Adaptive Data Structure Choice using Runtime Statistics for Compiling Query Engines

Patrick Barclay
TUM
patrick.barclay@tum.de

ABSTRACT

Hash joins are widely used in query engines but are not optimized for skewed data distributions, which are common in real-world data. For example, workloads with many duplicate keys ("heavy hitters") cause high hash collision rates for hash-based joins. Using specialized data structures for such cases can help, but skew detection at query optimization time is difficult and error-prone. We therefore propose a design that uses the HyperLogLog and Misra-Gries sketches to collect approximate statistics about join key distribution at runtime, and then adapt the data structure accordingly to improve join performance. We introduce two specialized data structures: One for handling heavy hitters and one for handling uniformly duplicated join keys, and implement them into an existing query engine. Our design incurs a maximum performance overhead of 0.4% in the worst case, and speeds up query execution by up to 220.1% for certain Zipf-like data distributions.

1 INTRODUCTION

Hash joins are very fast for in-memory database joins, but they are not optimized for skewed data distributions [1]. However, in real-world data, skew is common and can exist in different forms. When we talk about skew in this work, we refer to the distribution of join keys within a relation.

One type of skew is the "heavy hitter", which is a key that exists many times in one column of a relation. A common cause of a heavy hitter in real-world data is the use of a default value. In this case, the default value may occur many more times than any other value. Another real-world example of a heavy hitter can be found in datasets representing graph structures. If there is a graph node with many incoming or outgoing edges (for example, a popular influencer on a social media platform), its value occurs more often in the "edges" relation than others. Another type of skew is the uniform multiplicity skew. This type of skew occurs when each key appears n times instead of once. This can be caused by time-bucketed data.

These two types of build-side join key skew negatively impact join performance, as they create long collision chains, which cause 1) high contention during hash table build, and 2) many cache misses during probing chain traversal.

This guided research paper explores how we can use approximate statistics to optimize hash join performance for data distributions that are suboptimal for a regular hash join. We achieved this by leveraging the fact that most queries are memory-bound during materialization. This means we can calculate statistics on the data distribution during the materialization phase without incurring a large performance penalty. With these statistics, we can make runtime decisions about the join data structure to improve join performance.

We introduce a design for adaptive data structures to efficiently handle both uniform multiplicity skew and heavy hitters, along with a highly optimized implementation of a skew detection system. We evaluate the performance of our design across multiple synthetic datasets with varying build-side join-key distribution skew and show that it can speed up joins for heavily skewed data by up to 220.1%, while not significantly affecting non-skewed workloads, with a worst-case performance penalty of 0.4%.

2 BACKGROUND

The work in this guided research is based on a parallelized version of the query-compiling engine "p2c" developed at TUM [9]. Given a query plan, it generates C++ code that executes the query. For any arbitrary table join, p2c uses a hyper-style hash join [4]. The following section explains the principles behind the hyper hash join, the implications of skewed data, and methods for detecting such data skew.

2.1 Hyper-Style Hash Join

In general, hash joins insert all tuples of the smaller join relation ("build side") into a hash table. Then, it iterates over the larger relation ("probe side") and looks up the join key in the hash table. If a matching tuple exists on the build side for a probed tuple, the tuple pair is emitted as output.

The hyper hash join operates in three phases: materialization, hash table build, and probing.

1. Materialization. The materialization phase prepares the data on the build side for insertion into the join hash table. This can either mean reading the data from disk or applying the downstream operators in the query plan to the underlying data. The prepared data is then written to a buffer, where each tuple is stored densely, with extra space for a pointer

between each tuple. Since the query runs in parallel across multiple cores, the input data is partitioned and assigned to different threads. Each thread then materializes its assigned tuples into a thread-local buffer.

2. Hash table building. 2. After the tuple buffers have been populated, a hash table in the form of a pointer array is created. Each thread iterates over the tuples in its thread-local buffer and inserts them into the hash table. This is done by calculating the hash of the tuple, which determines the bucket (i.e., the index of the pointer array) into which the tuple will be inserted.

To insert the tuple, the pointer within the array is set to the tuple's address in the buffer. In the case that there is a hash collision, and the bucket already points to a tuple in the buffer, the existing pointer is moved to the new tuple's "next pointer" slot within the buffer. This creates a linked list of entries with hash collisions in the bucket. Since we are in a multithreaded scenario, each pointer update must be synchronized to avoid race conditions. By using this technique of writing the next pointers directly into the tuple buffer, we avoid copying the data when inserting into the hash table.

3. Probing. With the hash table built, the query engine iterates over all probe-side tuples and looks up the join key for each. If an entry with the same join key is present, the tuple pair is emitted as output. When multiple entries are present in the corresponding bucket, the whole linked list is walked to emit all matching tuples.

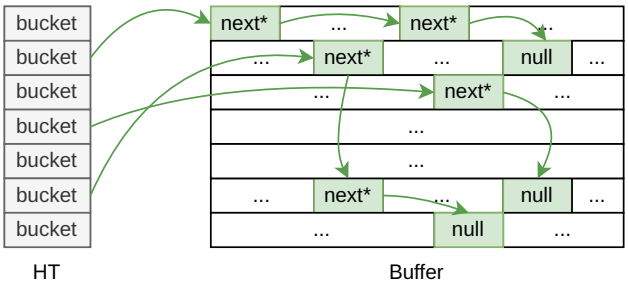

Figure 1: Example of the hash table and tuple buffer after the build phase.

