# Extending the MPSM Join

Martina-Cezara Albutiu, Alfons Kemper, Thomas Neumann

Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany
firstname.lastname@in.tum.de

**Abstract:**
Hardware vendors are improving their (database) servers in two main aspects: (1) increasing main memory capacities of several TB per server, mostly with non-uniform memory access (NUMA) among sockets, and (2) massively parallel multi-core processing. While there has been research on the parallelization of database operations, still many algorithmic and control techniques in current database technology were devised for disk-based systems where I/O dominated the performance. Furthermore, NUMA has only recently caught the community's attention. In [AKN12], we analyzed the challenges that modern hardware poses to database algorithms on a 32-core machine with 1 TB of main memory (four NUMA partitions) and derived three rather simple rules for NUMA-affine scalable multi-core parallelization. Based on our findings, we developed MPSM, a suite of massively parallel sort-merge join algorithms, and showed its competitive performance on large main memory databases with billions of objects. In this paper, we go one step further and investigate the effectiveness of MPSM for non-inner join variants and complex query plans. We show that for non-inner join variants, MPSM incurs no extra overhead. Further, we point out ways of exploiting the roughly sorted output of MPSM in subsequent joins. In our evaluation, we compare these ideas to the basic execution of sequential MPSM joins and find that the original MPSM performs very well in complex query plans.

## 1 Introduction

Hardware vendors are improving their (database) servers in two main aspects: (1) increasing main memory capacities of several TB per server, mostly with non-uniform memory access (NUMA) among sockets, and (2) massively parallel multi-core processing. These emerging hardware characteristics will shape database system technology in the near future. New database software has to be carefully targeted against the upcoming hardware developments. This is particularly true for main memory database systems that try to exploit the two main trends – increasing RAM capacity and core numbers. So far, main memory database systems were either designed for transaction processing applications, e.g., VoltDB [Vol10], or for pure OLAP query processing [BMK09]. However, industry thought leaders such as Hasso Plattner of SAP voiced the requirement for so-called real-time or operational business intelligence that demands complex query processing in "real time" on main memory resident data. SAP's Hana [FCP+11] and our hybrid OLTP&OLAP

database system HyPer [KN11], for which MPSM [AKN12] was developed, are two such databases. The query processing of in-memory DBMSs is no longer I/O bound and, therefore, it makes sense to investigate massive intra-operator parallelism in order to exploit the multi-core hardware effectively. Only massively parallel query engines will be able to meet the instantaneous response time expectations of operational business intelligence users if large main memory databases are to be explored. Single-threaded query execution is not promising to meet the high expectations of these database users as the hardware developers are no longer concerned with speeding up individual CPUs but rather concentrate on multi-core parallelization.

Merely relying on straightforward partitioning techniques to maintain cache locality and to keep all cores busy will not suffice for the modern hardware that increases main memory capacity via non-uniform memory access (NUMA). Besides the multi-core parallelization, also the RAM and cache hierarchies have to be taken into account. In particular, the NUMA division of the RAM has to be considered carefully. The whole NUMA system logically divides into multiple nodes, which can access both local and remote memory resources. However, a node can access its own local memory faster than remote memory, i.e., memory which is local to another node. Therefore, **data placement** and **data movement** such that threads/cores work mostly on local data is a key prerequisite for high performance in NUMA-friendly data processing.

Micro-benchmarks on our 1 TB, NUMA database server led us to state in [AKN12] the following three rather simple and obvious rules (called “commandments”) for NUMA-affine scalable multi-core parallelization:

C1 *Thou shalt not write thy neighbor’s memory randomly* – chunk the data, redistribute, and then sort/work on your data locally.

C2 *Thou shalt read thy neighbor’s memory only sequentially* – let the prefetcher hide the remote access latency.

C3 *Thou shalt not wait for thy neighbors* – don’t use fine-grained latching or locking and avoid synchronization points of parallel threads.

By design, MPSM obeys all three commandments. It scales almost linearly with the number of cores and outperforms state-of-the-art parallel join implementations. The IBM database research group at Almaden further improved our original MPSM algorithm with respect to an optimized data movement that avoids cross traffic between NUMA partitions [LPP+13]. In this paper, we go one step further and investigate the effectiveness of MPSM for non-inner join variants and complex query plan.

The remainder of the paper is structured as follows: In Section 2, we recap the MPSM algorithm. Then, we extend MPSM to compute non-inner join variants such as semi and outer joins in Section 3. In Section 4, we investigate the applicability of MPSM for complex query plans. We evaluate the presented MPSM variants in Section 5. Finally, we conclude our work in Section 6.

