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 favorable over binary joins in queries with large intermediate results. Freitag et al. proposed an implementation with hash-based data structures that can be built efficiently on the fly [1]. We implement and extend their proposal with our approach to combining the output tuples from their algorithm. We apply several optimizations from the Advanced Systems Lab (ASL) course and achieve an 11.6x speedup over the baseline implementation.

1. INTRODUCTION

Motivation. Relational databases are omnipresent in today's systems. Let it be for analytical or transaction processing; the queries these databases process overwhelmingly consist of joins. 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 have yet to be widely adopted. For instance, they require indices on all permutations of the attributes that take part in the join. The need for indices, in turn, causes storage and maintenance issues [7]. Furthermore, a traditional relational database management system (RDBMS) requires operations such as inserts and updates. However, 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 essential 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 implement the Worst-Case Optimal Join Algorithm proposed by Freitag et al. [1]. We further propose and apply multiple optimizations we learned during the ASL course. We adjust the data structure representing a table, i.e., a novel *hash trie* proposed by Freitag et al. Additionally, we implement an efficient way to combine output tuples into a single result tuple. Freitag et al. do not explain this part of the algorithm in detail in their paper and only mention it in one line of pseudocode. Moreover, our implementation of combining output tuples paves the way for other optimizations, including single instruction multiple data (SIMD) vector instructions. We further explain the optimizations with examples in Section 3. We only support integer values in our database to simplify our implementation.

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]. The LogicBlox system uses Veldhuizen's Leapfrog Triejoin algorithm, but it requires persistent precomputed index structures [11]. Veldhuizen also suggests using the trie index structures as nested hash tables [11]. Freitag et al. state that, to the best of their knowledge, they are the first 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 implement their suggested algorithms and apply various optimizations.

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). They, therefore, proposed a modified version that operates exclusively on the hash values of join keys, i.e., enumerates 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

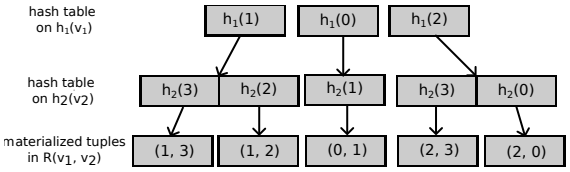


Fig. 1: Hash trie data structure for the table containing the tuples (1, 3), (1, 2), (0, 1), (2, 3), (2, 0).

of the hash trie for each table which we later use in Algorithm 3. Algorithm 4 is solely to determine whether a multi-way join is more beneficial over a regular binary join and is out of scope for this project. Below, we detail Algorithms 2 and 3, which are the essential parts of the multi-way join. Note that what we refer to below as Algorithm 3.5 is part of 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 next level's hash table, and the entries of the last hash table point to the materialized tuples. We will need one hash trie data structure for each relation taking 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 what the result table should look like. Therefore, we need to remove the duplicates, which is the goal of Algorithm 3.5. Since there is no description of this procedure in the paper, we came up with a straightforward way to implement it, which later turned out to be a bottleneck for Algorithm 3 on big input sizes.

Cost Analysis. The worst-case optimal join algorithm consists of iterators, comparisons, and data movements rather than computations. Since not much computation is involved, we cannot use popular metrics such as *flops/sec* learned in class. Since this algorithm primarily consists of all kinds of data reads and writes, which heavily skews the algorithm to be memory-bound, throughput and the total runtime in cycles are the main metrics we use in our profiling measurements. These metrics provide insights into 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

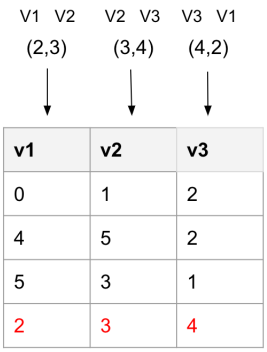
Profiling Tools. We used various profiling tools and some methods in order to discover different parts of the baseline implementation that require optimization and can be improved:

- `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 program's runtime 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. We can then understand the runtime of the commented part of the code through the difference between the two measurements.

Following the data structures and the algorithms provided in the paper, we developed 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.



v1	v2	v3
0	1	2
4	5	2
5	3	1
2	3	4

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

result table

v1	0	4	5	2 ← 2
v2	1	5	3	3 ← 3
v2	1	5	3	3 ← 3
v3	2	2	1	4 ← 4
v3	2	2	1	4 ← 4
v1	0	4	5	2 ← 2

(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.

