The 3D Hash Join: Building On Non-Unique Join Attributes

Daniel Flachs
flachs@uni-mannheim.de
University of Mannheim
Mannheim, Germany

Magnus Müller
magnus@uni-mannheim.de
University of Mannheim
Mannheim, Germany

Guido Moerkotte
moerkotte@uni-mannheim.de
University of Mannheim
Mannheim, Germany

ABSTRACT

One of the most prominent ways to evaluate an equi-join is based on hashing. We consider the problem of non-unique join attributes on the build side. In conventional hash tables where collisions are resolved by chaining, duplicates inevitably lead to long collision chains. This causes a high number of expensive main memory accesses and join predicate evaluations during the probe phase, increasing the runtime of the overall join. A related problem occurs when the query optimizer cannot determine the uniqueness of the join attribute of the build side.

We present the *3D Hash Join* to efficiently evaluate main-memory hash joins in the presence of duplicate build keys and skew. The main idea is to cluster the hash table collision chains based on the distinct values of the build attribute. We further introduce a technique called *deferred unnesting* to speed up the evaluation of multiple joins. In an experimental comparison with an implementation of a chaining hash table, our approach achieves a speedup of up to a factor of 3.53 for a single key/foreign key join and 5.67 for many-to-many joins.

KEYWORDS

hash join, hash table, skew, query processing, physical algebra

1 INTRODUCTION

Evaluating joins efficiently is a key performance goal of every relational database system. Hash joins have been around since the 1980s and have been studied extensively since, resulting in a multitude of different implementations [3, 6, 7, 11, 15, 18, 22]. However, the question for the best join implementation cannot be answered in general, as experimental studies have shown that there is no "one size fits all", depending on factors like data skew [27]. Hence, join algorithms remain a relevant research topic.

A hash join consists of two consecutive phases: building a hash table on the build relation, and then probing it with tuples from the probe relation. Database lore dictates that the smaller input is chosen as the build side. If the smaller relation is the one containing the foreign key of a key/foreign key join, duplicates occur on the build side. For many-to-many (N:M) joins, there is no way to avoid the non-uniqueness of the build side. A non-unique build side inevitably leads to collision chains growing proportionally to the frequency of the respective keys. During probe, this causes a large number of main memory accesses and join predicate evaluations when traversing the collision chains, resulting in high processing costs for probing.

A related problem occurs if the uniqueness of the join attributes in the build relation is not known at query compilation time. This happens if the *known functional dependencies* specified in the SQL standard do not allow to derive uniqueness.

In order to address non-unique build sides, we propose a simple but effective hash table organization, the *3D Hash Table*, that groups together equal keys, and a corresponding hash join algorithm, the *3D Hash Join*. "3D" refers to the three-dimensional organization of the hash table: (1) a hash directory, (2) main collision chains, and (3) sub chains beneath the main collision chain nodes. We also demonstrate the effectiveness of a new technique called *deferred unnesting*.

The remainder of this work is structured as follows: Section 2 briefly describes the standard hash table organization schemes and why duplicates are bad for them. Section 3 presents related work. We describe our approach and its main properties in Section 4 and evaluate it experimentally in Section 5. Section 6 concludes the paper.

2 THE UGLINESS OF DUPLICATES

Hash tables are unordered, associative arrays that store key/value pairs. In the context of main-memory hash joins, the keys are join attribute values, and the values are usually pointers to tuples or tuple identifiers (TIDs). The hash table's *hash directory* is an array of size m. Each directory entry (or *bucket*) points to the head node of an initially empty linked list, the *collision chain*. We refer to this organization as the *chaining hash table*. To insert a key/value pair (k, v) into the hash table, a hash function h maps k to a bucket $i \in \{0, \ldots, m-1\}$ where (k, v) is inserted into the respective linked list. List insertion is performed at the head, ensuring time complexity $O(1)$. We say that two distinct keys k, k' *collide* under h iff $h(k) = h(k')$. Colliding keys end up in the same bucket and, thus, in the same collision chain. To look up a key k, we compute the corresponding bucket index $h(k)$, and examine the nodes in the respective collision chain. We must check the key of each node, since there might be keys $k' \neq k$ present in the collision chain that just happen to collide with k. If there are no duplicate keys, we can terminate the lookup after the first (and only) matching key (*early termination*). Otherwise, or if k is not present in the table, we must traverse the collision chain until the end. This also holds if the uniqueness cannot be inferred.

In the presence of duplicate keys and especially a skewed distribution of the keys, collision chains of certain buckets containing frequent keys become very long. The performance problem arises during the probe phase: We must traverse those long collision chains, which causes numerous expensive main memory accesses and predicate evaluations for each and every collision chain node.

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well as allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2022.
12th Annual Conference on Innovative Data Systems Research (CIDR '22), January 9–12, 2022, Chaminade, USA

An alternative to hashing by chaining is open addressing (OA), which unfortunately still does not solve the problem. In open addressing, all data is stored in the hash directory. Therefore, an OA hash table can never store more than m entries. In case of a collision, an alternative bucket is searched according to a probing sequence. Robin Hood hashing [9] is an OA variant where keys are rearranged based on the length of their probing sequence. Our problem remains: in case of duplicates, probing sequences become very long.

Cuckoo hashing [23] is another OA variant. Inserting a key into an occupied bucket moves the key that is already present in that bucket to a second hash directory using a second hash function. The original paper suggests using two hash directories, but an extension to k directories is possible [26]. However, the implications of duplicate keys are rather severe.

3 RELATED WORK

This section is divided into (1) hash table organizations for joins (such as our own) and (2) orthogonal techniques. In general, we assert that building the hash table on the non-unique side of a join has never been reflected in the hash table organization itself.

3.1 Hash Table Organizations for Joins

The application of hashing for joins in database systems dates back to disk-based systems in the 1980s [8, 11, 14, 16].

A decade later, Shatdal et al. [28] propose and evaluate a main-memory hash join algorithm using a chaining hash table, analogous to the disk-based GRACE hash join [16].