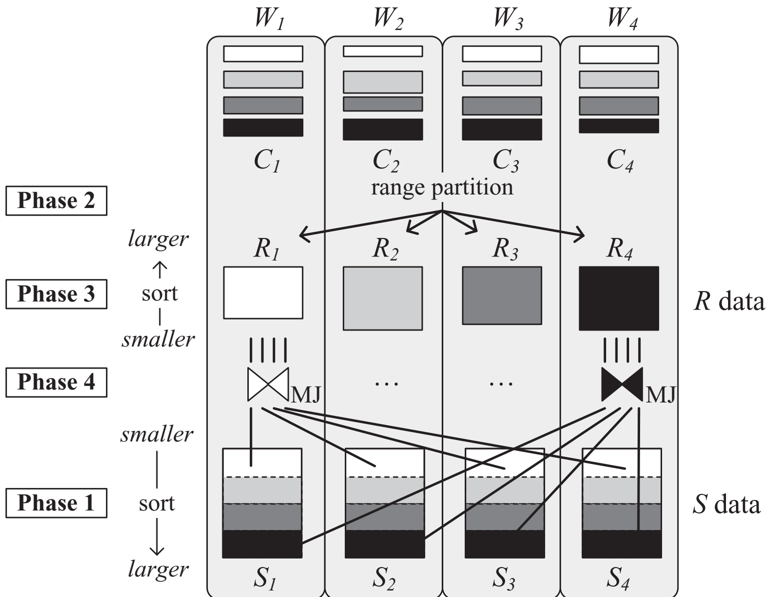



Figure 1: P-MPSM join with four workers $W_i$

## 2 The MPSM Algorithm

In [AKN12], we presented a suite of massively parallel sort-merge join algorithms for main memory and for disk-based systems. We limit the discussion of MPSM and its extensions to the range-partitioned version (P-MPSM) in this paper.

The MPSM join is designed to take NUMA architectures into account, which were not yet in the focus of prior work on parallel join processing for main memory systems. Though, we emphasize that MPSM is oblivious to specific NUMA architectures as it only assumes the locality of a RAM partition for a single core – without relying on multiple cores sharing the locality of RAM partitions or caches. As illustrated in Figure 1, each data chunk is processed, i.e., sorted, locally. Unlike traditional sort-merge joins, we refrain from merging the sorted runs to obtain a global sort order and rather join them all in a brute-force but highly parallel manner. During the subsequent join phase, data accesses across NUMA partitions are sequential, so that the prefetcher mostly hides the access overhead. We do not employ shared data structures so that no expensive synchronization is required. Therefore, MPSM obeys all three NUMA-commandments by design.

The P-MPSM processes its input in four phases as sketched in Figure 1 for a scenario with four workers. We call $R$ the private input and $S$ the public input. The input data is chunked into equally sized chunks among the workers, so that for instance worker $W_1$ is assigned a chunk $R_1$ of input $R$ and another chunk $S_1$ of input $S$. In phase 1, each worker sorts its public input run locally, resulting in runs $S_1$ to $S_4$. Subsequently, in phase 2, the private input chunks $C_1$ to $C_4$ are range partitioned. Thereby, the private input data is partitioned into disjoint key ranges as indicated by the different shades in Figure 1 ranging from white

over light and dark gray to black. In phase 3, each worker then sorts its private input chunk and in phase 4, each worker merge-joins its own private run $R_i$ with all public input runs $S_j$.

Besides the general algorithmic structure, several implementation details of the MSPM are essential for its good performance. This is discussed in detail in [AKN12], but in particular efficient sorting and efficient range partitioning are absolutely essential. Both steps must be executed largely branch-free, comparison-free, and synchronization-free, as otherwise they can easily dominate the join costs.

# 3 Outer, Semi, Anti Semi Joins

Depending on whether the private or the public input (or both) produces outer, semi, or anti semi join result tuples, additional data structures are required. As we will show, the costs for these data structures in terms of time and space, are negligible.

As each thread traverses its private input several times, we need to maintain joined-flags for private input tuples. We then only produce (additional) output tuples if

- no join partner has been found (outer and anti semi join) or if
- the respective tuple has not been joined yet (semi join).

On the contrary, each public input tuple is touched only once due to the implicit partitioning of the public input. Therefore, we can decide right away if outer, semi, or anti semi output tuples have to be produced. Before the join phase 4, outer, semi, and anti semi MPSM process their inputs the same way as inner join MPSM. If necessary, joined-flags in the form of bitmaps tracking whether private input tuples have (already) been joined are initialized when MPSM enters join phase 4.