Figure 1 shows an example of the finished hyper hash table after the build phase. On the left, there is the hash table array with pointers into a thread-local buffer on the right, which contains the next pointers for the collision-linked lists.

2.2 Problems Caused by Data Skew

We can observe that when a key on the build side occurs many times, the above-described algorithm results in a long linked list for the corresponding bucket. This causes the following problems:

Problem P1: During probing, we need to dereference the next pointer for each traversal step, which may result in a cache miss. This is because the next pointers may not reference a tuple within the same buffer, and thus within the same memory region, but rather a tuple in a buffer from a different thread in a different memory region.

Problem P2: Because we insert into the hash table concurrently across multiple threads, there is significant contention on the buckets with heavy hitters, as many threads try to insert values into the same bucket.

The same principles apply when there is not one value with many duplicates, but each value is duplicated n times (i.e., uniform multiplicity skew). Both of these problems can significantly impact query performance.

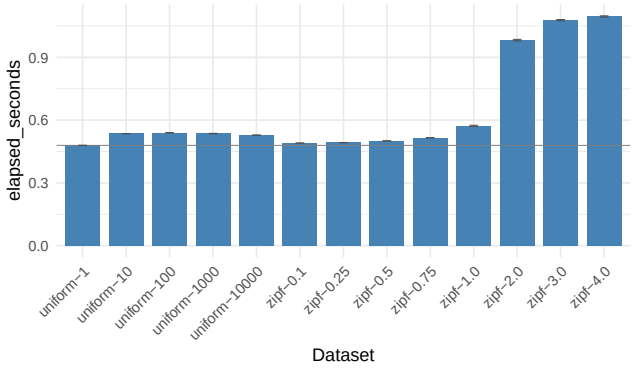

Figure 2: Query execution time for different Zipf distributions

Figure 2 shows how performance degrades as data skew is introduced for an example query. In the non-skewed case, the query takes 0.479 seconds. The runtime increases to 0.540 seconds with uniform multiplicity skew and up to 1.095 seconds with heavy hitters present. That is a significant runtime increase of 128.6%.

2.3 Sketches

To mitigate the above-described problem, we require a method to gain information about the data distribution, which would enable us to make an informed decision about the choice of a join data structure.

Previous work has shown that query optimizers are often unable to detect skew and correlation before runtime, frequently leading to suboptimal query plans in real-world datasets [5]. Therefore, to reliably detect non-uniformities in the data distribution, we need to compute distribution statistics at runtime. This is not a trivial problem in database systems, as we need to do so with a single pass through the data and minimal memory and CPU usage. A good way to

achieve it is to use *sketches* during the materialization phase. Sketches are data structures that can efficiently approximate various properties of data distributions with constant memory overhead in a single pass [7].

2.3.1 HyperLogLog Sketch. The first sketch we utilize in our work is the HyperLogLog Sketch [3], which is based on the Flajolet-Martin algorithm [2]. Given the sketch size is k bytes, it can approximate the number of unique values in a data stream with an expected relative error of $\pm\frac{1.04}{\sqrt{k}}$, using only a small amount of CPU cycles. For example, with an HLL sketch size of 256 bytes, a unique cardinality estimate in a stream of values has an expected error of 6.5%.

2.3.2 Misra-Gries Sketch. The second sketch we use is the Misra-Gries Sketch [6] (MG-Sketch). It can be used to detect values which a high frequency in a data stream. In a stream of n items, an MG-Sketch of size k can detect heavy hitters that occur at least $\frac{n}{k}$ times. For example, a sketch of size $k = 100$ detects heavy hitters that account for at least 1% of the build-side values. Thus, the larger we make the sketch, the more sensitive it becomes towards detecting heavy hitters. Section 4.1 explains in detail how the sketch works and outlines all optimizations we made to its implementation to minimize its performance overhead.

3 DESIGN

Our design focuses on the choice of data structure for the hash join, but in theory, it would also enable other query optimizations based on data skew. For the hash join, we can optimize hash table construction and probing. We leverage the fact that most large queries are memory-bound. This means that during materialization, the CPU is idle while waiting for data to be read from disk or memory. We can use these unused CPU cycles to compute statistics on the data while materializing it, which can then inform our decision about query execution. This may slightly slow down our materialization phase, but the performance gains from our optimized data structures in the hash table build and probing phases offset the overhead.

To inform our data structure choice, we calculate the following statistics: 1) minimum and maximum tuple value, 2) total number of tuples, 3) estimated unique tuple count using the HyperLogLog-Sketch, and 4) heavy hitters using the MG-Sketch. From the total number of tuples and the unique tuple count, we can estimate the average number of duplicates per tuple. Given these statistics, we developed the following decision tree to guide our choice of data structure for the hash join:

If the MG-Sketch detected any heavy hitters in our dataset, we use a specialized heavy-hitter hash table. If we did not detect any heavy hitters, we check if the estimated average

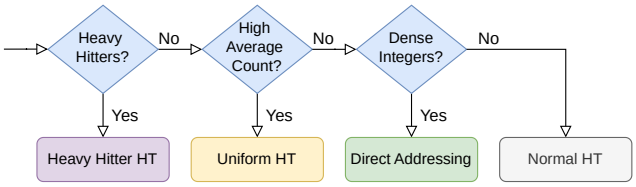

Figure 3: Decision tree for selecting the appropriate hash table

tuple frequency is greater than or equal to 5. If so, we use a specialized data structure to handle uniform multiplicity skew. If that is also not the case, we check whether our build values form a dense integer range (e.g., integers 1 to 10 000 000). This can be detected by calculating the range between the minimum and maximum values, then dividing by the estimated unique count. In that case, we can use a directly addressed array, where the key serves as the lookup index, eliminating the need for a hash table. If none of these value distributions can be found, we use the regular hyper hash table implementation.

