To Partition, or Not to Partition, That is the Join Question in a Real System

Maximilian Bandle
bandle@in.tum.de
Technische Universität München

Jana Giceva
jana.giceva@in.tum.de
Technische Universität München

Thomas Neumann
neumann@in.tum.de
Technische Universität München

ABSTRACT

An efficient implementation of a hash join has been a highly researched problem for decades. Recently, the radix join has been shown to have superior performance over the alternatives (e.g., the non-partitioned hash join), albeit on synthetic microbenchmarks. Therefore, it is unclear whether one can simply replace the hash join in an RDBMS or use the radix join as a performance booster for selected queries. If the latter, it is still unknown *when* one should rely on the radix join to improve performance.

In this paper, we address these questions, show how to integrate the radix join in Umbra, a code-generating DBMS, and make it competitive for selective queries by introducing a Bloom-filter based semi-join reducer. We have evaluated how well it runs when used in queries from more representative workloads like TPC-H. Surprisingly, the radix join brings a noticeable improvement in only one out of all 59 joins in TPC-H. Thus, with an extensive range of microbenchmarks, we have isolated the effects of the most important workload factors and synthesized the range of values where partitioning the data for the radix join pays off. Our analysis shows that the benefit of data partitioning quickly diminishes as soon as we deviate from the optimal parameters, and even late materialization rarely helps in real workloads. We thus, conclude that integrating the radix join within a code-generating database rarely justifies the increase in code and optimizer complexity and advise against it for processing real-world workloads.

CCS CONCEPTS

• **Information systems** → *Main memory engines*; **Join algorithms**.

KEYWORDS

Performance Evaluation; Partitioning; Join Processing; Modern Hardware; In-Memory Databases

ACM Reference Format:
Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452831

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452831

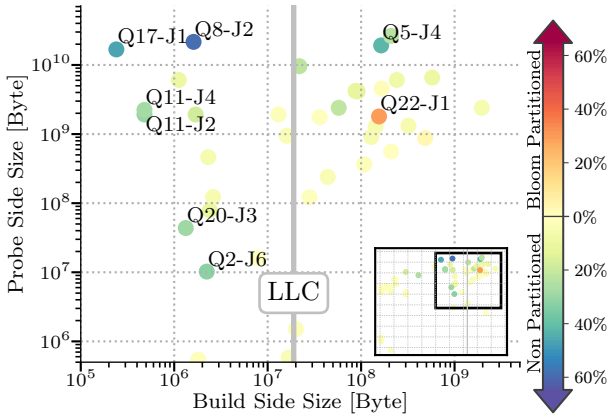


Figure 1: Relative performance of Bloom-filtered partitioned and non-partitioned hash join for every join of TPC-H SF 100 labeled as Q⟨*id*⟩-J⟨*order*⟩

1 INTRODUCTION

Architectural changes in modern processors have inspired a significant amount of research on finding the optimal join implementation. Over the years, the community has reached the conclusion that hash joins are better than sort-merge joins [3, 17], and that in general algorithm implementations should be tuned to the underlying hardware (i.e., be hardware conscious rather than oblivious) [4, 27, 32, 40].

Recent comprehensive studies have advised that the partitioned radix join performs better than the non-partitioned hash join [4, 40]. What is unclear, however, is if the radix join should completely replace the hash join as a major workhorse in the database engine, or if it should be used as a performance booster. The former is unlikely, as the radix-partitioning phase is only needed when the build side does not naturally fit into the caches; otherwise, the extra pass over the data and the necessary data materialization comes with a non-negligible overhead. The latter is a more difficult question. Using the radix-join as a booster implies that we should know *when* to use it. Unfortunately, existing research has only evaluated the performance of the two on synthetic microbenchmarks, which are not representative of what we typically get in real workloads.

In this work, we investigate *how* to best integrate the state-of-the-art radix join algorithm in a compiling main-memory DBMS and *when* to use it instead of the non-partitioned hash join. Our radix join performance is comparable to prior work's stand-alone implementations while also supporting all variants of equi-joins, including outer-, mark-, semi-, and anti-joins [33]. All query plans can use it as a drop-in replacement for the non-partitioned hash join used otherwise. Our system does data-centric query compilation [32] and applies relaxed operator fusion, which enables software-based

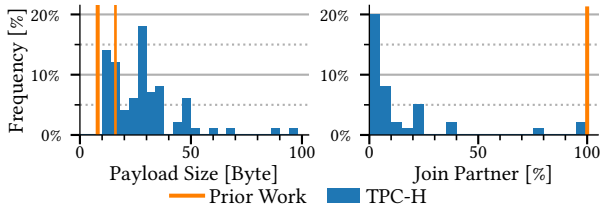


Figure 2: Tuple Size and Join Partners in TPC-H and prior work

prefetching [27]. This allows us to make a comprehensive comparison between the two join implementations in a much broader scope of workloads and factors than the analysis done by prior work. More specifically, we do the following:

Compare the hash joins in a system-wide setup: All joins under testing are integrated and tested within a compiling in-memory DBMS. Both the partitioned and the non-partitioned join are state of the art and hardware-conscious, using optimizations such as software-prefetching, software-write combining, and non-temporal stores [5].

Evaluate the holistic impact of the join on query execution: Existing work compares the joins in isolation and simplifies the settings by relying on materialized input data or omitting result materialization by merely counting the matching tuples [4, 16, 40]. Unfortunately, these simplifications cannot always be applied as joins appear in many stages of query execution. By integrating the joins within a full DBMS, we also investigate their *implicit* effects on the entire execution of a query.

Use representative datasets: In contrast to prior work that only considered narrow tuples (8–16 bytes) [47] and dense data-distributions [16, 40], we use the TPC-H benchmark for our performance evaluation (c.f., Figure 1). As shown in Figure 2, the TPC-H queries operate on a more extensive range of selectivities and tuple sizes.

To our surprise, despite the encouraging microbenchmark results from prior work, the optimized radix join [4, 40] was not competitive in the TPC-H benchmark. When analyzing the joins, we noticed that for most queries the majority of the shuffled tuples in the partitioning phase might not even be present in the final result (cf. Figure 2). Therefore, we introduced a filter for the probe phase that drops tuples as early as possible to save computation time and reduce unnecessary materialization overhead.

While this optimization makes the radix join more competitive, it provides measurable benefits in merely one of 59 equi-joins contained in the TPC-H workload (cf. Figure 1). By further investigation through a series of microbenchmarks we discovered that the benefits of the radix join diminish quickly when one of the workload's characteristics (e.g., payload size, data distribution, materialization strategy, join selectivity, etc.) deviates from the optimum. In fact, we can barely achieve any benefit for non-optimal cases.

This paper makes the following key contributions:

- We fully integrate the radix join into a main-memory DBMS. To the best of our knowledge, this is the first implementation of a radix join in a DBMS, using data-centric code generation [15].
- We embed a Bloom-filtered semi-join reducer that significantly reduces materialization overhead for queries with medium and high selectivity.
- We compare our radix join implementation and the Bloom-filtered version *within* Umbra [13] against a state-of-the-art hash join [18, 21, 27] using the TPC-H benchmark.
- With extensive microbenchmarks, we synthesize the range of values for the workload characteristics needed to observe *any* performance benefits when using the radix join.

Following on the insights from our extensive evaluation, we *express serious reservations to implementing the radix join.* Its usage as a booster is limited to a small set of workloads and thus rarely justifies the increase in code- and optimizer-complexity.

2 RELATED WORK

The majority of papers agree that in-memory hash joins are faster than sort-merge joins [3, 18]. There is further agreement that hardware-conscious joins are superior [5, 27]. However, it is not clear whether partitioning or prefetching for non-partitioning joins, makes the best use of the hardware resources. Balkesen et al., and Schuh et al. claim that radix partitioned joins are superior [3, 5, 40], while Lang et al. state the opposite [18].

Much attention has been given to parallel implementations of in-memory radix joins by our community in the last two decades. Here we give a brief overview. In 1999, Boncz et al. [9, 24, 25] proposed multi-pass radix-partitioning to overcome the TLB thrashing problem of the original hardware-conscious join by Shatdal et al. [42] (cf. Section 3.1) and investigated optimized materialization stragies [26]. Kim et al. [17] and Blanas et al. [7] have evaluated the radix join on multicore systems. Balkesen et al. [3–5] revisited partitioned and non-partitioned joins and optimized the implementation of Blanas [7] with write-combining and streaming instructions [3, 39, 46]. Fang et al. and Makreshanski et al. [12, 23] built theoretical models for the two hash joins and identified the tuple size as the most critical performance factor saturating the memory bandwidth.

Schuh et al. [40] introduced NUMA-awareness to radix joins, provided an extensive comparison against other NUMA optimized joins [2, 18], and motivated the use of the radix join as a booster. Among their other contributions, the authors also evaluated the radix joins in a stand-alone TPC-H Query 19 variation, where the size of the relations was significantly reduced by partitioning the reference to the original tuple and cutting all strings to one byte. Due to the lack of complete system integration, their analysis is based on the isolated join time without considering the cost of tuple reconstruction, which biases the conclusions [30].

