OPTIMIZING WORST-CASE OPTIMAL JOINS

Kyle Fearne, Jón Gunnar Hannesson, Luca Strässle, Roman Tarasov

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Worst-case optimal join algorithms are an attractive option for designers of relational database management systems. These algorithms offer an optimal runtime in the worst case, as they have a runtime which is bounded by the output size of the join. Relational database management systems are generally highly optimized, as particularly large analytical queries can take hours to run. This is most notable in the use of query optimizers and indexes, but equally important are optimized implementations of the underlying algorithms. In this report we present details on the implementation, systematic optimization and performance analysis of two algorithms: a generic worst-case optimal join and a hash-based worst-case optimal join algorithm. The defining feature of most of the proposed optimizations is that they are directed towards optimizing memory management, cache locality and bandwidth utilization, as the runtime of the algorithms is bounded by memory bandwidth as opposed to computation. We demonstrate methods that make the first algorithm up to forty thousand times faster and double the speed of the second, hash-based join algorithm for selected benchmarks.

1. INTRODUCTION

Motivation. Worst-case optimal join algorithms compute relational joins with a runtime that is bounded by the worst-case output size of the join, making them asymptotically faster than traditional binary join algorithms. This is especially true when there is an explosive number of intermediary results generated by the binary joins, which can grow larger than the result itself. The primary issue with worst-case optimal join algorithms is that for them to be truly optimal, indexes are required on all possible join-key orders. Having to store all these indexes means that there is a huge storage overhead, and an update to the table would be very expensive due to having to update the indexes. Freitag et al. [1] propose an implementation which generates these index structures on the fly in the shape of 'index tries'. Worst-case optimal joins are not better than binary joins in all cases, which leads to a second issue - the database system which implements worst-case optimal joins needs to also implement an optimizer which can decide when to implement which type of join algorithm. Freitag et al. also propose a hybrid optimizer which can detect growing joins, or joins with growing intermediary results, and replaces these joins with worst-case optimal joins.

Contribution. In this paper, we present our implementations of the algorithms proposed by Freitag et al. [1]. Base implementation for Algorithms 1-3 were done and these were then optimized as described in the next sections. A base implementation of Algorithm 4 was also done, however it was decided that this will not be pursued further as avenues of optimization were limited, and feasible input sizes did not produce high runtimes. The implementations of these algorithms address the main drawbacks brought about by worst-case optimal joins discussed previously.

Related work. The algorithms implemented by our team were all described, implemented and tested by Freitag et al. [1], however, this code was not made available. One of the most fundamental differences between Algorithms 2-3 and Algorithm 1 is the hashing of tuple values in order to construct the hash trie structures. Due to this, it is of utmost importance that the selected hashing function minimizes collisions and is performant. Freitag et al. propose using AquaHash or Murmurhash when discussing their implementation specifics. Ultimately, Murmurhash was selected for our implementation [2]. Many of the optimizations described in the next sections are built upon ideas found in other papers. From early on, it became apparent that changing the data layout to improve locality would be an important optimization. The advantages and disadvantages of vertically partitioning relations into a columnar format have been vastly researched in [3, 4]. A variety of implementations of this idea have made their way to the market, a popular example being MonetDB [5]. These examples inspired the optimized data layout of our algorithms, which involved storing the join attributes separately from the rest of the attributes. In Algorithms 2 and 3, Freitag et al. reference a two-pass radix partitioning, proposed by Balkesen et al. [6] to ensure that similar hash values reside in memory locations which are physically close to each other. The aim of this was to improve cache performance in the build and probe phases. Whilst the approach in the paper was not followed exactly,

this approach served as inspiration for one of the optimizations concerning Algorithms 2 and 3, where the tuples were partially sorted. An implementation similar to what was described in Algorithm 4 by Freitag et al. was found in Java, by Mhedhbi and Salihoglu [7]. Their work demonstrates how it is possible to seamlessly mix worst-case optimal joins and binary joins into a single hybrid query plan.

2. BACKGROUND ON THE ALGORITHM/APPLICATION

In this section a brief introduction is given to the algorithms described in Freitag et al. [1] that have been optimized within the scope of this report. The discussion is supplemented with metrics for cost analysis and asymptotic complexities of the algorithms.

