Simple, Efficient, and Robust Hash Tables for Join Processing

Altan Birler
altan.birler@tum.de
Technische Universität München

Tobias Schmidt
tobias.schmidt@in.tum.de
Technische Universität München

Philipp Fent
philipp@cedardb.com
CedarDB

Thomas Neumann
neumann@in.tum.de
Technische Universität München

ABSTRACT

Hash joins play a critical role in relational data processing and their performance is crucial for the overall performance of a database system. Due to the hard to predict nature of intermediate results, an ideal hash join implementation has to be both fast for typical queries and robust against unusual data distributions. In this paper, we present our simple, yet effective *unchained* in-memory hash table design. Unchained tables combine the techniques of build side partitioning, adjacency array layout, pipelined probes, Bloom filters, and software write-combine buffers to achieve significant improvements in $n:m$ joins with skew, while preserving top-notch performance in $1:n$ joins. Our hash table outperforms open addressing by 2× on average in relational queries and both chaining and open addressing by up to 20× in graph processing queries.

ACM Reference Format:
Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2024. Simple, Efficient, and Robust Hash Tables for Join Processing. In *20th International Workshop on Data Management on New Hardware (DaMoN '24), June 10, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3662010.3663442

1 INTRODUCTION

Relational database systems rely on hash joins to efficiently process join queries, and the runtime of most queries is dominated by join processing. The underlying hash tables have a huge design space, with considerable research focusing on partitioning [2, 20, 26], parallelization [1, 17], and skew handling [10]. In the following, we formulate goals that a good join hash table should strive for and present some simple but carefully tailored implementation techniques to achieve efficient and robust hash tables for join processing. A hash table optimized for join processing should be:

- efficient with memory and cache usage,
- efficient in executed CPU instructions,
- highly scalable for parallel execution, and
- robust against a wide variety data distributions.

These high-level criteria are often in conflict with each other. For example, memory could be traded for CPU cycles by adjusting the load factor of the hash table. Or data placement could be optimized, but at the cost of requiring intricate locking techniques. In our research DBMS Umbra, we experimented with a variety of different hash table designs, many of which used promising tricks that sped up certain cases. However, we found that a simple hash table design, employing minimal instructions in the hot path, performed better overall. In this paper, we present our in-memory hash table design, which achieves significant improvements in difficult $n:m$ joins with skew, while having top-notch performance in classical $1:n$ joins.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DaMoN '24, June 10, 2024, Santiago, Chile
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/10.1145/3662010.3663442

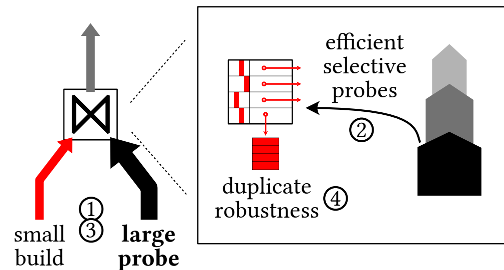

Figure 1: Design criteria for join hash tables: Input sizes are asymmetric and large, the join predicate is very selective, and tuples may find many matches due to duplicates.

There is no optimal design for a hash table; every design is a trade-off between different criteria. We base our design on the observations we have made of join queries found in many benchmarks and real-world workloads: ① Joins are asymmetric, with small build and large probe sides. ② Joins are selective, with many probes not finding a match in the hash table. ③ Join must be scalable, as inputs can be large, and CPU cores are plentiful. ④ Joins can have duplicates, many tuples sharing few keys. These observations guide our design choices: Our hash table combines the techniques of build side partitioning, adjacency array layout, pipelined probes, Bloom filters, and software write-combine buffers to achieve high performance on a wide range of queries.

In the following, we first formulate basic design goals for hash tables for join processing in Section 2. Afterward, we present our hash table technique in Section 3, and evaluate it in Section 4. Then, we discuss related work on hash tables in Section 5.

2 DESIGN CRITERIA

Based on our observations, we formulate four design criteria that a good join hash table implementation needs to strive for. Figure 1 sketches our hash table design and the goals.

Joins are asymmetric. Most joins are asymmetric, with one side often much smaller than the other. Hash joins exploit this asymmetry by building a hash table on the smaller side and probing it using

Table 1: Comparison of join hash table implementations.

Hash Table	Performant Probe	Parallel Build	Skew Robust	Impl. Effort
Open Addressing	~	✖	✖	~
Radix-Join	✖	✔	✖	✖
Chaining	✔	✔	~	✔
3D Chaining [10]	✔	✖	✔	✖
Unchained	✔	✔	✔	~

the larger side. As the sides are often orders of magnitude different in size, the probing phase must be kept extraordinarily efficient. Along these lines, we can afford to spend more CPU cycles on the build side. The following design decisions, further detailed in Section 3, exploit this asymmetry: (1) Our hash table's size is a power of two, to compute slot indexes with a single shift instruction at the cost of increased memory consumption. (2) The build side is fully materialized, partitioned, and copied. The probe side, in contrast, avoids materialization and pushes the tuples into the following operator [21]. (3) The tuples are stored in a compact adjacency array, which makes iterating over matches highly efficient.