Baseline / Optimization 0: Chain the Initialized Entries. While implementing the original version of algorithm 3 described by the paper, we already noticed an important 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. Instead of iterating over all the hash table entries to find initialized ones, the iterator can now directly jump to the next initialized entry from the current one using the linked list.

Optimization 1: Optimize Hash Table Format. Instead of allocating an array of `HashTrieEntry` objects which we all have to initialize upon allocating the array, we allocate an array of pointers to `HashTrieEntry` objects. Therefore, we only initialize the entries we need, which brings two advantages. Firstly, we reduce the hash table size, which improves cache locality and decreases total memory usage; secondly, allocating the hash table array takes less time since we do not 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 which had $O(n)$ complexity for a list of length n. Instead, we keep track of the length with a member variable every time an insertion occurs. When calling the length function, we simply return this variable.

Optimization 3: Optimize Memory Allocation in Tuple Builder. This optimization happens in the tuple builder (Algorithm 3.5). Since we know the amount of data needed beforehand, we reserve the memory for `data` to avoid reallocations in the vector data structure. Also, we use `std::copy` instead of `push_back` to avoid unnecessary bound 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 attributes to keep or drop was executed upon each tuple insert. The new version stores the result table in column order and naively appends the rows with duplication (Figure 2b). Once we have the entire table, we "compact" the table by only choosing one column for each attribute. We now only need simple column-to-column copies with a single condition that checks whether

we have already copied a column for the current attribute to get the final result table without duplicates. 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 3.