Further, we note that there is a lot of related research linked to join processing and radix partitioning: Polychroniou et al. [36] have investigated SIMD partitioning and provide an overview of partition variants [37], which is revisited for radix-partitioning by Schuhknecht et al. [41] and Zhang et al. [47]. Richter et al. [38] compare different hash table implementations, while Barber et al. [6] focus on memory-efficient hash joins. Pirk et al. [34] analyze hash joins in depth and Shrinivas et al. [43] and Abadi et al. [1] compare materialization strategies in column-store database systems.

Partitioning also applies to non-CPU centered data processing. For example, GPU- [35] or FPGA-accelerated [14] approaches have similar goals and use comparable algorithms to distribute the workload better.

3 PARTITIONED RADIX JOINS

Existing in-memory hash join algorithms can be divided into two camps [40]. On the one hand, we have the non-partitioning variants using a global hash table, which is accessed in parallel. They rely on software-based prefetching to avoid expensive cache misses and random memory accesses when the hash table does not fit in the caches [5, 27]. On the other hand, the radix joins directly reduce cache misses by joining the data partition-wise, where each partition is sized so that the hash table fits in the cache [42]. In this chapter, we assume that both probe and build side reside in already materialized form to be comparable with prior work [4, 40].

3.1 Basic Partitioned Join

On a high level, a partitioned join splits both input relations into partitions that are then joined individually.

A basic partitioned join implementation consists of two phases: First, in the partitioning phase, both the build and the probe side are partitioned by using a hashed value of the join condition as key. As a result, both sides are now split into partitions containing their respective join partners. In the second phase, the join is executed per partition. A union of all partitions' results yields the final outcome.

The partitioning algorithm operates in three steps [47]: The first step scans the input and builds a histogram, counting how many elements the partition will consist of. The second step uses the histogram to calculate the total number of tuples and the exact partition boundaries. We allocate an output buffer large enough to fit all tuples and assign each partition a region based on the partition boundaries. Finally, in the third step we scan the data again and materialize each tuple to the correct position in the output buffer. Each partition keeps track of the number of written tuples to determine the correct output position.

3.2 Parallel Radix Join by Balkesen et al.

Balkesen et al. [4] proposed an efficient, publically available[1] implementation of a radix join. Their join is a refined version of the one by Blanas et al. [7]. Figure 3a depicts an overview of their approach.

Two-pass Partitioning: Boncz et al. [9] observed that a single-split partitioned join has a performance problem. It occurs when writing to more partitions in-parallel than the translation lookaside buffer (TLB) has entries, which trashes the TLB. Boncz et al. mitigate the problem by applying multi-pass partitioning, called radix-partitioning, which performs multiple splits subsequently. This limits the number of partitions created in each pass so that it does not exceed the number of TLB entries. Each partitioning pass uses a different subset of bits from the hashed key. Balkesen et al. use two partitioning passes, as shown in Figure 3a.

Parallel Partitioning: Running the basic implementation in parallel is challenging because each worker writes to all partitions, leading to high congestion. Kim et al. [17] propose to split the input relation so that each slice can be processed in parallel. All equally sized slices are stored in a task queue. From there, each worker picks a task and performs the steps listed under Subsection 3.1. Following the histogram creation of all tasks, the prefix sums are computed combining all histograms ①. Based on the prefix sums, each task calculates a dedicated output location and scatters the tuples into partitions without any synchronization ②. The second pass takes the partitions from pass one and splits them again ③.

The final join is done in parallel, using task-based parallelism that also helps with skew.

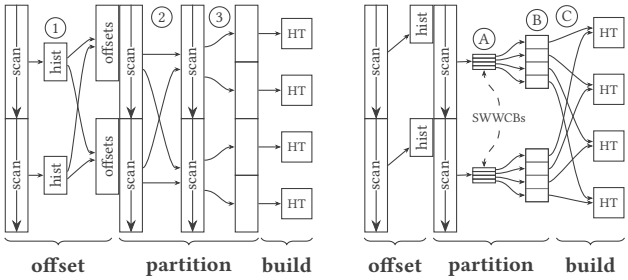


Figure 3: Radix-Partitioning in prior work

3.3 Optimized Radix Join:

Balkesen et al. further refined their radix join with software write-combine buffers (SWWCBs) and streaming instructions [3]. We compare our implementations against their optimized radix join in Section 5.2. Schuh et al. [40] further optimized the radix join by adding NUMA-awareness (cf. Figure 3b).

Software Write-Combine Buffers: Wassenberg et al. [46] propose SWWCBs to speed up radix sorting, which is also beneficial for radix-partitioning [3]. SWWCBs are software-managed data buffers residing in the cache, combining multiple writes. Each buffer is at least one cache line in size and stores the partitioned tuples instead of writing them to their destination directly Ⓐ. The buffer is flushed to its destination when it is full, which effectively reduces the pressure on the TLB, and the number of memory writes.

Non-temporal Streaming: Non-temporal streaming instructions mitigate the potential doubled number of writes introduced due to SWWCBs by writing full SWWCBs directly to DRAM. The write bypasses all caches and avoids their pollution Ⓑ. However, the data now needs to be aligned at cache line boundaries. This makes combined use of both optimizations sensible. The maximum width of the SIMD registers limits the size a single instruction can write at maximum. In 2016, this was half a cache line, or respectively 256B using AVX2. With AVX512 instructions, modern Intel processors can store a full cache line at once.

NUMA-awareness: Currently, the radix join is not NUMA-aware because each task writes to multiple partitions, which are located all over the output buffer. Thus, each worker potentially has to access different memory regions to store its tuples. Schuh et al. [40] keep the writes local by adding an output chunk per task, which stores the tuples in local partitions. However, now the final partitioning result is not located in one contiguous memory region but in one chunk per task. Hence, the join may have to read from different NUMA nodes Ⓒ. Their experimental evaluation shows that the advantages prevail, since only reads may be on different NUMA-nodes. Furthermore, NUMA access is much more balanced, and overall performance increases.

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

4 JOINS IN MAIN-MEMORY DBMS

Prior to this paper, all work on partitioned joins was evaluated with the join in isolation using microbenchmarks. To take the next step from a stand-alone radix join to a real database system, we integrated radix-partitioned joins into Umbra [13], whose performance is comparable to HyPer or MonetDB [11].

Umbra uses data-centric code-generation [32], relaxed operator fusion [27], arbitrary query unnesting [31], morsel-driven parallelism [21], and accepts the queries using a SQL frontend. We first describe how data-centric code generation works in general, and then how it works for both of our hash join implementations. Following these explanations, we focus on our novel radix-join implementation, which partitions two input dataflows.

4.1 Data-Centric Code-Generation

The main difference between a stand-alone join implementation and one integrated into a full-featured RDBMS system is the environment. In the former, the whole system focuses on the join. In the latter, the join is a part of operator pipelines that organize the dataflow, as shown in Figure 4. First, a pipeline's source operator loads the tuple from a materialized state into the CPU. Then, the tuple traverses the operators of the pipeline and it is finally materialized in the next pipeline breaker [29].

Umbra compiles each pipeline, in particular the dataflow from one source operator to the materialization point, in a bottom-up manner using the produce/consume model [32]. Each operator has to call *produce* on its inputs to delegate the responsibility for starting the pipeline. Eventually, the pipeline starter is reached, which cannot delegate further. It begins pushing tuples to its consumer up the pipeline. Once a pipeline breaker is reached, it generates code to materialize all incoming tuples. We use this abstraction to compile data-centric code for arbitrary SQL queries.

4.2 Materialization Strategy

Umbra stores relations column-wise in main memory [13]. We use early materialization to reduce random access during pipeline evaluation. Thus, the table scan only reads necessary columns, filters them using SIMD instructions, and stitches them together in tuples passed to the consumer. To avoid materialization overhead, we use sideways information passing [43]. The build side of our hash join, e.g., tells the probe pipeline the required tuples to filter them out early.

To compare effects of the chosen materialization strategy, we integrated Late Materialization. We traverse the query tree from top to find the earliest access to each column. If that does not happen immediately after a table scan, we introduce a late-load operator that retrieves columns based on their tuple id when needed.

4.3 Non-Partitioned Hash Join

The non-partitioned hash join does not have to write out the probe side, as shown in Figure 4. Each hash join passes the tuples on and performs the join within the pipeline [21]. This so-called operator fusion keeps the tuples in registers for as long as possible. Sadly, it might also hinder inter-tuple parallelism since the code structure is more involved. Relaxed Operator Fusion (ROF) counteracts this problem by loosening the original idea of data-centric code-generation in favor of intermediate materialization. It allows the DBMS to introduce staging points in the query plan, buffering the probe side in cache and trading pipelined tuples with cache-locality [27]. Reading from these buffers enables vectorization optimizations, e.g., branch-free primitives, and software-based prefetching to avoid cache misses. ROF effectively combines the advantages of data-centric code generation with vectorization.

