WORST-CASE OPTIMAL JOINS

Alexander Kvamme, Xianzhe Ma, Noah Schmid, Cagin Tanir

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Worst-case optimal joins are favourable over binary joins in queries with large intermediate results [1]. Freitag et al. proposed an implementation with hash-based data structures that can be built efficiently on-the-fly. We implement their proposal and extend it with our approach on their underspecified result builder. We apply several optimizations from the Advanced Systems Lab (ASL) course and achieve an 11.6x speedup over the baseline implementation.

1. INTRODUCTION

Motivation. Today, we depend on a lot of services that require a relational database. Let it be for analytical processing or transaction processing, the queries overwhelmingly consist of joins. Since they are frequently used, binary joins have a long history of fine-tuning and optimization [1]. However, they are suboptimal in certain cases [2, 3, 4]. Explicitly, the intermediate join results could grow much larger than the final query result [5]. Worst-case optimal join algorithms provide asymptotically better runtime in this type of queries [5, 6]. Nevertheless, due to their shortcomings, they are not widely adopted. For instance, they require indices on all permutations of the attributes that take part in the join. This, in turn, causes storage and maintenance issues [7]. Furthermore, a traditional relational database management system (RDBMS) requires operations such as inserts and updates. Yet the previous worst-case optimal systems like EmptyHeaded or LevelHeaded are read-optimized systems which make these operations potentially orders of magnitudes slower [7, 8]. It is important to consider that without the presence of growing intermediate results, multi-way joins are usually slower than binary joins [9]. In the paper *Adopting Worst-Case Optimal Joins in Relational Database Systems*, Freitag et al. propose a system where multi-way joins are only used in case there is a benefit. Additionally, performant index structures are built on-the-fly and without the requirement to persist to disk [1].

Contribution. We implemented the Worst-Case Optimal Join Algorithm proposed by Freitag et al. [1]. We further have proposed and applied multiple optimizations that we have learnt during the ASL course. We made adjustments to the data structure representing a table, i.e., a novel *hash trie* proposed by Freitag et al. Additionally, we implemented an efficient way to combine output tuples into a single result tuple. This part is omitted in the paper and is only referred by one line of pseudocode. What is more, our implementation paved the way for some other optimizations including single instruction, multiple data (SIMD) vector instructions. The optimizations are further explained with examples in Section 3.

Related work. Ngo et al. are one of the first to introduce a worst-case optimal join algorithm [10, 5]. It constitutes the foundation of other worst-case optimal join algorithms including the work of Freitag et al. [1]. In LogicBlox system, Veldhuizen's Leapfrog Triejoin algorithm is used but it requires persistent precomputed index structures [11]. Curiously, Veldhuizen also suggests using the trie index structures as nested hash tables [11]. Freitag et al. states that, to the best of their knowledge, they are the first ones to come up with a practical solution with acceptable performance. LevelHeaded and EmptyHeaded both require expensive index precomputation and crucially only work with static data [7, 8]. Freitag et al. address these practical challenges. As part of the ASL project, we built upon their suggested algorithms and applied various optimizations on top.

2. BACKGROUND ON THE ALGORITHM/APPLICATION

Throughout the paper *Adopting Worst-Case Optimal Joins in Relational Database Systems*, four algorithms are presented.

Algorithm 1 shows a generic worst-case optimal join. Freitag et al. pointed out the suboptimal performance of this approach (e.g., additional unnecessary comparisons) and therefore proposed a modified version that operates exclusively on the hash values of join keys, i.e., enumerate all tuples for which the hash values instead of the actual values of the join keys match, which they referred to as Algorithm 3 in the paper [1]. Algorithm 2 describes the building phase of the hash trie for each table which is then used in Algorithm

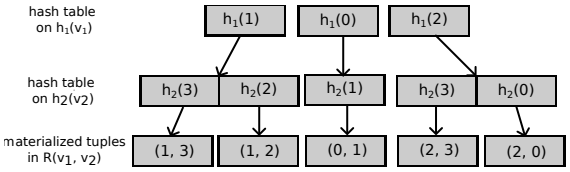


Fig. 1: Hash trie data structure for the table containing the tuples (1, 3), (1, 2), (0, 1), (2, 3), (2, 0).

