Practical Techniques for Hash Joins with Duplicate Keys

Parker Timmins
Technische Universität München
parker.timmins@tum.de

ABSTRACT

Hash joins are the dominant join algorithm in modern database systems. Despite their high performance, they are not without weaknesses. In particular, duplicate key values can significantly degrade their performance. A recent paper proposed the 3D hash join, which mitigates the issue of duplicate keys. This algorithm groups tuples that share keys into sub-lists within each hash table collision list. As this technique shows promise, we investigate it in a realistic join operator. Specifically, we combine the 3D algorithm with a hash join based on that of the Umbra database.

Unfortunately, the initial combination of these systems was not an improvement over the Umbra hash join. However, by estimating the number of unique keys and setting the hash directory size based on this value, we achieved up to a 77% speedup over the baseline. Based on these results, we also present a technique for optimizing the hash directory size in a standard chained hash join. This method produces up to a 45% speedup over the baseline while reducing hash directory memory usage by 70%. In this paper, we describe the implementation of both techniques and show that they are potentially useful tools for mitigating the effects of duplicate keys in hash joins.

1 INTRODUCTION

Despite being the most performant equi-join implementation [1], hash joins are relatively simple. The tuples of one relation are inserted into a hash table using the join key as the hash table key; this relation is denoted as the build relation. Then the second relation, known as the probe relation, is traversed. For each tuple in the probe relation, tuples with matching keys from the build relation are extracted from the hash table. For every match, the build and probe tuples are emitted to the output.

Much of the complexity of a hash join is within the hash table, so we start with a small introduction to hash table implementation.

There are several types of hash tables, but all share a primary part—the hash directory. This is simply an array containing either key/value pairs or pointers to key/value pairs. The pairs or pointers are placed at specific offsets in the array based on their hash value. Unfortunately, there is a problem with this scheme: multiple items may produce the same hash value and thus be located at the same offset. This is known as a collision. Generally, there are two ways of handling collisions: open-addressing, and separate chaining. We will only look into separate chaining here, as it is frequently used in database systems, such as Umbra [5].

In a separate chaining hash table, key/value pairs are stored in a linked list accessible from the hash directory. An example of such a linked list can be seen in the fig. 1a, which shows the directory and node structure of the Umbra hash table. In this scheme, pairs with the same key are stored in the same linked list. Likewise, different keys with the same hash value are stored in the same linked list.

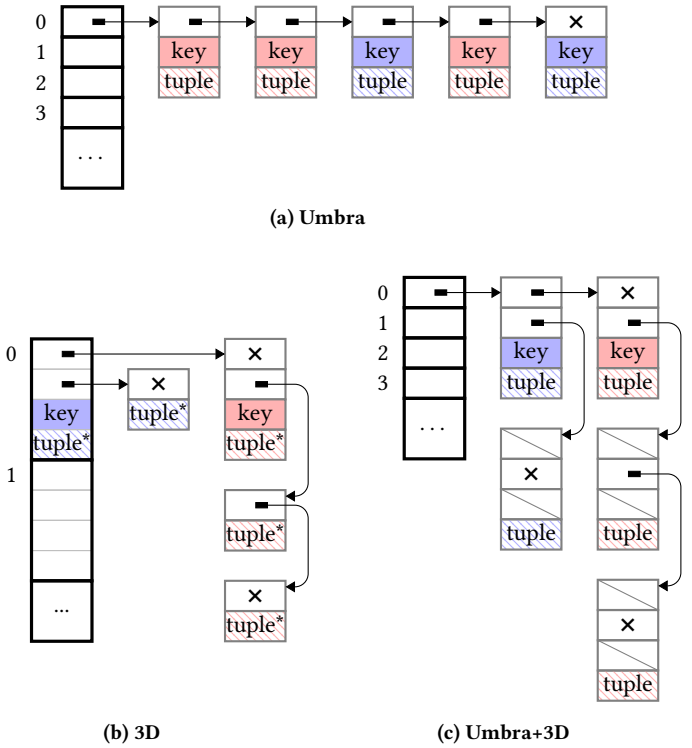


Figure 1: Hash Directory and Node structure of different implementations. Each contains the same data: a red key with 3 tuples and a blue key with 2 tuples.

Storing duplicates and collisions in a linked list is convenient, but has a downside—linked list traversal can be expensive. If the linked list nodes are not cached, dereferencing the *next* pointers requires expensive random-access memory reads. This issue is particularly apparent when there are both collisions and duplicates. Consider searching for all matches in a bucket that contains many duplicates of a different key. This entails the traversal of an arbitrarily long linked list, resulting in an $O(n)$ lookup complexity.

