 5: Experimental results of different data placement / memory allocation strategies

Performance Comparisons			
Setup, $\|R\|/\|S\|$	our NO	Radix [5]	VectorWise
16 M / 16 M	503 mtps	147 mtps	
16 M / 160 M	742 mtps	346 mtps	
32 M / 32 M	505 mtps	142 mtps	
32 M / 320 M	740 mtps	280 mtps	
1 G / 1 G	493 mtps	-	
1 G / 10 G	682 mtps	-	50 mtps

Table 1: Performance comparisons NO vs. Radix, using a uniform data set (key/foreign-key join)

We assume that in practice the relations are (equally) distributed across all memory partitions and we only need to assign the nearest input to each thread.

Table 1 shows comparisons with the Radix join implementation of [5]. Unfortunately, the Radix implementation was not able to handle extremely large workloads such as 1 G / 10 G (176 GB of data). For comparison, the TPC-H record holder VectorWise achieves 50 mtps for this large join [3].

4.3 Hash Functions

In accordance to previous publications, and in order to obtain comparable performance results, we used the modulo hash function (implemented using a logical AND, as discussed in Section 3.4) in all experiments. In this section we study the influence of hash functions on join performance. On the one hand, modulo hashing is extremely fast and has good join performance in micro benchmarks. On the other hand, it is quite easy to construct workloads that cause dramatic performance degradation. For example, instead of using consecutive integers, we left gaps between the join keys so that only every tenth value of the key space was

used. As a consequence, we measured a 84% decrease performance for the NO implementation of [5]. Whereas our implementation is affected by power-of-two gaps, and slows down by 63% when we use a join key distance of 16.

We evaluated a small number of hash functions (Murmur64A, CRC, and Fibonacci hashing) with our hash join implementation. It turned out that the Murmur hash always offers (almost) the same performance independent from the tested workload. At the same time it is the most expensive hash function, which reduces the overall join performance by 36% (over modulo hashing with consecutive keys). The CRC function is available as a hardware instruction on modern CPUs with the SSE 4.2 instruction set and therefore reduces the performance by less than 1% in most cases. However, it is less robust than Murmur, for a small number of workloads it caused significantly more collisions than Murmur. The Fibonacci hash function offered almost the same performance as modulo, but unfortunately had the same weaknesses.

Real-world hashing naturally incurs higher cost, but does not affect all algorithms equally. Employing a costly hash function affects the Radix join more than the NO join, because the hash function is evaluated multiple times for each tuple (during each partitioning phase, and in the final probe phase). Finally, using more realistic hash functions makes the results more comparable to algorithms that do not use hashing like sort/merge joins.

5 Related Work

Parallel join processing has been investigated extensively, in particular since the advent of main memory databases. Thereby, most approaches are based on the radix join, which was pioneered by the MonetDB group [7, 6]. This join method improves cache locality by continuously partitioning into ever smaller chunks that ultimately fit into the cache. Ailamaki et al. [8] improved cache locality during the probing phase of the hash join using software controlled prefetching. Our hash join virtually always incurs only one cache miss per lookup or insert, due open addressing.

An Intel/Oracle team [2] adapted hash join to multi-core CPUs. They also investigated sort-merge join and hypothesized that due to architectural trends of wider SIMD, more cores, and smaller memory bandwidth per core sort-merge join is likely to outperform hash join on upcoming chip multiprocessors. Blanas et al. [1, 9] and Balkesen et al. [4, 5] presented even better performance results for their parallel hash join variants. However, these algorithms are not optimized for NUMA environments.

Albutiu et al. [3] presented a NUMA-aware design of sort-based join algorithms, which was improved by Li et al. [11] to avoid cross-traffic.

6 Summary and Conclusions

Modern hardware architectures with huge main memory capacities and increasing number of cores have led to the development of highly parallel in-memory

algorithms for main memory database systems. The focus thereby is on hash-based algorithms [1, 2]. However, prior work did not yet consider architectures with non-uniform memory access. We identified the challenges that NUMA poses to hash join algorithms and based on our findings we developed our own algorithm based on optimistic validation instead of a costly pessimistic locking. Our algorithm distributes data carefully in order to provide balanced bandwidth on the inter-partition links. At the same time, no architecture-specific knowledge is required, i.e., the algorithm is oblivious to the specific NUMA topology. Our hash join outperforms previous parallel hash join implementations on a NUMA system. We further found that our highly parallel shared hash table implementation performs better than radix partitioned variants because these incur a high overhead for partitioning. This is the case although hash joins inherently do not exhibit cache locality as they are inserting and probing the hash table randomly. But at least we could avoid additional cache misses due to collisions by employing linear probing. We therefore conclude that cache effects are less decisive for multi-core hash joins. On large setups we achieved a join performance of more than 740 M tuples per second, which is more than 2 x compared to the best known radix join published in [5] and one order of magnitude faster than the best-in-breed commercial database system VectorWise.

References

1. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core CPUs. In: SIGMOD. (2011)
2. Kim, C., Sedlar, E., Chhugani, J., Kaldewey, T., Nguyen, A.D., Blas, A.D., Lee, V.W., Satish, N., Dubey, P.: Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. PVLDB **2** (2009)
3. Albutiu, M.C., Kemper, A., Neumann, T.: Massively parallel sort-merge joins in main memory multi-core database systems. PVLDB **5** (2012)
4. Balkesen, C., Teubner, J., Alonso, G., Özsu, T.: Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In: ICDE. (2013)
5. Balkesen, C., Teubner, J., Alonso, G., Özsu, T.: Source code. (`http://www.systems.ethz.ch/sites/default/files/multicore-hashjoins-0_1_tar.gz`)
6. Manegold, S., Boncz, P.A., Kersten, M.L.: Optimizing Main-Memory Join on Modern Hardware. IEEE Trans. Knowl. Data Eng. **14** (2002)
7. Boncz, P.A., Manegold, S., Kersten, M.L.: Database architecture optimized for the new bottleneck: Memory access. In: VLDB. (1999)
8. Chen, S., Ailamaki, A., Gibbons, P.B., Mowry, T.C.: Improving hash join performance through prefetching. ACM Trans. Database Syst. **32** (2007)
9. Blanas, S., Patel, J.M.: How efficient is our radix join implementation? `http://pages.cs.wisc.edu/~sblanas/files/comparison.pdf` (2011)
10. Porobic, D., Pandis, I., Branco, M., Tözün, P., Ailamaki, A.: OLTP on hardware islands. PVLDB **5** (2012)
11. Li, Y., Pandis, I., Mueller, R., Raman, V., Lohman, G.: NUMA-aware algorithms: the case of data shuffling. In: CIDR. (2013)