An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory

Stefan Schuh, Xiao Chen, Jens Dittrich
Information Systems Group
Saarland University
http://infosys.cs.uni-saarland.de

ABSTRACT

Relational equi-joins are at the heart of almost every query plan. They have been studied, improved, and reexamined on a regular basis since the existence of the database community. In the past four years several new join algorithms have been proposed and experimentally evaluated. Some of those papers contradict each other in their experimental findings. This makes it surprisingly hard to answer a very simple question: what is the fastest join algorithm in 2015? In this paper we will try to develop an answer. We start with an end-to-end black box comparison of the most important methods. Afterwards, we inspect the internals of these algorithms in a white box comparison. We derive improved variants of state-of-the-art join algorithms by applying optimizations like software-write combine buffers, various hash table implementations, as well as NUMA-awareness in terms of data placement and scheduling. We also inspect various radix partitioning strategies. Eventually, we are in the position to perform a comprehensive comparison of thirteen different join algorithms. We factor in scaling effects in terms of size of the input datasets, the number of threads, different page sizes, and data distributions. Furthermore, we analyze the impact of various joins on an (unchanged) TPC-H query. Finally, we conclude with a list of major lessons learned from our study and a guideline for practitioners implementing massive main-memory joins. As is the case with almost all algorithms in databases, we will learn that there is no single best join algorithm. Each algorithm has its strength and weaknesses and shines in different areas of the parameter space.

1. INTRODUCTION

Database research is full of traditions. One of our most prominent traditions is to publish new join algorithms every year. And, yes, we mean equi-joins, on relational data; not some fancy approximate similarity join on probabilistic JSON data streams. Isn't this kind of magic that after 40 years of database research, there is still progress in an area that is at the core of almost every query plan? Shouldn't relational equi-join algorithms be a solved problem anyway? So, why should we care?

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882917

When taking a deep look at the abundant recent related work on relational equi-joins from 2011 to 2015 [7, 6, 14, 3, 17, 4, 5], it quickly turns out that based on that literature it is surprisingly hard to decide which is the best join algorithm. For instance, it seems clear from [4] that a hash-based approach outperforms sort based approaches, at least on current hardware with the limited SIMD register width available today. However, for the different hash table-based join implementations it is unclear if hardware-conscious partition-based algorithm (the PRB-family of join algorithms) outperforms non-partitioning hardware-oblivious algorithms (the NOP-family of join algorithms) anyway. For instance, in 2011, the experiments in [7] show that NOP outperforms PRB. However, in 2013, the experiments in [4] show exactly the opposite: PRB outperforms NOP. In another work in 2013, [14] again, NOP outperforms PRB. So which algorithm is better? In odd years it is NOP? In even years it is PRB? The answer is: we do not know. The more interesting question is: why do those experiments report contradicting results? There are several reasons:

1. **different implementations** were used, e.g. in [7] a very basic NOP implementation was used where the concurrent chaining hash table was implemented using linked lists and two separate arrays were used for locks and head pointers. In contrast, [5] provided a more cache-efficient chaining hash table implementation which used a single array for both locks and tuples and removed head pointers. NOP from [14] implemented a lock free hash table by using linear probing and Compare-and-Swap instructions. Another example is PRB where [5] reimplemented the PRB algorithm from [7] since the implementation from [7] had too many function calls and pointer dereferencing in critical code paths.

2. **different optimizations** were applied to the different join algorithms, e.g. in [4, 6] software-write combine buffers were used for PRB, in [14, 7] they were not used, for [5] it is unclear wether software-write combine buffers were used for the results presented in the paper. Moreover, [14] used a linear probing hash-table as its hash-table implementation whereas [5, 4, 6] used chained hashing. In addition, [7, 4] did not consider NUMA-aware join processing at all, while in [14, 4] at least the join relations as well as working memory is distributed over all NUMA nodes.

3. **different performance metrics** were used, e.g. different definitions of "join throughput", e.g. in [4] it is defined as the number of join results produced per second, i.e. $Throughput = \frac{|R \bowtie S|}{|\text{total runtime}|}$. Notice, that this definition focusses on the amount of tuples output by the join algorithm. Hence, it is sensitive to the join selectivity. In contrast, in [14] throughput is defined as the ratio of the sum of the relation sizes and the total

runtime, i.e. $Throughput = \frac{|R| + |S|}{|\text{total runtime}|}$. This definition focusses on the input of the join algorithms. Hence, this definition is independent of the join selectivity. We will use the latter definition in our study.

4. **different machines**, e.g. the machine used in [7] is a single socket six-core Intel Xeon X5650 Nehalem. In contrast, [5] reports the results for several architectures with the best performance achieved on an eight-core Intel Xeon E5-2680 server. In [14] the authors used four eight-core Intel Xeon X7560 Nehalem CPUs in their experiments. Similarly, [5] used four eight-core Intel CPUs however with the Sandy Bridge architecture.

5. **difference of micro-benchmarks and real queries**, e.g. [7, 6, 5, 14, 3, 17, 4, 13] did not consider total query execution times of real world (or at least TPC-H) queries, that include several attributes that have to be considered for predicate evaluation after the join processing. However, simple techniques like selection push-down may considerably reduce the input sizes to a join algorithm (even though the unfiltered relations are huge). In such a situation the choice of the join algorithm may become less crucial. In addition, tuple reconstruction has been shown to have a substantial effect on the overall runtime of a query [2]. Whether a join is run with or without tuple reconstruction makes a huge difference in practice. However, none of these effects were evaluated in those works.

Any of these differences alone may have a substantial effect on the runtime of a join algorithm or its interpretation. Accumulating multiple of those differences makes a comparison very hard and comparing results from different papers becomes almost infeasible.

Therefore, we believe the time is ripe for a clean slate experimental comparison of relational equi-joins. This paper fills this gap. We will focus on hash-based join algorithms as almost all recent works suggest that these algorithms are the most promising ones. Still, we will use a modern sort-based approach as one baseline [4][1]. We do not further explore sort-based joins as the evidence from recent work suggests that sort-based joins cannot match the performance of other joins. We will evaluate all algorithms in the context of a modern NUMA (non-uniform memory access) architecture.

The algorithms we evaluate in our study are based on algorithms published in four recent papers [17, 4, 14, 5]. Notice that those papers in turn improve upon several other older papers [13, 7, 3]. We will introduce several variants of those algorithms. In total, in our study, we evaluate **thirteen different join algorithms**.

In this paper we make the following contributions:

1. **Black box Comparison.** We start with the core join algorithms from [14, 5, 17] which are already improved versions of join algorithms proposed in [7, 13] or have been shown to outperform join algorithms proposed in [3]. We will start with a black box end-to-end comparison of these four principal join algorithms in Section 4.

2. **White box Comparison.** We proceed by performing a white box comparison. We enable all optimizations mentioned in prior work and explore their effects on the runtime performance of the joins, see Section 5.1. We then proceed by evaluating the effect of different hash-table implementations in Section 5.2, for both NOP and PRB.

[1]We also wanted to use a second sort-based baseline [14]. However, that code was not available.

3. **Optimizing Join Algorithms.** In order to improve the join performance on NUMA architectures, we optimize the different versions of the PRB join algorithm. First, We will show how to make the partitioning phase NUMA-aware in Section 6.1 and also see that we can improve over PRB by 20%. Second, we will show in Section 6.2 that a NUMA-aware join task scheduling can also improve the performance by roughly 20%. These improvements are not cumulative as the NUMA-aware partitioning already makes use of all NUMA nodes in the join phase and therefore does not profit from a NUMA-aware scheduling.

4. **Comprehensive Comparison of Joins.** Finally, we will be in the position to perform a comprehensive comparison of all join algorithms, see Section 7. First, we define a common workload for all methods. We also do a reality check and use another meaningful baseline. For primary key columns it is common sense to use automatically generated integer IDs. This leads to a dense key domain of integers. For this distribution a simple array rather than a hash table may be a surprisingly good choice. Though the practicability of this approach may be questioned if the join key domain is sparse, it serves at least as a baseline on how good a join algorithm may get anyway. Additionally, for non-dense distributions a compressed array like the recently suggested [17] may be a good choice to avoid using a full-blown hash table.

   Second, we proceed by evaluating the page size of the virtual memory management on all phases of the different joins. This parameter has a surprising effect on runtime.

   Third, we look at the scalability of the join methods in terms of the input relation sizes. In this context we propose a strategy for choosing the right number of bits for partitioning the input relations in radix-based joins.

   Fourth, we explore the behavior of the join algorithms under moderately skewed and highly skewed data (Appendix A).

   Fifth, we examine the scalability of the different joins in terms of the number of threads and discuss possible reasons for the different scalability characteristics (Appendix B).

   Sixth, we evaluate the feasibility of using array joins in moderately dense domains (Appendix C).

   Seventh, we eveluate microarchitectural performance aspects (Appendix D).

5. **Effect on real queries.** We will evaluate the effect of the choice of the join algorithm in the context of a real TPC-H query in Section 8. We will start with a simple single join query, TPC-H Q19. We will analyze the portion of the query time spent in the actual join. We also provide an additional experiment changing the selectivity of Q19's predicates (see Appendix E). Additionally, see Appendices F and G.

6. **Key Lessons Learned.** Finally, we conclude by identifying the key lessons learned from our experimental study, see Section 9.

2. RELATED WORK

We focus on papers that discuss in-memory joins on multicore systems; we are aware that there is also a lot of related work on sorting, hashing or partitioning in memory, which is of course highly related to join processing. That work will be cited in other sections whenever appropriate. In the following we will discuss related work in chronological order.

Kim et. al. [13] revisited the sort vs hash argument in the context of main memory multi-core system by comparing a hash join and a sort-merge join that are optimized for modern multi-core systems. The hash join algorithm they introduced is called parallel radix hash join. Their experimental results showed that the parallel radix hash join outperforms the sort-merge join by a factor of two. They also developed an analytical model for the join performance and predicted that the sort-merge joins will become faster with the following two future hardware trends. First, *Wider SIMD instructions*: the sort-merge join algorithm scaled near-linearly with the width of SIMD instructions in their models, while the hash join algorithm hardly exploits the capability of SIMD instructions. Second, *Limited per-core bandwidth*: the hash join algorithm needs at least two pass partitioning for large data sets due to the limited number of TLB cache entries, which result in at least three trips to main memory, while the sort-merge join only needs two. With limited per-core bandwidth more memory trips would lead to worse performance.

Blanas et. al. [7] extended the categories of main memory multi-core join algorithms by introducing no-partitioning hash join. Unlike partition-based algorithm like parallel radix hash join [13], the no-partitioning hash join is a straightforward parallellised version of canonical (simple) hash join without partitioning the data at all. Blanas implemented the no-partitioning hash join with a lock-based concurrent chaining hash table and compared it with partition-based algorithms using three different partitioning algorithms: shared partitioning, independent partitioning, and the radix partitioning from Kim et. al. [13]. Their experimental results show that the no-partitioning hash join outperforms all partition-based hash joins for almost all data distributions and is only slightly slower than parallel radix hash join with uniform datasets.

A later work by Albutiu et. al. [3] on sort-merge join further extended the design space of main memory join algorithms by giving focus on optimizing for NUMA systems. The Massively Parallel Sort-Merge join (MPSM) proposed in their paper uses a carefully tuned memory access pattern and avoids inter-thread synchronization as much as possible. Their experimental results show that MPSM runs much faster than no-partitioning hash join [7] and radix hash join [15] (the latter implemented in Vectorwise on a 32-core, 4-socket machine). Unfortunately, the authors did not make their code available to us. Hence, we will use the sort-based join from Balkesen et. al. [4] as the sort-based baseline as it is freely available and additionally was shown to be superior to MPSM.