3.1 Heavy Hitter Hash Table (H3T)

The idea of the heavy-hitter hash table ("H3T") is to treat the heavy hitters differently from the normal tuples, since the heavy-hitter buckets will contain very long chains. Regular values are inserted like normal into a linked list, but the heavy hitters are stored in a chained array, where each chain link contains an array of n tuples. The differentiation between a heavy-hitter tuple and a regular tuple is possible, since the heavy-hitter tuples are known after the materialization phase. With our design, we reduce the chain lengths in heavy-hitter buckets, which reduces cache misses during probing. This is because the heavy hitters are stored in dense array chunks, with only a small number of next pointers connecting the array links. In other words, we reduce the number of next pointers from 1 per tuple to $1/n$ per tuple, thereby enabling more sequential reads during probing, thus mitigating problem *P1*.

Furthermore, each thread uses its own data structure, called a thread-local table. Each thread-local table stores one array of tuples for each heavy hitter, which were detected during materialization. These arrays serve as thread-local buffers for the heavy-hitter buckets, allowing each thread to insert heavy hitters without requiring synchronization. Only when one of the thread-local heavy-hitter arrays is full is it inserted into the global heavy-hitter chained array, which requires one synchronized pointer change. This means that during insertion, we only need to do one synchronized access to the heavy hitter bucket per n inserted items, thus

reducing contention during the hash table build phase and mitigating problem *P2*.

One drawback of our approach is that heavy-hitter tuples require an additional data copy, as they must be written to the tuple arrays rather than staying in the matrerialization buffer during insertion into the hash table, like non-heavy-hitter tuples.

3.2 Uniform Multiplicity Hash Table (UHT)

The data structure we use to handle uniform multiplicity skew ("UHT") relies on the same principle as the H3T: reducing the number of next pointers per tuple by storing tuples in a chained array to reduce cache misses during probing, thus mitigating problem *P1*. However, with this type of distribution, all tuples are stored in such chained arrays, as there is no constrained list of heavy hitters. Therefore, the approach of creating thread-local tables for insertion no longer works, since creating a complete hash table with thread-local arrays for all buckets would consume too much memory. Consequently, every single insert into the uniform hash table must be synchronized, so we cannot mitigate problem *P2* using our approach. Section 4.3 explains how the synchronization mechanism works in detail.

4 IMPLEMENTATION

The most relevant component of p2c for our work is the HashJoin operator, which joins two relations together using the hyper-style hash join [4]. This research is a proof of concept, meaning the design has not yet been implemented into p2c. Instead, we modified the generated C++ file and the relevant header files for a given query to reflect our design. The implementation of the HyerLogLog-Sketch is not included in the scope of this research.

Figure 4 shows the SQL of the query that we applied our design to. The query was purposely chosen not to exhibit differing intermediate or final result relation sizes when altering the data skew, to enable comparison of the query's performance across different data skews.

```
SELECT o_orderpriority, COUNT(c_custkey)
FROM orders
JOIN customer ON c_custkey = o_custkey
WHERE c_nationkey <= 25
GROUP BY o_orderpriority
```

Figure 4: SQL query for implementation

We parallelized p2c using morsel-driven parallelism [4], implemented with the oneTBB library [8]. The following outlines the implementations of the components of our design. Note that implementing the dense integer array is not covered in this research due to time constraints, but the performance benefit it would yield would not be surprising, given the results achieved by our design.

4.1 Misra-Gries Sketch Implementation

Developing a highly efficient implementation of the Misra-Gries sketch is crucial to making our design feasible, as the sketch must deliver information for every build-side join key distribution in every query. A slow sketch implementation would cause high overhead across all queries, even those that don't benefit from an intelligent data structure choice. Consequently, we want our sketch to have minimal instructions per tuple and a limited memory footprint, since there is already significant memory pressure during materialization. We chose $k = 33$ for our implementation, meaning we can detect heavy hitters that comprise at least 3.03% of the keys in a relation.

In principle, the sketch works by using a hash map that counts the occurrences of the most common values seen so far. The algorithm of the sketch is defined as follows:

```
A := new HashMap
for each i in s:
    if i is in keys(A):
        A[i] := A[i] + 1
    else if size(keys(A)) < k - 1:
        A[i] := 1
    else:
        for each k in keys(A):
            A[k] := A[k] - 1
            if A[k] = 0:
                remove k from keys(A)
return A
```

Figure 5: Misra-Gries Sketch algorithm

For each tuple, we look up its key in the hash map. If the key is present, we increment its counter. Otherwise, if the sketch is not full, we insert the key with a count of one. If the sketch is full, we decrement all counts by 1 and remove all keys with a count of 0. Notice that we only insert $k-1$ entries at most into our sketch, meaning we only need to store 32 key-count pairs. We use a 64-bit key and 64-bit counts. Thus, the sketch's minimum theoretical memory footprint is 4096 bytes.