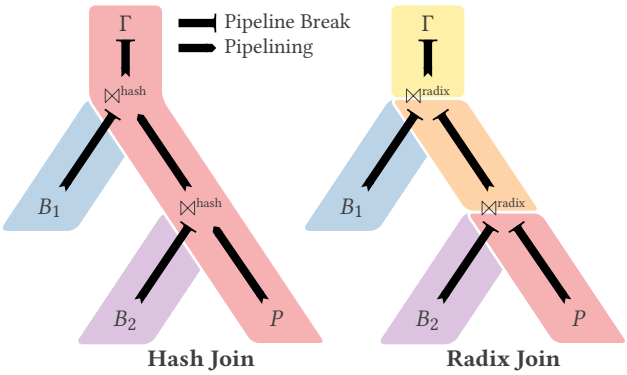


Figure 4: Pipelining in radix and hash joins. Hash joins can pass the probe tuples through multiple joins while radix joins have to materialize both inputs every time.

4.4 Partitioned Hash Join

In contrast, writing to memory is not optional for the radix join because it builds upon the radix-partitioning phase. These frequent writes loosen the original idea of data-centric code-generation, and they also counteract it. So when multiple radix joins are executed after one another, each join has to break the pipeline.[2]

Algorithm 1: Full Pipeline Breaker

```
Function Radixjoin::produce(requiredColumns):
    condition ← analyzeJoin(requiredColumns);
    build ← prepareBuild();
    left.produce(condition.left.requiredColumns);
    build.partition();
    probe ← prepareProbe();
    right.produce(condition.right.requiredColumns);
    probe.partition();
    joinTuples(build, probe);
```

Thus, the radix join is both a full pipeline breaker and a pipeline starter, as shown in Figure 4. Algorithm 1 follows along the three phases described in Section 3.1. The code first partitions the build side and then the probe side, which breaks both pipelines because

[2]When two subsequent joins use the same partition key, we could combine them in a pipeline to avoid the pipeline break with the resulting partitioning overhead.

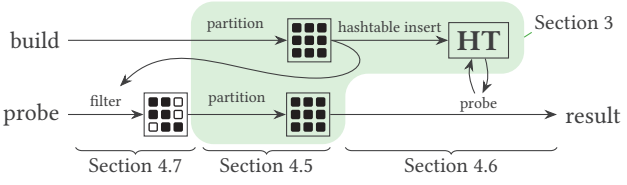


Figure 5: Schematic Overview of our Partitioned Join.

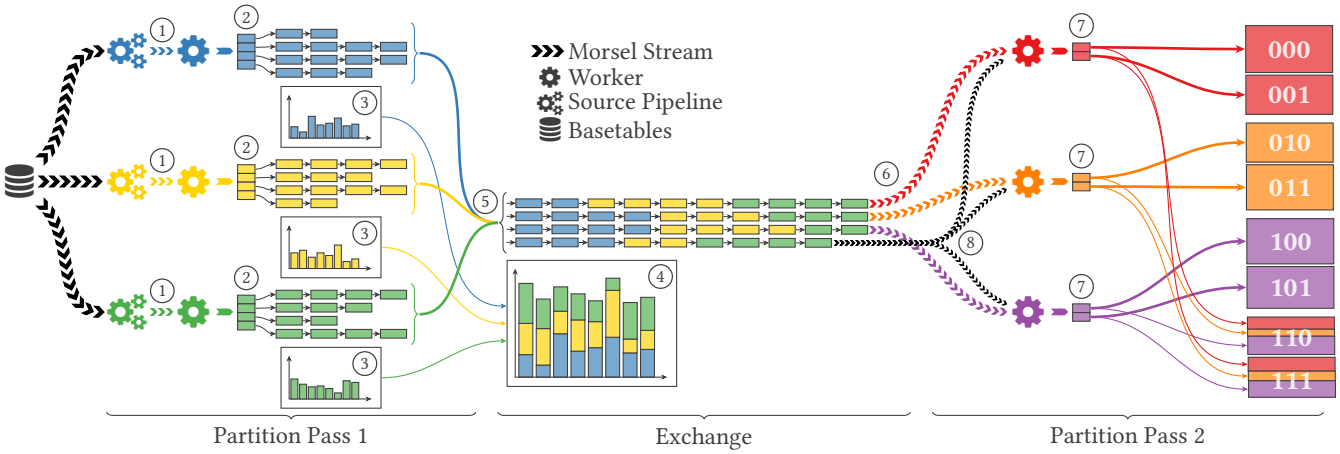


Figure 6: Schematic overview of our two-pass partitioning performing an eight-way split using 3 bits.

all data is now materialized. After both sides are partitioned, the new pipeline is started, which joins the tuples.

The join code of Algorithm 2 mainly consists of tight loops, which is characteristic for the produce/consume model [29]. These tight loops are advantageous for modern CPUs because they maximize data locality by keeping the data in CPU registers as long as possible. The algorithm has to loop over the partitions, build the hashtable, and check whether each tuple is contained. All matching tuples are passed to the consumer in the pipeline.

Algorithm 2: Starting a New Pipeline

```
Function Radixjoin::joinTuples(build, probe):
    for p_build, p_probe ← {build, probe} do
        hashtable = buildHashtable(p_build);
        for t_probe ∈ p_probe do
            for t_build ∈ hashtable.probe(t_probe) do
                consumer.consume(t_build ∘ t_probe)
```

Because the majority of the work is done during or after materialization, tuple collection is simple. Depending on the current input pipeline, the tuple has to be partitioned either on the build or on the probe side.

4.5 Morsel-Driven Partitioning

The pipeline execution is based on morsels, which divide the total workload into smaller blocks, enabling work-stealing [21]. Every source operator has to emit the data into the pipeline morsel-wise. Figure 6 shows a detailed overview of the tuple flow inside our partition step, which is used for both build and probe side.

The first pass consumes all morsels of the current source pipeline by picking them from the morsel stream once they finish their previous work ①. This technique allows fine-grained load balancing, even with skewed data. The worker determines the output partition based on the least significant bits of the hash value, which is then paired with the tuple. This is first materialized in the worker's own worker-local set of SWWCBs ②. As soon as a buffer is full, we use non-temporal streaming instructions to move the tuples to their temporary partition without polluting the caches.

One challenge lies in working with dataflow inputs. This means that we need to materialize the input first without relying on histograms, which is also the reason for using two passes. Hence, each temporary partition is implemented as a linked list of pages. Whenever a page is full, a larger page is prepended and used instead.

Afterward, each worker traverses the linked list and builds a local histogram for the next partition pass ③. Currently, there is no need for communication between the workers.

In the exchange phase, we do two things: First, ④ we compute the exact size of the output partitions based on the prefix sums of the worker-local histograms. Second, ⑤ all workers' linked lists are combined by concatenating the lists in so-called pre-partitions.

Hence, the database system does not need synchronization between the work packages in the second partitioning pass as each has its dedicated range.

We perform the second partitioning pass morsel-wise as well. The radix join generates its morsels based on the pre-partitions ⑥. We use the same worker to process the entire linked list of one pre-partition. Once again, we use SWWCBs to combine the writes and then scatter the tuple buffer to its final position ⑦. Further, we implement work-stealing to achieve proper load balancing among the workers, even under the presence of skew ⑧.

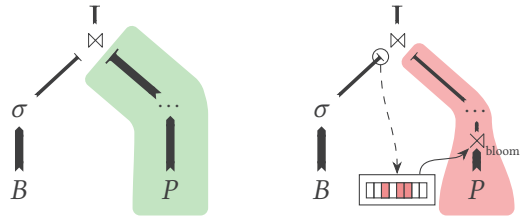
During the whole partition process, all workers are writing to either local or dedicated memory areas. Hence, there is no need for synchronization or writing to non-worker-local memory regions, which ensures scalability with different numbers of worker threads and on systems with multiple sockets.

4.6 Final Join Phase

Each morsel builds the hash table on the fly using robin-hood hashing, which provides the most robust performance for thread-local workloads [38]. Since moving tuples is expensive, we only store pointers. We avoid costly resizing of the hash table because we know its size in advance. In addition to that, we reuse the hash table's memory segment to avoid costly memory allocation. Thus, we only have to reallocate memory in case the partition size has significant skew.

4.7 Bloom Filters

We are now at the point where the join operates on cache-resident partitions, with the cost of partitioning dominating the execution time of the radix join [4, 40]. Materializing the probe side partitions



Radix Join Bloom Radix Join

Figure 7: Bloom filters with selective joins. Tuples without a join partner are filtered early and are not materialized.

can often become unnecessarily expensive in selective queries. One optimization is to reduce the number of stored tuples for the probe side. This is possible because most queries apply selections on the build side before joining the data [10].

Fuzzy semi-join reducers are an established technique for non-partitioned hash joins [19]. They improve the performance of selective joins, as already implemented in our non-partitioned join using tagged pointers [21]. The optimizer pushes the reducers down in the pipeline to prune tuples early (cf. Figure 7).

We introduce a Bloom-filter based reducer in our radix join to minimize the cost of materialization. The second pass over the build side generates the filter while partitioning. The filter is probed in the pipeline before partitioning the probe side and is also pushed down when possible.