Balkesen et. al. [5] investigated the parallel radix hash join and no-partitioning hash join algorithms from Blanas et. al. [7] and proposed better variants for both algorithms. They achieved higher throughputs by improving the cache efficiency of the hash table implementations for both algorithms and adopting a better skew handling mechanism for parallel radix hash join. We discuss this algorithm (called PRB in our paper) in more detail in Section 3. In their experimental results, using their new implementations, the parallel radix hash join outperforms the no-partitioning hash join for almost all architectures and workloads — except on Niagara2, which has 8 threads per core, for a workload using a very large probe relation. This result is contradicting the conclusions made by Blanas et. al. [7].

The picture changed again when Lang et. al. [14] published their NUMA-aware no-partitioning hash join. In their results, their no-partitioning hash join method outperforms the parallel radix hash join from Balkesen et. al. [5] by a factor of more than two on a 4-socket machine with 64 hardware contexts. We discuss the no-partition algorithm (called NOP in our paper) from Lang et. al. in Section 3.

Another work by Balkesen et. al. [4] improved the sort-merge join from Kim et. al. [13] and their own parallel radix join from [5]. Their sort-merge join uses wider SIMD instructions and uses range partitioning to allow for efficient multithreading without heavy synchronization. We discuss the proposed sort-merge join (called MWAY in our paper) in more detail in Section 3. In their experimental results, in contrast to the prediction from Kim et. al. [13], wider SIMD instructions did not yet make sort-merge join superior to parallel radix hash join. In fact, in their experiments, parallel radix hash join still always outperforms sort-merge join.

While all the previous research focused on the runtime of join algorithms, the work from Barber [17] studied the memory-efficiency of hash join methods. They proposed a highly memory efficient linear probing hash table called concise hash table (CHT). We also discuss this algorithm (called CHTJ in our paper) in more detail in Section 3. They compared their method with the no-partitioning hash join and the parallel radix hash join from [5]. Their experimental results show that they can reduce the memory usage by one to three orders of magnitude with competitive performance.

Finally, there has been work optimizing joins for specialized architectures including GPUs, e.g. [12], hybrid CPU-GPU architectures, e.g. [10], and coprocessors attached through PCI express cards like Intel's Xeon Phi, e.g. [11, 18]. Though these are all interesting works they are way beyond the scope of this paper. We will focus on modern server CPUs which are still abundant.

3. FUNDAMENTAL REPRESENTATIVES OF MAIN-MEMORY JOIN ALGORITHMS

In summary, we can identify three fundamental classes of join algorithms into which the most recently published join algorithms fall. Namely: (1) partition-based hash joins, (2) no-partitioning hash joins, and (3) sort-merge joins.

In the following we will discuss one or two modern variants of each class in more detail. This discussion will serve as the starting point of our study.

Join Class	**Modern Variants Introduced in Paper**
Partition-based Hash Joins	[13],[7],[5],[4]
No-partitioning Hash Joins	[7],[5],[14],[17]
Sort-merge Joins	[13],[3],[4]

Table 1: Join algorithms from Section 2 and their assignment to classes

Table 1 shows this classification assigning the papers discussed in Section 2 to their corresponding class. Notice that some of those papers, e.g. [7], presented algorithms from multiple classes.

3.1 Partition-based Hash Joins

Core Idea: *Partition-based Hash Joins partition the input relations into small pairs of partitions (co-partitions) where one of the partitions typically fits into one of the caches. The overall goal of this method is to minimize the number of cache misses when building and probing hash tables.*

PRB is the two-pass parallel radix hash join described in [5]. A problem with partitioning joins is that different partitions will most likely reside on different memory pages. Thus, randomly writing tuples to a large number of partitions may cause excessive TLB

Join	Description	Paper	Code
Fundamental Classes of Join Algorithms (Section 3) &Black box comparison (Section 4)			
PRB	Basic two-pass parallel radix join without software managed buffer and non-temporal streaming	[5]	Original
NOP	No-partitioning hash join	[14]	Original
CHTJ	Concise hash table join	[17]	Own
MWAY	Multi-way sort merge join	[4]	Original
White box comparison (Section 5)			
NOPA	Same as NOP except using an array as the hash table	This	Modified
PRO	One-pass parallel radix join with software managed buffer and non-temporal streaming	[5]	Original
PRL	Same as PRO except using linear probing hashing instead of bucket chaining	This	Modified
PRA	Same as PRO except using arrays as hash tables	This	Modified
Optimizing Parallel Radix Join (Section 6)			
CPRL	Chunked parallel radix join with software managed buffer and non-temporal streaming	This	Own
CPRA	Same as CPRL except using arrays as hash tables	This	Own
PROiS	PRO with improved scheduling	This	Modified
PRLiS	Same as PROiS except using linear probing hashing instead of bucket chaining	This	Modified
PRAiS	PRA with improved scheduling	This	Modified

Table 2: reference table for the algorithms evaluated in this paper

misses. In order to fix this problem, **PRB** uses two-pass partitioning to guarantee that the number of partitions does not exceed the number of TLB entries. The first partitioning pass starts by assigning equal-sized regions (chunks) to each thread. The algorithm precomputes the output memory ranges of each target partition by building histograms. Hence, each thread knows where and how much to write for each partition without the need for further synchronization. After histograms have been built, each thread scans the input relation and writes each tuple to its destination region. The first partitioning pass already produces a considerable number of partitions. Therefore, in order to perform the second partitioning pass, entire sub-partitions (rather than chunks of a partition as done in the first partitioning pass) are assigned to worker threads by using a task queue. If required, skew handling may be done to break up larger partitions further by assigning multiple threads to an individual partition. In the join phase, each thread takes one co-partition at a time and runs a textbook hash join algorithm on it using a chained hash table.

3.2 No-partitioning Hash Joins

Core Idea: *No-partitioning hash joins concurrently build a single global hash table. Simultaneous multi-threading and out-of-order execution is used to hide cache miss penalties automatically. In contrast to partition-based joins, no knowledge about the hardware cache sizes or number of TLB entries is required for tuning.*

NOP is the no-partitioning hash join described in [14]. It uses a lock-free synchronization mechanism for a linear probing hash table using compare-and-swap. The algorithm starts by assigning equal-sized regions (chunks) to each thread. Each thread then inserts its chunk of the build relation into the global hash table. After all threads are done inserting, each thread starts probing its chunk of the probe relation against the global hash table.

For inserts into the global hash table, each thread uses an atomic Compare-and-Swap (CAS) operation. This is a transactional and conditional operation that is only executed if the slot contains the empty key; in that case the empty key is overwritten by the key to be inserted. Otherwise the operation returns false. As entries are never removed or overwritten in a slot, the thread can copy the payload to the bucket in an additional non-transactional operation. In addition to the lock-free hash table, another optimization of NOP is to interleave hash table allocation among all available NUMA nodes for better memory bandwidth utilization.

CHTJ is the concise hash table join described in [17]. At its core a Concise Hash Table (CHT) consists of four major components. First, an array A of size n storing all n inserted tuples without any additional empty slots. Second, a hash function $hash : Key \mapsto [8 \cdot n]$[2], where Key denotes the domain of the join keys and $[8 \cdot n]$ denotes the set of all integers in the range from 1 to $8 \cdot n$. Third, a bitmap B of size $8 \cdot n$. This bitmap marks if a certain hash bucket is occupied. Fourth, a population count array PC of size $n/4$ with a running sum of the population count of the bitmap, i.e. for for every 32 bits of the bitmap we count the number of elements stored up to that point. A CHT is a static structure that is bulkloaded once and then used for lookups only. Hence, this structure is very suitable for join processing. Notice that Google sparse hash map [8] is very similar to CHT, but additionally allows for inserts and deletes.

In a CHT, a lookup for a *key* works as follows: we first check if the bit $B(hash(key))$ is set. If that is the case, this implies that array slot $hash(key)$ is occupied. In other words: there *may* be a result. In that case, from the population count array PC we retrieve the population count from position $\lfloor hash(key)/32 \rfloor - 1$ and add it to the number of bits set in B within the range

$$\big[\lfloor hash(key)/32 \rfloor \cdot 32;\ hash(key) - 1\big].$$

In order to speed up this process B and PC are physically interleaved in the same structure.

CHTJ works as follows: first it radix-partitions the build input into a small number of partitions very similar to PRB. Second, one global CHT is allocated where each thread bulkloads its partition to a disjoint region in that CHT. Hence, there is no need for additional synchronization at this point. Fourth, the probe relation is handled similarly to NOP: each thread probes one chunk of the probe relation against the global CHT. Again, as no inserts are performed in the CHT at this point, there is no need for synchronization.

We classify CHTJ as a no-partition hash join even though the algorithm uses partitioning on the build input. However, that partitioning is only used to build the global CHT in parallel. Afterwards the algorithms is equal to NOP as discussed above. Moreover, in CHTJ the partitioning is not used to run independent joins on co-groups like in PRB.

3.3 Sort-merge Joins

Core Idea: *Sort-merge joins belong to the oldest join methods used in databases. The idea is to first sort both input relations on their join keys, if they are not yet sorted, and to use an efficient merge step afterwards to find all matching tuples. Sort-merge join algorithms can exploit and create so-called interesting orders. Even if the performance of a single join in a complex multi-join query would be suboptimal, the overall performance of the sort-merge join plan could be superior.*

MWAY is the m-way sort merge join described in [4]. It is also very similar to the method described in [13]. **MWAY** partitions the data very similar to **PRB**, however using only a single partitioning phase and creating only few partitions. In addition, software

[2]For simplicity we assume here that n is a power of 2.

write-combine buffers (see Section 5.1) are used. After partitioning, each partition will be merge-sorted independently by a separate thread. The merge-sort is implemented with bitonic sorting- and merge-networks. Both sort- and merge-networks are vectorized using SIMD instructions. In addition, multi-way merging is used to save memory bandwidth.

4. BLACK BOX COMPARISONS

In this section we want to compare the representatives of the fundamental join algorithms that were also compared in prior work. We want to identify some of the reasons for contradicting results in previous works. Table 2 lists all evaluated join algorithms and the abbreviations used. The *paper* column refers to papers where the specific algorithm was published and "this" means that the algorithm is proposed in this paper. The *code* column shows the implementation we used for each algorithm. There are three values for this column: "Original" means we use the implementation from authors of the paper, "Modified" means we modify the original implementation to get the new variant and "Own" means we implemented the algorithm from scratch.

For our experimental evaluations we use a setup that is similar to the one used throughout all mentioned previous join papers. The tuples of the join relations consist of two four byte attributes, namely the join key and the payload. Furthermore, the keys in the smaller relation are dense and unique, like in a primary key column. Throughout this study we use the same 60 core machine, see Section 7.1 for details. Unless stated differently, all algorithms use at most 32 threads even though our machine has 60 cores available. The reason is that the code of the MWAY algorithm only works with a power of two number of threads. In Section B we will increase the number of threads beyond 32 threads. We measure the throughput of a join as $(|R|+|S|)/t_{join}$, i.e. the total number of input tuples of both relations divided by the algorithm runtime.

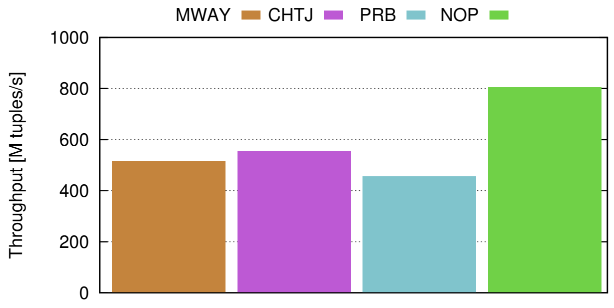


Figure 1: Black box comparison of the fundamental join representatives using 32 threads and relation sizes |R| = 128M and |S| = 1280M.

Figure 1 shows the performance of this black box comparison in terms of throughput. We use inputs of size 128 million and 1280 million tuples respectively. These results are comparable to the results found in [14] and [17], but do not match the findings in [4] as for instance in that study the performance of PRB was found to be much better than MWAY.