While semi joins are very similar to inner joins (output tuples are produced if a join partner has been found), outer and anti semi joins require some attention in order to avoid missing or duplicating output tuples due to range partitioning and interpolation search. In the following, we briefly describe the implementations of outer, semi, and anti semi joins. $R$ and $S$ denote the private and the public input, respectively.

## 3.1 Outer Joins

Outer joins bring together matching tuples like inner joins, and, in addition, they produce output tuples for input tuples that didn't find a join partner. The left (private input) outer join $R ⟕ S$ requires joined-flags indicating whether the private input tuples already took part in the join or not. Each time a regular join output tuple is generated, the corresponding flag is set. In Figure 2, this is indicated by the "set" arrows toward the bitmap, which are shown only for worker $W_1$ for the sake of readability. During its last merge join, in addition to the regular merge join computation, each thread checks the joined-flags and

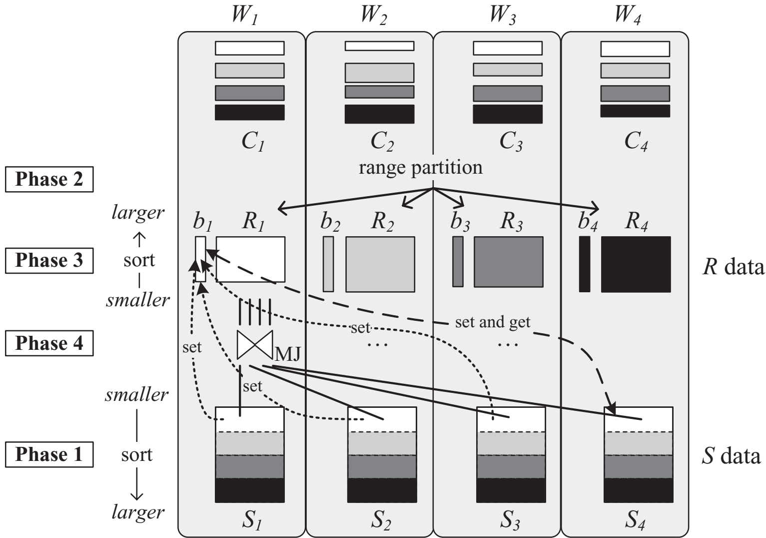
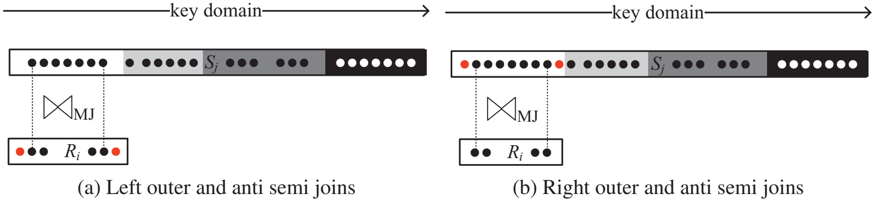



Figure 2: Outer and anti semi P-MPSM join with four workers $W_i$ maintaining each an additional joined-bitmap $b_i$ for their private input runs

produces output tuples for private input tuples for which the flag is not set. In the example in Figure 2, when worker $W_1$ conducts the last merge join of its private input run $R_1$ with $S_4$, it sets flags for joined tuples and gets flags to decide on the generation of additional output tuples ("set and get" arrow from and to bitmap $b_1$).

As opposed to the computation of inner joins, not only tuples finding a join partner have to be considered but also those that do not find a match. This requires special care: Due to interpolation search, the first private input tuples may be skipped. Further, the last tuples may be skipped as merge join stops early as soon as one of the inputs terminates. These two issues are illustrated in Figure 3a where the first and the last tuple of $R_i$ are not considered. In order to not miss outer output tuples, it is therefore crucial for each thread to scan its whole private input (at least) once. Thus, when executing the last merge join, the threads omit interpolation search on their private input runs and start scanning at the first tuple in their run. This actually mainly affects $R_1$ as for all other $R$ runs interpolation search is usually performed on $S$ runs. Further, the threads scan their private input run up to its last tuple irrespective of the occurrence of matching tuples in $S$.

The right (public input) outer join $R ⟖ S$ is straightforward as it can be decided at the time a tuple is processed whether it found a join partner or an extra output tuple has to be returned. However, here again due to interpolation search and early stop of merge join, the first and last tuples of the considered key range may be skipped as illustrated in Figure 3b. Therefore, interpolation search on public input runs $S_j$ is not based on tuple key values of $R_i$ but on the splitters determined in phase 2. That way, all $S$ tuples within a worker's key range (white to black shades in the figures) are considered in the join processing.