We implemented the sketch in many different ways to find the best optimization. First, we used the C standard library hash table to keep track of the counts. Then we implemented a custom chained hash table using an array to store keys and counts, performing a linear search for each key. Since our sketch is so small, the linear search outperformed the hash map variants. We continued optimizing by using SIMD,

storing the 32 key-count pairs in 16 `__m256i` registers, and using bit arithmetic to implement SIMD lookup, insertion, incrementation of a single value, and decrementation of all values greater than 1.

We then performed loop unrolling, further reducing the number of instructions per tuple. We tried unrolling by interleaving the operations and the iterations. An example of this is shown in Figure 6.

```
// Loop
for (int i = 0; i < 3; i++) {
  operation1(arr1[i]);
  operation2(arr2[i]);
}

// Unrolled - interleave operation
operation1(arr1[0]);
operation1(arr1[1]);
operation1(arr1[2]);

operation2(arr2[0]);
operation2(arr2[1]);
operation2(arr2[2]);

// Unrolled - interleave iteration
operation1(arr1[0]);
operation2(arr2[0]);

operation1(arr1[1]);
operation2(arr2[1]);

operation1(arr1[2]);
operation2(arr2[2]);
```

Figure 6: Loop unrolling example

We then added a fingerprinting mechanism to further optimize the lookup operation by storing the last 8 bits of each key in a single `__m256i` and using a bitmap to track empty slots.

The logic for updating the sketch is implemented using SIMD as follows: When processing a key k, the sketch first checks whether it contains a fingerprint in (A) that matches k's fingerprint using a single SIMD instruction. It then checks, using (B), whether the slot corresponding to the fingerprint is currently occupied. If no key with a fitting fingerprint is found and there is no free slot, we decrement all counts in (D) that are not zero, to avoid an unsigned integer overflow, and then re-calculate the free-mask (B). If there is a free slot, we use the free-mask to calculate which `__m256i` index and which lane within it the key should be inserted, and then do so by using the `_mm256_blendv_epi8` instruction to update (A), (B), (C), and (D) accordingly, as shown in Figure 8.

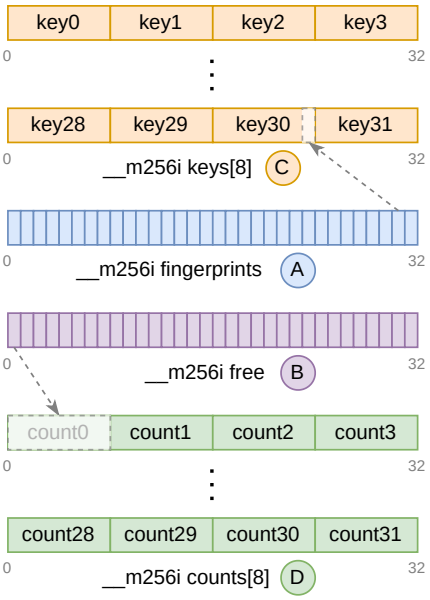

Figure 7: Data layout of the MG-Sketch with SIMD and fingerprinting

```
m256 idx = _mm256_set_epi64x(3,2,1,0);
m256 idx8 = _mm256_set_epi8(31,30,...,0);

// Get first free slot
u32 free_u32 = m256_movemask_epi8(free);
// lane from 0 to 31
u8 lane = __builtin_ctz(free_u32);
 // lane within 256i register
u8 internal_index = lane % 4;

// Mask to only modify correct lane
m256 insert_m = _mm256_cmpeq_epi64(idx,
    _mm256_set1_epi64x(internal_index));

// Update key and count at selected lane
keys[lane/4] = m256_blendv_epi8(
    keys[lane/4], key_broadcast, insert_m);

counts[lane/4] = m256_blendv_epi8(
    counts[lane/4], one, insert_m);

// Update fingerprints and free mask
m256 mask8 = m256_cmpeq_epi8(idx8,
    m256_set1_epi8(lane));

fngrprints = m256_blendv_epi8(fngrprints,
    fingerprint_broadcast, mask8);

free = m256_blendv_epi8(free, zero, mask8);
```

Figure 8: SIMD logic for sketch update

If a matching fingerprint is found, we increment the counts (D) for all slots that correspond with a matching key in (C). If merely the fingerprint matched one of the keys and no counts were incremented, we insert the key as described above.

Finally, we added a probabilistic disabling mechanism to the sketch that deactivates its calculation at runtime if no heavy hitters are found. It works by maintaining a confidence score between 0 and 10 to indicate whether heavy hitters are present. If the confidence drops below 3, we stop calculating the sketch and only enable it again with a 10% probability per block of data, giving a new chance to increase the confidence score. This further reduces performance overhead, especially in queries with no heavy hitters. To avoid additional synchronization, each thread has its own copy of the sketch, which is combined with the other sketches at the end of the materialization phase.