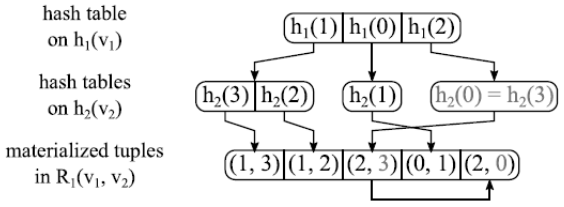

Fig. 1. Illustration of the Hash Trie structure, as described by Freitag et al. [1]

Join Algorithms. The first algorithm referenced in the paper by Freitag et al. [1] (Algorithm 1), is a generic worst-case join which is a conceptual backtracking algorithm. The algorithm is recursive, and takes a set of relations, a set of join attributes and the index of the currently processed join attribute as input. For each join attribute a subset of relations that contain the particular attribute is formed. The tuples of each relation of the subset are then checked, keeping only tuples in which the value of the currently processed join attribute is present in all other relations of the subset. The tuples that fail this validation are filtered out. When all join attributes are processed a Cartesian product is executed to merge all matched tuples.

The proposed modified version of generic worst-case join is called hash trie join and consists of two phases: build and probe (Algorithms 2 & 3). This join is based on a hash trie datastructure, which is shown in Fig. 1. The hash trie structure proposed in the paper is a prefix tree where nodes contain hash tables which map hash values to either a child node or to a linked list of tuples in the leaves of the trie structure. Each level of the hash trie stores the hash values for one join attribute. A path from the root to a leaf represents a linked list of tuples. There is more than one tuple in the linked list in case of hash collisions or when two tuples have the same values in all join attributes. The build phase focuses on creating and populating the hash trie data structure for each relation in the join, which is then used as an input to the probe phase. The probe phase performs a join that naturally follows the flow of the generic worst-case algorithm, but instead of producing unique set intersections it traverses the hash tries.

Cost Analysis. As there are no floating-point operations in the algorithms, and barely any integer operations, using either as a cost metric was not feasible. The cost metric that was optimized for in this paper is runtime, which was based on a few representative queries with a fixed input size from the TPC-H [8] benchmark. Total runtime is a crucial point for modern RDBMSs (Relational Database Management Systems) as they rely on fast execution. A secondary metric was also used, which follows from the nature of the algorithms, being that they are memory bound. Current computer architectures exploit caching principles to reduce the number of transfers from and to main memory. Consequently, we decided to focus on the number of transfers between LLC (last-level cache) and main memory in the form of the number of LLC misses during algorithm execution, as for larger queries these misses became a bottleneck.

Complexity. In their paper, Freitag et al. perform an in-depth analysis of all algorithms' time complexities [1]. They present a join as a query hypergraph structure, with each node representing an attribute, and hyperedges representing the relations. The generic worst-case join algorithm has the following time complexity:

$$O\left(mn^2 + n\sum_{j=1}^{m}|R_j| + 2n\sum_{j=1}^{m}|R_j|^2 + \prod_{j=1}^{m}|R_j|\right)$$

where n refers to the number of join attributes, m to the number of relations and $|R_j|$ to the cardinality of a particular relation. The space complexity is defined by the total space required to store all relations:

$$O\left(\sum_{j=1}^{m}|R_j|\right)$$

The build phase of the hash trie join has to go over all tuples to create a hash value for all join attributes which makes both the time and space complexities bounded by the number of join attributes and the cardinality of the relations:

$$O\left(n\sum_{j=1}^{m}|R_j|\right)$$

For defining the time complexity of the probe phase the concept of fractional edge cover is used [1]. The probe

phase has the following time complexity:

$$O\left(nm\prod_{j=1}^{m}|H(I_j)|^{x_j}\right)$$

where $|H(I_j)|$ is the size of the set of hashed join keys, that are present in the tuples stored in the leaves of the subtrie rooted in the node pointed to by the hash trie iterator I_j. $\mathbf{x}$ is an arbitrary fractional edge cover of the query hypergraph. The space complexity is:

$$O\left(nm\right)$$

3. YOUR PROPOSED METHOD

In the following section the base implementations and optimizations are explained in detail. The generic worst-case join algorithm is discussed first, followed by the hash-trie join algorithm. To reason about certain optimizations, experiments will be mentioned. Those experiments are discussed in more detail in Sec. 4.

3.1. Generic Worst-Case Join

Base Implementation. The base implementation is a recursive function which takes a set of relations and the current join attribute index as an input. In each recursive step, tuples in participating relations are checked and only the ones matching on the currently considered join-key are selected and passed to the next step. At the last step, a full Cartesian product is carried out. This means that all possible tuples are materialized, and then the join-keys are compared to each other. Only the tuples which match on all join-keys are outputted. In the base implementation, the relations are stored as structs and all values in the relation are stored as a contiguous array of strings. This means that all join-key comparisons are done using `strcmp()` and that cache locality is poor. When comparing join-keys, the cache is populated with neighbouring fields that are not relevant to the algorithm.