(a) Left outer and anti semi joins

(b) Right outer and anti semi joins

Figure 3: Outer and anti semi join require attention: due to interpolation search and early stop of merge join the outer-most (red) tuples are skipped by inner-join MPSM

### 3.2 Semi Joins

Semi joins produce output for tuples of one of the inputs which find a join partner in the other input. In contrast to inner joins, one input tuple may produce at most one output tuple. For this purpose, the left (private input) semi join $R \ltimes S$ requires joined-flags indicating whether a private input tuple already took part in the join or not. If so, the tuple will not produce any output again. If the flag is not set and a public input tuple matches, an output tuple is produced and the corresponding flag is set. As opposed to left outer joins, the joined-flags are not only checked at the end of the join phase 4 but need to be consulted for each matching tuple during each single merge join. This is illustrated in Figure 4 using "get and set" arrows for all merge joins.

For the right (public input) semi join $R \rtimes S$, the private input is scanned for a specific public input tuple until one match is found or the key of the private input is greater than the current public input key. If a match exists, an output tuple is generated and the worker moves on to the next public input tuple as the current may not produce any further output.

### 3.3 Anti Semi Joins

Anti semi joins are the opposite of semi joins. Output is produced for input tuples that do not find a join partner in the other input. The left (private input) anti semi join $R \triangleright S$ requires joined-flags indicating whether the private input tuples already took part in the join or not. Each time a public input tuple matches, the flag for the corresponding private input tuple is set (without producing an output tuple). In Figure 2, this is indicated by the "set" arrows toward the joined-bitmap $b_1$ of worker $W_1$. As for outer joins, during the last merge join, in addition to setting flags for joined tuples, each thread checks the bitmap and produces an output tuple for each private input tuple, for which the flag is not set ("set and get" arrow from and to bitmap $b_1$). Due to interpolation search and early stop of merge join, some tuples may be skipped as illustrated in Figure 3a. As for left outer joins, by omitting interpolation search on private input runs for the last merge join and scanning the private input completely, we make sure that no anti output tuples are missed.

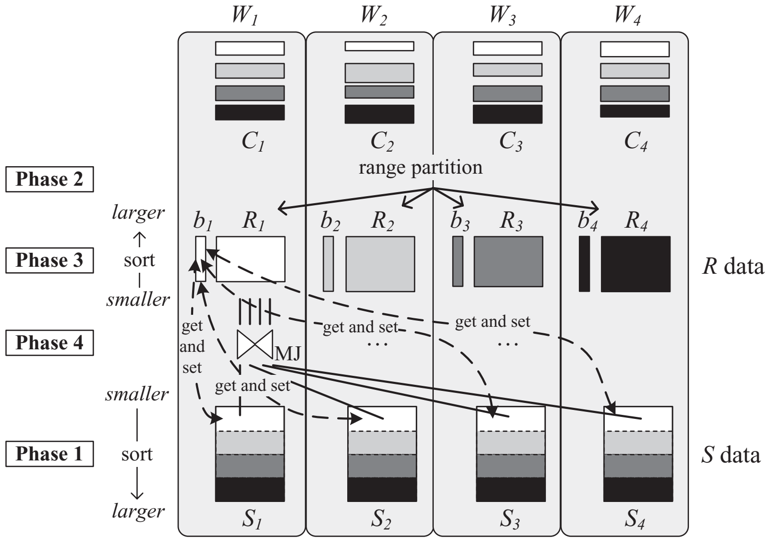



Figure 4: Semi P-MPSM join with four workers $W_i$ maintaining each an additional joined-bitmap $b_i$ for their private input runs

When computing the right (public input) anti semi join $R \triangleleft S$, it can be decided at the time a tuple is processed whether it found a join partner or – in case no join partner at all was found – an output tuple has to be returned. Again, we need to make sure that no output tuples are missed due to interpolation search and early stop of merge join as shown in Figure 3b. Therefore, as for right outer joins, interpolation search on public input runs $S_j$ is based on the splitters determined in phase 2, so that all $S$ tuples within a worker's key range are considered in the join processing.

# 4 Complex Query Plans: The Guy Lohman Test

After having covered MPSM for inner, outer, semi, and anti semi joins, we put MPSM to the Guy Lohman test [Gra93] stating that a join operator must not only be useful for joining two inputs but also in complex query execution. In particular, an operator is suitable in complex query processing if it does not require intermediate results to be materialized for further processing. MPSM is roughly order preserving, which can be exploited in subsequent join operations of a complex query plan as shown by [SAC+79, CKKW00]. We depict different ways of how to make use of the output sort order in a sequence of two MPSM joins. Here, we consider the intermediate result to be taken as private or public input for further processing. Teams even go one step further and combine multiple operations in a single one. Thereby, teams are usually more efficient than an equivalent sequential execution of the operations by an effective preprocessing of the data.