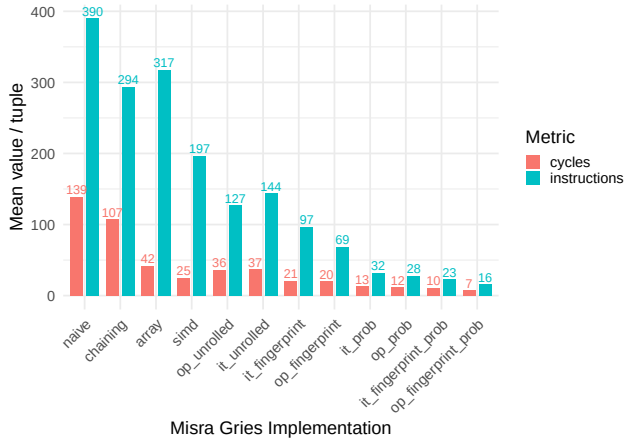

Figure 9: Performance comparison of different MG-Sketch implementations

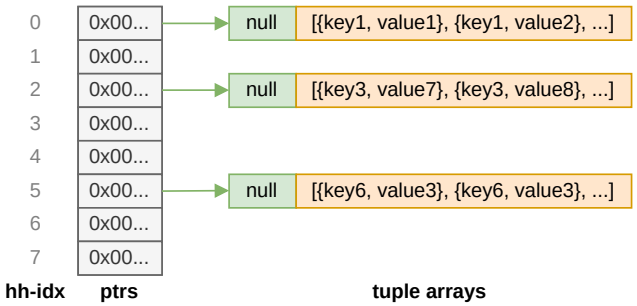

Figure 10: Thread-local heavy hitter hash table

Figure 9 shows how many instructions and cycles each MG-Sketch implementation requires to process each tuple during the materialization phase of the query in a dataset of uniform keys. We can see that each optimization step either decreased the number of instructions or cycles. As our final implementation, we chose the operation-interleaved probabilistically disabled version with fingerprint checking, as it consumes the least CPU cycles, with only 7.2 cycles and 15.65 instructions per tuple. Compared to the naive `std::unordered_map` implementation this is a 94.82% reduction in cycles and 95.99% reduction in instructions.

4.2 H3T Implementation

The Heavy-Hitter Hash Table implementation is split into two parts. The thread-local table and the global table. The global table is structured like a regular chained hash map, with an array of pointers that point to either a regular linked list or a chained array, where each chain link contains 1024 tuples. Figure 11 shows the layout of the global table.

The thread-local table stores an array of pointers to tuple arrays, where each pointer is associated with one heavy hitter, as shown in Figure 10.

The global table is initialized with the list of detected heavy hitters, where each heavy hitter key is mapped to an index value. Before inserting any values, the buckets associated with the heavy hitters are marked by setting the top byte of the pointer stored in each bucket to the corresponding index value. The pointer tagging serves two purposes: 1) When looking up or inserting tuples into the hash map, we can immediately determine if the respective bucket is a heavy hitter bucket without an additional lookup operation. 2) All inserts into heavy-hitter buckets of the global hash table are deferred to the thread-local table of the current thread. The executing thread can use the index stored in the pointer's address tag to get the information on which heavy-hitter array the tuple should be inserted into in the thread-local table.

If one of the thread-local tuple arrays is full after inserting a tuple, the array is moved to the global hash table via one `compare_exchange_strong` operation. Thus, the array becomes part of the chained array, where there is a "next" pointer attached to the beginning of the tuple array to refer to the next chain link. After the local array is moved into the global chained array, the thread allocates another local array for this heavy-hitter bucket, ready to receive additional tuples for insertion. At the end of the hash table build phase, all partially filled thread-local heavy-hitter arrays are flushed to the global hash table.

4.3 UHT Implementation

Since the use of the UHT assumes the absence of heavy hitters, there is no need for pointer tags to identify heavy-hitter buckets. Instead, *each* bucket's values are stored in

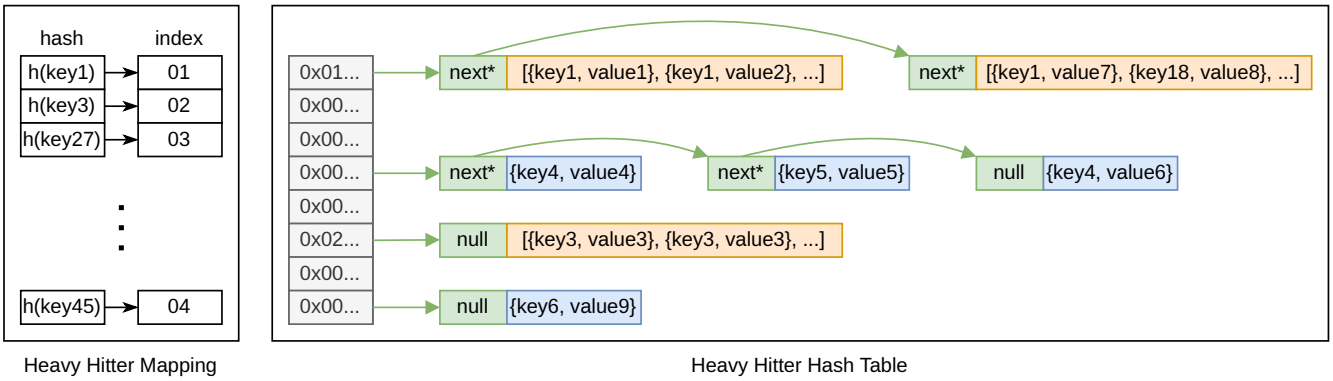

Figure 11: Global view of the H3T

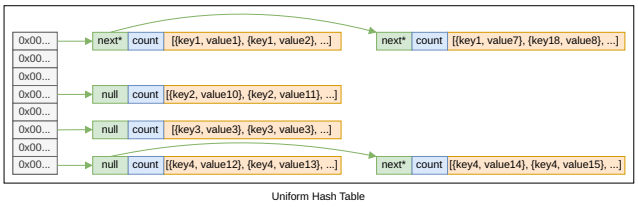

Figure 12: Uniform hash table diagram

chained arrays. However, because thread-local insertion is not supported, each tuple insertion must be synchronized. To do so, the UHT uses an additional atomic count value for each tuple array. It acts as a ticket allocator, assigning an array index for threads to insert new tuples into. If the current thread gets assigned the last slot in the array, it is responsible for creating a new array and performing a `compare_exchange_strong` to insert it into the chained array. If the current thread's index ticket exceeds the array size, it knows another thread is currently creating a new array, so it dereferences the bucket's pointer again and retries the insertion.