Separating Join Attributes. To address this, it was decided that the join attributes should be stored in separate arrays - each containing only a specific join attribute. This allows for two things - firstly, the join attributes in the TPC-H benchmarks are all integers so they can be stored as such in the structs. This makes integer instead of string comparisons possible. Secondly, this column-store like approach, makes better use of spatial locality. During selections and comparisons, only join attributes are brought into cache. This optimization improved the runtime of all our subfunctions, except for the function executing the Cartesian product, which was still dominating the algorithm's runtime.

Removing Recursion. The two recursive functions in this algorithm, the enumerate and the Cartesian product are both replaced with an iterative version in this optimization. This did not result in any runtime reduction, but simplified the algorithm flow and helped us to reason about further optimizations.

Partial Materialization in Cartesian Product. As mentioned earlier, the Cartesian product is by far the biggest bottleneck in all previous implementations. Before this optimization, a full Cartesian product was executed, materializing every possible tuple. Once materialized, a comparison was carried out on all the join attributes and discarded if there were any mismatches. In this optimized implementation of the Cartesian product, the comparison happens while the materialization is taking place instead of afterwards. This means that if the tuple is fully materialized, it implies that all the join attributes match. In case of any mismatch the function would continue with the next tuple instantly, meaning that no invalid tuples would ever be materialized. The number of last level cache misses and the total runtime can benefit a lot from this optimization.

Reduced Memory Activity in Selection. Following the successful optimization of the Cartesian product, it became evident that the select function constituted a significant portion of the overall runtime. Consequently, the focus of this optimization is directed towards improving the select function to further enhance the performance. In the initial implementation, the select function produced a new relation where all tuples that matched the selection predicate were copied into it using `memcpy()` function. With the optimized implementation, the original relation keeps track of selected tuples, by storing the tuple's indices in a newly introduced array. This reduces the memory footprint of the select function which resulted in almost halving its runtime.

3.2. Hash Trie Join

Baseline. In the base implementation the build and probe algorithms are both implemented recursively. The build phase uses Murmurhash [2] and linear probing to build the hash tries. It takes in a linked list of tuples, where each tuple is represented by a struct containing pointers to the next element in the list and to an array of strings that stores the tuple's values. The probe phase makes use of a stack to store the current node when moving down a level. When at the tuple level instead of performing a complete Cartesian product and then checking the join condition, early termination is used to avoid unnecessary computations. The Hash Trie that is built and then used in the probe phase is implemented using structs to represent the trie itself, as well as its nodes and buckets.

Remove Linked List. The build phase needs to know the size of the linked list considered in every recursive call to allocate a hash table of the correct size. In experiments the function that gets the length of the list turned out to be the bottleneck for the build phase. This is because it

involves traversing the linked list which not only leads to pointer chasing but also to bad locality, since the memory for every element is allocated separately. In the optimization two arrays are used instead of the linked list. One holds the values of the tuples while the other holds indices to the next tuples, substituting the pointers. This optimization led to the desired speedup for getting the list length and therefore further optimizations were built on top of this one.

Remove Recursions. There is typically an overhead that arises from recursive function invocations. To get rid of that overhead the recursions of the build and the probe phase were both replaced by iterative implementations. In the build phase a stack is used for this, much like a non-recursive implementation of the Depth First Search algorithm. In the iterative implementation of the probe phase, a loop is utilized to traverse the levels of the trie. Each level involves a scan relation that must be stored in case the algorithm descends to a lower level and subsequently returns to the original level. In the recursive implementation, this information is automatically preserved as the execution of the other level occurs through a recursive function call. In the iterative case, the scan relation needs to be saved in an array because a single variable would be modified at the lower level, rendering it unknown when transitioning back up to the previous level. Removing the recursions led to a speedup and was therefore used as a base for further optimizations.

Separate Attributes. In experiments, checking the join condition and materializing the results showed to be the bottleneck for the probe phase for large output sizes. Prior to this optimization string comparison was used to compare actual tuple values when checking the join condition. Leveraging the assumption that relations are exclusively joined on integer attributes, allows for the use of integer comparison. Furthermore, this optimization takes advantage of the fact that only the values of the join attributes need to be accessed in both the build and probe phases. As a result, an array of integers containing only the join attribute values can be used as an input instead of an array of strings holding all values. This reduces the size of the working set significantly and should lead to better locality. Finally, instead of performing a costly memory copy of the values to materialize the result, only the indices of the tuples forming a result tuple are stored. This optimization had a large impact on both the build and the probe phases and further optimizations were built on top of this one.

Remove Structs. The hash trie representation is implemented using arrays of structs, which can introduce overhead because it can lead to increased memory usage due to alignment. The goal of this optimization is to get rid of this overhead by representing the hash tries as arrays. To do so each struct element is turned into an array. This leads to a different memory layout because instead of storing all struct elements sequentially they are now stored in multiple different arrays. The new memory layout led to more last level cache misses. With it also the runtime increased and therefore the arrays instead of structs were not used in further optimizations.