Manegold et al. [22] use a chaining hash table as the underlying data structure for their radix-partitioning hash join algorithm. Kim et al. [15] build upon this partitioning approach, but use histograms and reordering of build tuples as their underlying data organization. This is comparable to a chaining hash table where all buckets are stored contiguously in memory. They mention duplicate build keys as a challenge, but do not investigate this further.

Blanas et al. [7] propose a simple, non-partitioning hash join algorithm with a shared chaining hash table in a multi-core environment. Balkesen et al. [4] base their implementations on a chaining hash table. The buffered non-partitioned hash join by Bandle et al. [5] uses a global chaining hash table.

Other approaches use open addressing hash tables, e.g., Barber et al. [6]. Lang et al. [18] use a lock-free open-addressing hash table for their NUMA-aware hash join.

The hash join algorithm by Zukowski et al. [30] is based on cuckoo hashing. "Problematic" keys in terms of collisions are stored in a single (!) separate linked list. A recent approach by Li et al. [21] also uses cuckoo hashing. However, a large number of duplicates on the build side can either not be handled, or would require a large number of (mostly empty) hash tables.

3.2 Orthogonal Techniques

The following techniques are orthogonal to the hash table organization and can be applied additionally.

The techniques by Shatdal et al. [28] for an efficient and cache-conscious implementation of query processing algorithms are generic and, thus, generally applicable.

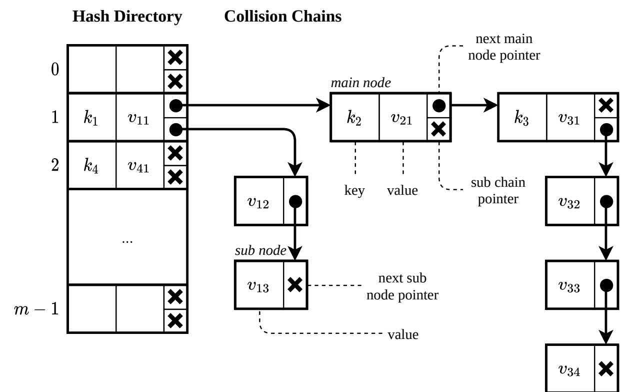


Figure 1: Layout of the 3D hash table: hash directory, main collision chains, and sub chains

Due to the random-access behavior of hash tables, **prefetching** can hide memory latencies [10, 15, 17]. Another optimization approach is **filtering**. For selective joins, non-qualifying tuples can be filtered out early using Bloom filters and/or semi-join reducers [5, 24]. **Partitioning** is a technique that is applied to the inputs of a join before evaluating it partition by partition. Partition sizes are chosen such that a partition fits into the cache, which reduces cache and TLB misses [5, 15, 22, 28]. **Parallelization** is concerned with load balancing, thread allocation and synchronization. Closely related is the notion of **NUMA-awareness** of implementations. A lot of work has been done in this area [1, 5, 12, 19]. There are also publications that **benchmark** and compare different approaches, e.g., Richter et al. [26], and Schuh et al. [27].

4 THE 3D HASH JOIN

The 3D Hash Join consists of a 3D hash table as its underlying data structure, and two physical algebra operators: *join* and *unnest*. This section introduces them, outlines the main properties of our approach, and points out some implementation-specific details.

4.1 3D Hash Table Design

Our 3D hash table aims to overcome the problem of long collision chains by organizing each bucket in a hierarchical fashion: Instead of a single, flat, linear linked list per bucket, we differentiate *main collision chains* and *sub chains*, each with its own node type, *main nodes* and *sub nodes*, respectively. The general layout is visualized in Figure 1. While a chaining hash table maintains one node for each inserted key/value pair[1], we maintain one *main* node for each inserted *distinct* key. As soon as the same key k is inserted for the second time, the respective value is stored in a sub node in the sub chain branching off from the main node for k. Hence, only main nodes store keys *and* values, sub nodes only store values, as they belong to exactly one main node with its respective key.

The main motivation for this design is to ensure short collision chains and, therefore, fewer main memory accesses and join predicate evaluations during probe.

[1] Recall that a value is a tuple pointer or a tuple identifier (TID).

Algorithm 1 3D Hash Table Insert

```
1:  function INSERT(H, h, k, v)
2:    Input: Hash table H, hash function h, key/value pair (k, v)
3:    dir_entry ← H[h(k)]
4:    if (dir_entry.EMPTY()) ∨ (dir_entry.key = k) then
5:      INSERTATMAINNODE(dir_entry, k, v)
6:    else                ▷ insert into some main collision chain node
7:      main_node ← dir_entry.next
8:      while main_node ≠ null do
9:        if main_node.key = k then
10:         INSERTATMAINNODE(main_node, k, v)
11:         return
12:       end if
13:       main_node ← main_node.next
14:     end while
15:     CREATENEWMAINNODE(dir_entry, k, v)
16:   end if
17: end function
18: function INSERTATMAINNODE(node, k, v)
19:   Input: main collision chain node node, key/value pair (k, v)
20:   if node.EMPTY() then
21:     node.key ← k; node.value ← v
22:   else
23:     INSERTINTOSUBCHAIN(node, v)   ▷ linked list insert (head)
24:   end if
25: end function
```

Algorithm 2 3D Hash Table Lookup

```
1:  function LOOKUP(H, h, k)
2:    Input: Hash table H, hash function h, key k
3:    Output: Pointer to the main node corresponding to k
4:    main_node ← H[h(k)]                    ▷ directory entry
5:    repeat                    ▷ traverse main collision chain
6:      if main_node.key = k then break
7:      main_node ← main_node.next
8:    until main_node = null
9:    return main_node
10: end function
```

The insert operation for a key/value pair (k, v) is slightly more complex than for the chaining hash table since we must first determine if k is already present. The procedures are outlined in Algorithm 1. We first check if the directory entry corresponding to k is empty or if the key stored there matches k. If so, we call INSERTATMAINNODE that either stores k and v in the empty directory entry or inserts v into the entry's sub chain. If the directory entry is neither empty nor has the same key, we traverse the main collision chain of this bucket (ln. 8) until we find a main node with the same key and insert v into its sub chain. If such a node is not found, k is not yet present in the table, and we append a new main node storing k and v to the main collision chain (ln. 15). The functions INSERTINTOSUBCHAIN and CREATENEWMAINNODE are not given here since they are simply insertions into linked lists (sub chain and main collision chain, respectively). Note that for all operations, the key k can be more complex than a scalar attribute value. We can easily use a multi-attribute key as long as the hash function accepts it as input.