3. Algorithm 4 is solely to determine whether multi-way join is more beneficial over a regular binary join and is not considered in this project. Below, we detail the Algorithms 2 and 3 which are the essential parts of the multi-way join. Noting that, what we refer below as Algorithm 3.5 is in fact part of the Algorithm 3 but we decide to emphasize its importance and the opportunities it provides for optimization.

Algorithm 2. Algorithm 2 denotes the *build phase* for the hash trie, which is a novel data structure described in the paper [1] and used to represent a table. A hash trie consists of a multi-level hash table, one level per join attribute. Algorithm 2 recursively builds the hash trie structure where entries of a hash table point to the hash table of the next level and the entries of the last hash table point to the materialized tuples. The hash trie data structure is built for each relation that takes part in the join query. Crucially, the hash trie data structure possesses iterators that allow Algorithm 3 to be carried out, e.g., moving the iterator up and down in the trie, to the next occupied bucket, to the bucket with the hash in question etc.

Algorithm 3. Algorithm 3 denotes the *probe phase* and its goal is to enumerate the joined tuples. It uses one hash trie data structure as created in Algorithm 2 for each table participating in the join and traverses these structures using iterators. The algorithm ensures that no unnecessary tuples are enumerated and therefore eliminates the overhead of large intermediate joins. To achieve this, it leverages the constant-time hash table lookup to find matching attributes and only traverses the smallest hash table for each attribute.

Algorithm 3.5. Algorithm 3.5 denotes the process of appending rows to the result table once we find joined tuples from the iterators of the joined tables. If we naively append the tuples we find from Algorithm 3, each attribute we join on will appear at least twice. This is however not the result table should look like, therefore we need to remove the duplicates which constitutes Algorithm 3.5. Since this procedure is not detailed in the paper, we came up with a straightforward way to implement it which later turned out to be a bottleneck for Algorithm 3.

Cost Analysis. The worst-case optimal join algorithm consists of iterators, comparisons, and data movements rather than computations. Since no computation is involved, we cannot use popular metrics such as *flops/sec* learned in class. On the other hand, since this algorithm consists of all kinds of the data reads and writes which heavily skew the algorithm to the memory bound, throughput is an important metric that we use in our profiling measurements alongside the number of cycles. They provide insights on which part of the implementation requires optimizations. The build phase, i.e., Algorithm 2, has time and space complexity in $O(n \cdot |R_j|)$ where $|R_j|$ are the relations that take part in the multi-way join and n the number of relations joined on [1]. The probe phase, i.e., Algorithm 3, has time complexity in $O(nm \prod_{E_j \in \mathcal{E}} |H(I_j)|^{x_j})$ and space complexity in $O(nm)$. Here, $x_j \in X$ where $X = \{x_1, ..., x_m\}$, contains the edges in the relations we are joining (and consequently $m = |X|$). $H(I_j)$ is the set of join keys present for some hash trie iterator $I_j \in I$.

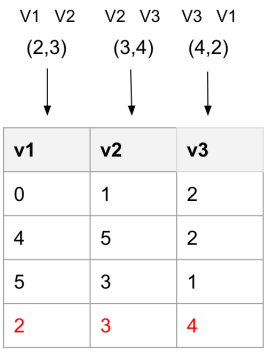
3. YOUR PROPOSED METHOD

Profiling Tools. We used various profiling tools and some methods in order to discover different parts of the baseline implementation that require optimization and can be improved:

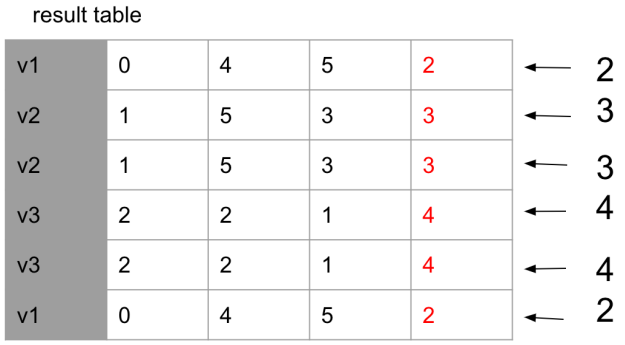
- `Linux Perf` [12] is a lightweight profiling tool to understand the performance of a program. It can show various aspects of program performance such as cache misses, page faults, and branch misses. `Perf` is already included in the Linux kernel.
- `RDTSC`. The instruction `RDTSC` is used to count the runtime of the program in CPU cycles. On x86 processors since `Pentium` there is a Time Stamp Counter (TSC) register that counts the number of CPU cycles. `RDTSC` returns the TSC in EDX:EAX [13].
- Breakdown Analysis. In order to find out which part of the program is the bottleneck for speedup, it is crucial to break the runtime down into several segments and analyze them separately. Therefore besides measuring end-to-end runtime, we also measure dedicated runtimes of Algorithm 2 and Algorithm 3. To further break the runtime down on a finer granularity, we comment out some parts of the code and re-do measurements. Through the difference between two measurements we can understand the runtime of commented part of code.

Following the data structures and the algorithms provided in the paper, we came up with a baseline version. Below, we list our iterative approach to optimize the baseline implementation. We will go over the observed results for each optimization in chapter 4.

Baseline / Optimization 0: Chain the Initialized Entries. While we implemented the original version of algorithms described by the paper, we already noticed an im-



(a) Before: deduplicate values on-the-fly.



(b) Now: make the result table in column order, first naively append rows to the table and later only copy one row per attribute.

Fig. 2: How rows are appended to the result table before and after Optimization 4.

portant optimization. Since this optimization is on the algorithm side and greatly simplifies the code logic, we decided to treat the version including this optimization as the baseline. This optimization utilizes a *linked hashmap* data structure [14] which chains the initialized entries in the hash table at each layer. The iterators in Algorithm 3 often need to find the next initialized entry in the hash table. Now instead of iterating over all the hash table entries to find initialized ones, the iterator can directly jump to the next initialized entry from the current one.

Optimization 1: Optimize Hash Table Format. Instead of allocating an array of `HashTrieEntry` objects which all have to be initialized upon the allocation of the array, we allocate an array of pointers to `HashTrieEntry` objects. Therefore, we only initialize the entries we actually need which brings two advantages. Firstly, we reduce the size of the hash table which improves cache locality and decreases total memory usage; secondly, allocating the hash table array takes less time since we don't need to initialize the hash entries.

```
// previously
hash_table =
new HashTrieEntry[allocated_size_arg];

// now
hash_table = (HashTrieEntry**)calloc(
    allocated_size_arg, sizeof(HashTrieEntry*));
```

Listing 1: Optimization 1 pseudocode

Optimization 2: Optimize Tuple List Length Calculation. Before this optimization, we were naively traversing the `TupleList` when we were calling the length function to determine the length of the tuple list. With the optimization, we keep track of the length with a member variable every time an insertion occurs. We simply return this variable.

Optimization 3: Optimize Tuple Builder. This optimization is from the initial version of our result tuple builder. Since we know the amount of data needed beforehand, we reserve the memory for `data` to avoid re-allocations. Also we use `std:copy` instead of `push_back` to avoid unnecessary checks.

```
// previously
int originalSize = data.size();
for (int j = 0; j < n; j++) {
    for(int i = 0; i < originalSize; i++) {
        data.push_back(data[i]);
    }
}

// now
int originalSize = data.size();
int newSize = originalSize * (n+1);

data.reserve(newSize);
for (int i = 1; i <= n; i++) {
    std::copy(data.begin(), data.begin() +
    originalSize, std::back_inserter(data));
}
```