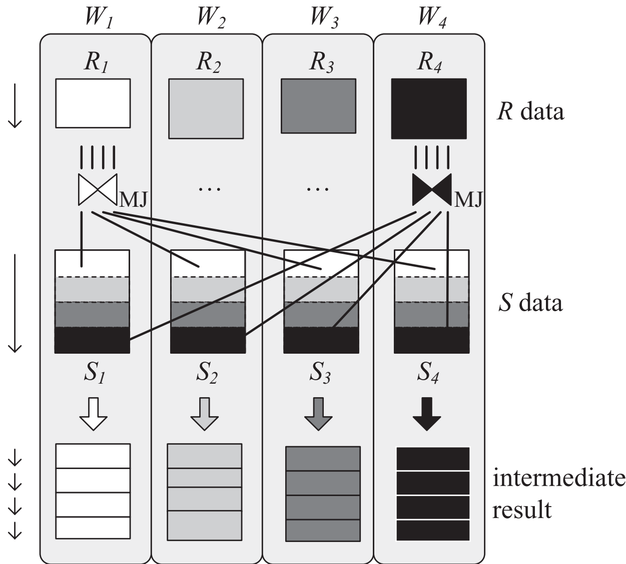



Figure 5: P-MPSM join with four workers $W_i$: Each worker produces four sorted (denoted by the arrows on the left) runs covering only its private input run part of the key domain and stores them locally

We present approaches for the use of MPSM in multiple join operations on the same column(s) and discuss their applicability for cases where the joins are executed on different columns.

### 4.1 Initial Situation

Figure 5 illustrates the situation after one MPSM join has been executed. Each of the workers produces several sorted output runs covering only part of the key domain. The intermediate result data is stored locally. A second MPSM join operator may take the intermediate result data as private or as public input depending on its size compared to the third relation to be joined. Assuming certain data distributions (in particular, similar data distributions of the inputs to the first and the second join), we can benefit from the given range partitioning of the intermediate result. When using it as private input we can omit re-partitioning the data. When using it as public input, this introduces location skew, i.e., most or all join partners of a private input run will be found in one local or remote public input run. As we showed in [AKN12], this reduces the effective number of merge joins and thus execution time.

Without any knowledge of the data distribution, however, the second MPSM join processing the intermediate result and a third relation is executed as usual. That is, the public

input is sorted, the private input is re-distributed among the workers and sorted, and the private input runs are each merge joined with all public input runs. We therefore use this scenario as the baseline and compare our approaches presented below to it.

### 4.2 Local Merge of Output Runs

Each worker's output consists of sorted runs within the worker's assigned key range. By merging the output runs one sorted run of the respective key range is produced. This intermediate result run can then be fed into the second join as private or as public input. Used as private input, we benefit from the given range partitioning. If there is key value skew within the inputs to the second join it is handled by computing new splitters and passing consecutive parts of the own run to other workers. As the workers' key ranges are disjoint and the data is already sorted, this only requires copying or linking run parts. When used as public input, this introduces location skew, i.e., basically only one merge join pass is required to find all join partners of a private input run as described above.

This variant requires each worker to store its complete intermediate result before it can be merged as the runs are produced subsequently. Furthermore, the sort order within one worker's output runs cannot be exploited if the second join is computed using different join column(s). It is possible, however, to sort the input chunks primarily by the first join column(s) and secondarily by the second. This requires the second join column(s) to be contained in the respective input relation (which is the case for one of the inputs) and incurs high merging overhead (potentially $n \cdot |D|$, where $n$ is the number of workers and $D$ is the key value dimension, runs have to be merged). In contrast to the scenario of merging in between joins on the same column(s), the merged output run then contains the complete key range, i.e., is not range partitioned.

### 4.3 Concatenation of Output Runs

When concatenating the workers' output runs instead of merging them, each worker obtains one sorted run covering the complete key range. This is achieved by letting each worker $W_i$ collect the $i$-th output run of all workers and append those runs. In contrast to merging, concatenating theoretically does not require the intermediate result to be materialized completely in case we know the size of the intermediate result runs. However, practically this is only applicable in case of non-filtered primary key–foreign key joins. Otherwise, additional buffer might be allocated for the result runs so that they are not completely dense. As the resulting runs cover the complete key range, feeding them into the second join as private input will not be beneficial as no work can be saved. We might only exploit the given sort order to copy whole run parts during scattering instead of considering each tuple on its own. When using the intermediate result runs as public input, sorting can be omitted.

This approach cannot be adapted to work for multiple joins on different columns.