A recent paper by Flachs et al. [3] proposed a scheme, known as the 3D Hash Join, to help rectify this issue. Instead of a single linked list of values, their algorithm groups tuples that share keys into separate sub-lists. The structure of these linked lists can be seen in fig. 1b. The benefit of this design is that the main list only contains nodes for the unique keys which collide in a given bucket. On average very few keys will collide per bucket, so this top-level list will be quite short. This design significantly reduces the number of linked list nodes that must be traversed when a hash table contains many duplicate keys.

In our paper, we investigate how to reduce the cost of duplicates in a realistic hash join. Given the results of [3], we start by adapting the 3D Hash Join technique to a highly performant, parallel join implementation. We base our hash join on that of the Umbra database system [5, 7], using its lock-free insertion, per-bucket Bloom filters, and morsel-driven parallelism. Integrating the 3D algorithm with these features was a main contribution to this work. We show that, after several modifications, the 3D algorithm can increase the speed of an already performant join implementation.

Additionally, we introduce a method for selecting the optimal hash directory size in a standard chained hash join implementation. Using this technique, we can improve the performance of joins with duplicate keys, while significantly reducing memory usage.

The remainder of this paper proceeds as follows: In Section 2 we will discuss the Umbra and 3D Hash Joins, as well as the problem of joining with duplicate keys. Then in Section 3, we describe our implementation, in particular the 3D+Umbra parallel insertion algorithm. Following this, in Section 4, we evaluate the performance of our implementation and show how to optimize the hash directory size. Finally, we cover related works and our conclusions, in Sections 5 and 6 respectively.

2 BACKGROUND

This paper is on practical techniques for improving state-of-the-art hash joins. As such, our baseline hash join implementation must be highly performant; we base it on the hash join of Umbra [5]. We start this section by describing several novel optimizations in the Umbra hash join and hash table. Despite these optimizations, like all separate chaining hash tables, the Umbra hash table is subject to the issues of duplicate keys. After describing this issue in the depth, we delve into the 3D Hash Join [3] which provides a means to mitigate this problem.

2.1 Umbra Hash Join

Umbra uses a Hash Join with a separate chaining hash table [2, 5, 7]. The join works as follows. The build side of the query is materialized, that is, it is spooled out in memory. During materialization, the tuples are padded with extra space for the key, hash value, and a pointer. This pointer is used to build the collision list.

After materializing the build side tuples, the hash directory is allocated; it consists of an array of pointers. This step is deferred until after materialization, so that the number of build tuples is known, and the hash directory can be sized accordingly. The hash directory size is set to a value slightly greater than the number of build tuples.

At this point, the hash table is constructed. The materialized tuples are traversed and inserted into the table. This happens in parallel, using as many threads as are available. For a given tuple, its key is hashed and the hash value is stored in the pre-allocated spot adjacent to the tuple. Then the hash value is mapped to a spot in the hash directory.

2.1.1 Hash Value Range Mapping. The hash value must be translated to an index in the hash directory. This entails mapping the 64-bit hash value into the range $[0, n)$ for a hash directory of size n. Traditionally, this is done by dividing the hash value by n and using the remainder as the bucket index; but this method is quite slow. Hence Umbra uses a well-known optimization: right shifting by k bits is equivalent to taking the remainder when dividing by 2^k. If the hash directory is 2^k for some k, the expensive remainder operation can be replaced with a cheap shift operation. Of course, this approach has the downside that the size of the hash directory must be a power of 2, potentially wasting a great deal of memory.

Another efficient range mapping approach is that of [6], which requires only a single multiplication and shift operation. We use this technique throughout much of the paper, as it allows arbitrary hash directory sizes.

2.1.2 Parallel Insert. Once the hash value is mapped to a specific bucket in the hash directory, it is inserted at the front of the bucket's collision list. This entails 1) setting the *next* pointer, which was allocated adjacent to the tuple during materialization, to the current first node of the collision list, and 2) setting the pointer in the hash table bucket to the new node. As the table is built in parallel, these updates must be thread-safe. Thus step 2 is performed using an atomic compare and swap instruction, which verifies that the current front of collision list has not changed since setting it to *next* in step 1. Using fine-grained atomic instructions for insertion allows a high degree of parallelism.

2.1.3 Bloom Filter Tagged Pointers. The keys in each collision list are maintained in a Bloom filter. During the probe stage, if a key is absent from the Bloom filter, the collision list does not need to be traversed. The Bloom filters consist of 16 bits stored in unused bits of the pointers in the hash directory. Since the hash directory pointers are updated with atomic compare and swap instructions, the Bloom filters are likewise updated atomically during linked list insertion.