Optimization 5: Optimize Hash Table Lookup. In the old implementation, when we were looking for a hash table entry, we did a costly modulo operation for each iteration. After observing that this function takes many cycles, we changed the approach to use two loops instead. In the optimized version, we first loop from *start* $\rightarrow$ *allocated_size*, and if we have not found the entry, we proceed to loop from *0* $\rightarrow$ *start*. This way, we eliminate the expensive modulo operation, resulting 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 we insert tuples to the hash trie structure. Previously, when inserting values, we iterated over the hash table, starting at the index derived from the hash value by incrementing the index modulo the hash table size until we arrived at a free space in the hash table. In the new version, we replace the expensive modulo operation in each iteration with a much cheaper *if* statement that checks whether the counter value overflows the hash table size. If so, it sets the counter value back to 0.

```
// 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 directly mapped the values in the relations to the hash table. We could do this because the relation values we are joining are already integers. 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 leads to a more compact hash table, which helps to increase cache locality.

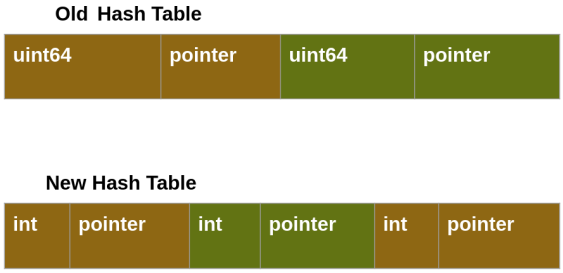


Fig. 3: Hash table format before and after Optimization 7.

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. We could safely remove this modulo operation, as the changes in 3 ensured that we only iterate over valid indexes.

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 individually. Since the contiguous integers along the column are contiguous in memory due to the column order way of storing the table, 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 the official IMDB dataset. The full IMDB dataset is too large to run benchmarks reasonably, so 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 $v1 \to v2, v2 \to v3, v3 \to v1$). This scheme provides us with a query with growing intermediate results, which is the problem the 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 different 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*.

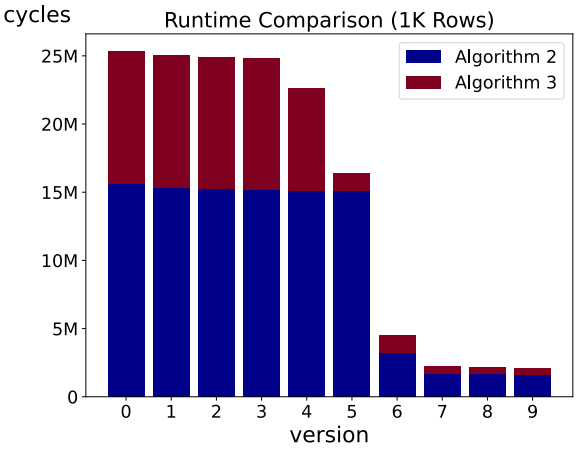


Fig. 4: Combined runtime of Algorithm 2 and 3 (including Algorithm 3.5) after each optimization, and breakdown of runtime per algorithm. Sizes of individual relations are 1000 rows each.

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 most significant impact. We first discuss the optimizations that caused the most significant performance increases 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 1. This optimization aimed to improve spatial locality in the hash table containing hash tries, and reduce the amount of unused memory reserved. A noticeable speedup was not measured in Algorithm 3, whereas Algorithm 2 got an approx. 2% speedup for 1K rows. This speedup in runtime was probably due to not having to initialize the entries upon allocating the hash table.

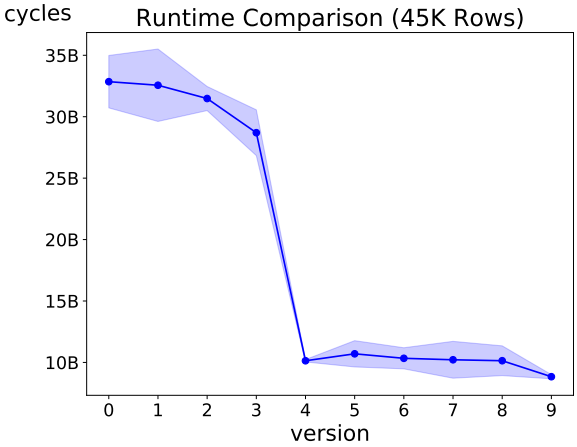


Fig. 5: End to end runtime of the worst-case optimal join program from Optimization 0 (Baseline) to Optimization 9 with tables having 45K rows.

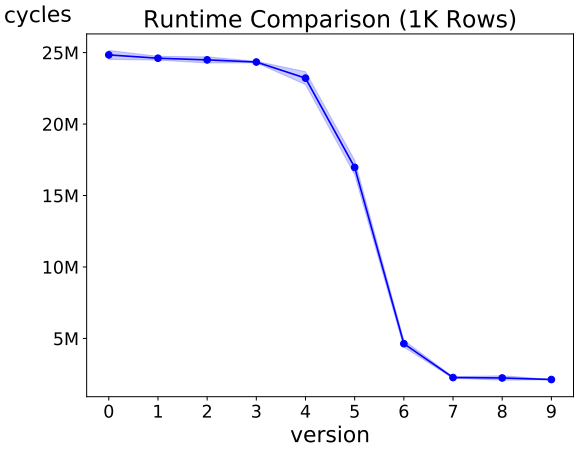


Fig. 6: End to end runtime of the worst-case optimal join program from Optimization 0 (Baseline) to Optimization 9 with tables having 1K rows.

Optimization 2. By changing to a constant time `size` lookup for `TupleList`, Algorithm 3 exhibited an approx. 1% speedup for small input sizes and up to 6% for larger sizes.

Optimization 3. We could observe an approx. 1% speedup in the 1K row case and an almost 10% speedup in the 45K row case for optimization 3, which got rid of unnecessary memory reallocations when inserting into *std::vector* data structures.

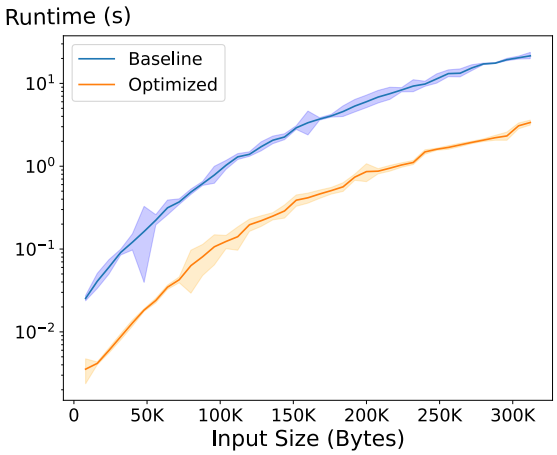


Fig. 7: Combined runtime, in seconds, of both Algorithm 2 and 3, where the runtime is related to the input size in bytes. The error bounds indicate the 99.7% running time. The y-axis uses a log-scale highlighting the 99.7% running time even for small time intervals.

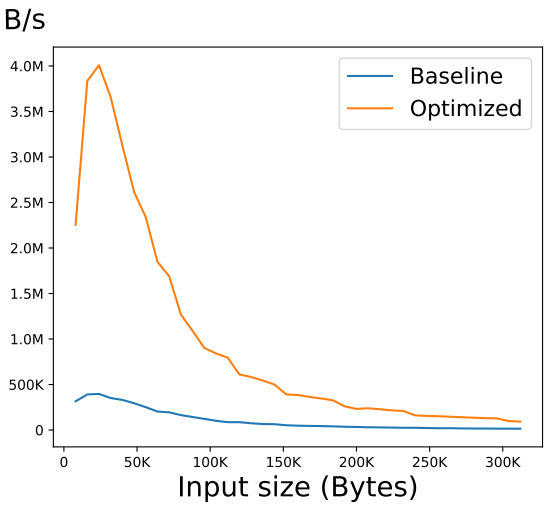


Fig. 8: For both the baseline and optimized implementation, the graph shows how many bytes are processed per second, relative to the input size.