Partition. This optimization uses partitioning of the input tuples to improve locality when building and searching tries. The use of a partitioning algorithm is mentioned in the paper by Freitag et al. [9] where a two-pass radix partitioning schema is employed [6]. The partitioning in our optimization makes use of the quicksort algorithm which repeatedly partitions an array based on a random pivot. The input tuples are sorted based on the hash value of the first join attribute. Because a full sort takes very long and becomes the bottleneck, partitions are not further sorted once they reach a certain size. This size is a tunable parameter that was chosen such that the total runtime of the complete join is minimized. The results of this tuning is discussed in the following section.

4. EXPERIMENTAL RESULTS

In the following section, all experiments that were conducted are described in detail along with the results. The experiments were generally done in three parts, an analysis of the runtime of subfunctions used within the algorithm, an analysis of last-level cache misses and a profiling of the total runtime of each algorithm. These experiments not only provide a comprehensive analysis of each method outlined in the previous section but also serve as a motivation for the implementation of the next optimization. Runtime measurements for all algorithms and subfunctions were made using the RDTSC (Read Time-Stamp Counter) method, which can be used to measure the runtime of a function in CPU cycles. Memory access analysis was done using Intel VTune, a profiling tool provided by Intel which, among other things, allows you to measure LLC misses, loads, and stores.

4.1. Generic Worst-Case Join

Experimental setup. Experiments for this algorithm were conducted on the performance core of Intel i7-1260P with 2.5GHz base frequency. Three levels of cache are available of sizes 480KB (288KB for data and 192KB for instructions), 7.5MB and 18MB respectively. All experiments were conducted with Turbo Boost disabled and with the compiler flag *-O1*. This was found experimentally, by trying out different permutations of compiler flags to see which yielded the best results. The chosen benchmarks were based on the TPC-H dataset [8]. In particular, two queries were selected to represent typical join use cases. The first query joins three tables containing a total of twenty one attributes, with two join attributes, and produces ten thousand output

tuples. The second query joins five tables with a total of forty attributes, with eight join attributes, and produces approximately seven hundred output tuples.

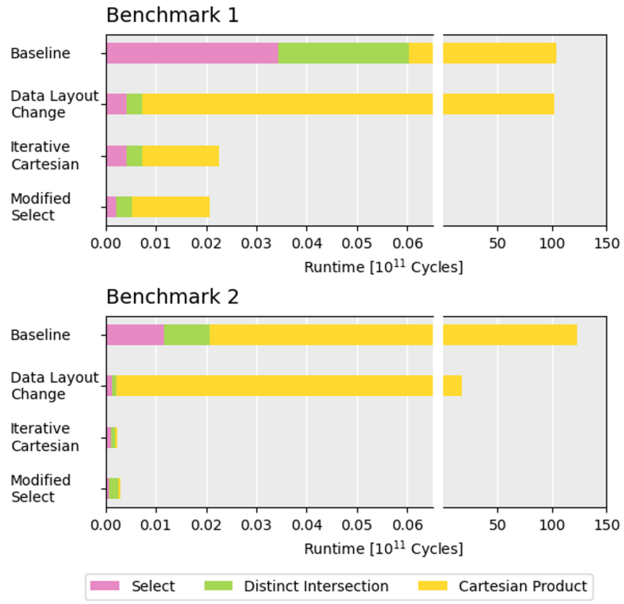

Fig. 2. Runtime in cycles for the generic worst-case join baseline and all optimizations.

Results: Subfunction Runtimes. The following subfunctions which contributed the most towards the total runtime of the algorithm were measured:

- Select: Filters tuples such that only tuples matching the selection predicate remain.
- Distinct Intersection: Provides the unique intersection of join values of relations for a particular join attribute.
- Cartesian Product: Materialization of the tuples in the output relation.

As can be seen in Fig. 2 there was a significant difference between the benchmarks in the base implementation as the *select* subfunction, which copied the tuples with values matching the join-attributes, was three times slower for the benchmark with a larger number of output tuples. At the same time *Cartesian product* dominated the runtime for both benchmarks. Interestingly, it took longer for the second benchmark to run. In this benchmark, the number of output tuples was much lower than the first benchmark, but the input was one hundred twenty five times larger. This can be explained by the fact that full tuple materialization was done, meaning that every single tuple was materialized and then a comparison took place which confirmed whether or not that tuple should form part of the result.

Making a change to the data layout yielded a big overall improvement for all of the subfunctions except for *Cartesian product* which remained a bottleneck.

Iterative Cartesian product was by far the most important algorithmic optimization. It resolved the bottleneck brought about by full tuple materialization and made the runtime almost five hundred times faster compared to the base implementation for the first benchmark. In the second benchmark, the gains were even greater due to the fact that there were not as many output tuples.