To understand this inconsistency between results from prior work, we will take a closer look at the parallel radix partitioning join in the next section.

5. WHITE BOX COMPARISONS

From the prior publications it is not always clear what optimizations were used for the parallel radix join. We therefore take a closer look at different possible optimization for the different methods to make them more comparable.

Let's start by taking a closer look at the code for PRB provided by [1]. We see at least two optimizations that can be enabled.

5.1 Optimizing Radix Partitioning

NUMA-Awareness. The first option is the `-basic-numa` flag that equally allocates the partition buffer on all NUMA nodes, as otherwise the buffer will be allocated randomly over different NUMA regions. This option was most likely enabled in all related work as otherwise the performance of the join algorithm decreases considerably on NUMA machines. Turning on this option has another important effect:

Memory Allocation Locality. Before running the actual join, all physical pages will be allocated locally and mapped in the virtual memory table. Hence, we will not trigger page faults and consequent allocations while running a join algorithm. As database system always have a buffer manager anyways, we believe it is a fair assumption that memory buffers were already physically allocated. We already used this option in the previous section and we will also keep this option turned on throughout all following experiments.

Software Write-Combine Buffers. The second option provided by the code of [1] is the `-enable-swwc-part` flag[3]: this flag enables the use of software write-combine buffers (SWWCB) and non-temporal streaming instructions for the parallel partitioning algorithm. SWWCBs, aka software managed buffers, have been known for quite some time [20]. The idea is to allocate a small local buffer for each partition and first put tuples into buffers instead of directly flushing them to the output. This is similar to buffered writes in disk-based partitioning, however, *the size of each buffer is only one cache line*. SWWCB reduce the pressure on the TLB as

Algorithm 1 Partitioning with Software Write-Combine Buffers

```
1: for all tuple ∈ relation do
2:     partition ← hash(tuple.key);
3:     pos ← slots[partition] mod TuplePerCacheline;
4:     slots[partition]++;
5:     buffer[partition].data[pos] = tuple;
6:     if pos == TuplePerCacheline - 1 then
7:         dest ← slots[partition] - TuplePerCacheline;
8:         copy buffer[partition].data to output[dest];
```

the buffers are very likely to reside in cache and on very few pages while only a full buffer is flushed to main memory. If a buffer can hold N tuples, then the number of TLB misses will be reduced by a factor of N. Algorithm 1 illustrates how SWWCB works. Turning on this option has another important effect:

Non-temporal streaming. This is a technique allowing programmers to write half a cache line to DRAM bypassing all caches. It prevents polluting the caches with data that will never be read again. Recently, Schuhknecht et.al [21] performed an in-depth study of the effects of using both software write-combine buffers and non-temporal streaming on the single-threaded radix partitioning algorithm. We follow the guideline provided in that paper.

[3]To be fair with prior work that did not enable this feature in their comparisons. It is marked as experimental and we applied some fixes that removed minor race conditions that did not influence the performance.

After enabling all those features we obtain a much more efficient algorithm called **PRO** (Parallel Radix with Optimized partitioning).

Single-pass Partitioning. The original PRO used two-pass radix-partitioning. We ran micro-benchmarks on PRO comparing single-pass and two-pass partitioning and also determined the optimal number of partitions to use for partitioning. Figure 2 shows that a single-pass partitioning using 14 bits leads to the highest throughput. We will use this setting in the following for all variants of PRO.

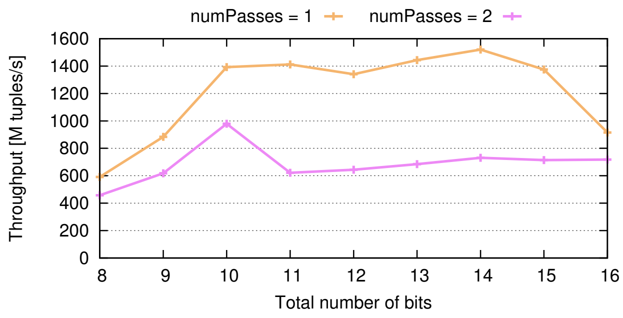


Figure 2: Throughput of PRO for different partition sizes and number of radix bits for partitioning (total join including partitioning and join phase); the two-pass algorithm divides the bits evenly over the two passes.

5.2 Choice of Hash Method

Linear vs Chained vs CHT. Another dimension that makes the available join algorithms harder to compare is the usage of different hash table implementations in the algorithms. All presented hash join methods use different hash table implementations. CHTJ of course uses a concise hash table; PRO uses a variant of chained hashing while the NOP algorithm uses linear probing to implement the hash table. We therefore also implemented a version of PRO that uses a linear probing hash table and call that method **PRL**. An in-depth study for different hashing strategies can be found in [19].

Arrays. When using unique and dense domains of join keys we can go one step further and use an even simpler hash table implementation — a simple array. Instead of storing key value pairs in a hash table we can simply use the key as an index in the array and store the value in that position. This assumption on the key domain may sound unrealistic on first sight, however, often joins are performed along 1:n or n:1 foreign key relationships using artificially created IDs (ID Integer PRIMARY KEY AUTOINCREMENT), this situation may occur frequently in practice. It can also be identified easily by the query optimizer through the available statistics. This simple array implementation can also be used in the NOP-family of joins and hence we get two new hash join variants called No Partition Array join (**NOPA**) and the Parallel Radix partition Array join (**PRA**). These joins are of course not as widely applicable as the other hash joins. We will also investigate the usefulness of these methods in the presence of holes in the key domain in Section C.

Performance Comparison. Considering the optimizations discussed in Sections 5.1 and 5.2, we take another look at the performance of the join algorithms and show their throughput in Figure 3. We can see that PRO clearly outperforms NOP and now the performance of PRO also resembles the results presented in [4]. However, surprisingly, there is almost no difference in runtime between PRA, PRO, and PRL. Therefore, we might conclude that the choice of the hash method does not have an effect on the runtime; however, later on we will learn that this conclusion would be wrong (see Section 6.2).

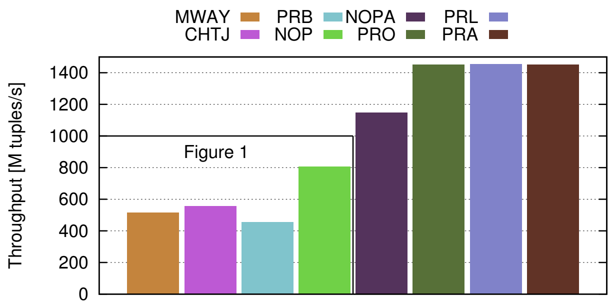


Figure 3: Join throughput including improved versions. We observe almost a twofold performance improvement over the black-box versions shown in Figure 1.

6. OPTIMIZING PARALLEL RADIX JOIN

We have observed in Figure 3 that the parallel radix partitioning joins PR* are providing the highest throughput so far. We are therefore looking for ways to further improve their performance. In the following, we will look at the partition phase and the join phase separately.

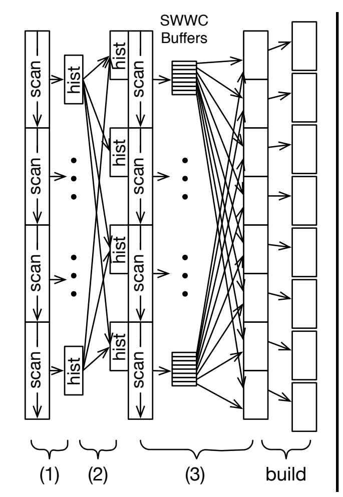
6.1 NUMA-aware Partitioning

First we investigate the partitioning phase. The Parallel Radix Partitioning algorithm is illustrated in Figure 4(a). It works as follows: (1) every thread sequentially reads a horizontal chunk of the input relation to create a local histogram. (2) a global thread merges the local histograms into a global histogram[4]. The goal is to exploit this global histogram later on as an index to the target partitions. To merge the local histograms, we need a synchronization barrier, as every thread has to complete the local histogram before the final output positions can be computed. (3) each thread reads again its horizontal local chunk of the input relation and partitions the data into its corresponding SWWCB. Each thread keeps as many SWWCB as it has target partitions. Whenever a SWWCB becomes full, it is flushed to the final output position in its target partition using non-temporal streaming. As the final output position of every partition was already determined in phase (2), no further synchronization is necessary. For the probe relation, phases (1)–(3) are executed similarly using the same partitioning function. Once both inputs have been partitioned, each pair of corresponding partitions is joined independently. This is done by building a hash table on the left input and probing the right input against that hash table. Hence, from a high-level perspective this join algorithm is a variant of Grace Hash Join applied to NUMA.

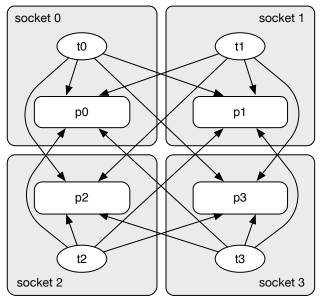
NUMA-partitioning is a task which triggers a considerable number of memory reads and writes, especially when writing out tuples to their destinations. However, in NUMA systems, careless memory access patterns can hurt the performance very badly as remote memory accesses have higher latency and lower bandwidth than local memory accesses. So we devoted our effort on analyzing memory access patterns of PRO and we found out that there are in particular random remote memory writes that we can avoid in this algorithm.

Figure 4(b) depicts the NUMA access pattern in a simplified case with four sockets, four threads, and four partitions when writing tuples to their target partitions. We see that the partitioning algorithm of PRO introduces many random **remote** writes when writing tuples to their corresponding partitions. Based on this observation,

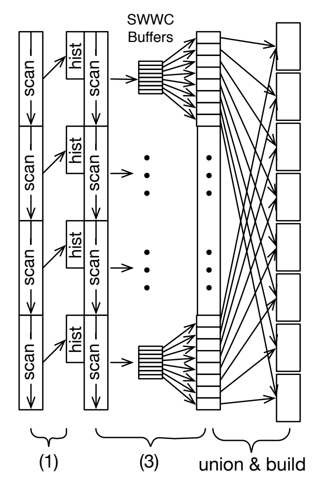
[4]Technically, this may also be implemented by letting the threads merge their histogram independently as in phase (3) each thread only requires a subset of the global histogram.



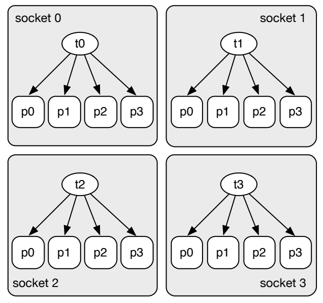
(a) Schematic view on build relation partitioning of PRO



(b) NUMA write pattern for PRO



(c) Schematic view on build relation partitioning of CPRL



(d) NUMA write pattern for CPRL

Figure 4: High-level schematic view and NUMA write pattern: PRO vs CPRL

we propose the Chunked Parallel Radix partition (**CPRL**) algorithm that eliminates remote writes when flushing tuples to partitions.