The length of the tuple arrays depends on the estimated average tuple count measured by our sketches, so that, on average, we can accommodate all tuples of one bucket within one array. To reduce overhead from allocating new arrays, we employ two strategies. Firstly, we use an arena allocator with preallocated memory to allocate new arrays. This ensures allocation overhead does not influence our measurements, and is in line with buffer-managed memory used by most modern database systems. Secondly, to avoid many allocations, we slightly overestimate the average tuple count. The formula for calculating the array size is the following:

$$\text{array\_size} = \text{estimated\_avg\_count} \cdot 1.05 + 1$$

Thus, even if the sketches underestimate the average tuple count, we are unlikely to have many arrays that are too short to fit all the duplicated values. Of course, this method uses more memory than necessary if the average count is estimated correctly or even overestimated.

5 END-TO-END EVALUATION

Given our optimized sketch implementation (see section 4.1) and the H3T and UHT implementations, we now evaluate the end-to-end system performance of our design. To do this, we run the same query 20 times on several synthetic TPC-like datasets with different build-side join key distributions.

5.1 The Datasets

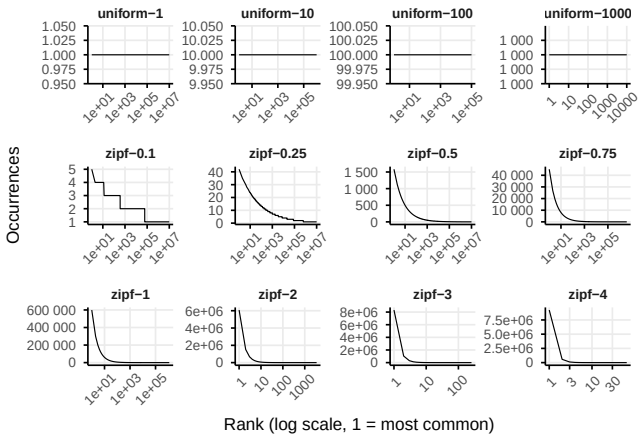

Figure 13: Key distributions in the datasets

The number of tuples is 10 000 000 on the build side, and 130 000 000 on the probe side. We evaluate 13 data sets with varying join-key distributions: five with uniform multiplicity distributions, and the other eight with Zipf-like distributions

with different parameters. Figure 13 shows the distribution of the join keys for some of the datasets (uniform-10000 was omitted for visual reasons). The y-axis shows the number of occurrences of a certain value. The x-axis is a log scale of occurrence rank. For example, the graph at the x-value 1 denotes the frequency of the most common key.

5.2 E2E Performance

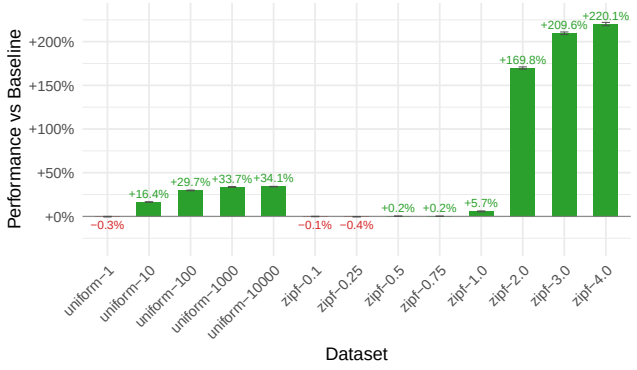

Figure 14: End-to-end performance comparison

The results of our end-to-end performance measurements are shown in Figure 14. They reveal that for the uniform multiplicity skew datasets (uniform-10 through uniform-10000), we improve performance by up to 34.1% by invoking the UHT, since the average frequency exceeds the UHT threshold. However, the most notable improvement is observed in datasets with detectable heavy hitters (zipf-1.0 through zipf-4.0), where using the H3T improves query performance by up to 220.1% compared to the default implementation. For all other datasets, the performance stays almost the same as the vanilla implementation of the query, with performance decreases of only up to 0.4%. These datasets do not contain any heavy hitters with sufficiently high frequencies to be detected by the Misra-Gries sketch of size 33.

The datasets zipf-0.5 and zipf-0.75 reveal a shortcoming of our Misra-Gries-based skew detection, which is that for large datasets, the size of the sketch might not be large enough to detect heavy hitters with sufficient sensitivity. For example, zipf-0.75 contains keys with up to 45,514 duplicates, which slows query execution compared to an unskewed workload, as seen in Figure 2. However, these frequencies do not exceed the threshold of 3.03% (requiring 303,000 duplicates) and are thus not large enough to be detected by the Misra-Gries algorithm.