Finally, the final optimization had a positive impact on *select* due to not having to copy every selected tuple into a brand new relation using `memcpy()`. The benefit can be seen clearly in benchmark one, but benchmark two shows that it is not as effective for queries with a lower selectivity hence why the runtime is slightly longer. The reason for this is that *distinct intersection* gets longer in these cases, as it now has to iterate over the full relation instead of over a relation containing only previously selected tuples. There was an attempt to modify *distinct intersection* to combat this, but this yielded worse results in *Cartesian product*, possibly due to an increased number of random accesses. Due to this, the most optimized version only contains the optimization made to *select*, but contains no changes in *distinct intersection*.

Results: Last Level Cache Misses.

Program	LLC cache misses
Base	2600182
Fully Optimized	260000

Table 1. LLC of the baseline and the most optimized version for the generic worst-case join.

In Tab. 1 it can be observed that last-level cache misses for the fully optimized implementation which includes the data layout change, iterative Cartesian product and modified select optimizations, dropped by an order of magnitude compared to the base implementation. This was a result of the data layout change which greatly improved spatial cache locality. Instead of polluting the caches with complete tuples containing fields irrelevant to the algorithm, the caches were only filled with the values of the join attributes in the optimized version. Removing full tuple materialization also played a big role in reducing the total number of last level cache misses since garbage tuples are not brought to the cache anymore when it's definitely known that this tuple will not be included in the results.

Results: Runtime Speedup. Fig. 3 illustrates the speedups achieved by all optimizations in comparison to the baseline. By far the most influential was the iterative Cartesian product which eliminated our biggest bottleneck. Data layout change also had a very positive impact on all subfunctions except *cartesian product*, due to these functions now gen-

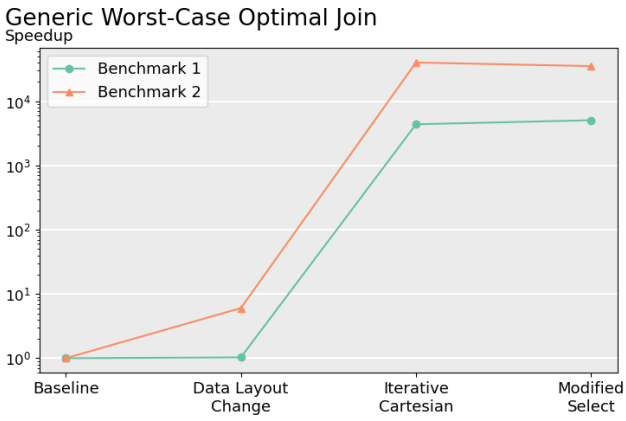

Fig. 3. Total speedup of the generic worst-case join for each optimization.

erating less last-level cache misses. Maximal speedup from the baseline was 4373x for the first benchmark and 40014x for the second benchmark. This difference was primarily due to the huge impact of the new cartesian product which is amplified by the differing number of output tuples.

4.2. Hash Trie Join

Experimental setup. For the hash trie join experiments, an Intel Core i7-10510U running on a Comet Lake microarchitecture was used. It has a 1.8 GHz base frequency and has cache sizes of 128KB for L1i, 128KB for L1d, 1MB for L2 and 8MB for L3. All experiments were conducted with Turbo Boost disabled and with the compiler flags *-O3*, *-fno-tree-vectorize*, *-ffast-math* and *-march=native* which proved to generate good results.
For the evaluation two representative TPC-H [8] benchmarks were used based on queries used in Freitag et al. [1, 10]. The first benchmark has four input relations, is joined on three attributes and has about two million output tuples. The second benchmark has six input relations, is joined on five attributes and has about one hundred twenty thousand output tuples.

Results: Subfunction Runtimes. The runtimes of the following subfunctions were measured:

- List Length (Build): Get the length of the linked list for which the hash trie is currently built.
- Allocate Hash Table (Build): Allocate hash table memory large enough to fit all elements of the linked list.
- Lookup Bucket (Build): Find the correct bucket in the hash table to place a given element in.
- Select Participants (Probe): Find relations that have a given join attribute.