Figure 4(c) shows a high-level schematic view on our algorithm CPRL. Notice, that the histogram phase (1) is the same as in PRO. However in contrast to PRO, in CPRL we leave out phase (2), i.e. we do not compute a global histogram. We proceed directly with phase (3), i.e. each thread partitions its data locally within its chunk only based on its local histogram. On a high-level this can be viewed as running a single-threaded histogram-based radix-partitioning inside a chunk. For the probe relation phases (1)–(3) are executed similarly using the same partitioning function. Once both inputs have been partitioned, each pair of corresponding partitions, i.e. each co-partition, is joined independently. In contrast to PRO, at this point we neither have physically contiguous probe nor build partitions available. Hence, we first have to read the different chunks belonging to the build input from its (possibly NUMA-remote) sources. In that process, we directly load that data into a local hash table. Then we also read the different chunks belonging to the probe input from its (possibly NUMA-remote) sources and probe them directly against the hash table. Hence, from a high-level perspective this join algorithm is also a variant of Grace Hash Join applied to NUMA. However, in contrast to PRO, we do not require the inputs to each join to be physically contiguous in main memory. Therefore, CPRL trades small random writes to remote memory for large sequential reads from remote memory. Notice that CPRL uses the same linear probing hash table as PRL since it was easier to integrate our own linear hash table implementation. In addition, the linear hash table also provided a slightly better performance compared to the chained hash table implementation by Balkesen et. al. When we use arrays rather than hash tables in the join phase of the Chunked Parallel Radix join, we call it **CPRA**. Again, optimizations like software write-combine buffers and non-temporal store instructions are also used in this algorithm. The write pattern of CPRL is illustrated in Figure 4(d).

Let's compare the performance of PRO, PRL, and PRA with our proposed CPRL and CPRA. Figure 5 shows the end to end join processing time broken down into partition and join phase. We see that the CPR*-algorithms outperform the PR*-algorithms by ∼20%. We also observe that the partitioning times of the CPR*-algorithms are indeed reduced as expected. However, surprisingly

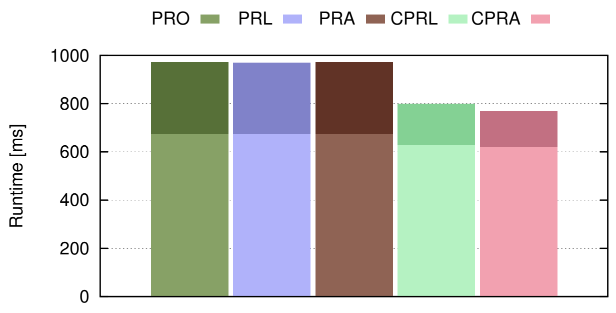


Figure 5: Runtime of PR* vs CPR*-algorithms. Relation sizes: $|R| = 128M$, $|S| = 1280M$. Lighter colors denote the partition phase and darker colors denote the join phase.

even the join time is reduced. This is counterintuitive to what we expected as we traded remote writes in the partitioning phase with remote reads from all sockets in the join phase. We will investigate the reason for this in the next section.

6.2 NUMA-aware Scheduling

All PR*- and CPR*-algorithms build co-partitions which eventually have to be joined independently. How are those individual joins scheduled? What effect does this schedule have on the overall performance of the join algorithms?

After partitioning, in both PR*- and CPR*-algorithms, all co-partitions are put into a LIFO-task queue (which is actually a stack), to be processed by different threads. Recall that the PR*-algorithms partitions an input array into p partitions where for any two partitions $i, j \in [0; p-1], i < j$ it holds that the starting address of partition j is greater than the starting address of partition i. In other words, the partition indices correlate with their virtual addresses. We observed that in all PR*- and CPR*-algorithms, co-partitions are inserted into the queue in ascending sequential indices order, i.e. for p co-partitions the insert order into the queue is $0, \ldots, p-1$. However, recall, that before executing any join, one quarter of each input relation is physically allocated on one of the NUMA-regions. In addition, any additional memory required for partititioning or building hash tables is also equally distributed across NUMA-regions. This memory allocation strategy was already present in the code used by [5]. Assume that the number of threads is considerably smaller than the number of partitions, i.e. $t << p$, typically $p = 16384$ and for our machine $t = 60$. This implies that the first $\lceil 16384/60 \rceil = 274$ partitions reside **on the same** NUMA-region. Hence, all of the first 60 threads removing tasks from the queue will have to read their input data from the same NUMA-region. Moreover, for three quarters of those threads, i.e. 45 threads, this is a remote NUMA-region. Similar bottlenecks can be observed for all other blocks of 274 partitions.

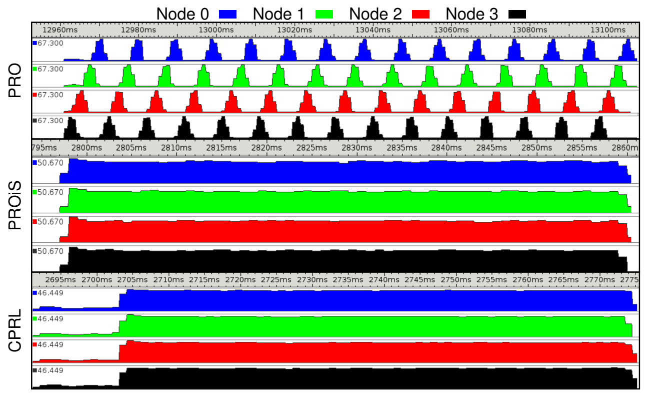


Figure 6: Bandwidth profiles for PRO, PROiS, and CPRL obtained with Intel VTunes

Figure 6 shows the bandwidth profile of PRO. We observe that most of the time PRO uses only a single NUMA-region.

We can improve this by carefully reordering the join tasks. We fixed this by changing the task scheduling strategy used by all PR*-algorithms as follows: we insert co-partitions into the task queue in a round-robin manner. Specifically, we first put a partition from the first NUMA region into the queue and then a partition from the second NUMA region and so on. An alternative would be to use a different queue for each NUMA-region. Like that it is very likely that all memory controllers are utilized simultaneously.

Figure 6 shows a bandwidth plot of the original PRO, the variant of PRO using improved scheduling, coined **PROiS**, as well as CPRL. In addition, we also introduce variants of PRL and PRA using improved scheduling called **PRLiS** and **PRAiS**. We observe that the improved scheduling of PROiS has a substantial effect on the total bandwidth utilization, i.e. all NUMA nodes are used at the same time. Even though the suboptimal scheduling is also used for CPRL, it does not affect the bandwidth utilization, as every partition has to be read from all NUMA nodes anyhow.

The improved scheduling results in a speedup of the join phase of PRL and PRA by more than a factor of 2, see Figure 7. As expected in Section 6.1, we can now observe that the join phase of the CPR*-algorithms is in fact slightly more expensive than the one of the PR*iS-algorithms. However, still, in total the CPR*-algorithms are slightly faster than the PR*iS-algorithms. Moreover, in contrast to our results from in Figure 3, we can now clearly observe that different hash table implementations have an effect on the runtime of the algorithms.

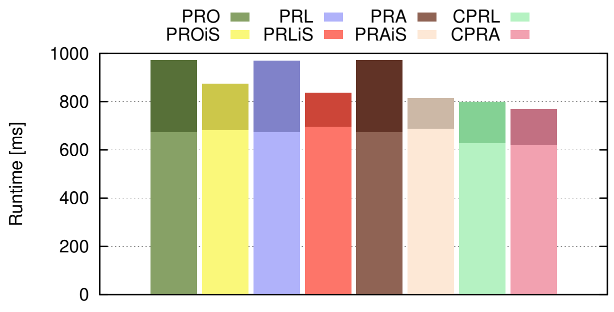


Figure 7: Runtime of PR* and CPR*-algorithms vs their variants with improved scheduling (PR*iS-algorithms). Relation sizes: $|R| = 128M$, $|S| = 1280M$. Lighter colors denote the partition phase and darker colors denote the join phase.

7. PUTTING IT ALL TOGETHER

After our initial black box comparison (Section 4), after having analyzed the effects of optimizing radix partitioning and using different hash tables (Section 5), and after optimizing the NUMA memory allocation and NUMA access pattern of the various radix algorithms (Section 6), we are finally in the position to perform a comprehensive comparison of *all* join algorithms.

In this section we will perform a large-scale experimental study of all thirteen algorithms mentioned above. Recall that Table 2 lists all algorithm abbreviations and their short descriptions. In the pdf of this paper all occurrences of algorithm abbreviations are hyperlinks pointing to their description. We evaluate all algorithms in the same benchmarking framework. Additional experiments including skewed data and semi-dense key domains can be found in the Appendix.

7.1 Settings

All our experiments are performed on a server with half a terabyte of main memory and four Intel Xeon E7-4870 v2 CPUs, clocked at 2.30 GHz (published in Q1 2014). This CPU has 30 hardware contexts executed on 15 physical cores that share a 30 MB L3 cache. Each core has one private 32 KB L1 data, one 32 KB instruction cache, and one 256 KB L2 data cache. Notice that the number of TLB entries when using 4 KB page is 256. However, if we use 2 MB pages, we only have 32 TLB-entries! The operating system we used is a 64-bit Debian 7 server with the kernel version of 3.2.0-4. The CPU supports AVX 1.0. Just like the original studies [13, 4, 3, 14, 6], we also assume a column-oriented storage model and adopt the configuration of using a <Key, Payload> pair as a tuple. We use a 4-byte integer key and a 4-byte integer payload to make a fair comparison between all methods, since some available implementations only work for this key size. We assume that the build relation follows a dense primary key distribution and the keys of the probe relation have a foreign key relationship to the keys of the build relation, if not mentioned otherwise. This setting was also used in the mentioned previous studies.

In the following experiments, we use the implementations of PRO, PRB, NOP from the original authors. PRA, PRL, PRAiS, PROiS, NOPA are implemented based on the authors' implementations. We implemented CPRL, CPRA, and CHTJ ourself from scratch. Since the build relation has dense primary keys, we use the identity hash function modulo the hash table size for all hash joins as it is very effective and efficient in this setting and was also used in the previous studies. Lang et. al. [14] additionally evaluated different hash functions like Murmur, CRC, and multiplicative hashing. We are not evaluating the effect of different hash functions on the join performance in this paper. All software is implemented in C/C++ and compiled by gcc/g++ version 4.7.2 with optimization level -O3.

7.2 Varying Page Sizes

The first dimension we want to explore is the page size of the virtual memory used for the join algorithms. As partitioning is very sensitive to TLB misses, this is a very important aspect. In the previous experiments we used huge page sizes for all allocations, i.e. 2 MB. To study the effect of the page size on the different algorithms we evaluate pages of 4 KB and 2 MB by switching the kernel setting transparent_hugepage/enabled between `never` and `always`. We also ensured that all allocations use the default `malloc` or `posix_memalign` methods.

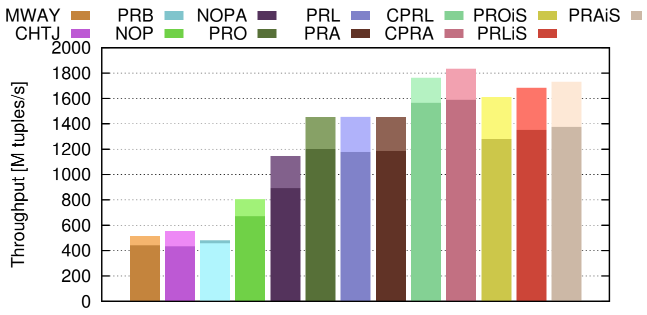


Figure 8: Performance of all thirteen join algorithms when using small (4 KB, dark color) and huge pages (2 MB, light color)

Figure 8 shows the performance of all thirteen join algorithms when using small (4 KB) and huge pages (2 MB). We observe that all algorithms except PRB improve by using huge pages. PRB is the only algorithm that has a slightly worse performance when using huge pages. This is due to the naive partitioning that does not use any software write-combine buffers for the different partitions. In each of the two radix passes PRB partitions along 7 bits = 128 partitions. The entries for all those pages fit into the TLB-cache when using small pages. However, when using huge pages, we only have 32 TLB-entries available. Hence, many write operations to partitions lead to TLB-misses. This effect is mitigated when using SWWCB as in PRO.

As some of the algorithms are clearly dominated by others, in the following, we do not report for results for PRB, PRO, PRL, and PRA anymore. In addition, for all following experiments we use huge pages.