Following the guidelines by Lang et al. [20], we implemented register blocked Bloom filters. These filters partition the Bloom filter into register-sized blocks. We have to access exactly one block for each probe, which reduces the number of cache misses to at most one per check. Consequently, the writes to the Bloom filter can be done in parallel without synchronizing as two partitions cannot share blocks. The Bloom-filtered radix join performs around 40% faster for 5% foreign key join partners (cf. Section 5.4.1).

5 EVALUATION

In the following, we present an experimental evaluation of our radix join against our non-partitioned hash join within Umbra, a full-fledged RDBMS. We answer when and whether partitioning pays off.

5.1 Experimental Setup

5.1.1 Joins under test. We have compared the following three joins inside Umbra [13]:

Radix-Partitioned Join (RJ): Our radix join implementation with SWWCBs, non-temporal streaming, two-pass partitioning, and thread-local output buffers. It implements all optimizations presented in Section 3.

Bloom Radix-Partitioned Join (BRJ): Our Bloom-filtered radix join implementation. It reduces materialization overhead by filtering the probe side (cf. Section 4.7).

Buffered Non-Partitioned Hash Join (BHJ): Our non-partitioned join implementation, using a global chaining hashtable with relaxed operator fusion [27]. It features a semi-join reducer based on tagged pointers [21].

We have validated our joins against state-of-the-art prior work:

Table 1: Workloads from Prior Work

workload		size [B]	tuple count		size [MiB]	
	used in	key/pay	build	probe	build	probe
A	[4, 7]	8/8	$16 \cdot 2^{20}$	$256 \cdot 2^{20}$	256	4096
B	[3, 4, 17]	4/4	$128 \cdot 10^{6}$	$128 \cdot 10^{6}$	977	977

Joins from Balkesen et al. (PRJ & NPJ): We have evaluated the aforementioned joins against the partitioned (PRJ) and non-partitioned join (NPJ) of Balkesen et al. [4], which they provide as stand-alone implementations. To allow for a fair comparison, we enabled all optimizations like SWWCBs and non-temporal storing for the PRJ and software-based prefetching for the NPJ.

5.1.2 Workloads. The major part of the evaluation was performed on the TPC-H benchmark [44], which we analyzed on a query and individual join level. It features 22 queries with different workload characteristics (c.f. Figure 2).

To compare against related work and refine the TPC-H analysis by isolating certain workload factors, we used microbenchmarks. As a base for these, we reused the workloads of Balkesen et al. [4], whose properties are listed in Table 1. We alter the workload for each microbenchmark to isolate particular workload factors that are of interest, e.g., different selectivities or payload sizes.

In our system, we reproduced the setup by generating the build and probe tables using the following SQL statement. We did not preprocess the data and particularly did not generate indexes.

```
CREATE TABLE b(key BIGINT NOT NULL, pay BIGINT NOT NULL);
```

For workload B, we used `INT` instead of `BIGINT` to generate 4 B sized columns.

5.1.3 Hardware. Unless otherwise noted, we used an Intel i9-9900X (Skylake-X) CPU with 10 cores and 64 GB RAM. By default, we used all available threads, including hyper-threads. Other experiments were conducted on a dual-socket Intel E5-2660v2 (Sandy Bridge) with 10 cores and 256 GB of RAM, and on an AMD 3950X (Ryzen 9) with 16 cores and 64 GB of RAM. Detailed specifications can be found in Table 2. We compiled the code with GCC 9 using the `march=native` flag to enable the AVX512 instruction set, if possible. The RDBMS uses LLVM and clang 9 to compile the queries itself.

To have a sound comparison, we did not include query compilation time[3] because the other implementations were hand-coded and pre-compiled. Before taking any measurements, we warmed up the system and ensured that all data is in memory. We ran all benchmarks at least five times and reported median performance.

[3] Query compilation takes negligible time, even for optimized settings.

Table 2: Hardware Platforms

	Skylake-X	Ryzen 9	Sandy Bridge
vendor	Intel	AMD	Intel
model	i9-9900x	3950X	E5-2660v2
sockets	1	1	2
cores (SMT)	10 (x2)	16 (x2)	20 (x2)
clock rate [GHz]	3.5-4.4	3.5-4.7	2.2-3.0
L1 data cache [KiB]	32	32	16
L2 cache [KiB]	1024	512	256
LLC cache [MiB]	19	16 (x4)	25
DRAM speed [GiB/s]	79.4	47.8	59.9

5.1.4 Key Questions. We separate the evaluation into three parts:

First, we ran experiments to ensure that our join implementations are competitive to related work (Section 5.2), to check how well they scale with the number of threads (Section 5.2.1) and in a NUMA system (Section 5.2.2), and to see how efficiently they use the memory subsystem (Section 5.2.3).

Second, we ran the TPC-H workload to check whether the radix join can completely replace the non-partitioned hash join in our database engine (Section 5.3.1). Since our hypothesis assumes no, we evaluate whether the radix join could be used as a performance booster by analyzing the TPC-H workload on a join-level (Section 5.3.2).

Finally, with an extensive series of microbenchmarks (Section 5.4) we searched for ideal range values of workload properties (e.g., selectivity, payload size, pipeline depth, etc.) that emphasize performance advantages of the radix join over the non-partitioned hash join.

5.2 Performance characterization and comparison to related work

We have aimed to evaluate benefits and drawbacks of partitioning within a DBMS objectively. At the same time, this evaluation is only insightful if our implementation offers reasonable performance.

We used PRJ and NPJ by Balkesen et al. [4, 5] with all optimizations enabled to compare its performance against our join implementations. To match the workloads used in the original paper, we have used the following query to join build and probe table and count the resulting tuples.

```
SELECT count(*) FROM probe r, build s WHERE r.k = s.k;
```

One key difference is that Balkesen et al. directly use the key for partitioning, while we create an equally sized hash value and store it with each tuple. This is compensated as we do not store the payload, which is not required for the tuple count.

5.2.1 Scalability. In this experiment, we first compared the performance of our implementations for that of the state of the art. The results are shown in Figure 8, which indicates that both the RJ and the BHJ are competitive to PRJ and NPJ. On the one hand, our RJ outperforms the PRJ for workload A, while on the other hand our BHJ is not as fast as the optimized NPJ on both workloads.

Another observation is that all implementations scale well with the number of hardware contexts, although radix joins experience bigger speed-up than non-partitioned joins. For 10 threads, our RJ implementation speeds up by a factor of 7.5 to 9.5 for workloads A and B, respectively. For workload A, the RJ does not fully scale to 10 threads because the system already reaches the memory bandwidth limit (as we will show in Section 5.2.3). For workload B, the hyper-threads give us about 10% additional performance, since the smaller tuples do not entirely saturate the memory bandwidth. As expected, both non-partitioned hash join implementations benefit more from hyperthreading because it hides their memory access latencies. The NPJ implementation, unlike the BHJ, is optimized for the given workload and performs better. For instance, the NPJ knows the exact hash table size and distribution beforehand.

5.2.2 NUMA effects. In this experiment, we evaluated how well algorithm implementations utilized available hardware resources by scaling the number of cores from one to the maximum number of logical threads available.

We used the other two machines, the dual-socket Intel Sandy Bridge and the AMD Ryzen 9, whose chip has four chiplets (cf. Table 2) to show the performance with NUMA.

The results in Figure 9 show that RJ scales well on the Sandy Bridge machine. Its performance increases by a factor of 10 to 16, depending on the workload. The smaller tuples put less pressure on the memory bandwidth, resulting in better scalability. As before, hyper-threads marginally sped up the performance.

On the Ryzen 9, however, we observed a different pattern and the RJs no longer exhibited the linear scalability beyond a certain point. The comparably small memory bandwidth is the key factor as the bandwidth per core is 60% of the Skylake-X's. Thus, the RJ scaled well initially, but reached the memory bandwidth limit much faster for workload A. As we increased the number of threads further, the RJ slowed down because of memory bandwidth contention. As before, the BHJ performed similarly on all machines and workloads and scaled more independently of the workload.

5.2.3 Memory bandwidth usage. As identified by the two previous experiments, the performance of RJ is significantly affected by its pressure on the memory subsystem. Both when increasing the payload size and when scaling the number of hardware contexts, the performance benefits diminish as we approach the bandwidth limits. Thus, in this experiment we analyzed the memory bandwidth usage

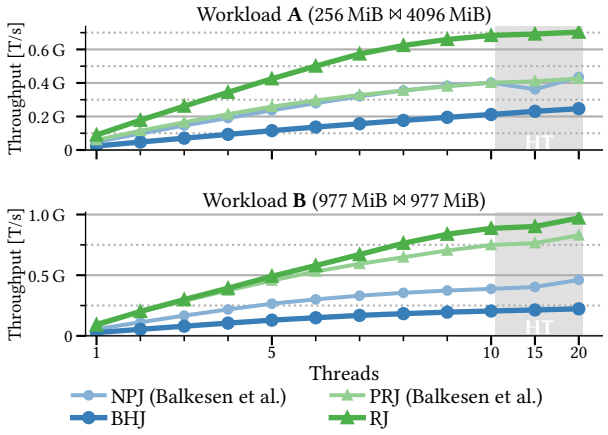


Figure 8: Scalability and comparison to Balkesen et al.