Joins are selective. The asymmetry of the sides also manifests in terms of join selectivity: Many probe-side tuples will not find matches. Therefore, it is crucial to efficiently eliminate tuples without a join partner from the large probe side. Bloom filters are ideal for this task: They offer minimal build overhead, high throughput, and good selectivity [16, 25]. In Section 3.2, we propose optimized register-blocked Bloom filters that minimize lookup times and can be embedded into the hash table without space overhead.

Another important aspect of an efficient hash table is the employed hash functions. Before probing the table, the join side computes the hash of the keys to determine the slot in the hash table. Hash joins must weigh the cost of computing the hash against a good distribution of the hash values. While fast hash functions are susceptible to skew, general-purpose hash functions are often too expensive. In Section 3.2, we discuss a cheap but well-distributed hash function based on the CRC32 CPU instruction.

Joins must be scalable. While the build side of a hash join is usually small and fits into CPU caches, it can still grow large and be expensive to build. In addition, recent hardware trends have led to an increase in the number of cores; hence, parallelizing the construction of hash tables is essential to exploit modern hardware. Most general-purpose hash tables do not support efficient parallel building, as they (1) rely on locks that limit scalability, and (2) resize the hash table as more tuples are inserted. These issues can be avoided with a multi-stage build process as described in Section 3.3.

Joins can have duplicates. Multiple tuples with the same key in the build side is another hazard for join hash tables. For common relational benchmarks like TPC-H, this is not a problem as joins typically build hash tables on the duplicate-free primary-key side. However, duplicates often occur in graph workloads and can then be devastating for naïve hash tables. This is pronounced in open addressing schemes, where duplicates are stored inline, colliding with other entries of the hash table. Hash tables with a separate directory (i.e., chaining) are more robust in this aspect because they restrain the impact of duplicate entries to one slot in the directory.

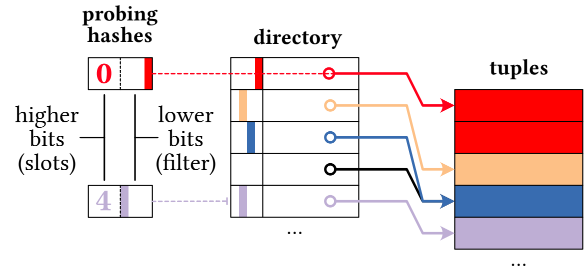

Figure 2: Probing the hash table. Tuples are stored in a contiguous buffer ordered by their hash prefixes. The directory consists of entries representing ranges of tuples sharing hash prefixes. Each entry contains a tiny Bloom filter and a pointer to the range. When probing, the hash prefix is used to index into the directory, suffix is used to check the Bloom filter, and the pointer is followed to the range of matching tuples.

Table 1 summarizes benefits of different hash table designs. We do not consider hash tables that support mixed insert and lookup, which would be needed, e.g., for streaming joins. As we will see in Section 4, open addressing schemes handle skew poorly, since collisions affect neighboring buckets, and they can not efficiently filter selective probes. Chaining allows efficient probes using embedded Bloom filters and are simple to build in parallel [17]. However, for workloads with duplicates, long chains degrade performance as they cannot be efficiently scanned, which degrades skew robustness. In Section 3.1, we discuss our *unchained* table, which allows efficient iteration over duplicate tuples. We further discuss the performance characteristics of related work in Section 5.

3 APPROACH

We now describe the design and layout of our hash table. The design follows the goals formulated in the previous section to allow performant probes, parallel builds, and robustness against duplicates. The microbenchmark results shown in this section in Figures 3, 5, and 12 are further detailed in Section 4.

3.1 Layout

Open addressing schemes store all tuples in a single contiguous array, which is efficient for probing with non-selective predicates and no duplicate keys. When the join is selective, open addressing schemes cannot efficiently filter out non-matching tuples. When there are tuples with duplicate keys, these tuples collide with other keys leading to high build and probe costs. In contrast, chaining hash tables build a *directory* that is separate from the tuples. The directory entries point to the tuples that share the same hash prefix, which are then chained in linked lists, reducing collisions due to duplicates. However, the linked list traversal is costly, especially for long chains. To address these issues, we propose the *unchained* hash table layout that combines the benefits of both open addressing and chaining. Figure 2 illustrates the basic layout of our hash table with a directory and tuple storage. As an alternative to chaining, we use a dense adjacency array for collision resolution: The tuples are stored in a contiguous buffer ordered by their hash prefixes. This way, the tuples are stored in the same order as the pointers in the directory. Adjacent directory entries give the range of tuples with