7.3 Scalability in Dataset Size

The next dimension we will explore is the scalability in the size of the input data to the joins. We will explore two workloads: (1) the probe relation is ten times the size of the build relation. The factor ten is motivated by the typical ratio in the TPC-H benchmark and the observation that in a star schema, often used in OLAP applications, the dimension tables are typically much smaller than the fact table. (2) both relations have the same size. This is close to a worst case for hash joins, as the typically more expensive build phase is followed by a rather short probe phase. If the build relation becomes smaller than the probe relation, the optimizer should actually have switched the roles of the relations in the first place. At the same time, this case is close to a best case for sort-based methods, as sorting has super-linear costs and is therefore minimized if both relations have the same size. Previous studies [4, 14] also used similar workloads.

Fine-tuning the partition-based joins. When we started benchmarking the effects of scaling the input datasets, we quickly noticed that the radix-based algorithms are very sensitive to the number of bits used for partitioning. Recall, that in Section 5.1 we already explored this effect for *a fixed-size input dataset*. Following the results of our micro-benchmark in Figure 2 we assumed that when doubling the data size, we take one additional bit for partitioning and hence end up with the same partition size and obtain good performance. But is that really true? Let's take a second look:

Figure 11 shows the average partitioning time per tuple for a varying number of partitions and corresponding data set sizes. On the horizontal axis we use a log scale doubling the number of partitions at every tic. The number of partitions is chosen such that a chained hash table that is built on a single partition fits into L2 cache.

We observe that the average partition time per tuple stays almost constant up to including 2^{15} partitions. Starting with 2^{16} partitions the performance deteriorates. This can be explained with the size of the software write-combine buffers. Recall that we use 32 threads. If we create 2^{15} partitions using a single cache line per partition as an SWWCB, all SWWCBs together including other working variables, e.g. histograms, still fit into the shared last level cache (LLC). However, when using 2^{16} partitions, this is no longer the case.

We conclude from this experiment that partitioning data into too many partitions might overshadow the performance gains obtained in the join phase. We therefore micro-benchmarked the performance of all partition-based joins with a varying number of bits used for partitioning and show these results in Figure 9. In Figures 9(a)&(b) we choose the number of radix bits such that the hash table on a partition fits onto L2. In contrast, in Figures 9(c)&(d) we depict the number of radix bits actually leading to the lowest overall runtime. We can see that choosing the number of bits such that the partitions fit into L2 cache is close to the optimal choice as long as the SWWCBs still fit into the shared LLC. However, for larger datasets, we observe in Figure 9(b) that the partitioning costs increase sharply. Hence, we conclude that for these input sizes it is better to balance the partitioning cost with the join costs. The sweet spot for the number of bits used in partitioning seems to be the minimal number of bits such that the partitions still fit into the shared LLC.

Predicting the optimal number of radix bits. This leads to the following formula for the number of bits for partitioning n_p. Given the size of R as $|R|$, the size of a tuple of R as s_t, the intended load factor of the join hash tables l, the size of a partition buffer as s_b, the size of the L2 cache as $L2$, and the size per thread of the last level cache as LLC_t [5]:

$$n_p(|R|) = \begin{cases} \log_2\left(\frac{|R|\cdot s_t}{l\cdot L2}\right), & \frac{|R|\cdot s_b\cdot s_t}{L2\cdot l} < LLC_t \\ \log_2\left(\frac{|R|\cdot s_t}{l\cdot LLC_t}\right), & \text{otherwise} \end{cases} \quad (1)$$

Figure 12 shows the observed runtime for CPRL[6] when varying the number of radix bits from 8 to 18 bits (black points) versus the performance observed when setting the bits according to Equation (1) (red line). We can see that the number of bits computed by Equation (1) leads in almost all cases to the lowest runtime.

Back to Figure 9, we can also make an additional observation: we see that the different hash table implementations have an effect on the optimal number of bits for partitioning. This is reasonable, as the different hash table implementations differ in their space efficiency. For instance, array joins use a tight array that only keeps the payload, the key however is implicitly represented through the array index. In contrast, a linear probing hash table has to store the key explicitly.

Based on these results, from now on, we will use Equation (1) to set the bits of all PR*- and CPR*-algorithms and are in the position

[5]As the LLC is shared between cores, the available share per thread is dependent on the number of concurrently running threads.

[6]Similar results for the other algorithms are not shown due to space constraints.

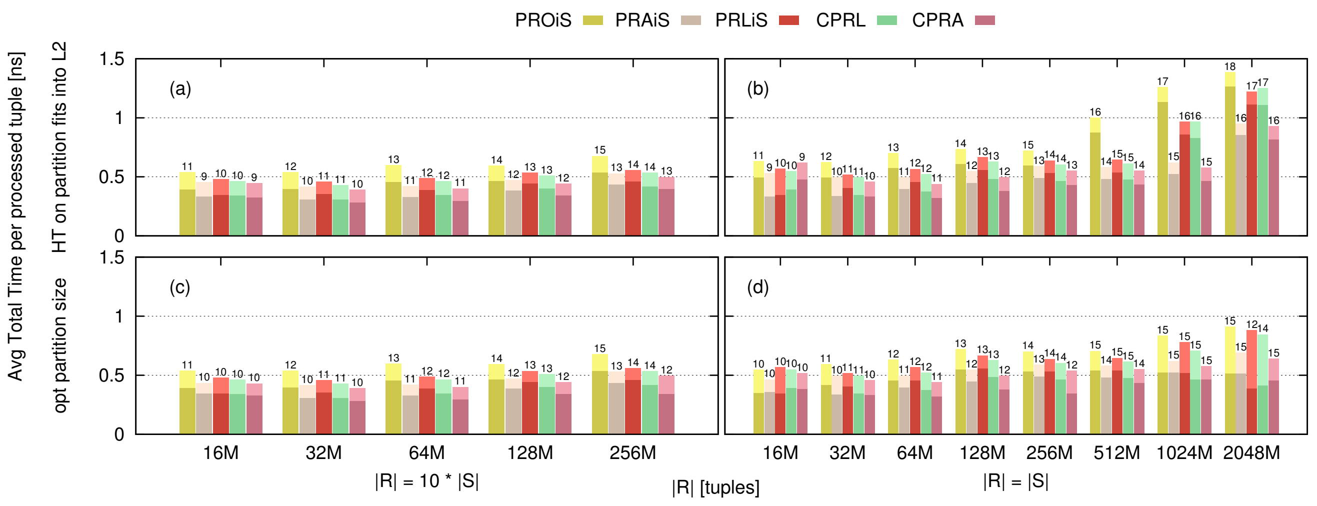


Figure 9: Average total time per tuple (partitioning and join) when varying the number of radix-bits used for partitioning. The dark color marks the time for partitioning; the light color marks the time for joining. In (a) and (b) we choose the number of radix bits such that the hash table on a partition fits onto L2. In contrast, in (c) and (d) we depict the number of radix bits leading to the lowest overall runtime. In particular for $|R| = |S|$ (right column) and $|R| \geq 512$ M tuples we see that our assumption, (a) and (b) diverges heavily from the optimal number of bits, (c) and (d). Notice that we can observe in (b) that the partitioning costs increase heavily whereas the join costs stay the same.

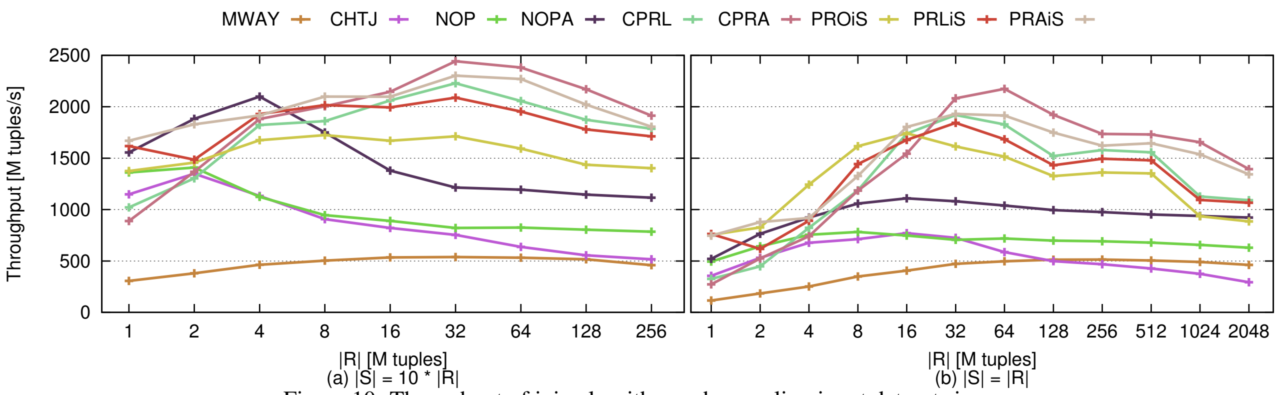


Figure 10: Throughput of join algorithms when scaling input dataset sizes

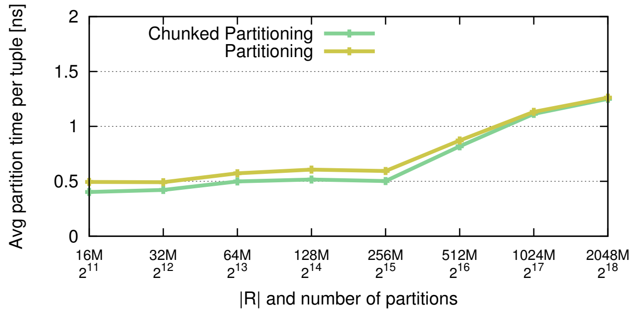


Figure 11: Scalability of the partition phase for chunked and non-chunked partitioning

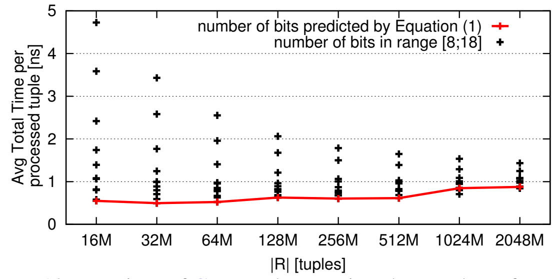


Figure 12: Runtime of CPRL when setting the number of partitioning bits according to Equation (1)

to evaluate the join performance when scaling the sizes of the input datasets.

Scalability results. Finally, Figure 10 contains the performance results for all join methods when scaling the input data size. For the partitioning joins we observe that for very small input sizes, i.e. up to 4M tuples, the various algorithms show similar performance. However, if we scale the input data to larger sizes, we observe that the PR*- and CPR*-algorithms outperform the NOP*-algorithms, CHTJ, and MWAY. In particular, for the NOP*-algorithms we can see from Figure 10 that the throughput is very good up to 4M. However, for larger inputs the throughput decreases. This matches our expectation since no-partitioning join methods need to build a big global hash table. With growing data sizes the global hash table won't fit into the LLC anymore, which in our case is just 30 MB. Hence, the bigger the build relation, the higher the probability for an LLC miss as well a TLB-miss. This effect can be observed very well in Figure 10(a): up to an input size for $|R|$ =32 M tuples there is a decrease in performance. Afterwards the performance does not deteriorate (visibly) anymore as almost all hash table accesses trigger LLC and TLB misses anyway (due to lacking spatial proximity in the caches). In other words, the NOP*-algorithms are already bound by this bottleneck.

The *inverse argument* to this holds for the PR*- and CPR*-algorithms. These algorithms perform *more* memory operations than the NOP*-algorithms. The underlying assumption of these algorithms, however, is that memory accesses to individual tuples may be very expensive, i.e. they may lead to a costly LLC miss. This effect is similar to external memory algorithms trying to avoid individual seeks on disk by bundling operations into larger granules. The PR*- and CPR*-algorithms try to access (and implicitly cache) memory according to larger granules, i.e. partitions and/or memory pages. This algorithmic pattern does not have much of an effect if the input data fits into LLC anyways (then, the underlying assumption of the algorithms simply does not hold). However, once the underlying assumption holds, i.e. the input data exceeds the size of the LLC, these algorithms can efficiently avoid costly DRAM accesses to individual tuples.