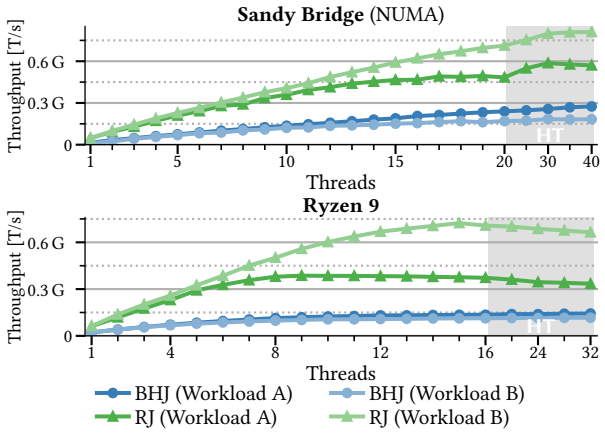


Figure 9: Scalability on different machines

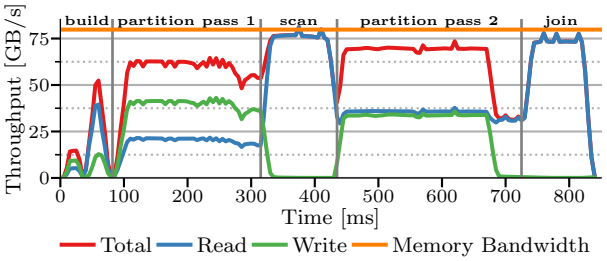


Figure 10: Memory Bandwidth for 24B wide tuples

(read and write) for the individual stages of the RJ as measured using the PCM Tools.[4]

Figure 10 shows the read, write, and total memory bandwidth while performing the RJ for the SQL query stated in Section 5.4.2. The x-axis shows the time spent to highlight how expensive each phase of the join is. The build pipeline takes a fraction of the execution time, given that it is 30 times smaller in size than the probe side. The probe pipeline dominates the execution time, mainly due to the materialization phase during the two partitioning passes. We deliberately chose this query since it demonstrates the effects introduced by padding. It is required for the use of SWWCBs and non-temporal streaming instructions, which outweigh the negative effect of padding. We notice that both partitioning steps and the join are bandwidth-bound, which confirms the futility of adding more hardware contexts, and why increasing the payload size hurts the performance.

The prior three experiments verified the competitiveness of our implementation. It can fully utilize the memory bandwidth and is bound by it, leaving minor room for improvement.

5.3 TPC-H Evaluation

The TPC-H benchmark offers a variety of queries that put pressure on different parts of the RDBMS at varying scaling factors (SFs): i.e., string comparisons, large base table scans, or joins with different selectivities [10]. To address whether the RDBMS should use a radix join as the sole workhorse, we have compared the performance of our join implementations by replacing all joins in the query tree with the join under testing for different scaling factors.

Figure 11 shows the results of our experiments for relevant TPC-H queries as we vary the dataset size (i.e., scaling factor). We used processed tuples per second as a metric with the number of tuples being the sum of all tuples counted at the pipeline sources.[5] *Queries 1, 6*, and *13* were not included in our measurements since they do not use joins.[6]

We make the following key observations. First, the BHJ delivers the best overall performance, especially apparent for SFs under 30. Second, BRJ is faster than RJ for all queries because foreign keys mainly use filtered build sides (cf. Figure 2, [10]). Third, the BRJ outperforms the BHJ only in Query 22 for SF 30 and 100. Fourth, Late Materialization appears to be orthogonal to the question of whether to partition or not. Therefore, if one needs to choose to implement only one hash join in their system, the BHJ is the apparent implementation choice.

[4]https://github.com/opcm/pcm

[5]For example, the number of tuples in "`SELECT count(*) FROM a, b WHERE a.key = b.key;`" is *tablescan* + *tablescan* + *groupby scan* = size(a) + size(b) + 1.

[6]Our system uses a groupjoin for Query 13, which combines join and group by [28].

This conclusion confirms our hypothesis that replacing all joins is not desired because the radix join is most promising for selected workloads [4, 5, 40]. We continue our analysis in more detail for individual query plans to explain why BRJ and RJ cannot always replace the BHJ as the primary join.

5.3.1 End-to-End Query Performance. In this section, we analyze the selected TPC-H queries based on their query plan.[7] Since the queries in TPC-H have different characteristics, we have split them into several groups and discuss the performance difference between BHJ, BRJ, and RJ based on the join sizes in SF 100.

Small Build Size (Q2, Q11): These queries contain only joins with a small build side, which fits in the caches. This is advantageous for the BHJ because there are no cache misses. *Query 2* contains nine different joins, whose build sides, even for SF 100, are smaller than the LLC. The 2 GB probe side causes materialization overhead, which is more significant for the RJ than for the BRJ.

In *Query 11*, the largest build side is 480 KB, so the global hash table fits in the L2 cache, making the partitioning phase redundant. The BRJ performs better than the RJ in both queries because it can avoid most partition overhead by pre-filtering the tuples.

Single Join Queries (Q4, Q12, Q14, Q19): For these queries, the number of pipelines is overseeable, and the join mostly dominates the query runtime. *Query 4* contains one join of `orders` and `lineitem` that clearly dominates the query. The Bloom filter pays off since the join's build is pre-filtered. It can discard around 80% of unjoined tuples, for a predicate with 3% selectivity, and thus reduces the partitioning overhead. Even though its build side does not fit in the LLC for SFs larger than 10, the BHJ's performance remains constant, thanks to the buffers introduced by relaxed operator fusion. *Query 12* spends most of its time scanning the `lineitem` relation using it as the build side for a join with the `orders` relation. Once again, the bottom-most selection discards the majority (99.5%) of the tuples, but the resulting build side is 87 MB for SF 100, which is four times the LLC size. As before, the prefetching keeps the BHJ's performance stable, and the RJ cannot keep up with the BRJ. *Query 14* joins 1% of the tuples from `lineitem` with `part` which are 209 MB and 560 MB in size, respectively. As both sides are roughly equal in size, both BRJ and RJ perform well for a high enough SF. *Query 19* divides its runtime between filtering and joining the `lineitem` relation. The build side is only 2 MB in size, and fits in the LLC. The BHJ cannot significantly outperform the BRJ because the Bloom filter drops 90% of tuples before the partitioning phase.

Otherwise dominated Queries (Q3, Q10, Q15, Q16, Q17, Q18): In these queries, joins account for less than 40% of the total runtime, which limits the effect that the join implementation has on the overall performance. *Queries 15, 16, 17*, and *18* are dominated by grouping of tuples. *Query 10* is dominated by scanning and selecting the base table, while *Query 3* is dominated by a group join. As a result, the differences in the join performance are minor for large SFs, as other operators dominate the query runtime. For small scale factors, however, the BHJ is superior.

Complex Queries (Q5, Q7, Q8, Q9, Q21, Q22): These queries contain various joins with different build and probe side sizes. We

[7]All generated query plans are similar to the ones reported by the Umbra Webinterface umbra-db.com/interface .

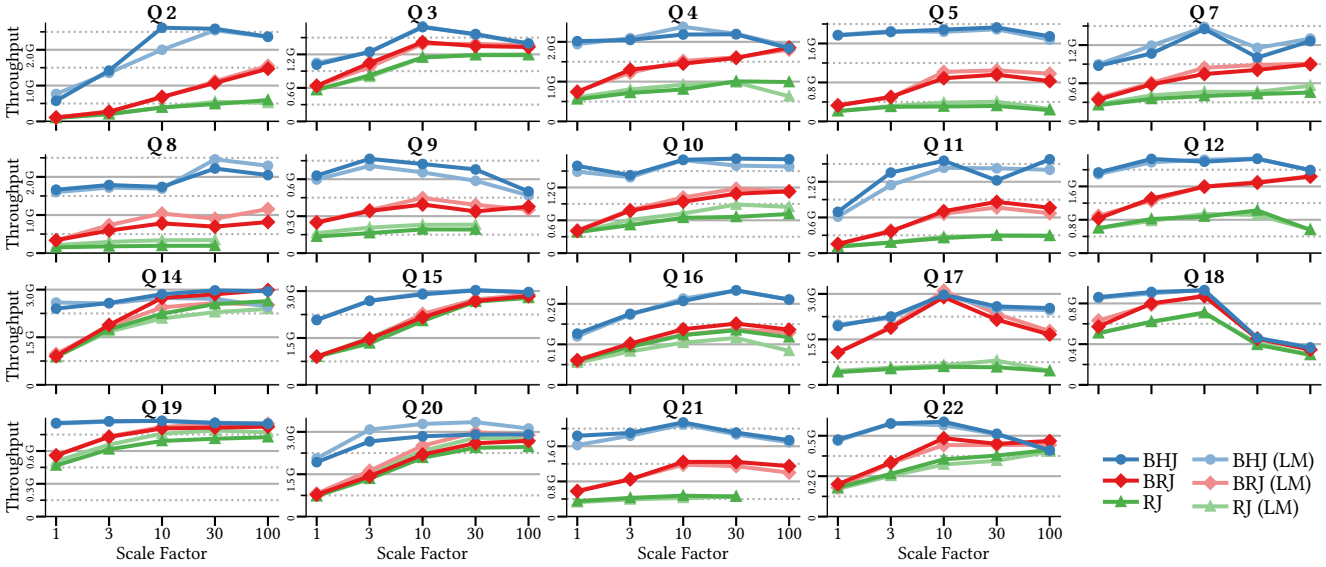


Figure 11: Throughput of all TPC-H queries containing joins with every join replaced by the one under testing[8]

cannot explain the effect of the join performance solely based on the query plan and the total execution time. In the following sections we check if there might be a case to use the BRJ as a performance booster for each join.