2.1.4 Probe Stage. The probe stage iterates over the probe tuples, hashes their keys, and looks up keys in the hash table. Before iterating down a collision chain, the probe stage checks the Bloom filter in the pointer at the head of the list. If the hash is not present in the Bloom filter, the key cannot be present in the collision list, and the expense of iteration can be avoided. If the hash is in the Bloom filter, the collision list is traversed to find any matches. If it is known that the build side has unique keys, the list only needs to be traversed until the first match; otherwise, the list must be traversed until the end is reached. For each match in the collision list, the build tuple and probe tuple are emitted to the output.

2.2 Duplicates in Hash Joins

Despite the effectiveness of hash joins, they have a weakness—build side duplicates. Tuples on the build side with the same key will be hashed to the same bucket and added to the same collision list. If the collision list only contains tuples for a single key, this does not pose a problem. This is because for a given probe key all matching build tuples must be emitted, thus the list of tuples with duplicate keys must always be traversed. But if a collision list contains tuples for more than one key, there is a potential for inefficiency. For example, consider a collision list with two keys, one with a single tuple and one with many associated tuples. When probing for the single tuple, we need to traverse an arbitrary number of unrelated tuples. The linked list traversal consists of pointer dereferences—expensive operations if the build side does not fit in the cache.

This can cause a significant slowdown, thus finding ways to deal with duplicates could improve hash join performance.

2.3 3D Hash Join

A recent paper by Flachs et al. [3] proposed an optimization to help mitigate the negative effects of duplicate keys in hash joins. Rather than using a single linked list per hash bucket, they propose a list of sub-lists, where each sub-list consists of all tuples which share a given key. Thus the main collision list has a length equal to the number of unique keys. With this structure, when more than one keys collide in the same hash bucket, probes with one key can avoid iterating past multiple tuples of a different key.

2.3.1 3D Hash Directory & Node Structure. A diagram showing the structure of the 3D hash table can be seen in fig. 1b. It shows a single collision list with five tuples, but only two unique keys. The main list thus has two main nodes, one with two tuples in its sub-list and one with three tuples.

Unlike Umbra, the 3D hash directory consists of the first node of each collision list, rather than pointers to collision lists. This is possible because 3D nodes contain pointers to their associated tuples, rather than the tuples themselves. The main nodes along the collision list require two pointers, one to the next node in the main collision list, and one to the sub-list of tuples sharing the same key. Nodes in the sub-list only need one pointer to the next node in the sub-list. Additionally, since the tuples in a sub-list share the same key, the associated hash value will be equal for all duplicate nodes. Thus, to save space, hash values are not included in sub-nodes.

The key value is not stored in the node, as it can be checked through the tuple pointer. With 64-bit hash values, it is rare that two different keys share the same hash value, thus hash value equality can be used as a proxy for key equality. Using hash equality means that tuples in a given sub-list share the same hash value, but may in fact have different key values. Of course, the actual keys must be checked for equality when emitting tuples. Though this change does make the 3D collision lists less effective, it is more efficient overall, as operations on 64-bit hash values are likely cheaper than on arbitrary key types. We adopt the same behavior in all hash join implementations in this paper, and use the terms *hash* and *key* largely interchangeably.

2.3.2 Insertion. After finding the correct hash bucket, the 3D insertion algorithm traverses the main collision list searching for a main node with a matching hash value. If such a node is found, the new tuple is inserted in the sub-chain of this main node. If no matching main node is found, the new tuple is inserted as a main node at the end of the main collision list.

The need to traverse the main collision list when inserting new keys is a potential downside of the 3D algorithm. Umbra's hash table insertion algorithm can insert at the beginning of the list since new tuples do not need to be aggregated with existing tuples sharing the same key. Though the main collision list should be short, this additional iteration has a non-zero cost.

2.3.3 Probe. The probe stage is where the benefits of the 3D algorithm become apparent. The main collision list will be quite short, so traversing it during probe can be a significant speedup compared to a standard collision list containing lots of tuples. Additionally, once a matching main node is found, its sub-nodes can be easily emitted to the output without the need to filter them from other non-matching nodes.

2.3.4 Deferred Unnesting. In addition to reducing iteration during probe, the 3D hash table design allows another optimization called deferred unnesting. When there are multiple joins on the same key, deferred unnesting allows joins higher in the join tree to operate on an iterator of tuples rather than tuples themselves. Such an iterator is simply a sub-chain of tuples from a collision list lower down in the join tree. Though this technique shows promise, we will not be investigating it in this paper. We will be focusing on the 3D hash table's ability to reduce the cost of iterating over collision lists containing duplicate keys.