5.3 Per-Phase Evaluation

To get a more precise overview of where these performance gains come from, we measured the number of CPU cycles

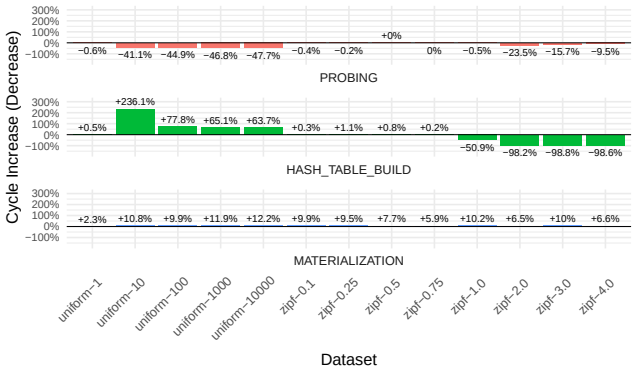

Figure 15: Per-phase performance comparison

in each phase of the hash join separately. The results can be seen in Figure 15. We observe that in the UHT datasets, most of the speedup comes from the probing phase by solving problem *P1*, whereas in the H3T datasets, mitigating problem *P2* during the hash table build phase makes the greatest difference.

In all datasets, the probe phase is sped up by some degree or remains the same as the baseline. We notice that in the uniform-10 dataset, the UHT build phase is much slower than in all other runs, which warrants further investigation in future work. All distributions exhibit an increase in cycle count during the materialization phase, requiring up to 12.2% more cycles to compute the sketches.

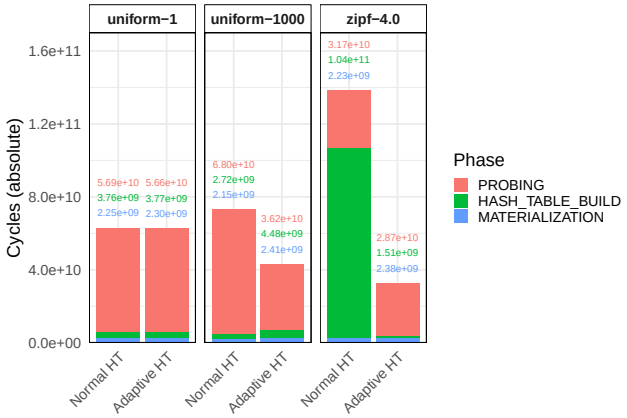

Figure 16: Absolute cycle count per phase for selected datasets

To show the absolute difference in cycles across phases, we provide an excerpt of measurements for the most notable datasets in Figure 16.

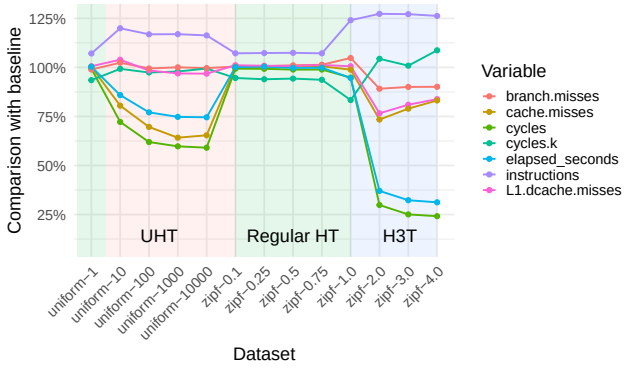

Figure 17: Comparison of all metrics across datasets

5.4 Additional Metrics

Figure 17 displays additional metrics that provide insights into the performance gains our design delivers. We can see that across all datasets, the number of instructions increases due to sketch calculation and the additional mechanisms for building and probing the UHT and H3T data structures. However, the additional instructions do not incur a noticeable performance penalty, as they either occur during the memory-bound materialization phase or are amortized by the significantly improved cache behavior of our design. The improved cache behavior is also evident in the graph, where the UHT exhibits up to 35.8% fewer cache misses, and the H3T 26.6% fewer than the baseline. The increased number of kernel cycles in the H3T is also a subject of future investigation.

5.5 Skewed Probe Side

To see how our design behaves when we additionally introduce skew to the probe-side join keys, we reran all benchmarks with the *probe-side* keys exhibiting a uniform-100 skew. Figure 18 shows the performance increase our design yields on these tests.

We see that for the uniform-x build-side datasets, we now have a greater performance increase, whereas we do not gain as much performance for the higher Zipf-factor skews.

The graph in Figure 19 explains the cause. It shows that the vanilla implementation experiences a much larger slowdown in the uniform-x cases when the probe side also contains duplicated join keys. An example can explain why this is the case. The uniform-10000 build-side dataset contains only 1000 unique values, leaving most hash table buckets empty. With unique probe-side keys, most probe operations access an empty bucket, triggering no cache misses. However, when the probe-side keys are also skewed (and the same keys are used for duplication), more probe operations access a populated bucket with a long chain.