Materialization Strategies: Late Materialization (LM) only helps when we substantially reduce the tuple width at selective joins. For example, in *Query 8*, LM reduces the build side in four out of seven joins. Or *Query 20*, where the result consists of two text columns, which are only present in the output. Materializing them late pays off, reducing the probe side size by two-thirds. When using LM in *Query 14*, however, we only reduce the build size by 8 B. The random access for all build side tuples outweighs the positive effect.

5.3.2 Individual Join Comparison. The analysis in the previous section shows that most TPC-H queries perform multiple joins. Using just one join implementation for the whole query can lead to suboptimal performance. However, analyzing the impact for each join in a query plan in our system is challenging because all joins are part of pipelines (cf. Section 4.1), where all operators of a pipeline are fused to pass the tuples in registers and efficiently organize the code in tight loops.

Thus, we have examined all possible permutations of the join plan to compare BRJ and BHJ with TPC-H SF 100. To evaluate each join in the query plan (e.g., the 2nd join), we computed the pairwise difference in performance when all other joins were fixed with one implementation, and we only varied the hash join algorithm used for that join. We show results for selected queries in Figure 12, where the x-axis denotes the join number within the query plan, and give an overview for all the joins in TPC-H in Figure 1, where we break down the measurements in build and probe side sizes.

One key observation is that most joins are not relevant for the total execution time. However, for some of the *expensive* joins, choosing the optimal implementation makes a big difference. For example, the execution time can be up to 60% slower or up to 30% faster when selecting the BRJ instead of the BHJ. Therefore, we focus the rest of our analysis on queries with multiple joins, where the join implementation choice has the most significant impact.

In *Query 5*, a single join dominates the runtime difference between the BRJ and BHJ. This join uses the unfiltered `lineitem` relation as the probe side and has a much smaller build side. Even though the build side does not fit in the LLC, the size difference between build and probe side is 1:117 and too big for the BRJ to pay off (cf. Figure 12). *Query 8* also uses the unfiltered `lineitem` as the probe side, which is 20 GB in size in the differentiating join. The build side is a 1 MB filtered relation. As a result, the hash table fits in the cache, and the BHJ is 60% faster in total execution time.

In *Queries 7 and 9*, the topmost two joins dominate the runtime difference. Each has a large build and probe side. RJ and BRJ still cannot outperform the BRJ, because the build tuple sizes are over 48 B, making partitioning too expensive to pay off. Prefetching in the BHJ also reduces cache misses more effectively.

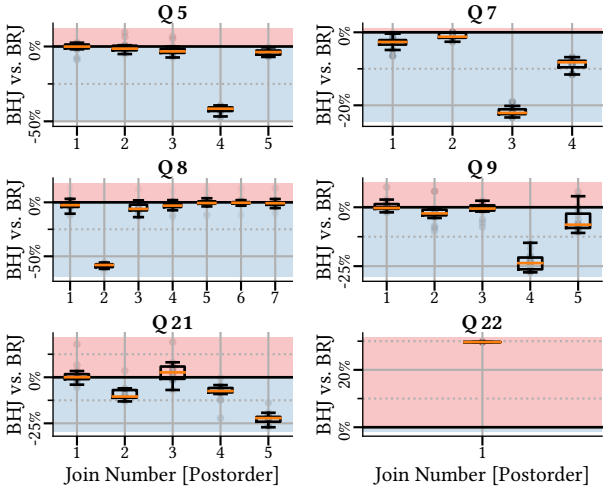


Figure 12: Relative impact per join for selected TPC-H queries (without Late Materialization)

[8] Due to the materialization overhead, the RJ cannot finish processing Q8, Q9, and Q21 for SF100 within our memory budget.

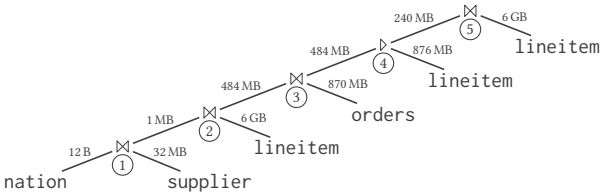


Figure 13: Q21 Join Tree annotated with build and probe size

Query 21 is dominated solely by joins, and each join has different characteristics, as shown in Figure 13. The query has a left-deep join tree, which prevents long pipelines.

① is negligible because of its size. For ②, the build side fits in the LLC. The Bloom filter can reduce the materialization overhead, so the BHJ is only 10% faster. ③ has narrow tuples and comparable sizes, so BRJ and BHJ perform equally. In ④ and ⑤, multiple factors lead to a suboptimal performance. The build side tuples are 33 B in size and the difference between build and probe size is not optimal. While Figure 12 shows that ③ is on average faster with BRJ, using BHJ for all leads to the overall fastest runtime.

Query 22 consists of two joins. One is a non-equi join, which cannot be handled by the hash join, so we do not enlist it in Figure 12. The anti-join reads the `customer` relation which is 155 MB in size as its build side and the unfiltered `orders` relation which is 1.8 GB as its probe side to evaluate a *not exists* predicate. Thus, each probe tuple is only 12 byte in size, including the hash value. Since small tuples work well for the BRJ, using the BRJ for this join improves the total query performance by 30% over the BHJ.

5.4 Isolating the effects of different factors

The analysis done so far has focused on the TPC-H benchmark, where join performance is concurrently affected by different factors. The combination of these factors leads to a completely different view on the RJ than in prior work (c.f. Section 5.2, [40]). In order to pin down the individual effects of each factor, we ran an extensive series of microbenchmarks. Combining all, we could isolate the cases where BRJ and RJ are superior to non-partitioned joins.

5.4.1 Effect of foreign key selectivity. One common pattern in all queries is that the BRJ outperformed the RJ due to selective foreign key joins (cf. Figure 2). In this experiment, we analyzed how varying selectivity affects each join's performance.

Our workload was based on workload A by Balkesen et al. [4], on which the radix join generally performs well (cf. Section 5.2). The build side remained unchanged for all selectivities. We modified the foreign key selectivity in the probe side while preserving its size to ensure that the number of processed tuples remained constant.

The results of the experiment are shown in Figure 14. We observe that both the BRJ and the BHJ are significantly affected by the varying selectivity. The BRJ is up to 50% faster than the RJ for low selectivities. However, when more than 50% of the foreign keys find a join partner, the RJ overtakes the BRJ because the computation time required to perform the filter lookup does not pay off — as it introduces up to one cache miss per lookup. We overcome this problem by sampling the probe side tuple while probing the Bloom filter. This allows us to switch off the filter adaptively in case almost all tuples pass the filter, which introduces a minor overhead, mostly below 10%. We note, however, that TPC-H and real-world queries

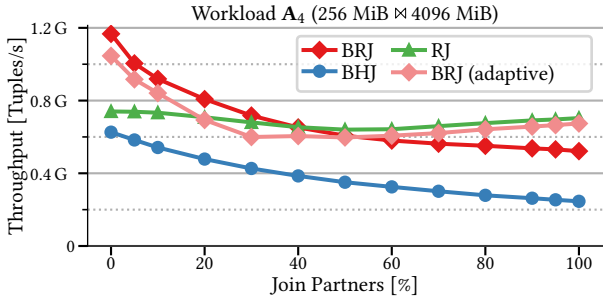


Figure 14: Impact of pre-filtering the probe side using a Bloom-filter based early probe

usually have selectivities below 25% (cf. Figure 2, [10]). This experiment shows why the BRJ performs better than the RJ in TPC-H. We further note that the RJ is 10 to 40% faster than the BHJ for low selectivities, when all other parameters are near-optimal.

5.4.2 Effect of payload size. The payload size also influences join performance. Some joins have small payloads, but that is not always the case since the columns, e.g., may contain strings (cf. Figure 2). To isolate the effects that the payload size has on the performance of the RJ and BHJ, we set the foreign key selectivity to 100%.

Once again, we based our workload on the unskewed workload A by Balkesen et al. [4], where the radix join generally performs well (cf. Section 5.4.5). The build side remained unchanged. We modified the probe tuple size by adding multiple 8 B wide columns with randomized integers. We used up to 8 payload columns, leading to a maximum payload size of 64 B. Together with the join key and its hash value, our tuples were at most 80 B wide.