Algorithm 3 Unnest

```
1:  function UNNEST(X)
2:    Input: Set X of nested tuples of the form [r, B]
3:    Output: Set of unnested result tuples
4:    Result ← ∅
5:    for each [r, B] ∈ X do
6:      for each s ∈ B do
7:        Result ← Result ∪ {[r ∘ s]}
8:      end for
9:    end for
10:   return Result
11: end function
```

The operation to look up a given key k is outlined in Algorithm 2 and is essentially identical to the lookup operation in a chaining hash table without duplicate keys: Using h, we compute the bucket index corresponding to k and examine the keys in the directory entry and the main collision chain nodes until either k is found or there are no more nodes to examine. In case of a successful search, we return *a pointer to the main node* (!) to the caller, who can then decide to either "unpack" all the matches by traversing the sub chain, or just leave the search result in its packed form. We call the packed search result a *nested tuple* and the process of unpacking it *unnesting*, see Section 4.3.

4.2 3D Hash Join

When computing the join of two relations R and S with some equality predicate p, e.g., $R.x = S.y$, we need to distinguish the logical join operator $\bowtie_p$ and its physical counterparts. We denote the standard hash join operator using a chaining hash table by $\bowtie_p^{\text{hj}}$ and our 3D hash join operator based on the 3D hash table by $\bowtie_p^{\text{3D}}$. The input of both physical operators are, in general, two expressions, e.g., two relations R and S. As a convention, we will always build the hash table on the right input, and probe it with the tuples from the left input. With regard to the output, while the standard hash join operator yields the result of $R \bowtie_p S$, the 3D hash join produces *nested tuples* of the form $[r, B]$, where r is a tuple from R (or a pointer or a TID), and, conceptually, B is a list of tuples s from the build side S that fulfill the join predicate $p(r, s)$.

A physical hash join operator typically consists of two consecutive phases: build and probe. In the build phase, each tuple r from the build input is inserted into the hash table, e.g., according to Algorithm 1 for our approach. In the probe phase, for each tuple s from the probe input, the 3D hash join traverses the respective main collision chain and evaluates the join predicate on each main node's key until a match is found or the chain ends. If a match is found, it terminates the search and produces a single nested output tuple, consisting of the probe tuple s and a *pointer to the main collision chain node of the match*. Therefore, the approach produces at most one output tuple per probe tuple.

Algorithm 4 Computing $R \bowtie_p S$ using the 3D hash table and an optional unnest operation.

```
1:  function HashJoin3D(R, S, p, h_R, h_S)
2:      Input: probe relation R, build relation S,
3:              equality join predicate p,
4:              probe and build hash functions h_R, h_S
5:      Output: the join result R ⋈_p S
6:      H ← empty hash table
7:      for all s ∈ S do                                    ▷ Build
8:          key ← ExtractJoinAttributes(s, p)
9:          insert(H, h_S, key, s)
10:     end for
11:     ResultNested ← ∅
12:     for all r ∈ R do                                    ▷ Probe
13:         key ← ExtractJoinAttributes(r, p)
14:         main_node ← lookup(H, h_R, key)
15:         if main_node ≠ null then
16:             ResultNested ← ResultNested ∪ {[r, main_node]}
17:         end if
18:     end for
19:     Result ← unnest(ResultNested) ▷ Unnest (can be deferred)
20:     return Result
21: end function
```

4.3 Unnest Operator

To get the final join result, we need to *unnest* the nested right-hand side B of each nested tuple. To that end, we introduce the *unnest operator* which operates on a set X of nested tuples and "expands" each tuple's right-hand side: $\mu_B(X) := \bigcup_{[r,B]\in X}\{r \circ s \mid s \in B\}$, where $\circ$ denotes tuple concatenation. Hence, the following equivalence holds: $R \bowtie_p^{\text{hj}} S \equiv \mu_B\left(R \bowtie_p^{\text{3D}} S\right)$. The unnest operation is outlined in Algorithm 3. It does not require any further predicate evaluations.

Note that in the physical implementation, B is a pointer to a main node whose sub chain is then traversed by the unnest operator, cf. end of Section 4.1.

The reason why join and unnest are separate operations is that it allows to *defer unnesting*, e.g., until after a second join. This is possible if the predicate of the second join only refers to attributes from the probe side like in $\left(R \bowtie_{R.a=S.a}^{\text{3D}} S\right) \bowtie_{R.b=T.b}^{impl} T$, where *impl* is an arbitrary physical join implementation. Then, unnesting can happen further up in the operator tree. This results in better performance as the second join processes far fewer tuples and, additionally, may eliminate some nested tuples, see Sec. 5.3.

The whole algorithm to compute $R \bowtie_p S$ using the 3D hash table is shown in Algorithm 4. Note that the algorihm can, e.g., be implemented in a push- or a pull-based manner, where tuple processing is typically pipelined.

4.4 Main Properties

This section summarizes the main properties of our approach.

The time complexity of an insert operation is linear in the length of the respective main collision chain as we must first check if the insert key is already present. Once the main node is found, insertion into the sub chain has constant time.

Since keys in the main collision chain are unique, we can always apply early termination and, thus, have to traverse on average only half the main collision chain during a successful probe. Further, the main collision chain is rather short as it only contains hash collisions. As a consequence, we need only few memory accesses and join predicate evaluations. This is visualized in Figure 2.

Last but not least, the 3D hash join allows for deferred unnesting.

4.5 Implementation Remarks

This section outlines deviations between the conceptual description from the previous subsections and the actual implementation.