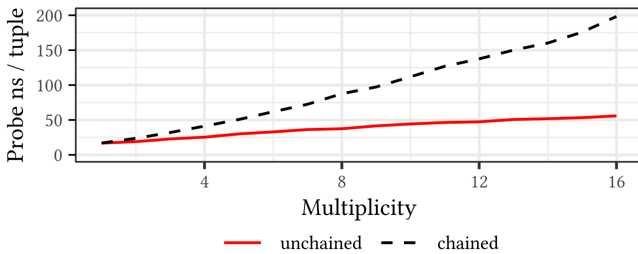

Figure 3: Time in nanoseconds each probe takes with varying multiplicity. The amount of time grows faster for chained hash tables as opposed to unchained.

```
1  u64 shift; // Used to reduce a hash to a directory slot
2  u64 directory[1 << (64 - shift)];
3  void lookup(K key, u64 hash) {
4    u64 slot = hash >> shift;                       // shr
5    u64 entry = directory[slot];                    // mov
6    if (!couldContain((u16)entry, hash)) return;   // Fig. 6
7    produceMatches(key, slot, entry);
8  }
9  void produceMatches(K key, u64 slot, u64 entry) {
10   T* start = directory[slot - 1] >> 16;
11   T* end = entry >> 16;
12   for (T* cur = start; cur != end; ++cur)
13     if (cur->key == key)
14       produce(cur);
15 }
```

Figure 4: Lookup logic for tuples in the hash table: Directory lookups compile to a few instructions.

the same hash prefix, and we can now perform a sequential scan to access all those tuples. Compared to chained hash tables, unchaining eliminates costly pointer chasing. Unlike open addressing schemes, the entries in the directory only reference the buckets with the values, and duplicates do not propagate to neighboring entries. We have measured the cost of traversing linked lists as opposed to compact arrays in our microbenchmark, as shown in Figure 3. As expected, the cost of traversing linked lists grows faster than the cost of scanning adjacency arrays.

Since joins are typically selective, only a fraction of the tuples on the probe side are passed on. It is, therefore, crucial to eliminate tuples without a join partner efficiently and early on. We embed a register-blocked Bloom filter in every slot in the directory to probabilistically discard tuples that definitively do not have a join partner. Since current systems only use the lower 2^{48} bytes of address space, the upper 16 bits of the pointers are unused, and we can use these store the filter. Hyper pioneered this technique [17], and we extend it to our unchained hash table. For efficient access, we store the pointer in the upper bits and the filter in the lower bits. The Bloom filters are checked before accessing the tuple storage and can also be pushed into other operators as semi-join reducers [25].

3.2 Efficient Probes

Probing the hash table is typically the most expensive part of join execution. The probe side can be orders of magnitude larger than the build side. Therefore, optimizing hash table lookups and minimizing

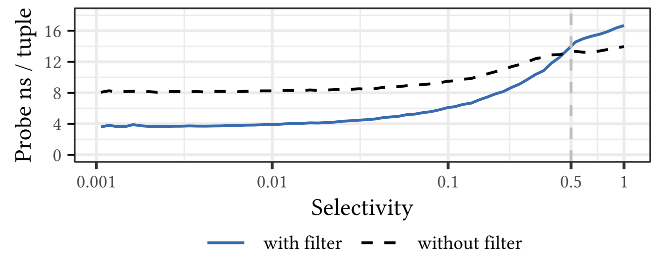

Figure 5: Time in nanoseconds each probe takes with varying selectivity. The amount of time increases as selectivity increases. For selective joins, Bloom filters are advantageous. For joins that are not selective, the overhead is low.

```
1 u16 tags[1 << 11]; // Precalculated 4 bit Bloom filter tags
2 bool couldContain(u16 entry, u64 hash) {
3   u16 slot = ((u32)hash) >> (32 - 11);    // shr
4   u16 tag = tags[slot];                   // mov
5   return !(tag & ~entry);                 // andn
6 }
```

Figure 6: Filtering logic to check if hash value is likely present in the hash slot. We use 4 bit tags from a precalculated lookup table to avoid calculating the tag on the fly.

the per-tuple work is paramount. In the following, we describe our efficient probes for unchained hash tables. Figure 4 summarizes the logic to probe for tuples in the hash table. We optimize several aspects like hash functions, Bloom filters, and the layout of the hash table to achieve this. As a result, filtering non-matching tuples requires only a few instructions.

Due to join selectivity, the hottest path during join processing is filtering out the tuples that do not find a join partner. For filtering, we use a per-slot register blocked Bloom filters with 16 bits [16, 25]. To achieve a low false positive rate, we set 4 bits for each tuple in the slot, which allows us to discard probe tuples where any of the corresponding bits in the filter are not set.