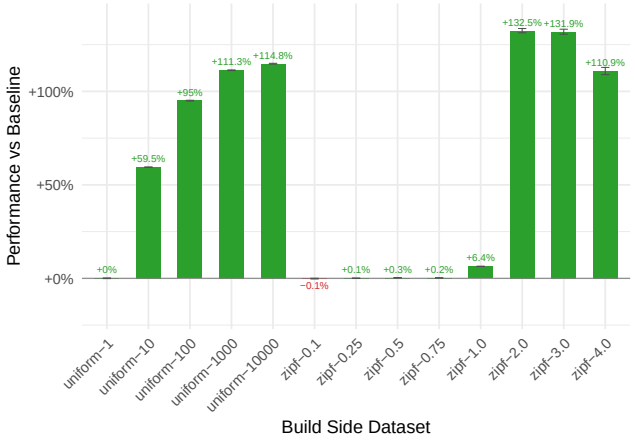

Figure 18: Performance increase with skewed probe-side keys

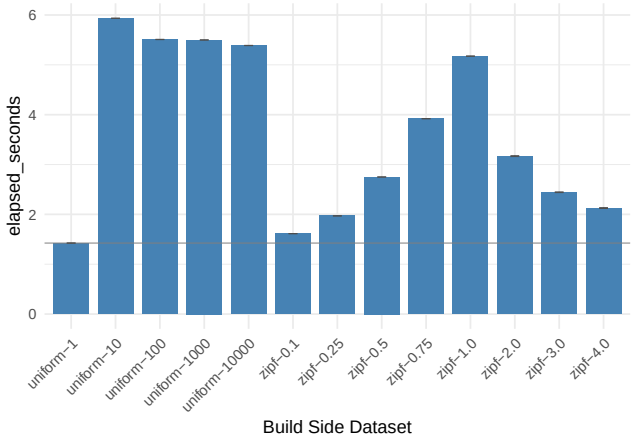

Figure 19: Performance slowdown of the vanilla implementation with skewed probe-side keys

Furthermore, we notice that the vanilla implementation improves its performance again once the build-side skew exceeds zipf-1.0. This behavior is quite unexpected, but due to time constraints, we were unable to investigate how extremely long chains can be probed so efficiently. However, it explains why our adaptive design does not improve the performance as drastically for these datasets.

6 FUTURE WORK

Although we achieved a large speedup for skewed hash joins, our design and evaluation have some shortcomings that could be an interesting topic for future research.

Sketch Scale: One drawback of our design is the limited skew detection for less heavily skewed data. Our Misra-Gries

implementation can only detect heavy hitters with a value share > 3.03%. For large datasets, this is not sufficient to alleviate the performance penalties caused by less-dominant heavy hitters. To mitigate this problem, the sketch size would need to scale with the number of build-side tuples. However, increasing the sketch size might incur a larger performance penalty on unskewed datasets. Especially for our highly optimized loop-unrolled implementation, varying the sketch size is not easily feasible and therefore subject to future work.

Query Engine Integration: The next step in developing this project is to implement the data structure for handling dense integer keys, then integrate the design into the p2c query engine. One path to consider is adaptive code generation at runtime, meaning that the code for the hash table build and probing phases is generated after materialization and sketch calculation are complete, thereby reducing the size of the query program.

Real-World Evaluation: After integration into a query engine, it is advisable to run additional benchmarks across a range of queries on different real-world datasets to obtain a more accurate indication of how the design performs across a wide range of non-synthetic workloads.

Investigate Probe Side Skew: As shown in section 5.5, when skew is introduced on the probe side, the vanilla implementation shows a speedup in some cases. It would be interesting to further investigate this behaviour.

Optimize More Operators: In this work, we only focused on creating adaptive data structures for the hash join operator. However, the same principle could be applied to other operators such as `GROUP BY` or `UNIQUE`. One could design a system to forward the collected runtime statistics to other operators in the query plan, thereby creating more opportunities for adaptive data structures higher up in the operator tree.

7 CONCLUSION

In this guided research paper, we explored how collecting runtime statistics can enable adaptive data structures to improve the performance of the hash joins on skewed workloads. Specifically, we used the HyperLogLog and Misra-Gries sketches to create a highly optimized skew detection system, and designed custom data structures, the H3T and the UHT to handle different kinds of skew. These data structures follow the principle of using chained arrays rather than linked lists in buckets with hash collisions. By doing so, we greatly improved the cache behavior of the probing phase and reduced contention during the build phase on skewed workloads.

We evaluated our implementation on custom synthetic datasets with varying build-side skew. The measurements show, that our design can speed up joins for heavily skewed data by up to 220.1%, while not significantly affecting non-skewed workloads, with a worst-case performance penalty of 0.4%. This demonstrates that incorporating lightweight runtime statistics with skew-aware data structures offers a practical path toward significantly accelerating hash joins under skewed workloads without compromising much performance in the general case.

REFERENCES

[1] Bryce Cutt and Ramon Lawrence. Using intrinsic data skew to improve hash join performance. *Information Systems*, 34(6):493–510, 2009.

[2] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31 (2):182–209, 1985.

[3] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science*, (Proceedings), 2007.

[4] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 743–754, 2014.

[5] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[6] Jayadev Misra and David Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

[7] Brian Tsan, Asoke Datta, Yesdaulet Izenov, and Florin Rusu. Approximate sketches. *Proceedings of the ACM on Management of Data*, 2(1): 1–24, 2024.

[8] UXL Foundation. oneAPI Threading Building Blocks (oneTBB). https://github.com/uxlfoundation/oneTBB, 2026. GitHub repository, accessed 2026-04-03.

[9] Viktor Leis. P2C: Plan-to-C Query Compiler. https://github.com/viktorleis/p2c, 2026. GitHub repository, accessed 2026-04-04.