For the implementation of the main collision chain nodes, there are three alternatives: additionally to the next and the sub chain pointer, (1) store only a tuple pointer, (2) store both the key and the tuple pointer, (3) store the hash value and the tuple pointer. Alternative (1) has low memory consumption, but requires tuple pointer dereferencing and memory access for each join predicate evaluation. Alternative (2) does not have to access the tuple for predicate evaluation, but the size of a node depends on the key, which negatively affects caching behavior, e.g., for composite keys. We chose alternative (3): Nodes remain compact and allow fast comparisons. We can use the hash values for a cheap pre-test during

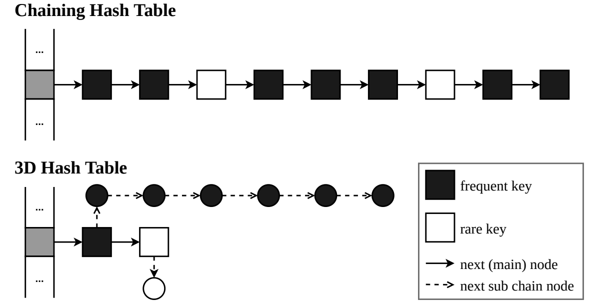


Figure 2: Comparison of rare and frequent keys in both hash table designs. The 3D hash table clusters inserts by key, making the main collision chains short.

Listing 1: C++ structs of main collision chain nodes and sub nodes in the 3D hash table.

```
struct MainNode {
  MainNode*       _next;
  SubNode*        _subchain_head;
  tuple_t*        _tuple;
  hashvalue_t     _hashvalue;
};

struct SubNode {
  SubNode*        _next;
  tuple_t*        _tuple;
};
```

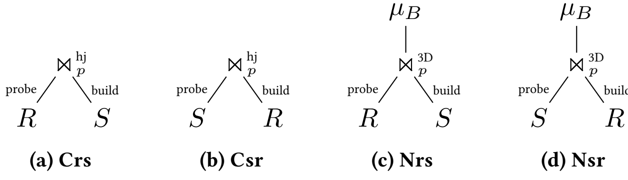


Figure 3: Join Trees (C: standard hash join, N: 3D hash join)

insert (build) and lookup (probe). Only if the (full) hash values match, the tuple is accessed to perform the actual key comparison. Note that we previously stated that the hash function h maps keys to buckets. However, this was simplified: the hash function's codomain is generally larger than the number of buckets m. The bucket index for a key k is calculated by $h(k) \mod m$. Therefore, the nodes within one bucket can have different hash values.

Listing 1 shows the C++ structs for the main and sub nodes. Assuming 64-bit pointers and hash values, the memory consumption of a `MainNode` instance is 32 B, a `SubNode` instance requires 16 B. Common cache line sizes, e.g., 64 B or 128 B, are multiples of these numbers, which ensures proper alignment to cache line boundaries.

The hash directory is implemented as an array of main nodes. This avoids additional cache misses for accessing the first entry of a bucket, since no pointer dereferencing is necessary [26]. Therefore, an empty hash table with m slots already has m main nodes allocated. To distinguish empty from occupied nodes, we use the least significant bit of the pointer to the head of the main collision chain (`MainNode::_next`) as an indicator. This avoids "wasting" a byte of memory for a Boolean member. Lang et al. [18] use a similar approach but set one bit of the hash value instead.

Similar to Richter et al. [26], we use memory pools for main and sub nodes to avoid expensive fine-grained memory allocations on insert. More specifically, main and sub nodes are allocated in chunks of size 1024. Since m main nodes in the hash directory are always allocated, the total memory consumption of a 3D hash table with m buckets built on n entries from d distinct values is at most

$$\begin{aligned} & m \cdot \texttt{sizeof(MainNode)} \\ & + \left\lceil \tfrac{d-1}{1024} \right\rceil \cdot 1024 \cdot \texttt{sizeof(MainNode)} \\ & + \left\lceil \tfrac{n-d}{1024} \right\rceil \cdot 1024 \cdot \texttt{sizeof(SubNode)} \end{aligned}$$

depending on the number of hash collisions of different distinct values.

5 EVALUATION

We conducted an experimental evaluation of our approach and implemented both a chaining and a 3D hash table as well as a small push-based [13] physical algebra with operators for the standard and the 3D hash join in C++.

In the experiments in Sec. 5.1 and 5.2, we evaluated a single join between two row-store relations R and S with two 32-bit integer attributes k and a each. k is each relation's unique key. a serves as the foreign key or as a general join attribute, depending on the experiment. As a hash function, we use the 32-bit finalizer of Murmur3 [2], cf. [26].

Table 1: Key/Foreign Key Experiment Parameters

Parameter	Values	Purpose
$\lvert R\rvert, \lvert S\rvert$	$2^{10}, 2^{11}, \ldots, 2^{25}$	relation cardinality
t	$0, 1, \ldots, 9$	foreign key scale parameter
dist	uniform, std. Zipf	foreign key distribution

There are four possible ways to evaluate the join $R \bowtie_p S$, which are visualized in Figure 3. We encode the join order and implementation with three letters: the first capital letter denotes the hash join implementation (C: standard hash join, N: 3D hash join). The two lowercase letters describe the join order (left: probe, right: build). For instance, "Crs" corresponds to $R \bowtie_p^{\text{hj}} S$.

All experiments were executed on machines with the following specifications: two Intel Xeon E5-2690 v3 processors with twelve cores each, and 256 GB RAM. All code was compiled with clang 11, optimization level `-O3`.

5.1 Key/Foreign Key Joins

We measured the runtime of different query execution plans involving a single key/foreign key join. We identified the size of the input relations, the number of distinct build keys, and the build key distribution as the relevant parameters that influence hash join performance. Our experiments cover this parameter space extensively. Note that our approach dominates the standard hash join only for certain parameter combinations. For use in practice, we leave it to the query optimizer to choose the best alternative for a given query. Further, we show only a subset of our results due to the large number of parameter combinations.