Notice that among the NOP*-algorithms, CHTJ is very sensitive to the data size as it needs at least two random accesses for every operation on its CHT. MWAY sort-merge join is another very stable algorithm, it even outperforms the CHTJ for large datasets.

8. EFFECTS ON REAL QUERIES

Up to now we focused on benchmarking raw performance of multithreaded joins. Like that we followed the micro-benchmarking philosophy of previous work [7, 6, 5, 14, 3, 17, 4, 13]. However, micro-benchmarks always trigger the same (and important) question: how big is the impact observed in micro-benchmarks in a larger context, e.g. when joins are used inside a larger query?

The development of a full-fledged multithreaded NUMA-aware query execution engine is beyond the scope of this paper. However, state-of-the-art main-memory databases use code compilation anyways [16], i.e. at query time they translate incoming SQL to machine code (a standalone program if you wish) and then execute that program kind of independently from the remaining system (if you do not require locking which is the case for us). Therefore we decided to simply emulate a column store in C++. Similar to MonetDB we represent every column as a separate array consisting of <virtual oid,value> pairs, where the virtual oid is given implicit by the position of the value in the array. More details can be found in Appendix F. We choose TPC-H query 19 (Listing 1 in Appendix F) as that query is the only query that contains a single join followed by an aggregation without any subqueries. This query joins the `Lineitem` table with the `Part` table.

Both tables are stored as a struct of arrays and additionally, we dictionary-compress all string columns. We used float values instead of arbitrary precision numeric values. All foreign and primary key columns are represented as <Key , Payload> pairs with the row ID as the payload, this made it easier to use the join implementations with minimal modifications.

On a high-level we execute this query according to the query execution plan depicted in Figure 13. We obtained this plan through textbook query optimization. Notice that the same plan is also used by HyperDB according to the explain functionality of their web interface [7]. In this plan, the selection, that was pushed down to the scan of the `Lineitem` table, has a selectivity of 3.57%. This means that for scale-factor 100 the build relation Part has 20 M tuples and the probe relation Lineitem has 600 M tuples (600 M$\cdot$ 3.57% = 21.42 M tuples after filtering), i.e. in the join both relations have roughly the same size.

[7]http://hyper-db.de/interface.html

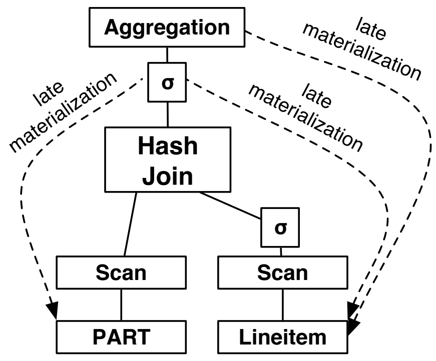


Figure 13: Optimized semi-physical query plan for TPC-H Q19 plus materialization strategy in the column store

For scale-factor=100 this corresponds to the results of our micro-benchmark of Figure 10(b) for $|R| \approx 20$ M tuples. Notice that the plan in Figure 13 is actually only *semi-physical* as it does not specify *when* to reconstruct tuples. We used late materialization, i.e. all attributes are only touched when required by an operation[8].

Figure 14 shows the runtime of Q19 for TPC-H for scale-factors 100. This figure includes a cost breakdown where the colored bars represent the time spent for the join. In contrast, the black bars represent the time spent in other parts of the query. We obtained the numbers for the colored bars by executing each join just like in the micro-benchmarks above, i.e. each of the four join algorithms receives a build input of 20 M tuples and a pre-filtered (and pre-materialized) probe input of 21.42 M tuples. The difference of the execution time of the query and the join smicro-benchmark yields the black bars[9].

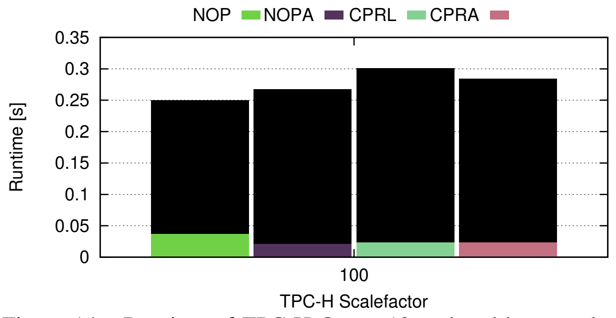


Figure 14: Runtime of TPC-H Query 19, colored bars mark the fraction of the time spent in the actual join; the black bars mark the time spent for the rest of the query.

We immediately observe that even for this relatively simple query a major part of the runtime is spent in the non-join parts of the query. The time spent in the actual join is only about 10%–15% of the total runtime! In addition, for some methods more time is spend outside the join than for others.

There are several reasons for this: First, the join key column is a primary dense key and the `Part` table is even generated in sorted order according to this key. This means that we have an ideal sequential access pattern for NOPA when building the join array (also compare our discussion in Section 5.2 as well as additional experimentation in Section C). Second, scanning and filtering 600 M tu-

[8]This strategy is also used by MonetDB.

[9]This method is not entirely fair as the join- and non-join parts of a query may overlap. However, it gives a good indication on how much of the total query time is actually due to the actual join.

ples from Lineitem down 21.42 M tuples simply eats up some time. Third, in contrast to the micro-benchmark experiments, for Q19 we have to access several attributes other than the join key. This happens in multiple places: in order to evaluate the complex predicate after the probe and to aggregate the final result, i.e. we have to perform implicit positional joins (for tuple reconstruction). In particular, In NOPA neither the `Lineitem` table nor the `Part` table have to be partitioned. Therefore, all other attributes *stay aligned with the join array and the probe relation*. This is especially beneficial for all attributes of the probe relation since they are accessed sequentially when evaluating the complex join predicate. In contrast, for the CPR*-algorithms those benefits do not apply. If we access attributes that are not the join key, we have to follow the row ids contained in the narrow join tuples. Those row-ids point to arbitrary locations after partitioning the data. This means that we pollute our cache and TLB with data from other attributes and lose locality in our accesses. Therefore, it would be beneficial to explore tuple reconstruction strategies for CPR*-joins in more detail.

9. LESSONS LEARNED

(1.) Don't use CPR* algorithms on small inputs. For input relations with less than 8 million tuples we do not observe a benefit of partitioning local chunks instead of the global relation. On the contrary, we even observe a performance degradation. This is mainly due to two things: (1) the overhead of creating all the threads does not pay off for the small input data. This is also true for all other presented algorithms. (2) the size of a chunk becomes smaller than a page. This leads to a random allocation of the pages to different NUMA nodes. Several threads will have to read from and write to remote memory, even though the main advantage of the CPR* algorithms over the PR* algorithms should be to avoid remote writes. For very small inputs the NOP* algorithms become very interesting, especially if the build relation starts to fit into the LLCs.

(2.) Clearly specify all options used in experiments. This sounds like common sense for any experimental study. Still, we list it here again as a gentle reminder since we ran into this problem when we were trying to interpret results from different papers. As the implementations provided by authors typically have multiple optimization options, it is sometimes hard to understand which optimizations were actually used in a paper. Rather over-specify than under-specify your experiments.

(3.) If in doubt, use a partition-based algorithm for large scale joins. In this paper, we have studied the performance of three variants of no-partitioning algorithm and eight variants of partition-based algorithms with different workloads by varying ratio between build and probe relations, data size, number of threads and skewness. **All** Partition-based algorithms outperform **all** no-partitioning algorithms in almost all except for only two cases. The first case is when the size of input relations scales to 32 GB where the worst partition-based algorithm is a little bit slower than the best no-partitioning algorithm. The second case is when the probe relation is highly skewed such that partition-base algorithms suffer from unbalanced loads between threads while no-partitioning algorithms have less cache misses. No-partitioning algorithms start outperforming partition-based algorithms only for a Zipf factor > 0.9.

(4.) Use huge pages. As we discussed in Section 7.2, all algorithms except PRB benefit from using huge pages. Using huge pages means less pages are needed for a certain amount of data, thus reduces the pressure on TLB system.

(5.) Use Software-write combine buffer. Software-write combine buffers are a very effective technique to reduce the number of TLB misses. Hence, using SWWCBs allows us to use single pass algorithm which accelerates partition-based algorithms.

(6.) Use the right number of partition bits for partition-based algorithms. Partition-based algorithms are very sensitive to choosing the right number of radix bits. For different data sizes, one has to choose different number of partition bits to get the optimal performance. As shown in Figure 9, choosing suboptimal number of bits can lead to performance degradation by up to a factor of 2.5.

(7.) Use a simple algorithm when possible. In our study, simple ideas turned out to be surprisingly efficient and effective. For instance, array joins are very efficient in the case of dense primary key distributions. They outperform other non-array variants under all workloads by up to 44%. Chunking is another simple idea we used to create faster join algorithms. Chunking eliminates remote memory writes in partition phase and improves the join performance by up to 26%.

(8.) Be sure to make your algorithm NUMA-aware. Back at the time when PRO and PRB [5] were published, the authors ran experiments on several single socket machines. These algorithms were designed for multicore systems but not yet for NUMA systems. Directly running these algorithms on NUMA systems will yield suboptimal performance. We evaluated chunking to eliminate writes to remote memory and explored NUMA-aware scheduling to avoid bandwidth saturation on a single memory controller. These two optimizations improve the performance over non-NUMA-aware algorithms by up to 26% and 20%, respectively.

(9.) Be aware that join runtime$\neq$query time. Our experiments with a TPC-H query clearly indicated that the join time may actually only be a 10%–15% share of the total runtime of the query. We identified multiple reasons that lead to interesting avenues for future work. However, already at this point it again emphasizes that micro-benchmarks alone may be misleading in case we want to understand performance in a bigger context, e.g. an entire query.

10. CONCLUSIONS

In this paper, we evaluated thirteen main-memory join algorithms in a common setting. We resolved some contradicting results and showed that hardware-conscious partition-based approaches typically outperform hardware-oblivious no-partition based joins on modern multi-core NUMA architectures. At least this is the case if the probe relation is not highly skewed; for very skewed data the unpartitioned hash table can match or even outperform the partition-based approaches. We also presented new partition-based approaches, called CPRL and CPRA that often outperform the PR*-algorithms from prior work. Overall CPRL and CPRA achieve a remarkable join throughput of up to 3.4 billion input tuples per second.

So should we finally consider relational joins a solved problem? The bad news is, we will probably never be able to label it 100% solved as there will always be some fancy new SIMD instruction or whatever "new" hardware that may impact the relative performance differences. The good news, with this study we believe we made major steps forward in understanding the performance of state-of-the-art join algorithms as of 2015. However, as stated above, almost all previous works, e.g. [7, 6, 5, 14, 3, 17, 4, 13], evaluated micro-benchmarks only. We departed from that and evaluated the runtime of the most promising join algorithms when used in a simple TPC-H query. This initial experimentation already reveals that only a fraction of the query runtime may be spent in the actual join, a majority may be spent in other parts of the query including scanning, filtering, and tuple reconstruction. Hence, as future work we would like to evaluate the cross product of different join algorithms and the large space of tuple reconstruction algorithms, in particular for the very promising CPR*-family of join algorithms.

11. REFERENCES

[1] https://www.systems.ethz.ch/node/334.

[2] D.J. Abadi, D.S. Myers, D.J. DeWitt, and S.R. Madden. Materialization Strategies in a Column-Oriented DBMS. *ICDE*, pages 466–475, 2007.

[3] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB*, 5(10):1064–1075, 2012.

[4] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Ozsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB*, 7(1):85–96, 2013.

[5] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Ozsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. *ICDE*, pages 362–373, 2013.