Listing 2: Optimization 3 pseudocode

Optimization 4: Optimize Algorithm 3.5. As Figure 2 shows, before this optimization, every time we appended a new tuple to the result table, we removed the duplicate attributes on the fly (Figure 2a). For example, if we combine matching tuples from relations $R_1(v_1, v_2)$ and $R_2(v_2, v_3)$, we will have duplicated values for attribute v_2. The same logic to decide which attribute to keep or drop was executed upon each tuple insert. Instead, we store the result table in column-order and naively append the rows with duplication (Figure 2b). Once we have the entire table, then we "compact" the table by only choosing one column for each attribute. We now only need to do simple column-to-column copies with a single condition that checks whether we have already copied a column for the current attribute. Since the table is in column-order, the copy process benefits from *spatial locality* and gives way to apply SIMD instructions which we explain under Optimization 9.

Optimization 5: Optimize Hash Table Lookup. In the old implementation, when we were looking for a hash table entry in the table, we were doing a costly modulo operation for each iteration. After observing that this function takes a lot of cycles, we changed the approach to instead use two loops. In the optimized version, we first loop from *start* → *allocated_size* and if we have not found the entry, we proceed to loop from *0* → *start*. This way, we get rid of the expensive modulo operation which results in a huge speedup.

```
// previously
do {
    index = (index+1) % allocated_size;
    // processing logic
    [...]
} while (start != index);

// now
for (int i = start; i < allocated_size; i++) {
    // processing logic
    [...]
}
for (int i = 0; i<start; i++) {
    // processing logic
    [...]
}
```

Listing 3: Optimization 5 pseudocode

Optimization 6: Optimize Hash Trie Tuple Insertion. This optimization focuses on optimizing how tuples are inserted to the hash trie structure. When inserting values, the code used to increment a counter and do modulo for each iteration in case the counter value exceeded the size of the hash table. This was done until a free space in the hash table was found. To optimize, we replace the expensive modulo operation in each iteration with a much cheaper *if* statement. Then, if the incremented counter value is equal to the size of the hash table, the counter value is set to 0 to *wrap-around* to the first value in the table.

```
// previously
while(hash_table[index]->hash != hash) {
    index++;
    index %= allocated_size;

    [...]
}

// now
while(hash_table[index]->hash != hash) {
    index++;
    if(index == allocated_size)
        index=0;

    [...]
}
```

Listing 4: Optimization 6 pseudocode

Optimization 7: Optimize Hash Table Format. This optimization is two-fold. First, we removed the hash function, and mapped the values in the relations directly to the hash-table. We could do this because the relation values that we are joining are already integer values. Secondly, as shown in Figure 3, instead of using `uint_64` values to store the hashed values in the hash-table, we opted to use `int` values instead. These `int` values only need 4 bytes, as opposed to 8 bytes for `uint_64`. As the values in the relation were only 4 bytes large, using `uint_64` only took up unnecessary space. This change lead to a more compact hash-table, which helps to increase cache locality.

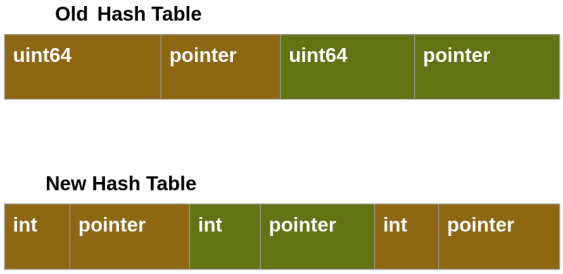


Fig. 3: Hash table format before and after Optimization 7.

Optimization 8: Remove Modulo when Calculating Linear Probing Index. In the hash table, when calculating which index to start the linear probing on, we used to do an unnecessary modulo operation. This modulo could safely be removed, as the changes in 3 ensured that only valid index values would be iterated over.