Our queries are similar to the following with one payload:

```
SELECT sum(s.p1) FROM build r, probe s WHERE r.k = s.k;
```

This query materializes 32 B per tuple: 8 B for the payload, 8 B for the key, 8 B for its hash value, and 8 B padding. We show the results in Figure 15 and notice that the performance of the RJ is more affected by the payload size than the BHJ. The RJ performance degrades by a factor of 7, while the BHJ remains constant for five times larger tuples. Also, the use of SWWCBs is visible as the tuple sizes are padded to the next power of two. We do not use buffers for tuples larger than 64 B because padding would lead to higher performance losses than the benefits of non-temporal streaming.

LM lowers the performance, since the selectivity is at 100% and we have to additionally store the tuple id, leading to 24 B wide tuples. The RJ performs strictly worse due to cache misses introduced by random access after the join phase which could be addressed by radix decluster [26]. The BHJ is not affected by LM because there are no intermediate results.

The performance of the BHJ is memory bound (i.e., affected primarily by the latency of random memory accesses). Hence the

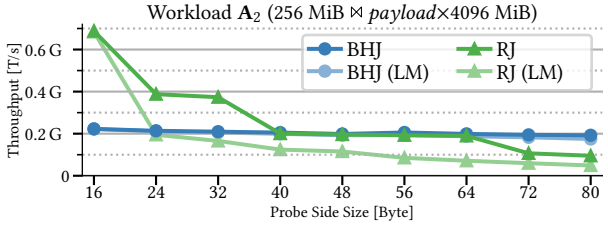


Figure 15: Impact of payload size on join performance

tuple size does not affect its performance significantly. The RJ, however, is bandwidth bound. The materialization costs heavily influence its performance in the partitioning phase, which is directly dependent on the payload size (cf. Section 5.2.3). The RJ is up to three times faster than the BHJ for small tuples, but it completely loses the advantage once the tuple size exceeds 32 B.

5.4.3 Combined effect of payload size and selectivity. Both our previous benchmarks cannot individually show the benefits of LM. However, if we vary and analyze selectivity and payload size, we can see its benefits. We modified the workload with 5 % selectivity from Section 5.4.1 by adding columns to the probe side, like in Section 5.4.2. We used four 8 B columns which total 40 B including the hash value. Using LM, we only had to materialize 24 B before and could fetch the remaining 24 B after the join.

Table 3: Throughput [T/s] w and w/o Late Materialization

	LM	no LM	benefit
BHJ	452 M	453 M	±0 %
BRJ	656 M	487 M	+35%
RJ	341 M	153 M	+122%

Analyzing the results from Table 3, LM doubles RJ's performance because it halves the necessary materialization. The thereby introduced random access has no negative consequences since only 5% of the tuples require it. Yet, it is still slower than the BRJ without LM, which follows the idea of sideways information passing [43] to prune most rows even before partitioning. However, LM gives the BRJ a significant boost by reducing the materialization, making it almost 50% faster than the BHJ. The BHJ does not materialize the intermediate result, so there is no benefit.

5.4.4 Effect of pipelining. When lining up multiple joins in a pipeline, the effects of both factors (selectivity and payload size) amplify each other. This is particularly bad for chaining RJs. Each RJ in the pipeline requires materialization and adds its column to the payload size, effectively enlarging the tuple size as the pipeline depth increases. This workload is a typical case for queries operating on a star schema where the central table connects various fact tables for additional information.

To evaluate the effects of the pipeline depth, we used the same workload as before, but instead of summing up the payloads, we used them as keys for fact tables, which resulted in a star-schema benchmark. Thus, we added multiple copies of our build side table containing randomly permutated rows. So we still achieved 100% selectivity and could investigate the pipelining effect isolated. The optimizer had to use the central table every time because its keys connect the fact tables, finally resulting in a query plan with a single long pipeline (cf. BHJ in Figure 4).

We show the results in Figure 16, where we observe the throughput for each join in the pipeline. In the ideal case, the throughput is constant, which means that pipeline depth and join execution time

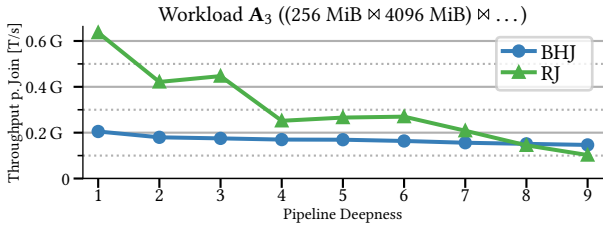


Figure 16: Impact of pipeline depth

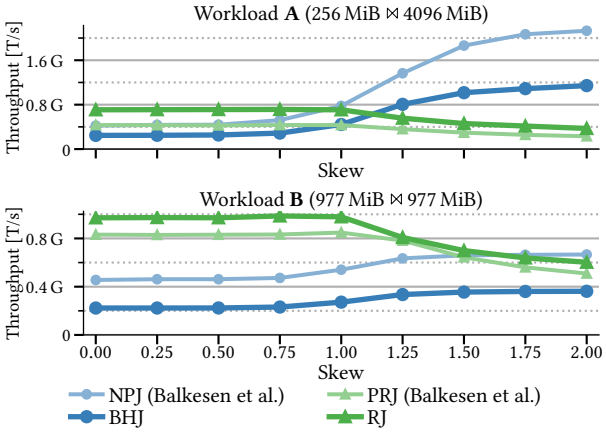


Figure 17: Impact of different Zipf factors

do not correlate. This is indeed almost the case with the BHJ.

The performance of the RJ, however, decreases proportionally to the length of the pipeline. Materialization overhead and memory bandwidth limitations add up, ultimately slowing down the join.

5.4.5 Effect of skew. To evaluate the effect of skew, we populated the foreign column in the probe relation with Zipf distributed data and varied the Zipf factor between a uniform distribution and $z = 2$, which resembles high skew. The same set-up was used by Balkesen et al. to evaluate the implementation of their PRJ (cf. Section 3.2) and their NPJ. The results of the experiment are shown in Figure 17.

We note that both the NPJ and BHJ benefit from an increase in skew as the workload exhibits better temporal cache locality and incurs less random memory accesses during the probe phase. Blanas et al. [7] already reported similar observations. For both radix joins, however, the skew has adverse effects. The partitioning of skewed data leads to heterogeneous partition sizes, which complicates the partition scheduling. This is especially visible when $z > 1$, meaning more than 50% of the tuples find their join partner in the first 20% of the build relation.

For workload **A**, BHJ outperforms RJ once the skew is higher than $z = 1$, and is more than five times faster for $z = 2$. For workload **B**, the intersection happens later for the NPJ and not at all for the BHJ since both relations are equally sized and have narrower tuples, both of which are more favorable to the radix joins. Comparing PRJ and our RJ, both show similar runtime characteristics. Our implementation is up to 50% faster because it parallelizes better, as we have already seen in Section 5.2.1. The BHJ profits from increased skew because it improves the cache locality. In contrast, the RJ loses performance for $z \geq 1$ since it throws partition sizes and scheduling out of balance.

5.4.6 Effect of build size. Prior work extensively studied this effect [3, 4, 7, 40]. As long as the build side fits into the LLC, the global hashtable does not suffer from cache misses, rendering partitioning useless. For larger hashtable sizes, prefetching reduces the cache misses for BHJ, while partitioning shows its strength for the BRJ.

We observed this behavior in the TPC-H measurements (c.f. Figure 11), where the BRJ only began to pay off in larger SFs. The in-depth join analysis presented in Figure 1 also shows that the LLC size is crucial: having a build side smaller than the LLC means there is no need for partitioning.

5.4.7 Effect of size difference. The difference in size between the build and the probe side has also been analyzed in prior work as we can see from the chosen datasets A and B, with size differences of 1:1 and 1:22. Schuh et al. also used a maximum difference of 1:10 [4, 40]. The reason is that a limited size difference ensures that the cost of materializing the partitions is in the same order of magnitude for both the build and the probe side.

We already observed the negative effect of a too-large size difference in the TPC-H measurements (c.f. Figure 1). When build and probe side are in the same order of magnitude, the RJ performs well and might outperform the BHJ (depending on the values of the other factors). The BRJ can operate on a broader range of workloads since pre-filtering decreases the materialization overhead. For example, the size difference in Query 22 is 1:11 and the BRJ leads to a speed-up of 30%. In contrast, for Join 4 in Query 5 the size difference is 1:100 and the BHJ is 40% faster.

6 DISCUSSION AND CONCLUSION

In this paper, we have addressed one of the most important join questions of the last decade: *When does radix partitioning pay off?* To do that, we integrated a state-of-the-art radix partitioned hash join into a main-memory DBMS and compared it against an optimized non-partitioned hash join implementation. Given the results from prior work, our expectation was to use it to boost some expensive analytical queries (e.g., from the TPC-H workload).