3 COMBINING UMBRA & 3D

This section is the main contribution of our paper. Here we show how the Umbra and 3D algorithms described in the previous section can be combined. We start by describing the hash directory and node structure of our 3D+Umbra algorithm. Following this, we explain our insertion algorithm and argue that it is correct and thread-safe. Then we show how Umbra's Bloom filters and atomic pointers increase the efficiency of the 3D algorithm. Lastly, we explain the need to estimate the number of unique keys in a given join, and describe how the HyperLogLog algorithm was used to perform this estimation efficiently.

3.1 Hash Directory & Node Structure

Combining the techniques used by Umbra with the 3D hash join requires a number of modifications to the 3D node and directory structure. Umbra uses pointers as the values in the hash directory, whereas 3D uses inline hash table nodes. We use a directory of pointers, as we do not know the number of build tuples while materializing, thus cannot allocate a contiguous piece of memory of the required size.

Similarly, the 3D hash table is able to save space by only storing a duplicate hash once at the head of its sub-list. All subsequent tuples for this hash are stored in smaller nodes which only need space for the tuple and the sub-list pointer. But before the table is built, it is not known which of the duplicate tuples for a given hash value will be inserted first. Thus we do not know which of the tuples will be in the main collision list and will thus need space for a main node's hash value and next pointer. Unfortunately, this means every node must have enough space to be a main node. An example of this can be seen in fig. 1c, where every sub-node has unused space for a hash value and next pointer.

3.2 Insertion Algorithm

The insertion algorithm can be seen in algorithm 1. We start by converting the hash value into a hash directory index and getting the associated pointer from the hash directory. In line 5, we check the Bloom filter in the bucket pointer to see if the hash may already be present in the bucket. If the hash is not present, we attempt to atomically insert the node at the head of the linked list, in line 8. We simultaneously update the Bloom filter as it is part of the pointer. If the insert succeeds, we are done.

Algorithm 1 Umbra+3d Hash Table Insertion

```
1  insertNode(hash, node):
2    slot := table[hash % size(table)]
3    currSlot := slot
4
5    while hash not in bloomTag(slot):
6      newSlot := node | bloomTag(slot) | bloomUpdate(hash)
7      // insert start main chain
8      if slot.CAS(currSlot, newSlot):
9        node->next := ptr(currSlot)
10       return
11
12   prev := null
13   curr := ptr(currSlot)
14   while True:
15     while curr not null:
16       if curr->hash == hash:
17         // insert in sub-chain
18         node->sub := curr->sub.exchange(node)
19         return
20       prev := curr
21       curr := curr->next
22
23     // insert end main chain
24     if prev->next.CAS(curr, node):
25       return
```

Otherwise, the compare and swap failed due to another node being inserted first. This node may have had the same hash value or a different hash value, so we check the Bloom filter again. If our key is still absent, we know it was a different node that was inserted and try again until we either succeed, or our key becomes present in the Bloom filter (lines 5-10).

If the key is in the Bloom filter, either the key is really present in the bucket, or there was a Bloom filter false positive. Either way, we must iterate until we find a node with a matching hash value, or reach the end of the bucket list. Lines 14-25 consist of this collision list traversal. If we find a node with a matching hash value, we insert at the head of this node's sub-chain in line 18.

Otherwise, we iterate to the end of the main collision list and append a new main node at the end of the list using an atomic compare and swap in line 24. This append could fail if another node was appended before our new node was appended. In this case, we repeat lines 14-25, until we find a node with a matching hash value or append at the end of the main list.

3.2.1 Benefits of Bloom filter & Atomic Pointers. The Bloom filters provide an opportunity to improve an area of inefficiency in the 3D algorithm. On every insertion, the main collision list must be traversed until a main node with the matching key is found, or the end of the collision list is reached. This is a downside of 3D, as with a standard hash table a new node can be inserted at the head.

The Bloom filter allows us to avoid this traversal in most cases. If the key is not present in the Bloom filter, we know that there is no main node in the collision list with the same hash. Hence we can insert at the beginning of the list. This relies on the fact that the Bloom filter is updated with a compare and swap along with the rest of the containing pointer. If another node with the same hash has been added between checking the Bloom filter and inserting at the head, the CAS instruction would fail.

It is fortuitous that the Bloom filter is updated atomically with the pointer, as otherwise, insertion at the front of the collision list would not be safe.

3.3 Counting Unique Keys

It is useful to know the number of unique keys in the build relation, as this value is used to determine the appropriate hash directory size. Unfortunately, this value may not be available. In a realistic database system, if the build relation is a base table we may know the actual or estimated number of unique keys. If the build relation is the result of a query, we may still have an estimate of the number of unique keys, but the quality of this estimate could be quite poor.