Optimization 9: SIMD instructions. Due to optimization 4, where we changed Algorithm 3.5 to operate in a column-order fashion, we now need to copy adjacent integers one by one. Since the contiguous integers along the column are contiguous in memory due to the column-order, we can use SIMD instructions to copy multiple integers per iteration. Pseudocode is shown below.

```
int row_index = 0;
for (; row_index < num_rows-7; row_index += 8) {
    __m256i t=__mm256_loadu_si256(col+row_index);
    _mm256_storeu_si256(base+row_index, t);
}

// the remainder of num_rows devided by 8
for(; row_index < num_rows; row_index++) {
    base[row_index] = col[row_index];
}
```

Listing 5: Optimization 9 pseudocode

4. EXPERIMENTAL RESULTS

We synthesized a dataset derived from IMDB dataset. The full IMDB dataset is too large to reasonably run benchmarks with, which is why we ran the following queries on reduced

tables with 1000 and 45,000 rows. We joined the tables on triangles (e.g., on attributes v1, v2, v3 where tables have relations $v1 \rightarrow v2, v2 \rightarrow v3, v3 \rightarrow v1$). This scheme provides us with a query with growing intermediate results, which is the problem multi-way join algorithm tries to overcome.

Experimental setup. The computer we did our measurements on has the following specs: 11th Gen Intel(R) Core(TM) i7-1165G7 (Tiger Lake) @2.80GHz L1-I: 32KB, L1-D: 48KB, L2: 1.25MB, L3:12MB and the compiler version is GCC (9.4.0).

Compiler flags. We experimented with compiler flags, however ffast-math, ftree-loop-vectorize, -m64 did not provide consistently better results so we opted these out. In the end, we used *-march=native* and *-O3*.

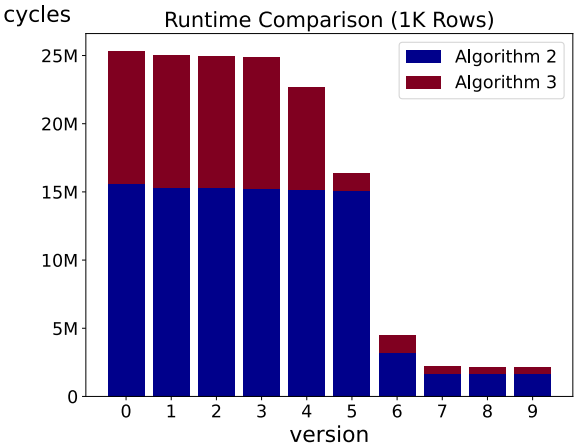


Fig. 4: Combined runtime of Algorithm 2 and 3 (including Algorithm 3.5) after each optimization, and breakdown of runtime per algorithm. Sizes of individual relations are 1000 rows each.

Results. Figure 4 shows that each optimization had a measurable impact on the Worst Case Optimal Join algorithm, though optimizations 4, 5, 6 and 7 had the biggest impact. We first discuss the optimizations that caused the most significant increases in performance, and why they helped. After that we outline the results from the rest of the optimizations, and analyze the performance between the base and optimized implementation with varying input sizes.

Optimization 4. This optimization aimed to improve spatial locality when creating the joined tuples. We were hoping that storing the intermediate data in a column-ordered table, instead of row-ordered, would increase spatial locality in the cache when joining the final result. Empirical data support this hypothesis, as data from `perf` show that this change reduced the number of LLC load misses by approximately 25%. In the 1000 row case we went from 4293 to 3348 misses, which was a $100 \cdot (1 - \frac{3348}{4293}) \approx 23.3\%$ LLC

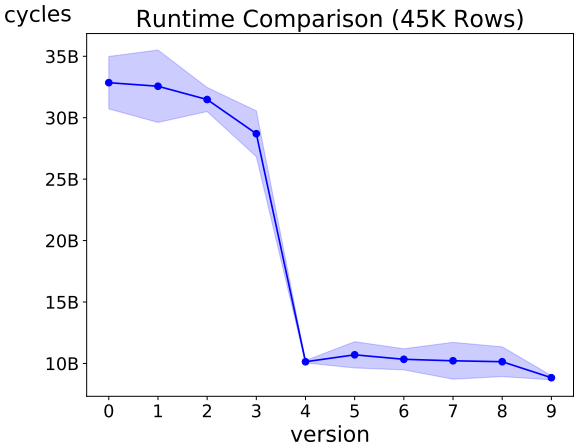


Fig. 5: End to end runtime of the worst-case optimal join program from Optimization 0 (Baseline) to Optimization 9 with tables having 45K rows.