Surprisingly, the benefits of the optimized radix join (with NUMA-awareness, SWWCBs, and non-temporal streaming instructions) are barely noticeable for any join in TPC-H. After an in-depth inspection, we identified that partitioning (and materializing) tuples — which are not present in the join result — dominates its runtime, especially for selective joins. We tried Late Materialization to reduce tuple width, which sped up the RJ in some microbenchmarks but did not make a big difference in TPC-H. Lastly, we addressed this issue by implementing a Bloom filter in the probe side (BRJ). While this slightly slows down the join in the microbenchmarks, it is significantly faster for the TPC-H queries, as shown in Figure 18.

However, even with that optimization, the non-partitioned hash join (BHJ) achieved comparable speed and a more stable performance than the BRJ for all queries. In fact, the BRJ is faster than the BHJ for SF 100 only for one join in TPC-H, and even then only by 30%. This shows a severe discrepancy with the insights obtained by prior work when the analysis was done only on microbenchmarks.

The second major contribution of our work comprises an extensive analysis of the performance of each individual TPC-H join (c.f. Figure 1) and isolating the effects of different workload factors with a series of microbenchmarks. The end goal was to synthesize the range of values for the key workload properties when using the radix join (and partitioning the data) actually brings benefits. Our findings are summarized in Table 4.

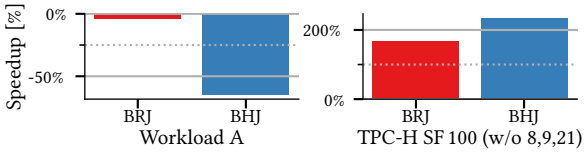


Figure 18: Speedup of different join implementations over the optimized radix join

Table 4: Workload Characteristics for Partitioned Joins

Factors	Workable	Beneficial
Selectivity	handled by Bloom filter	
Payload Size[9]	$\leq$ 32B	$\leq$ 16B
Pipeline Depth	$<$ 8 Joins	$<$ 2 Joins
Skew (Zipf)	≤ 1	≤ 0.5
Build Size	$> LLC$	$\gg LLC$
Size Difference	$<\times 50$	$<\times 10$

One key observation is that the RJ is very sensitive to any deviation from the near-optimal workload characteristics. While the BRJ delivers competitive performance for a large range of queries (c.f. Figure 1), it seldom can reveal its full potential and bring performance improvements over the non-partitioned alternative. Theoretically, we can expect up to a 300% improvement by choosing the radix join. In reality, for some cases we even observe a performance drop because the required workload conditions are not met, e.g., the payload is not narrow enough. This makes it difficult for the optimizer to reliably predict the expected improvement from choosing the radix join over the hash join.

Putting the previously researched datasets and TPC-H into perspective, it becomes clear that past research took place on a relatively narrow range of data. We extended the applicability of the RJ to varying payload sizes and selectivities. While this makes it easier for practitioners to use it, it is still difficult to judge if the insights obtained from that evaluation are also applicable to their workloads. Although, TPC-H is synthetic, it still provides a broader range of queries and data properties (Table 5).[10] Actual real-world data is even less suitable for the radix joins with its non-negligible data skew and high emphasis on string processing (and wider payloads).

We have shown that integrating the optimized radix join in an RDBMS is a non-trivial process and requires additional modifications to make it competitive for selective queries. Even then, choosing *when* to use it to gain a performance advantage requires many parameters to be satisfied and be accurately known by the optimizer at runtime. So unless the radix join is beneficial for other reasons, e.g., larger than main-memory working sets, we express reservations that implementing the radix join in a general-purpose production system justifies the added complexity.

[9] Late Materialization can handle large payloads when they occur with selectivity.
[10] TPC-DS did lead to similar insights. In the Join Order Benchmark [22], the RJ performed worse because it is string-processing heavy.
[11] JCC-H [8] provides a more realistic drop-in replacement for TPC-H with skew. It puts even more pressure on the radix join.

Table 5: Workloads for Join Processing

Factors	Prior Work	TPC-H	Real World [45]
Skew (Zipf)	$0-2$	none[11]	yes
Payload Size	$8-16$ B	≈ 32 B	large (strings)
Pipeline Depth	1 Join	$1-5$ Joins	various
Selectivity	100%	low selectivity	low selectivity
Size Difference	$1-25$	mostly high	mostly high
Build Size	$\gg LLC$	mostly small	mostly small

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).

REFERENCES

[1] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, Istanbul, Turkey*, pages 466–475. IEEE Computer Society, 2007.

[2] M. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[4] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 362–373, 2013.

[5] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on modern processor architectures. *IEEE Trans. Knowl. Data Eng.*, 27(7):1754–1766, 2015.

[6] R. Barber, G. M. Lohman, I. Pandis, V. Raman, R. Sidle, G. K. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *Proc. VLDB Endow.*, 8(4):353–364, 2014.

[7] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 37–48, 2011.

[8] P. A. Boncz, A. G. Anadiotis, and S. Kläbe. JCC-H: adding join crossing correlations with skew to TPC-H. In *Performance Evaluation and Benchmarking for the Analytics Era - 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers*, pages 103–119, 2017.

[9] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65, 1999.

[10] P. A. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*, pages 61–76, 2013.

[11] M. Dreseler, M. Boissier, T. Rabl, and M. Uflacker. Quantifying TPC-H choke points and their optimizations. *Proc. VLDB Endow.*, 13(8):1206–1220, 2020.

[12] J. Fang, J. Lee, P. Hofstee, and J. Hidders. Analyzing in-memory hash joins: Granularity matters. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2017, Munich, Germany, September 1, 2017.*, pages 18–25, 2017.

[13] M. Freitag and T. Neumann. Umbra: A disk-based system with in-memory performance. In *CIDR*, 2020.

[14] K. Kara, J. Giceva, and G. Alonso. Fpga-based data partitioning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 433–445, 2017.

[15] T. Kersten, V. Leis, and T. Neumann. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. *VLDB J.*, 30, 2021.

[16] O. Khattab, M. Hammoud, and O. Shekfeh. Polyhj: A polymorphic main-memory hash join paradigm for multi-core machines. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, pages 1323–1332, 2018.

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

[18] H. Lang, V. Leis, M. Albutiu, T. Neumann, and A. Kemper. Massively parallel numa-aware hash joins. In *In Memory Data Management and Analysis - First and Second International Workshops, IMDM 2013, Riva del Garda, Italy, August 26, 2013, IMDM 2014, Hongzhou, China, September 1, 2014, Revised Selected Papers*, pages 3–14, 2013.

[19] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 311–326, 2016.

[20] H. Lang, T. Neumann, A. Kemper, and P. A. Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high-throughput. *PVLDB*, 12(5):502–515, 2019.

[21] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754, 2014.

[22] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[23] D. Makreshanski, G. Giannikis, G. Alonso, and D. Kossmann. Many-query join: efficient shared execution of relational joins on modern hardware. *VLDB J.*, 27(5):669–692, 2018.

[24] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a join? dissecting CPU and memory optimization effects. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 339–350, 2000.

[25] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[26] S. Manegold, P. A. Boncz, and N. Nes. Cache-conscious radix-decluster projections. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 684–695, 2004.

[27] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, 2017.

[28] G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *PVLDB*, 4(11):843–851, 2011.

[29] T. Neumann. Engineering high-performance database engines. *PVLDB*, 7(13):1734–1741, 2014.

[30] T. Neumann. Comparing join implementations. http://databasearchitects.blogspot.com/2016/04/comparing-join-implementations.html, Apr 2016.

[31] T. Neumann and A. Kemper. Unnesting arbitrary queries. In *BTW, Germany*, pages 383–402, 2015.

[32] T. Neumann and V. Leis. Compiling database queries into machine code. *IEEE Data Eng. Bull.*, 37(1):3–11, 2014.

[33] T. Neumann, V. Leis, and A. Kemper. The complete story of joins (in hyper). In *BTW, Germany*, pages 31–50, 2017.

[34] H. Pirk, O. R. Moll, M. Zaharia, and S. Madden. Voodoo - A vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.*, 9(14):1707–1718, 2016.

[35] C. Pohl, K.-U. Sattler, and G. Graefe. Joins on high-bandwidth memory: a new level in the memory hierarchy. *The VLDB Journal*, 07 2019.

[36] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1493–1508, 2015.

[37] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 755–766, 2014.

[38] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB*, 9(3):96–107, 2015.

[39] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious SIMD sort. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 351–362, 2010.

[40] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1961–1976, 2016.

[41] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich. On the surprising difficulty of simple things: the case of radix partitioning. *PVLDB*, 8(9):934–937, 2015.

[42] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 510–521, 1994.

[43] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. Materialization strategies in the vertica analytic database: Lessons learned. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1196–1207. IEEE Computer Society, 2013.

[44] Transaction Processing Performance Council (TPC). *TPC BENCHMARK^{TM} H (Decision Support) – Standard Specification Revision 2.18.0.* 1993-2018.

[45] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then. Get real: How benchmarks fail to represent the real world. In *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 1:1–1:6, 2018.

[46] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*, pages 160–169, 2011.

[47] Z. Zhang, H. Deshmukh, and J. M. Patel. Data partitioning for in-memory systems: Myths, challenges, and opportunities. In *CIDR*, 2019.