### 4.4 MPSM Teams

Teams prepare all inputs to be joined before starting the join phase such that both joins can then be done in one pass. For hash based join algorithms, this means partitioning all input relations, then loading the corresponding partitions of all relations and producing output tuples [GBC98]. We adapt this idea to MPSM by pre-processing the relations in the following way: we range partition the two smaller relations (i.e., treat them as private inputs) and sort chunks of the third one (public input). Then, each worker is in charge of merge-joining the two corresponding range partitioned chunks to all sorted runs of the public input.

MPSM Teams are not directly applicable for joins on different columns. In case of joins along primary key-foreign key chains with $1:N$ functionalities, it is however possible to map the join keys to new values and allow for teams processing even for different join columns. Of course, key mapping incurs extra overhead.

### 4.5 Pipelined Execution

The approaches above share the disadvantage that intermediate results have to be stored. This contradicts the Guy Lohman requirements for join operators. We now present a pipelined execution of two subsequent MPSM joins, which exploits the fact that each worker produces sorted output runs and that these runs can immediately be joined with the third relation. In total, each worker then executes quadratic as many merge joins as there are workers (and thus output runs), however, sorting of the third relation and the merge joins between intermediate result runs and runs of the third relation are executed in parallel with the first join processing.

The pipelined MPSM is applicable to joins on different columns. The pipelined intermediate result run parts are then sorted, thereby probably losing the range partitioning of the private input.

## 5 Experimental Evaluation

We implemented the MPSM join variants in C++, and compiled the query execution plans to machine code as employed in our HyPer query processor [Neu11]. In all our experiments, the input data is completely in main memory. To minimize interactions with other parts of the system, we consider the case where the input relations are scanned, a selection is applied, and then the results are joined. Thus, no indexes or referential integrity constraints (foreign keys) can be exploited during query processing. In the following, we vary the data sets regarding input sizes, join multiplicities, and data distributions to explore the application space thoroughly. Note that join multiplicities are expected multiplicities, individual tuples might have more or less join partners, or even none at all.

### 5.1 Platform and Benchmark Scenarios

We conduct the experiments on a Dell PowerEdge R910 Linux server (kernel 3.0.0) with 1 TB main memory and four Intel(R) Xeon(R) X7560 CPUs clocked at 2.27 GHz with 8 physical cores (16 hardware contexts) each, resulting in a total of 32 cores (and due to hyperthreading 64 hardware contexts). The machine has four NUMA regions, one for each CPU socket. Due to its large main memory of 1 TB that Dell "squeezed" into this comparatively low-cost server (ca. 40,K Euro) it has quite noticeable NUMA effects. Some micro-benchmark results with NUMA effects for this precise server are included in [AKN12].

In the experiments, each tuple consists of a 64-bit key within the domain $[0, 2^{32})$ and a 64-bit payload:

$$\{[\textit{joinkey}\text{: 64-bit}, \textit{payload}\text{: 64-bit}]\}$$

Each dataset consists of two relations $R$ and $S$. $R$ is $1600M$, the cardinality of $S$ is scaled to be $1{\cdot}|R|$, $4{\cdot}|R|$, $8{\cdot}|R|$, and $16{\cdot}|R|$. Our datasets of cardinality $1600M\times$ (1+multiplicity) have sizes ranging from 50 GB to 425 GB, which is representative for large main memory operational BI workloads. The multiplicities between the relations $R$ and $S$ further cover a wide range, including not only the common cases (4, as specified for instance in TPC-H and 8 to approximate the TPC-C specification) but also extreme cases (1 and 16). The data was generated by uniformly generating 32-bit integers that were padded to 64-bit join keys. Thereby, referential integrity was not given, which results in "real" outer join result tuples. For the experiments on multi-way joins, we extended the datasets by a third relation, which is scaled in the same way.

The experiments are conducted using a parallelism of 32 (equal to the number of physical cores) if not stated otherwise.

### 5.2 Performance of Outer, Semi, and Anti Semi Join compared to Inner Join

We execute an equi-join on the tables:

```
SELECT count(*)
FROM R <join variant> S
WHERE R.joinkey = S.joinkey
```

This query is designed to ensure that the payload data is fed through the join while only one output tuple is generated in order to concentrate on join processing cost only. Further, we made sure that early aggregation was not used.