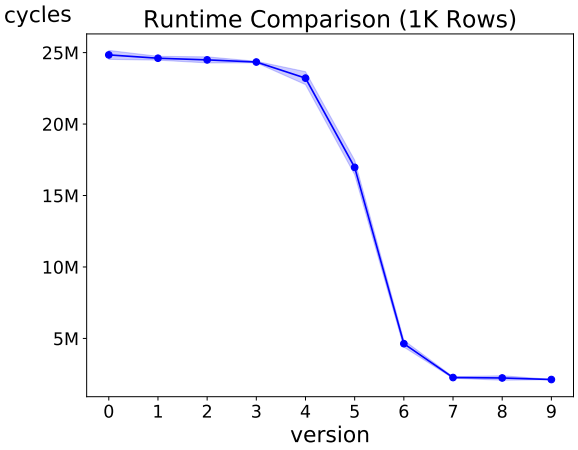


Fig. 6: End to end runtime of the worst-case optimal join program from Optimization 0 (Baseline) to Optimization 9 with tables having 1K rows.

load miss decrease. Similarly we saw an approx. 10% decrease in total cache read misses. This in turn lead to an approx. 28% speedup for Algorithm 3.

Optimization 5 and 6. These optimizations drastically improved performance through removing modulo operations. Optimization 5 yielded an approx. $7x$ speedup for Algorithm 3, which implied that these modulo operations during hash-trie lookup occupied the majority of the algorithm's running time. Similarly in optimization 6, moving away from modulo operations gave an approx. $3,5x$ speedup in Algorithm 2. Hence the modulo operations during hash-trie insertion took the majority of Algorithm 2's runtime.

According to Agner's tables [15], for the Tiger Lake mi-

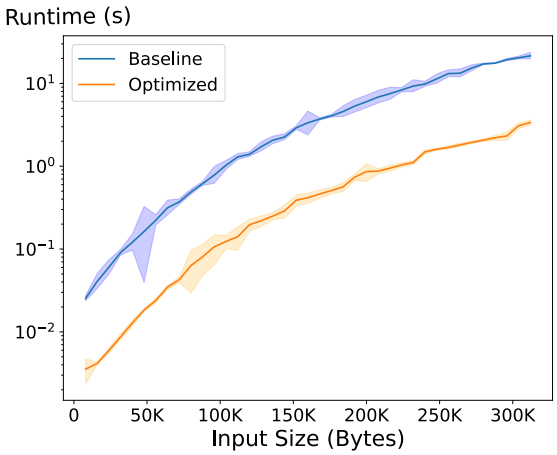


Fig. 7: Combined runtime, in seconds, of both Algorithm 2 and 3, where the runtime is related to the input size in bytes. The error bounds indicate the 99.7% running time. The y-axis uses a log-scale highlighting the 99.7% running time even for small time intervals.

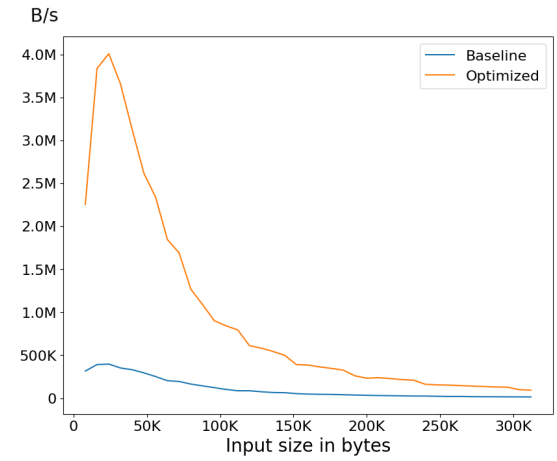


Fig. 8: For both the baseline and optimized implementation, the graph shows how many bytes are processed per second, relative to the input size.