5.1.1 Setup and Data Generation. R is the key relation, S the foreign key relation. The join predicate is $R.k = S.a$. We varied the cardinalities of the two relations between 2^{10} and 2^{25} in steps of powers of 2, which covers input sizes between a couple of thousand up to several million tuples[2]. The domain of the foreign key $S.a$ is defined as $dom(S.a) := \{0, \ldots, |R|/2^t - 1\}$, where $t \in \{0, \ldots, 9\}$ is a scale parameter controlling the fraction of keys that is referenced, i.e., tuples that find join partners. For instance, $t = 1$ means that only half of the keys in $R.k$ will be referenced by foreign keys in $S.a$. Values for $S.a$ are randomly sampled from $dom(S.a)$, either without skew according to a uniform distribution, or with skew according to a standard Zipf distribution. The number of duplicate foreign keys is controlled by the cardinalities and parameter t. For instance, for $|R| < |S|$, the absolute frequency of each distinct value in $S.a$ is $|S|/(|R|/2^t)$ on average in the uniform case.

Let $dv(e)$ denote the set of distinct values returned by some algebra expression e. For the hash table, we set the number of buckets m to $|dv(x)|$, where x denotes the attribute on which the hash table is built. Note that the 32-bit finalizer of Murmur3 is collision-free on the whole attribute domain $[0, 2^{25} - 1]$. Hence, collisions are solely the result of the modulo operation.

The parameters and their ranges are summarized in Table 1. In total, we evaluated 5120 parameter combinations and all four plans (see Fig. 3) to allow for a detailed analysis. This should cover a large

[2]Large build sizes $> 2^{25}$ tuples should be handled by partitioning, see Sec. 3.

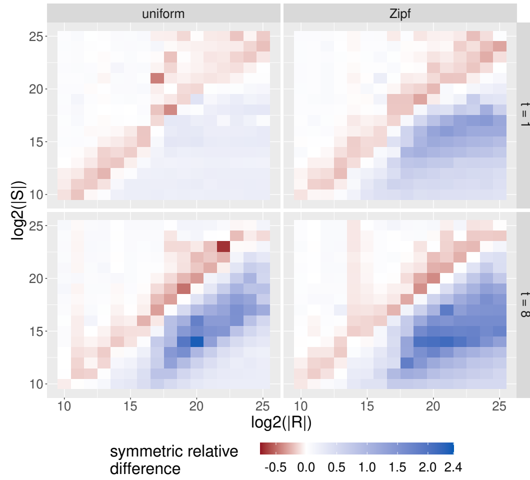


Figure 4: Heatmap of relative differences between runtime of 3D and standard hash join with early termination

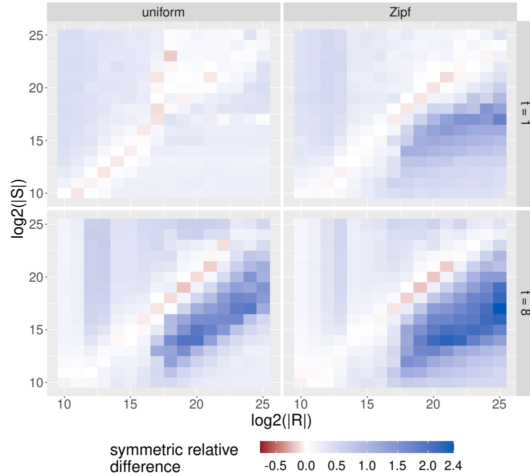


Figure 5: Heatmap of relative differences between runtime of 3D and standard hash join without early termination

fraction of the different possibilities the query optimizer is going to see in practice. No single join implementation is superior to all others for all parameter combinations. Thus, it remains the query optimizer's task to determine which implementation to choose in which context.

5.1.2 Results. For each parameter combination, we compare the best join tree for the standard and the 3D hash join, respectively, by taking the minimum of the total runtimes: $t_{\text{hj}} := \min(t_{\text{Crs}}, t_{\text{Csr}})$ and $t_{\text{3D}} := \min(t_{\text{Nrs}}, t_{\text{Nsr}})$, where the subscripts refer to the join trees from Figure 3. As a relative measure for performance, we choose the symmetric relative difference $srd\left(t_{\text{hj}}, t_{\text{3D}}\right)$, defined as $srd\,(x,y) := (x-y)/\min\,(x,y)$. Values above 0 indicate that the 3D hash join is faster. For values below 0, the standard hash join is faster. The speedup/slowdown factor for a symmetric relative difference of x is $|x| + 1$.

Figure 4 shows the results for both foreign key distributions (uniform and Zipf) and for two example values of the foreign key scale parameter t as a heatmap. The axes show the binary logarithm of the cardinalities of the argument relations R and S. Blue-shaded tiles indicate that our approach is superior to the standard hash join, while red-shaded areas indicate the opposite. The darker the color, the higher the relative difference. In white areas, both algorithms perform equally well. Note that the color scale is not symmetric in the sense that the darkest colors at both ends of the scale have different absolute values. The diagonal from bottom-left to top-right is dominated by the standard hash join, i.e., if both argument relations have about the same cardinality, the chaining hash table has the advantage. In the bottom-right triangle, $|R| > |S|$ holds and our approach outperforms the standard hash join. This effect increases (1) in the presence of skew (right column of Fig. 4), and (2) when increasing the number of duplicates (bottom row of Fig. 4), as both lead to longer collision chains in the chaining hash table. Over all parameter combinations, the relative differences range from −0.70 to 2.53 for uniformly distributed foreign keys and from −0.40 to 2.21 in case of skew. This corresponds to maximum speedup factors of 3.53 (uniform) and 3.21 (skew), respectively.

For the join *Csr* (Fig. 3b), the chaining hash table can benefit from early termination of lookups during probe, as it was built on the duplicate-free key of R. Figure 5 shows what happens when early termination is disabled, which is mandatory in case of lacking knowledge about key uniqueness as introduced in Section 1. The heatmap shows that the dominance of the chaining hash table diminishes. Now, the relative differences over all parameter combinations range from −0.41 to 2.27 for uniformly distributed foreign keys and from −0.20 to 3.10 in case of skew. This corresponds to maximum speedup factors of 3.27 (uniform) and 4.10 (skew), respectively.