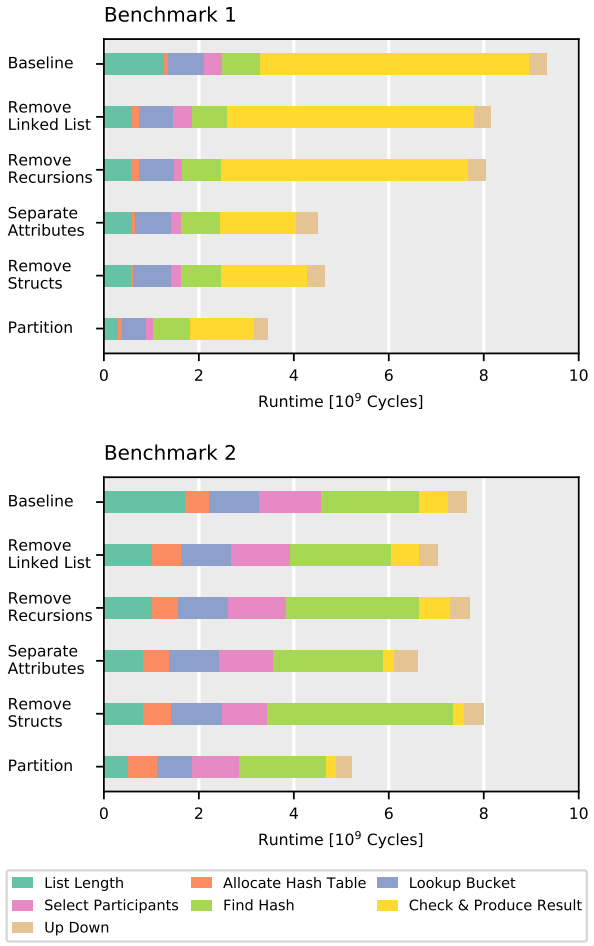

Fig. 4. Runtimes in cycles of the subfunctions of the build and probe phases for the baseline implementation and all optimizations.

- Find Hash (Probe): Find a bucket based on a hash value in a given hash trie.
- Check & Produce Result (Probe): Check join condition and produce result tuples given a linked list of tuples for each relation.
- Up Down (Probe): Move iterator up or down a level in the hash trie structure.

The runtimes of the subfunctions referred to in the following are shown in Fig. 4. In the baseline implementation the difference between the two benchmarks is clearly visible. In benchmark one which has a large output size, the *check & produce result* subfunction is the bottleneck. In benchmark two which has a large input size subfunctions operating directly on the input like *list length* or on the hash tries like *find hash* are dominant. Changing the data layout

to remove the linked list most notably reduced the runtime of *list length*. Removing the recursions had little impact on the subfunctions since the implementation for most of the subfunctions did not change much. As detailed in Sec. 3, the transition to an iterative approach necessitated the introduction of supplementary data structures. This modification noticeably impacted the runtime of *find hash*, leading to a visible increase in execution time. Separating the attributes of the input tuples and casting the join attributes to integers had an especially large impact on the runtime of the *check & produce result* subfunction which reduced by more than a factor of two for both benchmarks. Removing all structs and replacing them with arrays made the runtime of *find hash* much worse for the second benchmark and led to no significant speedup for any of the subfunctions. Partitioning the input tuples had a positive impact on the runtime of all subfunctions most notably on *list length* and *find hash*. Compared to the baseline, the final version including all optimizations except for removing all structs, decreased the runtime of all subfunctions except for *allocate hash table* where there was an insignificant increase. The most relevant subfunction speedups were achieved in the *list length* and *check & produce result* functions.

Results: Last Level Cache Misses. The primary objective of the optimizations was to minimize runtime. Although reducing the number of LLC misses was not the main focus, it was still an essential metric to consider due to its impact on runtime. In Fig. 5 the evolution of the number of LLC misses is depicted for both the build and the probe phase. Removing the linked list improved the cache locality of the build phase significantly because the tuples were now stored sequentially in one array. The probe phase, which does not access the tuples sequentially could not profit from that change. Instead the optimization led to more LLC misses in the probe phase because the next index replacing the pointer was now stored in a separate array. This made two array accesses necessary when accessing both the tuple and the index to the next tuple. Removing the recursions incurred more LLC misses for both algorithms due to the additional state variables that were needed to make the functions iterative, however this increase was not significant. In separate attributes the tuples that the algorithms operated on were shorter, since they only contained the join attributes instead of all attributes. The utilization of shorter tuples resulted in a reduction in the size of the working set, thereby leading to an anticipated decrease in LLC misses as a consequence of this optimization. This expectation was observed during the build phase. However, during the probe phase, the optimization unexpectedly resulted in an increase in LLC misses. The underlying cause of this increase remains uncertain despite our thorough investigation. All modifications implemented as part of this optimization should have a positive impact on cache locality, yet the specific reason for the observed increase in LLC misses could not be identified. Replacing all the structs with arrays led to more LLC misses in all cases. This was due to worse spatial locality because rather than storing variables belonging to a single node sequentially in a struct, this optimization stores the values across all nodes of a particular variable in an array. Pre-partitioning the input data aimed to minimize the occurrence of LLC misses. This optimization turned out to be successful as a decrease in number of LLC misses could be observed for all cases. For the probe phase the final version including all optimizations except for removing all structs had more LLC misses than the baseline especially in the output heavy benchmark one. For the build phase however the number of LLC misses reduced by about a factor of three for both benchmarks. The number of LLC misses for the complete hash trie join algorithm reduced for both benchmarks.