Optimization 4. This optimization aimed to improve spatial locality when creating the joined tuples. We hoped that storing the intermediate data in a column-ordered table instead of a row-ordered would increase spatial locality in the cache when joining the final result. Empirical data supports this hypothesis, as the output from `perf` shows that this change reduced the number of LLC load misses by approx. 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 load miss decrease. Similarly, we saw an approx. 10% decrease in total cache read misses. This optimization, in turn, leads to an approximately 28% speedup for Algorithm 3.

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

According to Agner's tables [15], for the Tiger Lake microarchitecture used for benchmarking, a `div` instruction has a latency of 19 cycles and a gap of 6 cycles. We care about `div` because in order to do a `modulo` operation, we 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`. Compared to `div`, Agner's tables [15] claim that a `cmp` instruction only has a latency of 1 cycle and a gap of 0.25 cycles.

A very conservative estimate on the number of hash table insertions and lookups that assumes neither hash- nor index-collisions is $m \cdot n$ 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 shows 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 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 change, 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. Reducing the size of a hash entry 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 concerning 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 leads to an approx. 92% speedup for Algorithm 2, and 100% speedup for Algorithm 3.

Optimization 8. This optimization only had a very marginal performance benefit of roughly $\approx 1\%$ in the 45K row case and $\approx 1.5\%$ speedup in the 1K row case.

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 baseline's runtime grows exponentially faster than the optimized version's runtime as the input size in bytes grows.

We modeled the input such 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 $\approx 400k$ bytes/second. The optimized solution's peak is much more pronounced at the input size of around $25k$ bytes with $4M$ bytes/second. As the input size grows, more time is spent moving data around and calculating the final output, eventually dominating the running time and making the optimizations less pronounced. This effect can be seen by how the *Bytes/Second* move closer as the input size increase. However, there is still a noticeable difference between the baseline and optimized solution. At input size $300k$ bytes, the solutions have a performance of $15k$ bytes/second and $106k$ bytes/second, respectively.

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. We designed each dataset to yield large intermediate result tables using binary joins. The comparison between the two provides excellent insights. For example, Optimization 5 and 6, i.e., removing modulo operations in insert and lookup, bring the most considerable speedup with an input size of 1K cycles. Also, the increase in performance after *SIMD operations* is barely visible in Figure 6. The relatively small size of the result table can explain this. 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 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 a considerable performance increase at Optimization 4 and a more noticeable performance increase at Optimization 9. Ultimately, the effect of different optimizations on different input sizes aligns with our expectations.

5. CONCLUSIONS

SQL join is an essential query type in an RDBMS that occurs frequently and consumes considerable 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 provides fewer 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. Despite this, we still managed to apply several techniques learned in class, such as improving cache locality to reduce data movement, introducing SIMD operations for more efficient calculations, using instruction tables to optimize which operation we do (`div` vs. `cmp`, for example), among many others. The optimizations presented in this paper improved performance (measured in CPU cycles) of the *Worst Case Optimal Join* algorithm by up to 11.6 times, compared to the baseline implementation.

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.). In optimization 5 3, improved how hash trie lookup was 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, worked on debugging base implementation and optimizations 1, 2, 7, 8, 9.

Xianzhe Ma. implemented the basic data structures (tuple list, hash trie, and hash table) for algorithms 2 and 3 and optimizations 0 and 4; also patched memory leaks using `Valgrind`.

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.