We repeated the above experiments with strings instead of integers as the join attributes' data type to investigate more costly join predicates: string comparisons instead of integer comparisons. We used character arrays with a fixed length of 20 for attributes k and a. The data generation remained the same, but all integers were converted to their decimal string representation, padded with zeros at the front. The results were consistent with the integer experiments. Over all parameter combinations, the relative differences range from −0.24 to 2.25 for uniformly distributed foreign keys and −0.19 to 2.99 in case of skew, corresponding to maximum speedup factors of 3.25 and 3.99, respectively.

5.2 Many-to-Many Joins

We conducted two experiments with a single N:M join between two relations R and S, which show that our approach allows for significant time savings compared to the standard hash join. Both experiments use generated data and $p :\equiv (R.a = S.a)$ as the join predicate. We compared the total runtime of $R \bowtie_p^{\text{hj}} S$ and $\mu_B(R \bowtie_p^{\text{3D}} S)$, i.e., the hash tables were built on S and probed with tuples from R. This corresponds to the plans *Crs* and *Nrs* from Figure 3.

5.2.1 Experiment 1: Uniform Distribution. The experiment has three parameters: the cardinalities of the two relations, and the

Table 2: N:M Join, Experiment 1: Result Excerpt

$\|R\|$	$\|S\|$	d	*srd*	$L^{\text{hj}}_{\text{avg}}$	$L^{\text{hj}}_{\text{max}}$	$L^{\text{3D}}_{\text{avg}}$	$L^{\text{3D}}_{\text{max}}$
2^{11}	2^{10}	1024	0.16 (min)	1.58	5	1.58	5
2^{16}	2^{12}	1024	0.97	6.33	20	1.58	5
2^{23}	2^{13}	1024	2.04	12.66	40	1.58	5
2^{25}	2^{14}	1024	4.67 (max)	25.32	80	1.58	5
2^{25}	2^{19}	1024	2.51	810.34	2560	1.58	5

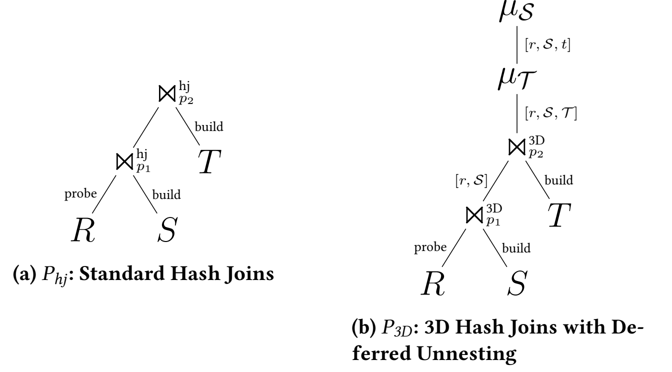
domain size d of attribute a. The attribute values of $R.a$ and $S.a$ are generated uniformly from $\{0, \ldots, d-1\}$. We chose a domain size of 1024, which is also the number of hash table buckets, and varied the cardinalities between 2^{10} and 2^{25} with $|S| < |R|$. A subset of the results can be found in Table 2, where we show the parameters as well as the symmetric relative difference $srd\,(t_{\text{Crs}}, t_{\text{Nrs}})$. We also give the average and maximum collision chain lengths (L_{avg}, L_{max}) of non-empty hash table buckets for both implementations. Each distinct value in $S.a$ has a frequency of $|S|/d$. The collision chains are way longer for the chaining hash table, e.g., 25.32 on average for $|R| = 2^{25}, |S| = 2^{14}$, while the 3D hash table maintains a constant average collision chain length of 1.58, which is exactly the theoretically expected value; for a derivation, see Appendix A. The 3D hash join achieves a speedup factor of up to a 5.67 compared to the standard hash join. Notably, our approach is never slower: Over all parameter combinations, the minimum speedup factor is 1.16, the average 2.82.

5.2.2 *Experiment 2: Extreme Case.* For this experiment, we construct a scenario to showcase that the standard hash join is not robust. We fix some domain size d for $R.a$ and $S.a$ and let the respective hash table have d buckets. Then, we choose two distinct keys k, k' such that they collide. On the build side S, the d distinct values in $S.a$ are uniformly distributed with a frequency of $n := |S|/d$, with the exception of k, which occurs $|S|/2 + n$ times. We call k the *hot build key*. On the probe side R, the d distinct values in $R.a$ are uniformly distributed, with the exception of k', which occurs $|R|/2 + |R|/d$ times.

After building, the chaining hash table contains one collision chain with a length of at least $|S|/2 + 2n$ – the bucket where k and k' are stored. Unfortunately, k is probed only a few times while k' is probed $|R|/2 + |R|/d$ times. Hence, the probe operator has to traverse a long collision chain frequently, only to find that most of the entries do not match since $k \neq k'$. The 3D hashtable benefits from both fewer join predicate evaluations and fewer "hops" when traversing the collision chain (cf. Fig. 2). For $|R| := 2^{23}$, $|S| := 2^{12}$, and $d := 1024$, the standard hash join is 137 times slower than our approach. This factor, however, can grow arbitrarily.