In Figure 6, we compare the execution time of the inner MPSM join presented in [AKN12] and the non-inner join variants for multiplicities between 1 and 16. The non-inner join variants described in Section 3 incur no (in case of right outer, semi, and anti semi joins) or only little overhead for tracking whether one tuple of the left input already found a join partner in the right input (in case of left outer, semi, and anti semi joins). The modification

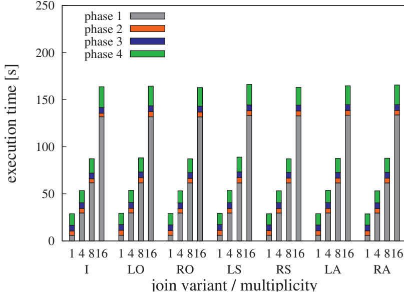



Figure 6: Inner (I), left outer (LO), right outer (RO), left semi (LS), right semi (RS), left anti semi (LA), and right anti semi (RA) join

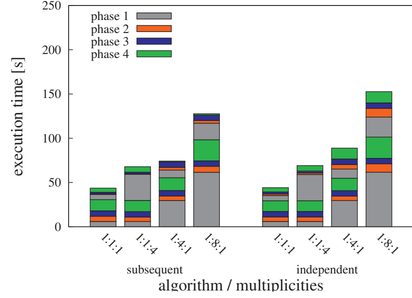



Figure 7: Comparing two subsequent MPSM join executions to two independent MPSM joins

of interpolation search required for outer and anti semi joins does not incur additional overhead. In total, we find that the performance decrease caused by the additional data structures is negligible.

### 5.3 Exploiting MPSM Characteristics in Complex Query Plans

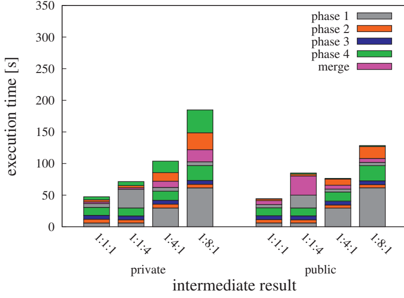
We examine the suitability of MPSM for complex query plans on the example of a three-way-join on the same join key:

```
SELECT max(R.payload + S.payload + T.payload)
FROM R, S, T
WHERE R.joinkey = S.joinkey AND S.joinkey = T.joinkey
```

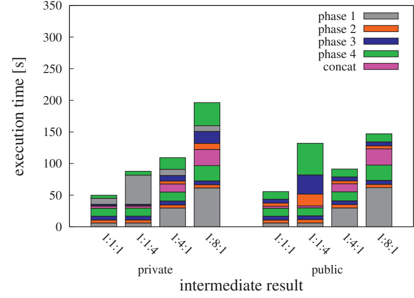
We compare the alternatives of exploiting the rough sort order of the MPSM output presented in Section 4 to the base case where two MPSM joins are executed subsequently without post-processing the intermediate result. We report experiments using the multiplicities $1:1:1$, $1:1:4$, $1:4:1$, and $1:8:1$. In a perfect scenario, an optimizer would always join smaller relations first, i.e., the third and fourth case wouldn't occur. However, we included those experiments to cover cases in which the intermediate result is smaller than the third table and in which it is larger, without modifying the key ranges or uniformity of the data distribution. In our experiments, for multiplicity $1:1:1$, the intermediate result is a little smaller than the third relation, for $1:1:4$ it is much smaller, for $1:4:1$ it is a little larger, and for $1 : 8 : 1$ it is much larger.

#### 5.3.1 Implicit Benefits of Subsequent MPSM Joins

We first want to point out that two subsequent MPSM joins in one query plan implicitly benefit from locality of the data and range partitioning. As illustrated in Figure 5, each worker's output runs are stored locally and cover only part of the key domain. A




(a) Merging the intermediate result runs between two subsequent MPSM joins




(b) Concatenating the intermediate result runs between two subsequent MPSM joins

Figure 8: Two subsequent MPSM joins exploiting the rough sort order of the intermediate result runs

second operator (not changing the affinity of threads to cores in between) can therefore initially work on local data and (in case of a second MPSM) profits from the location skew introduced by the first operator. In Figure 7, we compare the execution times of two independent MPSM joins and two subsequent MPSM joins. The positive effect shows in the two leftmost bars in the second join's phase 2 (upper red) and phase 3 (upper blue) execution times and in the two rightmost bars in the second join's phase 1 (upper gray) and phase 4 (upper green) execution times.

### 5.3.2 Merge and Concatenation of Intermediate Result Runs

We applied merge and concatenation to the intermediate result runs and then fed the result into the second join, once as private input and once as public input. As shown in Figure 8, the findings in [AKN12] that the smaller relation should always be picked as the private input were confirmed. This is due to the efficient scans on local memory compared to remote memory. In the following, we therefore assume the optimizer to correctly assign private and public input roles to the smaller, respectively larger relations after the first join.