[6] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Ozsu. Main-Memory Hash Joins on Modern Processor Architectures. *TKDE*, 27(7):1754–1766, 2015.

[7] Spyros Blanas, Yinan Li, and Jignesh M Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. *SIGMOD*, pages 37–48, 2011.

[8] Google Inc. Google Sparse and Dense Hashes. https://code.google.com/p/sparsehash/.

[9] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. Quickly Generating Billion-Record Synthetic Databases. *SIGMOD*, pages 243–252, 1994.

[10] Jiong He, Shuhao Zhang, and Bingsheng He. In-cache Query Co-processing on Coupled CPU-GPU Architectures. *PVLDB*, 8(4):329–340, 2014.

[11] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *PVLDB*, 8(6):642–653, 2015.

[12] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU Join Processing Revisited. *DaMoN*, pages 55–62, 2012.

[13] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[14] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. Massively Parallel NUMA-aware Hash Joins. *IMDM*, pages 3–14, 2013.

[15] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing Main-Memory Join on Modern Hardware. *TKDE*, 14(4):709–730, 2002.

[16] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[17] R Barber G Lohman I Pandis, V Raman R Sidle, G Attaluri N Chainani S Lightstone, and D Sharpe. Memory-Efficient Hash Joins. *PVLDB*, 8(4):353–364, 2014.

[18] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. *SIGMOD*, pages 1493–1508, 2015.

[19] Stefan Richter, Victor Alvarez, and Jens Dittrich. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. In *PVLDB*, 2016.

[20] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. *SIGMOD*, pages 351–362, 2010.

[21] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning. *PVLDB*, 8(9):934–937, 2015.

APPENDIX

A. SKEWED DATA DISTRIBUTIONS

Until now we only looked at uniformly distributed data sets. In the next set of experiments we use different skew factors for the probe relation. We used an algorithm proposed by Gray et. al. in [9] to quickly generate large amounts of skewed join keys. To achieve a more realistic distribution and to avoid that the key occurring most often, i.e. the smallest keys, are all in a single partition, we map the 10 smallest keys to random keys in the full domain. Figure 15 shows the performance of all algorithms with different zipf factors θ ranging from zero to 0.99. For every method we choose the number of threads such that the throughput was the highest. This means the no-partition algorithms make use of all hyper threads while the partition based algorithms only use a single thread per core to not pollute the private caches. We can see that lower levels of skew have no real impact on the performance of the algorithms.

When the skew factor is high we observe a shift in the throughput towards methods that do not partition the input. This has two reasons. First, the partition based methods have to handle skewed partition sizes, which is for now only handled automatically by a task queue. This means that the threads responsible for larger partitions are processing less partitions. We do not exploit the possibility to use multiple threads to process the join on the largest partitions in parallel. Second, a high skew factor makes the caches more effective, as the keys that are accessed most often are likely to be cached. For the partition based algorithms this effect is not helping, as the partitioning already makes the caches effective. We can see that the partition based approaches are still competitive with the no partition joins for the presented data size.

B. SCALABILITY IN NUMBER OF THREADS

In this section we explore the scalability of the different join methods in terms of multithreading. All previous experiments were run using 32 threads. As the implementation of MWAY only works with a power of two many threads, we cannot report numbers for MWAY with more than 32 threads[10].

We take as a starting point four threads where each thread is assigned to one of the four NUMA regions. From that starting point we increase the number of threads distributing threads evenly across NUMA regions.

Figure 16 shows the results when scaling the number of threads from 4 to 120. All methods achieve good performance when using all physical cores, i.e. 60 threads. All partitioned-based approaches perform worse when using hyper-threading. This is understandable, as then even the private caches have to be shared among the hyper-threads. Even for the NOP*-joins the benefit of hyper-threading is minimal. This is also understandable for our fast hash function, as we do not have many computations that could hide the memory latency. A more computationally intensive hash function could also benefit more from hyper-threads.

Table 3(a) shows a summary of the relative speedup for the join algorithms. We calculate the relative speedup as runtime($T > 4$ threads)/runtime(4 threads) where T is the number of threads used. Hence, the perfect speedup for $T = 60$ threads would be 15. Of course no method scales to this theoretically achievable perfect speedup, but CPRA and CPRL come close with a speedup of almost 12.

C. HOLES IN THE KEY RANGE

All previous experiments used a dense key range. In this section we want to study the effect of holes in the key range on the performance of the different join algorithms, especially on the array-based methods. We generate the build relation with a domain k times the size of $|R|$ for increasing values of k. Figure 17 shows the

[10] We also evaluated MWAY using 64 threads, but the results are not competitive and also not fair, as only four out of 60 cores have to work on two threads.

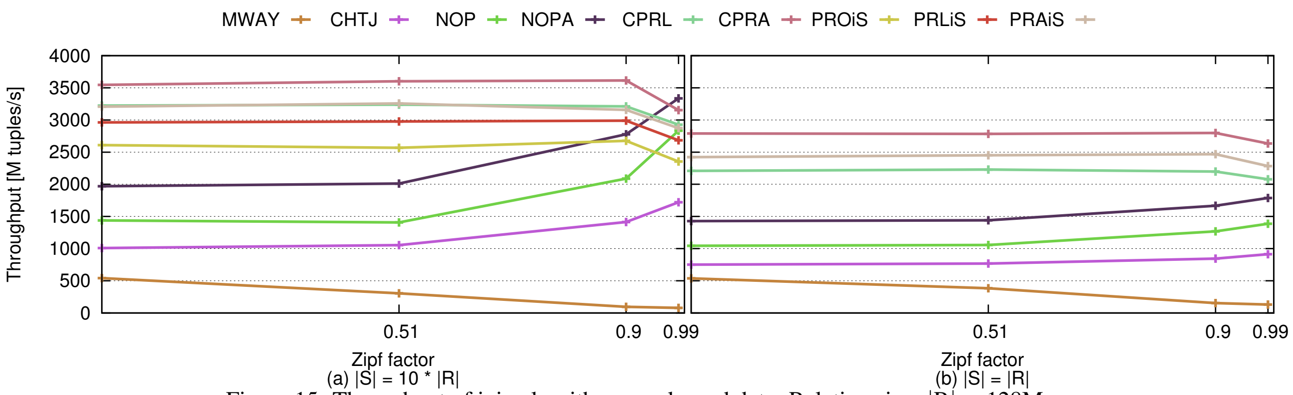


Figure 15: Throughput of join algorithms on skewed data. Relation size: $|\mathrm{R}| = 128\mathrm{M}$.

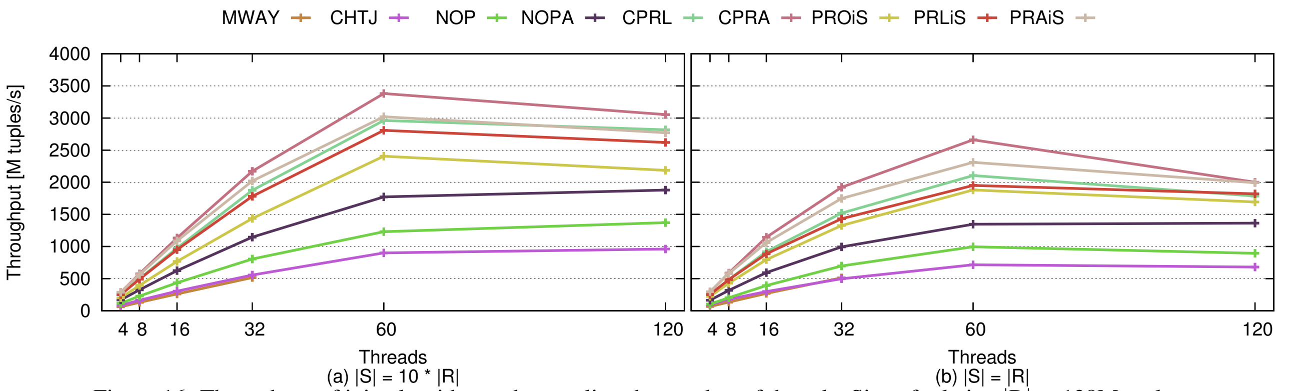


Figure 16: Throughput of join algorithms when scaling the number of threads. Size of relation $|\mathrm{R}| = 128\mathrm{M}$ tuples

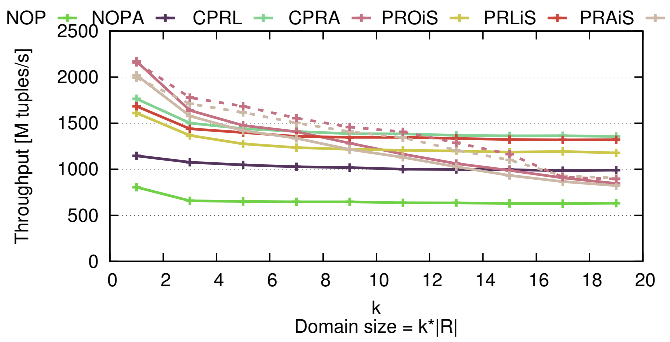


Figure 17: Performance of join algorithms with increasing domain size. $|R| = 128$ M and $|S| = 1280$ M. The dashed lines for CPRA and PRAiS denote the throughput when adapting the number of partitions to the domain size

performance of the different join algorithms for varying domain sizes $k \cdot |R|$. We can see that the performance of the NOPA join is not influenced that much. This is expected, even without any holes in the domain is it very unlikely that neighboring elements in the array are probed in a short enough sequence for the second tuple to still reside in the caches. This slight chance is simply eliminated in the case of very large domains, as neighboring elements are very unlikely to even be present at all. The size of the used array is of course growing linear with the domain size and occupies a larger and larger part of the available memory for the whole time of the join processing. The partition-based array joins on the other hand suffer greatly from large domains, as the array does no longer fit into the caches for larger and larger domains. A possible remedy for the partition-based methods is to use more fine grained partitioning in the case of larger domains, such that the array again fits into the cache. We applied this technique to PRAiS and CPRA and depicted the performance as dashed lines in Figure 17. Please note, that the main memory consumption of PRAiS and CPRA is much lower compared to NOPA as we only construct temporary arrays on small partitions, that can be freed after processing the co-partition join. On another note, all our hash table implementations suffer a small performance hit when increasing the domain, as now we can observe some collisions in the hash table when inserting and probing the join keys.

From the results it looks like NOPA stays very competitive for arbitrary large domains, as long as you are willing and able to pay the additional memory overhead. PRAiS and CPRA can also deal with domains that are reasonably dense, especially when adapting the partition strategy to the domain size.

D. MICRO-ARCHITECTURAL PERFORMANCE ASPECTS

We also measured the performance of all presented join algorithms with respect to the number of cache misses and the instructions per cycle (IPC) metric. Table 4 shows all measurements. The number of cache misses are measured in millions while the instructions retired (IR) count is given in billions. We see that the partition-based joins indeed lead to a dramatic reduction in cache misses and reach a cache hit rate of up to 99% for the join phase. Furthermore, the CHTJ suffers from roughly two times the number of cache misses compared to NOP, due to the additional bitmap lookup, as expected. We can also observe that the partition-based algorithms need more instructions to perform the join but they also have a much higher IPC rate, that allows them to perform the join faster than the no-partitioning joins. Please note, that the different amount of cache misses for PRA, PRL, PRO in the partition phase stem from the fact that we use 12, 13, or 14 radix bits respectively according to Equation (1).