5.3 Deferred Unnesting for Two Joins

To support our claim from Sec. 4.3 that the deferral of unnest operations can be beneficial, we conducted the following experiment. We now consider two key/foreign key joins between the key relation R and the two foreign key relations S and T. Like before, all relations are row-store relations with two 32-bit integer attributes k (unique key) and a (foreign key). We compute the join $(R \bowtie_{p_1} S) \bowtie_{p_2} T$



(a) P_{hj}: Standard Hash Joins

(b) P_{3D}: 3D Hash Joins with Deferred Unnesting

Figure 6: Deferred Unnesting for Two Joins: Join Trees

with the join predicates $p_1 :\equiv (R.k = S.a)$ and $p_2 :\equiv (R.k = T.a)$. Hence, the query is an "inverted star join query". In a regular star schema, a large, central fact relation contains foreign keys that reference various smaller dimension tables. A query on a star schema is a star join query [25, p. 689 ff.]. An "inverted star schema" is the opposite: a central key relation is referenced by various foreign key relations. This shape can, for instance, be found in the IMDb dataset from the Join Order Benchmark (JOB) [20]: `title` with its key `id` is the central key relation referenced by numerous foreign key relations like `movie_info` or `movie_companies`. The 3D hash join is most suitable when the build relation is smaller than the probe relation, and the build key contains duplicate values, as in, e.g., JOB query 13 [20]. This experiment focuses on this case.

5.3.1 *Setup.* R, S, and T are generated as follows: We choose the cardinality of R and set $R.k$ to $0, \ldots, |R|-1$. For the generation of the foreign keys $S.a$ and $T.a$, we partition the keys $R.k$ into four disjoint subsets according to the choice of three fractions $\alpha, \beta, \gamma \in [0, 1]$: A subset of $\alpha \cdot |R|$ keys in R is referenced by foreign keys in both S and T. We call this subset R_α. A subset of $\beta \cdot |R|$ keys in R (called R_β) is referenced exclusively by foreign keys in S, while another subset of $\gamma \cdot |R|$ keys (called R_γ) is referenced exclusively by foreign keys in T. The remaining subset of $(1 - (\alpha + \beta + \gamma)) \cdot |R|$ keys is not referenced at all. Note that the keys in R_α survive both joins, whereas the keys in R_β and R_γ survive one join but not the other. Further, foreign keys are generated uniformly regarding their respective partition: Foreign keys referencing keys in R_α occur c_α times, foreign keys referencing keys in R_β occur c_β times, and foreign keys referencing keys in R_γ occur c_γ times.

We compared two possible physical plans to compute the join, as shown in Figure 6. The plan P_{hj} uses two standard hash joins. The plan P_{3D} uses two 3D hash joins followed by two unnest operations. In each case, the hash tables are built on the foreign key relations S and T. The number of hash table buckets is chosen equal to the number of distinct values of $S.a$ and $T.a$. Note that in P_{3D}, the first join $\bowtie^{\text{3D}}_{p_1}$ yields nested tuples of the form $[r, \mathcal{S}]$, where r is a tuple from R and, conceptually, $\mathcal{S}$ is a list of matching tuples from S[3]. Analogously, the second join in P_{3D} yields nested tuples of the form $[r, \mathcal{S}, \mathcal{T}]$ with an additional list $\mathcal{T}$ of matching tuples from T. The

[3]In reality, it is a pointer to a main collision chain node in the hash table built on $S.a$.

Table 3: Deferred Unnesting for Two Joins: Results (Excerpt): $|R| := 2^{22}, \alpha := 1/2^{22}, \beta := \gamma := 1/2^6$

c_β, c_γ	$|S|, |T|$	*srd*		
		total	build (S)	probe (R)
1	65 537	0.20	−0.37	0.22
2	131 073	0.84	−1.01	1.01
3	196 609	2.74	−1.30	3.39
4	262 145	3.47	−1.57	4.57
5	327 681	6.61	−1.24	8.91
6	393 217	7.73	−1.66	11.12
7	458 753	9.12	−1.56	13.79
8	524 289	10.20	−1.81	16.58
9	589 825	10.63	−1.92	18.02

subsequent unnest operations $\mu_{\mathcal{T}}$ and $\mu_{\mathcal{S}}$ unpack the nested tuples and yield tuples of the form $[r, s, t]$ as the final result.

The first join between R and S produces $|R| \cdot ((\alpha \cdot c_\alpha) + (\beta \cdot c_\beta))$ output tuples in P_{hj}, but only $|R| \cdot (\alpha + \beta)$ in P_{3D}. For $c_\alpha, c_\beta > 1$, this means that the second join with relation T must process fewer input tuples in plan P_{3D}.

For the experiment, we choose the parameters such that the overall join between R, S, and T is very selective, while each single join between R and S, and R and T is not: We vary $|R|$ between 2^{10} and 2^{25}, and set $\alpha := 1/|R|$, $c_\alpha := 1$. β is varied between $1/2^0$ and $1/|R|$, c_β between 1 and 9. γ is set to β, c_γ is set to c_β. Note that since $\beta = \gamma$, $|S| = |T|$, $c_\beta = c_\gamma$, and $|R \bowtie_{p_1} S| = |R \bowtie_{p_2} T|$, the join order is immaterial.

5.3.2 Results. An excerpt of our results for $|R| := 2^{22} \approx 4.2 \cdot 10^6$ and $\beta := \gamma := 1/2^6$ can be found in Table 3. The number of distinct values of $S.a$ and $T.a$ is $(\alpha + \beta) \cdot |R| = 65\,537$.

We show the symmetric relative differences (srd, cf. Sec. 5.1.2) between the execution times of the plans P_{hj} and P_{3D} for the whole plan, the build on S, and the probe on R. The build times on T are similar to S. Recall that for $srd < 0$, P_{hj} performs better, and for $srd > 0$, P_{3D} is superior.

Increasing the multiplicities c_β, c_γ leads to more duplicates and a higher cardinality of the foreign key relations. This results in superior overall performance of the 3D hash join: srd_{total} increases from 0.20 to 10.63; this corresponds to speedup factors ranging from 1.20 to 11.63. This is due to a reduced number of join predicate evaluations and a smaller number of input tuples to the second join, resulting in better probe performance (column "*srd* probe (R)"). At the same time, though, the build performance decreases as builds are more expensive for the 3D hash join due to lookups during insert. This can be observed in column "*srd* build (S)".

In our experiment, we measured speedups up to a factor of 20. Real-world data might additionally be skewed, leading to even larger factors. Note that the factor by which the 3D hash join is faster than the standard hash join can grow arbitrarily, depending on the properties of the data.