## 5.4 Comparison to Baseline

In Figure 9, we compare the total execution time of two MPSM joins using the approaches described in Section 4 to that of two subsequent MPSM executions without any additional processing of the intermediate result. Overall, the simple execution of two MPSM joins shows the best performance.

Merging the intermediate result runs to use them as private input to the second join allows for omitting the sort phase (in case of uniform distribution). However, merging shows approximately the same performance as the optimized sorting technique of MPSM combining Radix sort and Introsort. When using the merged runs as public input this has the same positive effect as location skew [AKN12]. Two subsequent MPSM joins benefit from location skew as well (see Section 5.3.1) and thus has the same performance.

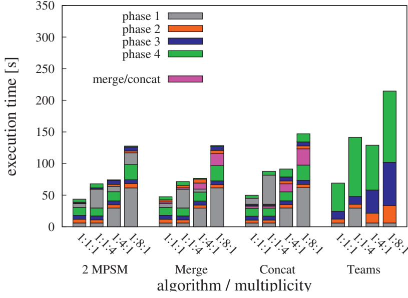



Figure 9: Performance comparison of two subsequent MPSM joins without post-processing of the intermediate result (2 MPSM), with merging each worker's runs of the intermediate result (Merge), and with concatenating the intermediate result runs (Concat)

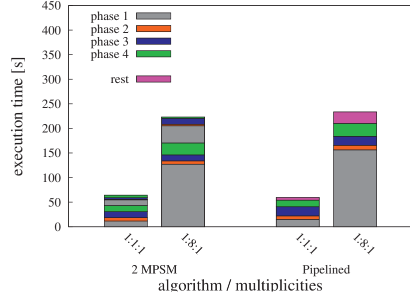



Figure 10: Performance comparison of two subsequent MPSM joins without post-processing of the intermediate result (2 MPSM) and pipelined MPSM (16 threads)

Concatenating the result runs is less beneficial when using the outcome as private input to the second join. This is because those runs cover the complete key range and thus must be range-partitioned as usual. When using the intermediate result runs as public input, the performance even degrades because concatenating the runs from multiple remote NUMA partitions is even slower than sorting the own runs within the local partition.

The MPSM Teams are not competitive at all as three-way-merge joining incurs a very high overhead. We conclude that subsequent merge joins are more efficient than three-way-merge joins.

### 5.5 Pipelined MPSM

For the evaluation of pipelined MPSM we instantiate 16 threads to process the first join and another 16 threads to which the intermediate results are piped. The total number of threads thus equals the number of physical cores on our server.

Figure 10 shows the comparison of two subsequent MPSM joins to pipelined MPSM. Here, "rest" denotes the time from the completion of the first join until the second join execution finishes. Due to the additional bandwidth incurred by the parallel processing of the first join and operations (sorting of the third relation and merge joining) of the second join, the performance of the first join degrades slightly. Overall, there is no significant performance difference between the two three-way-join variants.

## 6 Conclusions

In this work, we developed the algorithmic details of MPSM for other join variants, i.e., outer, semi, and anti semi joins. We also worked on exploiting the rough sort order that MPSM inherently generates due to its range-partitioned run processing. We compared the effect of merging and concatenating intermediate result runs to MPSM Teams execution processing a three-way-join in one operation. Furthermore, we investigated a pipelined version of MPSM. The experimental evaluation revealed that the efficient sort and merge phases of MPSM leave almost no room for improvement by replacing sort by merge or concatenation or by overlapping the merge phase with subsequent operations. Although some of the proposed optimizations for MPSM in complex query processing are applicable for multiple joins on different columns, we are confident that executing two subsequent MPSM operations results in the most robust and efficient performance.

## References


[AKN12] 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.

[BMK09] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2):1648–1653, 2009.

[CKKW00] Jens Claussen, Alfons Kemper, Donald Kossmann, and Christian Wiesner. Exploiting Early Sorting and Early Partitioning for Decision Support Query Processing. *The VLDB Journal*, 9:190–213, 2000.

[FCP+11] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database: Data Management for Modern Business Applications. *ACM SIGMOD Record*, 40(4):45–51, 2011.

[GBC98] Goetz Graefe, Ross Bunker, and Shaun Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. In *VLDB*, pages 86–97, 1998.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System based on Virtual Memory Snapshots. In *ICDE*, pages 195–206, 2011.

[LPP+13] Yinan Li, Ippokratis Pandis, Ippokratis Pandis, Vijayshankar Raman, and Guy Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[Neu11] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *VLDB*, 2011.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, pages 23–34, 1979.

[Vol10] VoltDB LLC. VoltDB Technical Overview, Whitepaper, 2010. `http://voltdb.com/_pdf/VoltDBTechnicalOverviewWhitePaper.pdf`.