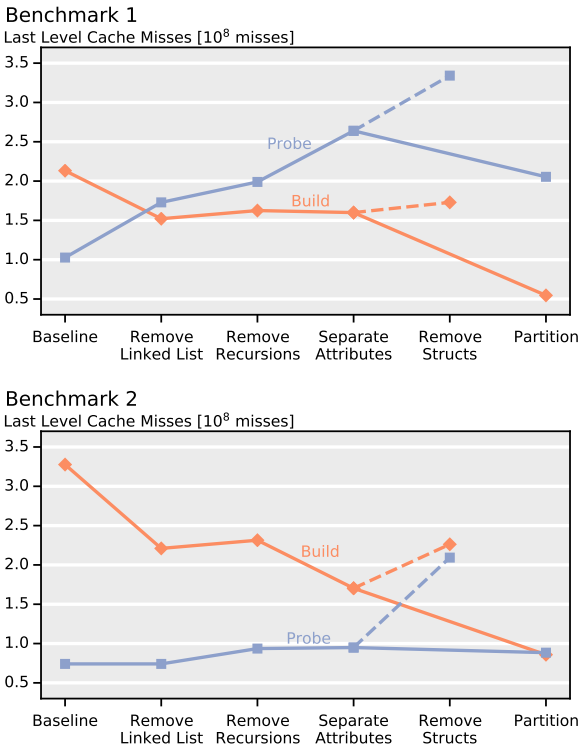

Fig. 5. Number of cache misses of the build and probe phases for the baseline implementation and all optimizations.

Results: Runtime Speedup. The total runtime of the build and probe functions decreased with almost every optimization as shown in Fig. 6. Removing the linked list had a negative impact on the probe function due to worse cache locality as explained above. Even though removing the recursion did not show much speedup in the subfunctions,

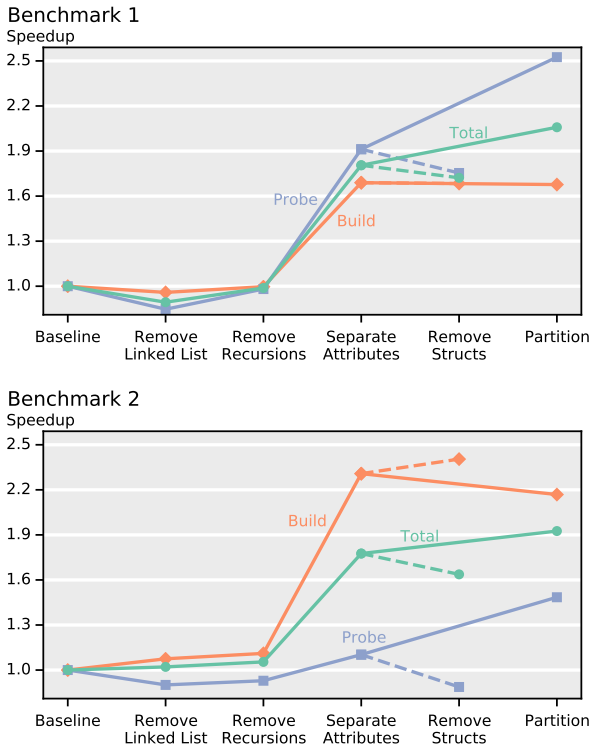

Fig. 6. Speedup of the runtime in cycles of the build and probe phases for the baseline implementation and all optimizations.

there was a noticeable speedup in the runtime of the complete function. So overall the gain of removing the overhead of the recursion was larger than the additional overhead introduced by making the functions iterative. Separating the join attributes was the optimization that had the largest impact on performance. In the output heavy benchmark one the probe function gains more from the optimization while in the input heavy benchmark two, the build function gains more. Removing all the structs made the runtime of probe much worse and the runtime of build only slightly better. Partitioning was then built on top of separate attributes and led to a large speedup for the probe phase and for the build phase without partitioning. In Fig. 6 the runtime of the partitioning was included in the build phase which resulted in a slightly worse runtime for build compared to the previous optimization. However the total speedup for build with partitioning and probe combined was significant for both benchmarks. From the baseline to the final version that included all optimizations except for removing all structs, there was a speedup of about two for the complete join for both benchmarks.