As we cannot assume that we know the number of unique keys, the 3D+Umbra join algorithm must find this value. But determining the exact number of unique keys in the build relation would require either building a hash set or a sorted list of the keys. Clearly, either option is far too expensive.

Thankfully, the HyperLogLog algorithm of Flajolet [4] provides a fast way to estimate the number of unique keys while using a small, constant amount of space. Thus we maintain a local HyperLogLog counter in every thread during materialization. We also hash the keys during materialization so the hash value can be used in the HyperLogLog algorithm. The hash is stored in the materialized tuple, to be used later during the build phase. At the end of materialization, the HyperLogLog counters can be merged and a final distinct key count estimate is produced.

4 EVALUATION

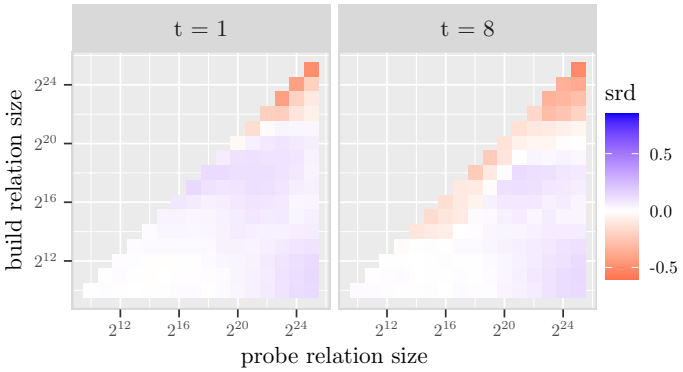
In this section we cover how our 3D+Umbra algorithm was evaluated against the Umbra-style baseline. We start with the experimental structure, which we base on that of [3]. From these experiments, we discovered that the initial hash directory size was too large. Thus in the next section we describe how we rectified this issue by setting the hash directory size from the number of unique keys. Following this is another main contribution of this paper. Here we show how we optimized the hash directory size of the baseline Umbra-style algorithm and increased its performance while significantly reducing memory usage.

4.1 Experiment Structure

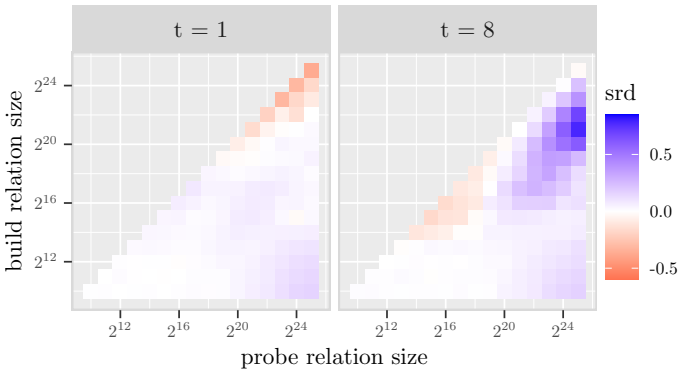
For ease of comparability, we base our experimental evaluation on that of [3]. Though they used both foreign key and many-to-many joins, we restrict our tests to foreign key joins.

4.1.1 Relation Size. The 3D Hash Join was evaluated using various sizes of key and foreign key relations. Each relation was varied between 2^{10} and 2^{25} elements.

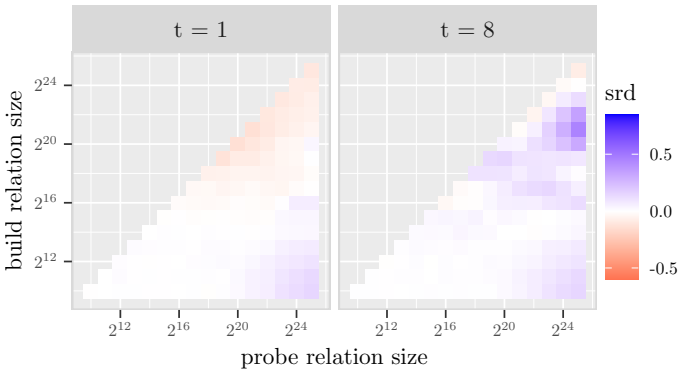
4.1.2 Duplicates. The number of duplicate keys was controlled through a foreign key scale parameter which specifies from what subset of the key relation, the foreign keys are sampled. For a key relation of size $|R|$, and foreign key scale parameter t, the first $\frac{|R|}{2^t}$ elements of R are sampled to obtain the foreign keys. At t-scale 0 the entire key relation is sampled, at t-scale 1 half is sampled, at t-scale 2 a quarter is sampled, etc. The original sampling was done using both uniform and Zipf distributions, though we restrict our tests to uniform sampling.