croarchitecture used for benchmarking, a `div` instruction has latency of 19 cycles, and gap of 6 cycles. We care about `div`, because in order to do a `modulo` operation we actually need to complete a `div` instruction behind the scene (as the modulo is the remainder of a division). By moving from `modulo` to `if`-statements, we only do a `cmp` instruction instead of `div`. Comparing to `div`, Agner's tables [15] claim that a `cmp` instruction only has latency of 1 cycle, and gap of 0.25 cycles.

Each entry that is joined on, needs to be added to the hash table once. Also, each value that is joined on needs to be looked up at least once. A very conservative estimate on the number of hash table insertions and lookups that assumes neither hash- nor index-collisions is therefore $m \cdot n$ insertions and $m \cdot n$ lookup operations, where n is the number of relations joined on, and m the number of rows in each relation. Empirical data, however, show that in practice the number of lookup operations is approximately $m \cdot n \cdot 1.858$, and for insertions $m \cdot n \cdot 1.013$. This analysis helps explain why the speedup in Algorithm 3 (where we do lookups) is about twice to the speedup to that of Algorithm 2 (where we do insertions).

Optimization 7. Since we were dealing with integers all along, there was no benefit in hashing these values before inserting them in the hash table. Therefore we omitted the hash function in this optimization, which allowed us to use `int` instead of `uint64_t` in the hash table. By doing this, the size of a hash entry got reduced from 16 bytes (8 bytes for the hash value and 8 bytes for the pointer) to 12 bytes. This not only means that more hash table entries can fit in cache ($1 - \frac{12}{16} = 25\%$ more) at once, but also that less time is used to move data around. Although a hash table is generally not optimal with regards to cache locality, spatial locality will still be beneficial during the linear probing phase when we iterate over adjacent values whenever index collisions occur. In this case, LLC load misses decreased by approx. 20%. This change lead to an approx. 92% speedup for Algorithm 2, and 100% speedup for Algorithm 3.

Optimization 8.

Optimization 9. The SIMD operations improved the runtime by approx. 13% for the benchmark with 45K rows, but were less noticeable for fewer rows. In the case of 1K rows, it only yielded an approx. 5% speedup. This is probably because for smaller input sizes there are not enough values being copied such that it could make a big impact. LLC load misses in the 1K rows case got decreased by approx. 10% from 3317 to 3075.

Comparing the baseline implementation to the final implementation. Figure 7 shows how the runtime grows together with increasing input size. The plot shows that the runtime of the baseline grows exponentially faster compared to the runtime of the optimized version as the input size in bytes grows.

The input is modeled in a way that the number of output results grows together with the input size. More specifically, the number of joined results in the final result is $1250 \cdot (\frac{x}{6000})^3$, where x is the input size in bytes. Hence, the output size grows cubically as the input size increases linearly. Relative to the input size, Figure 8 shows the performance of the baseline implementation peaks at the input size of $15k - 25k$ bytes with $400k - 450k$ bytes/second. The optimized solution's peak is much more pronounced at the input size of around $35k$ bytes with $4M$ bytes/second.

Comparison of runtime with input tables of 1K and

45K rows. We ran each version from baseline to optimization 9 with tables of 1K rows in Figure 6 and with tables of 45K rows in Figure 5. Each dataset is designed to yield a lot of intermediate result. The comparison between the two provides great insights. For example, Optimization 5 and 6, i.e., removing modulo operations in insert and lookup, bring the largest speedup with input size of 1K cycles. Also, the increase in performance after *SIMD operations* is barely visible in Figure 6. This can be explained by a relatively small size of results. On the other hand, in Figure 5 where we have an input with tables of 45K rows, we see the immense benefits of optimizing the result tuple builder. In this case, the result is much larger and the process of removing the duplicates from the result table, i.e., copying column-ordered rows with SIMD instructions is a more efficient solution than to *deduplicate* on-the-fly. For this reason, we notice the huge performance increase at Optimization 4. In the end, the effect of different optimizations on different input sizes are in line with our expectations.