Results: Partition Tuning. As described in Sec. 3, the partitioning optimization involves sorting the input tuples until the partitions reach a specific size, adjustable as a tunable parameter. Smaller partition sizes generally yield improved locality for both the build and probe phases, although at the cost of longer partitioning times. Experimental results in Fig. 7 demonstrate that a full sort, which corresponds to a partition size of 1, is too time-consuming. Larger partition sizes reduce the time spent on partitioning and still offer positive impacts on both the probe and especially the build phase. Notably, partition sizes between 2^{16} and 2^{20} tuples outperform non-partitioned joins, with the most optimal performance observed at a partition size of 2^{19}.

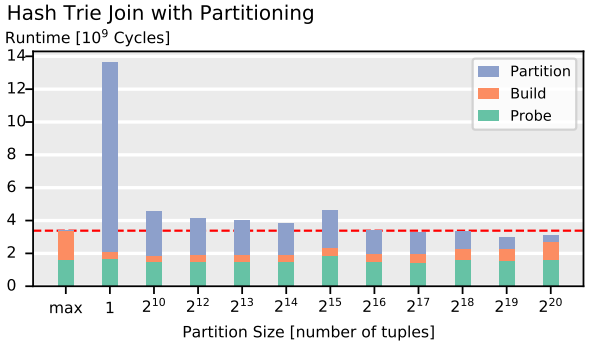

Fig. 7. Results from tuning the partition size parameter based on benchmark one. The two leftmost bars represent no partitioning (*max*) and a full sort (*1*).

5. CONCLUSIONS

This paper focused on optimizing two memory-intensive join algorithms. Due to their nature, traditional optimization techniques such as vectorization were not suitable. Instead, the optimizations that have been made centered around data layout and implementation details, aiming to reduce storage overhead and enhance cache locality. Most notably, by utilizing only the join attribute values rather than all tuple values, significant improvements were achieved in both join variants. These and further findings presented in this paper highlight the substantial impact that different data layouts can have on program performance and emphasize the considerable potential for improvement in straightforward algorithm implementations. In a next step the partitioning of the final implementation of the hash trie join algorithm could be further improved. While in our work a partitioning based on the quicksort algorithm was used, Freitag et al. [1] based their partitioning on radix join as proposed in [6] which has better asymptotic complexity.

6. CONTRIBUTIONS OF TEAM MEMBERS

Kyle. Implemented several sub-functions in the base implementation of Algorithm 1; initial cartesian product, distinct intersection and other supporting functions. Implemented the change of data layout optimization, as well as the iterative cartesian product optimization. Made the subfunction plots for Algorithm 1. Worked with Roman on all implementations, optimizations, visuals and presenttaion for Algorithm 1. Work on base implementation of Algorithm 4 was split with Roman.

Jón Gunnar. Implemented the base implementation of algorithm 2 and removed its recursion. Set up the validation infrastructure for the code base. Made the optimization to remove linked list for algorithms 2 & 3 as well as the partitioning optimization. Handled runtime and memory access measurements, and the code changes needed to run these measurements on large enough input sizes. Worked closely with Luca on all implementations, optimizations, visualizations and interpretations for algorithms 2 and 3.

Luca. Implemented the base implementation of algorithm 3 and removed its recursion. Made the separate attributes and remove all structs optimizations which affected both algorithms 2 and 3. Wrote the plotting scripts and made the plots for algorithms 2 and 3 for the presentation. Work on the base implementation of the data structures, timing utilities and the TPC-H setup used in both algorithms was split with Jon Gunnar. Worked closely with Jon Gunnar on all implementations, optimizations, visualizations and interpretations for algorithms 2 and 3.

Roman. Implemented selection, projection and union for algorithm 1. Removed recursion (from Cartesian product and main body) and implemented modified (cheap) version of select function for algorithm 1 optimizations. Made the speedup plot for algorithm 1. Worked with Kyle on implementations, optimizations, visualizations and presentation for algorithm 1 as well as base implementation for algorithm 4.

7. REFERENCES

[1] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann, "Adopting worst-case optimal joins in relational database systems," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 1891–1904, 2020.

[2] Austin Appleby, "Murmurhash3," https://github.com/aappleby/smhasher/, GitHub repository.

[3] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreaos, and Samuel Madden, *The Design and Implementation of Modern Column-Oriented Database Systems*, 2013.

[4] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem, "Column-stores vs. row-stores: How different are they really?," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2008, SIGMOD '08, p. 967–980, Association for Computing Machinery.

[5] Stratos Idreos, F. Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten, "Monetdb: Two decades of research in column-oriented database architectures," *IEEE Data Eng. Bull.*, vol. 35, 01 2012.

[6] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 362–373.

[7] Amine Mhedhbi and Semih Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1692–1704, 2019.

[8] Transaction Performance Council, "TPC-H specification," http://www.tpc.org.

[9] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann, "Combining worst-case optimal and traditional binary join processing," 2020.

[10] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann, "Experimental queries," https://github.com/freitmi/queries-vldb2020, 2020, GitHub repository.