6 CONCLUSION

We proposed a novel, simple, yet powerful hash table organization that resolves collisions using hierarchical chains and clusters duplicate keys. The additional costs for key lookup during build and for the unnesting operation can pay off during the probe phase in many cases, especially in the presence of duplicates due to data skew. In the experimental evaluation, we identified the cases in which our approach outperforms the standard hash join. For a single key/foreign key join, the speedup factor of our approach is up to 3.53. For many-to-many joins, we observed a speedup factor of up to 5.67 while never being slower than the standard hash join. Furthermore, deferred unnesting can be beneficial if more than one join is involved.

For future work, we plan to further enhance the implementation of the 3D hash join using the known techniques, e.g., prefetching, filtering, partitioning, parallelization, and NUMA-awareness.

REFERENCES

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *VLDB* 5, 10 (2012), 1064–1075.

[2] A. Appleby. 2016. MurmurHash3. https://github.com/aappleby/smhasher.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *VLDB* 7, 1 (2013), 85–96.

[4] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. 2013. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *ICDE 2013*. 362–373.

[5] M. Bandle, J. Giceva, and T. Neumann. 2021. To Partition, or Not to Partition, That Is the Join Question in a Real System *(SIGMOD '21)*. 168–180.

[6] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. 2014. Memory-Efficient Hash Joins. *VLDB* 8, 4 (2014), 353–364.

[7] S. Blanas, Y. Li, and J. M. Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs *(SIGMOD '11)*. 37–48.

[8] K. Bratbergsengen. 1984. Hashing Methods and Relational Algebra Operations *(VLDB '84)*. 323–333.

[9] P. Celis. 1986. *Robin Hood Hashing*. Ph.D. Dissertation. University of Waterloo.

[10] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. 2007. Improving Hash Join Performance through Prefetching. *ACM Transactions on Database Systems* 32, 3 (2007), 17–es.

[11] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. 1984. Implementation Techniques for Main Memory Database Systems *(SIGMOD '84)*. 1–8.

[12] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. 1992. Practical Skew Handling in Parallel Joins. In *VLDB '92*. 27–40.

[13] L. Fegaras. 2004. The Joy of SAX. In *Proceedings of the First International Workshop on XQuery Implementation, Experience and Perspectives*. 61–66.

[14] G. Graefe. 1993. Query Evaluation Techniques for Large Databases. *Comput. Surveys* 25, 2 (1993), 73–169.

[15] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *VLDB* 2, 2 (2009), 1378–1389.

[16] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. 1983. Application of Hash to Data Base Machine and Its Architecture. *New Generation Computing* 1, 1 (1983), 63–74.

[17] O. Kocberber, B. Falsafi, and B. Grot. 2015. Asynchronous Memory Access Chaining. *VLDB* 9, 4 (2015), 252–263.

[18] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. 2015. Massively Parallel NUMA-Aware Hash Joins. In *In Memory Data Management and Analysis (Lecture Notes in Computer Science)*. Springer International Publishing, Cham, 3–14.

[19] V. Leis, P. Boncz, A. Kemper, and T. Neumann. 2014. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age *(SIGMOD '14)*. 743–754.

[20] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. 2018. Query Optimization through the Looking Glass, and What We Found Running the Join Order Benchmark. *VLDB* 27, 5 (2018), 643–668.

[21] Y. Li, Q. Zhu, Z. Lyu, Z. Huang, and J. Sun. 2021. DyCuckoo: Dynamic Hash Tables on GPUs. In *ICDE 2021*. 744–755.

[22] S. Manegold, P. A. Boncz, and M. L. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering* 14, 4 (2002), 709–730.

[23] R. Pagh and F. F. Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.
[24] B. Răducanu, P. Boncz, and M. Zukowski. 2013. Micro Adaptivity in Vectorwise *(SIGMOD '13)*. 1231–1242.
[25] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database Management Systems* (2. ed. ed.). McGraw-Hill, Boston, Mass.
[26] S. Richter, V. Alvarez, and J. Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *VLDB* 9, 3 (2015), 96–107.
[27] S. Schuh, X. Chen, and J. Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory *(SIGMOD '16)*. 1961–1976.
[28] A. Shatdal, C. Kant, and J. F. Naughton. 1994. *Cache Conscious Algorithms for Relational Query Processing*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.
[29] C. Stein, R. L. Drysdale, and K. P. Bogart. 2011. *Discrete Mathematics for Computer Scientists*. Pearson/Addison-Wesley.
[30] M. Zukowski, S. Héman, and P. Boncz. 2006. Architecture-Conscious Hashing. In *Proceedings of the 2nd International Workshop on Data Management on New Hardware (DaMoN '06)*.

A PROBABILITY THEORY FOR HASHING

The following remarks are based on Stein et al. [29, p. 310 ff.].

Let m denote the size of the hash directory and n the number of keys inserted into the hash table. Assume that all keys are distinct. The probability that a fixed key is hashed to a fixed slot i is $1/m$, since each slot is equally likely. Conversely, the probability that a fixed empty slot i remains empty after a key has been inserted is $1 - \frac{1}{m}$. Further, hashing one key is independent from hashing another. Therefore, the probability that slot i remains empty after n insertions is $(1 - 1/m)^n$, and the expected number of empty slots is $m \cdot (1 - 1/m)^n$. The expected number of non-empty slots is $m - m \cdot (1 - 1/m)^n = m \cdot (1 - (1 - 1/m)^n)$.

For the experiments in this paper, the size of the hash directory is set equal to the number of distinct keys that are inserted into the hash table, i.e., $m := n$. For this specific case, the probability that slot i remains empty after a large number of insertions ($n \longrightarrow \infty$) converges to $1/e \approx 36.8\,\%$ [29].

We can now calculate the expected average collision chain length of non-empty slots, which is the number of insertions divided by the expected number of non-empty slots: $\frac{n}{m \cdot (1-(1-\frac{1}{m})^n)}$. This simplifies to $\frac{1}{1-1/e} = e/(e-1) \approx 1.58$ for $m = n$ and large n.