(a) 3D+Umbra with a *total key*-based directory size versus baseline. This version performs poorly, with a maximum *srd* of only 0.13 and a minimum of -0.51.



(b) 3D+Umbra with a *unique key*-based directory size versus baseline. This version has a better maximum *srd* of 0.77, but still a minimum of -0.37.



(c) Umbra with a unique/total key-based directory size versus baseline. Though the maximum *srd* is only 0.43, the minimum is decent, with a value of only -0.14.

Figure 2: Comparison of join implementations with a baseline consisting of an Umbra-style join implementation using a total key-based directory size.

4.1.3 Relation Order. In the 3D paper, tests were run with both the key and foreign key relations being the build side of the join. Since both relations varied over the range of relations size, this means that the larger relation was tested as the build side. In contrast, we only consider the smaller relation for the build side. This is due to a pragmatic concern—before a join we may not know if one of the sides of the join has duplicate values. Since we cannot assume that we know, it is safer to always use the smaller relation for the build side.

4.1.4 Early Termination. Additionally, we only consider the foreign key relation, being the relation with duplicates, for the build side. This is based on the assumption that a regular hash join will perform better when the build side does not contain duplicates. If the build side has no duplicates, we can likely infer this fact, in which case we can stop probing after a single match. The tests in fig. 4 of [3] support this viewpoint, thus we chose to focus our testing on foreign key build sides.

4.1.5 Performance Metric. We also adopt the performance metric used by [3], the symmetric relative difference of join times. This function is defined as $srd(b,t) = \frac{b-t}{min(b,t)}$, where b is the join time of the baseline implementation, and t is the join time of the test implementation. A positive *srd* is equivalent to the $speedup - 1$, and a negative *srd* to $1 - slowdown$.

4.2 Hash Directory Size Effect

The evaluation of the initial version of our 3D+Umbra hash join can be seen in fig. 2a. This can be compared with the evaluation of the 3D hash join in fig. 4 of [3].

Unfortunately, these initial results show little improvement over the baseline.

A notable difference between the initial 3D+Umbra version and the 3D version is the size of the hash table used. The original 3D hash join uses a hash directory size equal to the number of unique keys, whereas the 3D+Umbra version uses a hash directory size based on the number of the total tuples in the build relation. This is because the first 3D+Umbra version did not include the HyperLogLog unique key count logic described in section 3.3, and thus had no way of determining the number of unique keys.

As the hash table size was the main difference from the better-performing 3D Hash Join, we tested the 3D+Umbra version using a hash directory size equal to the unique key count; a value we only knew due to having designed the test cases. As this showed promise, we implemented the HyperLogLog counters to estimate the number of unique keys.

The value produced by HyperLogLog is an estimate and contains some expected error. We found that a too-small hash directory performed worse than a too-large hash directory. Thus, to be safe, we inflate the estimated unique count by a constant of 1.5 to obtain the hash directory size.

The updated version of 3D+Umbra using HyperLogLog key count estimation and unique key-based hash directory size can be seen in fig. 2b. This version performs much better than the original version, despite the overhead of the distinct key estimation.

The fact that the directory size had a large effect on the performance of the 3D+Umbra implementation leads naturally to the idea

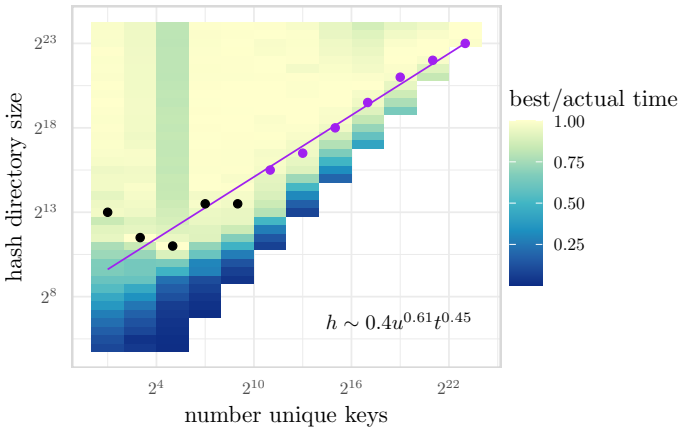


Figure 3: Regression for optimal hash directory size from the number of unique keys and total tuples. The data points used for regression are the smallest hash directory sizes with join times within 5% of the best time for the same input relations.

that the chain hash table's directory size may not be optimal either. We investigate this in the next section.

4.3 Optimizing Hash Directory Length

Given that the size of the hash directory plays a pivotal role in the performance of the 3D+Umbra Hash Join, it is worth reconsidering the choice of hash table size for the standard Umbra-style chained hash join. The baseline version uses a hash directory size equal to the number of build tuples multiplied by a constant of 1.5. We ran the following tests to find the optimal hash directory size.