			Relative Speedup		
Join	**4 Threads [M/s]**	**60 Threads [M/s]**	**Total**	**Build or Partition Phase**	**Probe or Join Phase**
CHTJ	87.3	945.7	10.8	8.4	10.9
NOP	122.1	1291.2	10.6	9.4	10.7
NOPA	176.2	1859.5	10.6	6.8	11.2
CPRL	264.3	3105.4	11.7	12.1	10.6
CPRA	300.1	3545.1	11.8	12.4	10.1
PROiS	212.0	2522.9	11.9	11.3	13.2
PRLiS	263.7	2944.0	11.2	11.3	10.8
PRAiS	302.3	3168.0	10.5	10.7	9.8

(a) With |R| = 128M and |S| = 1280M

			Relative Speedup		
Join	**4 Threads [M/s]**	**60 Threads [M/s]**	**Total**	**Build or Partition Phase**	**Probe or Join Phase**
CHTJ	96.6	751.6	7.8	8.5	7.7
NOP	109.8	1043.7	9.5	9.7	9.3
NOPA	173.0	1413.1	8.2	7.1	9.8
CPRL	260.2	2207.9	8.5	8.5	8.4
CPRA	304.9	2790.2	9.2	9.2	9.1
PROiS	234.2	1971.2	8.4	7.3	12.9
PRLiS	263.5	2046.0	7.8	7.3	10.4
PRAiS	312.3	2422.4	7.8	7.3	9.6

(b) With |R| = |S| = 128M

Table 3: Relative speedup when scaling from 4 to 60 threads.

	Sort or Build or Partition Phase						Probe or Join Phase					
Join	**L2 Misses [M]**	**L3 Misses [M]**	**L2 Hit Rate**	**L3 Hit Rate**	**IR [B]**	**IPC**	**L2 Misses [M]**	**L3 Misses [M]**	**L2 Hit Rate**	**L3 Hit Rate**	**IR [B]**	**IPC**
MWAY	430	388	0.64	0.10	260	1.36	10	10	0.01	0.04	17	1.78
CHTJ	559	353	0.20	0.37	15	0.40	1911	1561	< 0.01	0.18	29	0.25
PRB	558	555	< 0.01	0.01	65	0.33	59	40	0.98	0.33	30	1.46
NOP	394	393	0.38	< 0.01	8	0.36	957	955	0.39	< 0.01	20	0.39
NOPA	409	391	< 0.01	0.04	6	0.27	335	320	< 0.01	0.05	5	0.28
PRO	981	209	0.51	0.79	42	0.87	60	45	0.98	0.26	30	1.34
PRL	791	209	0.45	0.74	41	0.90	86	52	0.93	0.40	24	1.08
PRA	396	110	0.61	0.72	41	1.05	88	52	0.92	0.42	17	0.83
CPRL	730	193	0.47	0.74	43	1.06	72	25	0.94	0.65	29	2.26
CPRA	341	85	0.65	0.75	44	1.24	64	24	0.94	0.62	22	1.94
PROiS	976	209	0.51	0.79	40	0.83	31	13	0.99	0.59	30	2.10
PRLiS	788	209	0.45	0.73	40	0.87	55	23	0.95	0.58	24	2.28
PRAiS	398	110	0.61	0.72	40	1.00	63	31	0.94	0.50	17	1.80

Table 4: Performance counter for the join with |R| = 128M and |S| = 1280M and 32 threads.

E. VARYING THE SELECTIVITY OF THE SELECTION IN Q19

We also measure the query performance with a varying selectivity on the probe relation. Figure 18 shows that the partition based joins indeed outperform the no partition based joins when the actual probe relation in the join becomes large.

F. TPC-H QUERY USED

Listing 1 contains the full SQL code of TPC-H query 19.

Listing 1: TPC-H Query 19

```
select
        sum(l_extendedprice* (1 - l_discount)) as revenue
from
        lineitem,
        part
where
(
p_partkey = l_partkey
and p_brand = 'Brand#12'
and p_container in ('SM␣CASE', 'SM␣BOX', 'SM␣PACK', 'SM␣
     PKG')
and l_quantity >= 1 and l_quantity <= 1 + 10
and p_size between 1 and 5
and l_shipmode in ('AIR', 'AIR␣REG')
and l_shipinstruct = 'DELIVER␣IN␣PERSON'
)
or
(
 p_partkey = l_partkey
and p_brand = 'Brand#23'
and p_container in ('MED␣BAG', 'MED␣BOX', 'MED␣PKG', 'MED
     ␣PACK')
and l_quantity >= 10 and l_quantity <= 10 + 10
and p_size between 1 and 10
and l_shipmode in ('AIR', 'AIR␣REG')
and l_shipinstruct = 'DELIVER␣IN␣PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#34'
and p_container in ('LG␣CASE', 'LG␣BOX', 'LG␣PACK', 'LG␣
     PKG')
and l_quantity >= 20 and l_quantity <= 20 + 10
and p_size between 1 and 15
and l_shipmode in ('AIR', 'AIR␣REG')
and l_shipinstruct = 'DELIVER␣IN␣PERSON'
)
```

Listing 2: Data-structures to represent the Lineitem and Parts tables

```
struct LineitemTable {
    size_t numTuples;
    float *l_extendedprice;
    float *l_discount;
    tuple_t *l_partkey;
    unsigned int *l_quantity;
    uint8_t *l_shipmode;
    uint8_t *l_shipinstruct;
};
struct PartTable {
    size_t numTuples;
    tuple_t *p_partkey;
    uint8_t *p_brand;
    uint8_t *p_container;
    unsigned int *p_size;
};
```

Our simulated column store represents the TPC-H tables as structs of column pointers as depicted in Listing 2. Please note, that we only represent the columns accessed in TPC-H Q19. The type

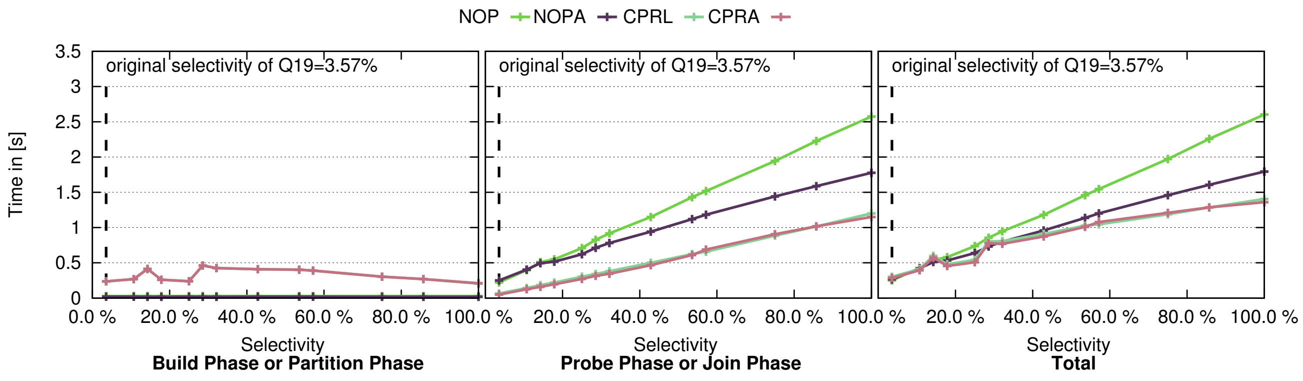


Figure 18: Runtime of TPC-H Query 19 (sf=100) when varying the selectivity of the pushed-down selection predicate

`tuple_t` is a <key,payload> pair with the rowID as the payload. We depict the filter predicate implementation in Listing 3.

Listing 3: Filter conditions

```
inline bool preJoin(LineitemTable *l, size_t rowID) {
    return (l->l_shipinstruct[rowID] == DELIVER_IN_PERSON
         &&
           (l->l_shipmode[rowID] ==AIR||l->l_shipmode[
               rowID] ==AIR_REG));
}
inline bool postJoin(LineitemTable *l, PartTable *p,
     size_t rowIDL, size_t rowIDP) {
    uint8_t p_brand = p->p_brand[rowIDP];
    uint8_t p_container = p->p_container[rowIDP];
    auto l_quantity = l->l_quantity[rowIDL];
    auto p_size = p->p_size[rowIDP];
    return (p_brand == BRAND12
           && (p_container == SM_CASE || p_container ==
               SM_BOX || p_container == SM_PACK ||
               p_container == SM_PKG)
           && l_quantity >= 1 && l_quantity <= 1 + 10
           && 1 <= p_size && p_size <= 5) ||
          (p_brand == BRAND23 &&
           (p_container == MED_BAG || p_container ==
               MED_BOX || p_container == MED_PKG ||
               p_container == MED_PACK)
           && l_quantity >= 10 && l_quantity <= 10 + 10
           && 1 <= p_size && p_size <= 10) ||
          (p_brand == BRAND34 &&
           (p_container == LG_CASE || p_container ==
               LG_BOX || p_container == LG_PACK ||
               p_container == LG_PKG)
           && l_quantity >= 20 && l_quantity <= 20 + 10
           && 1 <= p_size && p_size <= 15);
}
```

These predicates correspond one-to-one to the predicates in the SQL query depicted in Listing 1.

Listing 4 shows the pseudo code for the Q19 implementation using the NOP join. All threads build a hash table on `p_partkey` concurrently. Afterwards, every thread is responsible for a fixed chunk of tuples of the probe relation and first accesses the necessary attributes in `LineitemTable` to evaluate the preJoin predicate. All passing tuples from the `LineitemTable` are probed against the hash table and as soon as a join partner is found the postJoin predicate is evaluated. If the matched tuples pass this predicate, the `l_extendedprice` and `l_discount` attributes are immediately accessed and added to the final aggregate. With this execution strategy it is not necessary to materialize a join index for further processing. This execution strategy also corresponds to the strategy described for the HyperDB system [16].

Listing 4: Q19 Pseudo code for NOP

```
query_result_t
NOPQ19(LineitemTable *L, PartTable *P, int threadCount) {
 parallelBuild(P,threadCount);
 parallel for in chunks numTuples/threadCount
 for (int i=0;i < L-> numTuples;++i) {
   if (preJoin(L,i)) {
     auto tuple=probe(L->l_partkey);
     if (postJoin(L,P,i,tuple.rowID)) {
       res+=L->l_extendedprice[i] * (1.0 - L->l_discount
            [i]);
 }}}}
```

G. FURTHER COST-BREAKDOWN OF Q19

We designed an additional experiment for NOP just to find out how much individual components of that query contribute to the overall runtime of that query. The core idea of this experiment is to start with the "naked join", i.e. the microbenchmark, and then gradually morph the microbenchmark into TPCH-Q19. Like that we see at each step the overhead introduced by that step.

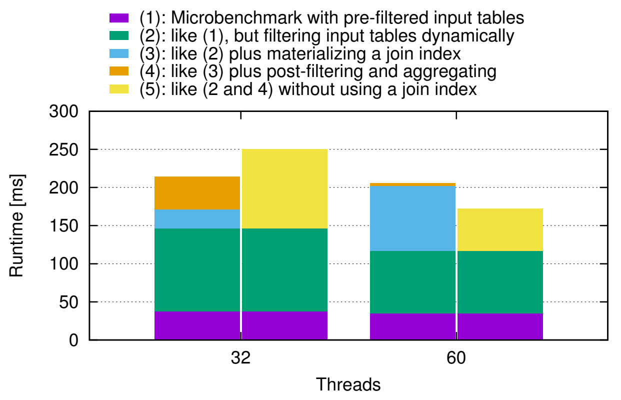


Figure 19: Additional cost-breakdown morphing a microbenchmark stepwise into Q19.

From the results in Figure 19, we can already learn many things:

(1.) Tuple reconstruction is **not** the main culprit for the overheads. In fact, for this query filtering the input rows eats up most of the additional time for both 32 and 60 threads.
(2.) Even writing out join results first into a join index and then doing all additional work like post-filtering, tuple reconstruction (in the same order as before!), and aggregating is faster than running all of this in a pipeline! But only for 32 threads! For 60 threads this does not hold anymore and the results turn upside down: here the overheads for creating and using a join index do not pay off anymore.
(3.) for 32 threads there is room for a tuple reconstruction algorithm, i.e. at most ∼20% performance improvement seem possible (for a tuple reconstruction running in zero time).