5. CONCLUSIONS

SQL join is an important type of query in an RDBMS that occurs frequently and consumes a lot of time. In this work, we implemented the worst-case optimal joins algorithm proposed by Freitag et al. [1]. We applied optimizations learned in the Advanced Systems Lab course and iteratively improved the performance.

Unlike matrix computation, Fourier transformation, or other problems presented in the class, the problem of worst-case optimal joins provide less opportunities for optimizations because it consists of data structure traversals, integer comparisons and data movements. It involves almost no computation. Also, the nature of hash trie and the recursive algorithms present less temporal and spatial locality. Nevertheless we managed to apply the optimizations in various locations and obtained an 11.6x speedup of the program in CPU cycles.

6. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

Here are contributions of team members.

Alexander De Battista Kvamme. redesigned the hash maps implementations, did debugging on the base implementation (fixing memory leaks, etc.), and set up an automated testing and benchmarking framework. In optimization 5 [3] Alexander improved how hash trie lookups were implemented. He also improved on hash trie insertions, which among other things included optimization 6 [3]. This optimization made hash trie insertions faster.

Cagin Tanir. Debugging on base implementation. Implemented optimization 3, removed some procedure calls and made data structure changes to avoid library calls. Worked on benchmarking.

Noah Schmid. Implemented baseline algorithms 2 and 3 and optimizations 1, 2, 7, 8, 9.

Xianzhe Ma. implemented the basic data structures for algorithms 2 and 3 and optimizations 0 and 4; also patched all memory leaks.

7. REFERENCES

[1] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann, "Adopting worst-case optimal joins in relational database systems," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 1891–1904, 2020.

[2] Ron Avnur and Joseph M. Hellerstein, "Eddies: Continuously adaptive query processing," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, Eds. 2000, pp. 261–272, ACM.

[3] Goetz Graefe, Ross Bunker, and Shaun Cooper, "Hash joins and hash teams in microsoft SQL server," in *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, Ashish Gupta, Oded Shmueli, and Jennifer Widom, Eds. 1998, pp. 86–97, Morgan Kaufmann.

[4] Alfons Kemper, Donald Kossmann, and Christian Wiesner, "Generalised hash teams for join and group-by," in *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, Eds. 1999, pp. 30–41, Morgan Kaufmann.

[5] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra, "Worst-case optimal join algorithms," *J. ACM*, vol. 65, no. 3, pp. 16:1–16:40, 2018.

[6] Albert Atserias, Martin Grohe, and Dániel Marx, "Size bounds and query plans for relational joins," *SIAM J. Comput.*, vol. 42, no. 4, pp. 1737–1767, 2013.

[7] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré, "Emptyheaded: A relational engine for graph processing," *ACM Trans. Database Syst.*, vol. 42, no. 4, pp. 20:1–20:44, 2017.

[8] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré, "Levelheaded: A unified engine for business intelligence and linear algebra querying," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. 2018, pp. 449–460, IEEE Computer Society.

[9] 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.

[10] Hung Q. Ngo, Dung T. Nguyen, Christopher Ré, and Atri Rudra, "Beyond worst-case analysis for joins with minesweeper," in *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, Richard Hull and Martin Grohe, Eds. 2014, pp. 234–245, ACM.

[11] Todd L. Veldhuizen, "Leapfrog triejoin: a worst-case optimal join algorithm," *CoRR*, vol. abs/1210.0481, 2012.

[12] "Perf Wiki," `https://perf.wiki.kernel.org/index.php/Main_Page`, last visited 20 June, 2023.

[13] "Time Stamp Counter," `https://en.wikipedia.org/wiki/Time_Stamp_Counter`, last visited 20 June, 2023.

[14] "Java LinkedHashMap," `https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html`, last visited 20 June, 2023.

[15] Agner Fog, *4. Instruction tables*, Technical University of Denmark, 2022.