4.3.1 Test setup. We use equal size build and probe relations, varied in sizes between 2^{15} and 2^{25}, increasing by multiples 4. The data generation is similar to the design of the foreign key tests, with the probe relations using unique keys and the build relation keys taken from a subset of the probe relation key range. As opposed to the previous tests, the number of unique key values in the build relation was varied in the tests, between 2 and the number of total keys. To achieve this, each key in the build relation was duplicated $d = \frac{t}{u}$ times, for u the number of the unique keys, and t the number of total keys. In the cases where this was not an integer, the number of duplicates was varied between $\lfloor d \rfloor$ and $\lceil d \rceil$ in proportion so that on average the correct number of duplicates would be attained.

The hash directory size was also varied between the number of unique keys and the number of total keys.

4.3.2 Modelling Hash Directory Size from Unique/Total keys. By benchmarking joins for each combination of unique key count, total key count, and hash directory size, we find the optimal hash directory size for each total/unique key configuration.

We select the best hash directory size as the smallest hash directory with join performance within 5% of the fastest time for a given total/unique key configuration. These points appeared to follow a line in the log-log plot of hash directory size versus unique key; though points with a unique key count less than 2^{10} were noisy and did not follow this trend. As these latter configurations seemed less realistic, we removed them before running log-log regression. This resulted in the relationship $h \sim 0.4u^{0.61}t^{0.45}$, for h the hash directory size and u and t as defined above. This fits intuition as it is nearly the geometric mean of the total and unique keys, multiplied by a constant.

Using this fitted curve, we have a simple model to choose a suitable hash directory size. In practice, we use a much larger constant of 1.5, and use an upper bound for the maximum hash directory size of $1.5t$. Results for the standard chained hash table join using this hash directory sizing model can be seen in fig. 2c. Though it does not perform as well as the 3D+Umbra Hash Join, given its simplicity, it is worth considering in a hash join implementation.

We used a similar analysis to find the optimal hash directory size for the 3D+Umbra hash join, but found that using a size based on the number of unique keys, as in [3], did indeed perform the best.

4.4 Effects of Hash Table Collisions

Hash tables are widely used as the expected runtime of their insert and lookup operations is $O(1)$. Unfortunately, their worst-case performance is $O(n)$, as resolving hash collisions requires inspecting multiple keys.

Though 64-bit hash functions do have collisions, there are rare enough that they can often be ignored. But a 64-bit hash must map to a location in the hash directory, which must be small enough to fit well in memory and caches. A variety of mapping functions are used, as described in section 2.1.1, but all result in hash bucket collisions.

Separate chaining and open addressing are two techniques used to handle such collisions; both Umbra and 3D use the former technique. Separate chaining tables are easier to build in parallel, and unlike open addressing tables, allow different-sized values may be held directly in the hash table [5]. Despite the benefits of chained hash tables, resolving collisions is expensive as it requires traversal of linked lists; this entails dereferences and the associated random-access reads.

The issues of collisions are exacerbated by duplicate keys. These increase the length of collision lists, and unlike collisions, there is no way to reduce the number of duplicates.

4.4.1 Optimizing costs due to hash collisions. There are three aspects of the 3D+Umbra algorithm which control the costs of hash collisions. These are 1) the 3D structure, 2) Bloom filter pointer tags, and 3) the size of the hash directory.

The *3D Hash table algorithm* can be seen as a method to reduce the cost of hash collisions. As mentioned previously, if a slot has only one key with many duplicate tuples, the tuples must be traversed when output. In this case, the 3D algorithm does not reduce the amount of required iteration. 3D becomes beneficial when multiple keys with duplicates collide in the same slot. In this case, the 3D algorithm avoids iteration over tuples with keys different from the current probe key.

Umbra's *Bloom filters* also reduce the cost of hash collisions. The Bloom filters allow the probe algorithm to avoid traversing the collision list when the key is not present in the Bloom filter. If the key is present, whether in actuality or due to a false positive, the collision list must be traversed in search of matching nodes. If the collision list is traversed and keys have duplicates, unlike the 3D

algorithm, Umbra must iterate through each of the duplicate nodes. But in many cases the Bloom filters allow collision list traversal to be avoided entirely.

The *size of the hash directory* also affects the costs of collisions by reducing the number of collisions. Assuming there are no collisions of the underlying hash function, the expected number of unique keys per non-empty hash bucket is $\frac{n}{m \cdot (1-(1-1/m)^n)}$ where m is the size of the hash directory, and n is the number of unique keys [3]. By increasing the size of the hash directory, the expected number of unique keys per non-empty bucket can be made arbitrarily close to 1. Though increasing the hash directory size reduces collision, it comes at the cost of using additional memory. As the hash directory is accessed with a random access pattern, minimizing its size to promote good caching is pivotal.

We can see the effects of each of these factors in fig. 4. Here we see a comparison of the performance of joins while varying each of the above three factors. The probe relation size is held constant at 2^{25}, while build relation size is varied between 2^{10} and 2^{25}. The baseline configuration is the Umbra-style chained hash join with Bloom filters and hash directory size based on the total number of tuples.

Several aspects of the plot are of note. First, all configurations without Bloom filters perform notably worse than the same configuration with a Bloom filter. Despite this, the 3D versions without Bloom filters perform fairly well at high build relation sizes. The 3D version with hash directory size based on the number of total keys has similar performance to the baseline at low build relation sizes but drops in performance at high build sizes. The best version is the 3D algorithm with Bloom filters and directory size based on the number of unique keys, though the chained version with Bloom filters and unique key-based hash directory size performs decently as well.

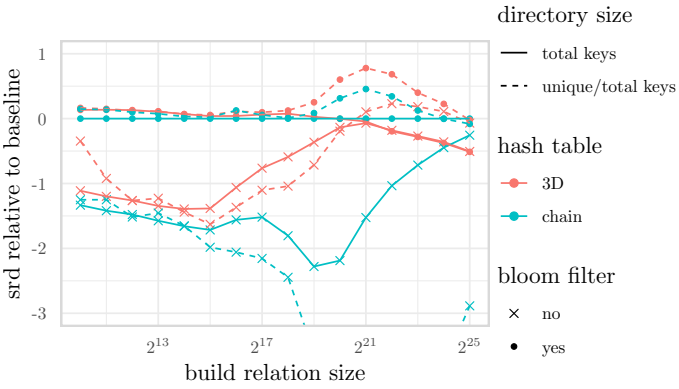


Figure 4: Comparison of three factors affecting join performances. Probe relation size is 2^{25} and t-scale parameter is 8 (256 maximum expected duplicates).

5 RELATED WORKS

This works rests on the significant existing work on hash tables and hash joins. We have previously discussed this is based on: the 3D Hash Table Join [3], and the morsel-drive parallelism of Umbra [5].

This work assumes that joins are small enough to avoid the need for partitioning. Bandle et al. discuss when to use radix partitioning in Umbra [2]; a real system employing the work in this paper would likely use radix partitioning for large joins.

We discussed above the optimal hash directory size relative to the number of unique and total keys. This closely relates to the notion of load factor in hash tables. Load factor is one of several factors affecting the performance of Hash Joins analyzed by Richter et al [8]. They also consider other types of hash tables, in particular open addressing tables using various probing schemes.

6 CONCLUSION

In this paper, we investigated techniques for handling duplicate keys in hash joins. Specifically, we looked at the marriage of two existing algorithms: the 3D Hash table joins of Flachs et al [3], and the parallel hash join implementation of Umbra [2, 5].

We evaluated a hash join combining these algorithms and found that the 3D structure, Bloom filter pointer tags, and an increased hash directory size have a similar effect of reducing the cost of duplicates and collisions. Since the combination of 3D structure and Bloom filters reduced this collision cost significantly, we found we could decrease the hash directory size to around the number of unique keys, without significantly adversely affecting the cost of collisions/duplicates. This decrease in hash directory size improves the algorithm's speed, likely due to improved cache usage. To size the hash directory based on the number of unique keys we added a HyperLogLog sketch to estimate this value during tuple materialization.

Additionally, we found that reducing the hash directory size to a function of both the unique and total key sizes improves the performance of the chained hash join. We found a suitable function by regressing the relationship between unique keys, total keys, and the best directory size per join configuration.

Our final results are two algorithms—Umbra+3D, and an Umbra join with unique key-based directory size—both of which help mitigate the negative effects of duplicates in hash joins.

REFERENCES

[1] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96.

[2] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event , China.*

[3] Daniel Flachs, Magnus Müller, and Guido Moerkotte. 2022. The 3D Hash Join: Building On Non-Unique Join Attributes. In *12th Annual Conference on Innovative Data Systems Research (CIDR '22), January 9–12, 2022, Chaminade, USA.*

[4] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science.* Discrete Mathematics and Theoretical Computer Science, 137–156.

[5] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.*

[6] Daniel Lemire. 2016. A fast alternative to the modulo reduction. https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/. Accessed: 2023-02-01.

[7] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR.*

[8] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB* 9, 3 (2015), 96–107.