Our implementation uses an optimization to calculate and subsequently test all four bits of the tag at once. In a naïve implementation, calculating this tag requires over a dozen instructions to calculate the positions of the bits. To avoid these in the hot path, we instead use a precomputed lookup table [22] that stores the $\binom{16}{4} = 1820$ distinct bit patterns and load them in a single instruction. Note that we pad this lookup table to $2^{11} = 2048$ entries with uniformly random sampled tags to be able to calculate an index with a single shift instruction. To test if all are set in the Bloom filter, we bitwise complement the filter and bitwise AND it with the computed tag. This also functions as a NULL-pointer check and translates to efficient instructions, i.e., on x86 to a single ANDN. Therefore, this hot path that filters out tuples consists of only five instructions and a branch. Figure 6 shows the logic for this containment check.

Using the precomputed lookup table has several desirable properties compared to computing tags on the fly: First, the precomputed tags only require 4 KB of memory, at most consuming 1 TLB entry. Thus, loading a tag is almost as cheap as computing a single bit tag

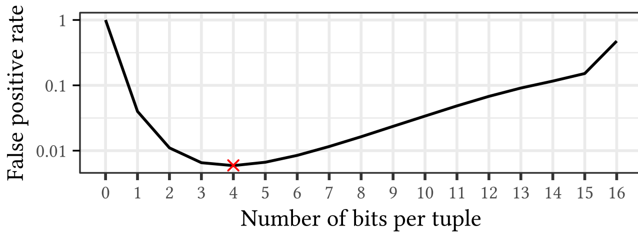

Figure 7: False positive rate of lookups for varying number of Bloom filter bits per tuple. The hash table is assumed to have size $m = 2^{\lceil \log_2(1.125n) \rceil}$ where n is the number of tuples, which leads to a fill rate of $n/m \approx 65\%$. The probability distribution of the number of tuples in a slot can be approximated by the Poisson distribution with $\lambda = n/m$. Given the distribution, the expected false positive rate of a lookup is computed. The optimal number of bits per tuple is 4.

```
1  u64 hash32(u32 key, u32 seed) {
2    u64 k = 0x8648DBDB;              // Mixing constant
3    u32 crc = crc32(seed, key);      // crc32
4    return crc * ((k << 32) + 1);    // imul
5  }
6  u64 hash64(u64 key, u32 seed1, u32 seed2) {
7    u64 k = 0x2545F4914F6CDD1D;      // Mixing constant
8    u32 crc1 = crc32(seed1, key);    // crc32
9    u32 crc2 = crc32(seed2, key);    // crc32
10   u64 upper = crc2 << 32;          // shl
11   u64 combined = crc1 | upper;     // or
12   return combined * k;             // imul
13 }
```

Figure 8: Specialized logic to compute hashes for integers. We use hardware accelerated CRC instructions that allow to cheaply compute well-distributed hashes.

on the fly. Additionally, the 4 bit tag has a significantly lower false positive rate. For a single hash within the list, the false positive rate is ~1/1820 instead of 1/16 with a single bit tag. We evaluated the probe performance with and without filters in Figure 5. We find that the overhead of the Bloom filter is low for non-selective joins, and it is highly advantageous for selective joins.

The number of bits per tag influences the false positive rate. If we use too few bits per tuple, hashes are assigned to fewer distinct tags, leading to more tag collisions. If we use too many bits, unions of tags (in slots with multiple tuples) lead to collisions with more unrelated tags. Both cases increase the false positive rate. Thus, given a load factor for the hash table, which determines the distribution of number of tuples per slot, we can determine the optimal number of bits per tuple. Figure 7 shows the false positive rate for varying numbers of bits per tuple. For our hash table load factor of 65%, 4 bits per tuple with 1820 tags is optimal with false positive rate 1/169. The padded lookup table with 2048 tags has a slightly higher false positive rate of 1/168.

In comparison to probing, generating a well distributed hash from an input key is surprisingly compute intensive. For example, the xxh3 uses avalanche mixing that alone consists of more instructions than our probing logic, i.e., for the common case of 8 byte keys, the full xxh3 hash needs about 4× the instructions of our filter path. We use specialized functions designed for the common case of hashing four or eight byte integer keys, shown in Figure 8.

To get few collisions in the directory slots, we need a well-distributed hash function. We use a large amount of the bits from the 64-bit hash values, since slot selection uses the upper bits and the Bloom filter tags the lower bits. For efficient calculation of these hashes, we use CRC instructions which are well supported and have good properties for hash tables. As shown in Figure 8, for 32-bit inputs, we use a single CRC32 instruction and a multiplication with a mixing constant. For 64-bit inputs, we require two CRC32 instructions as the individual CRC digests are only 32-bits. The resulting specialized hash functions allow efficient filtering of tuples in the hash table, and only need about 10 instructions between loading the value from its base table to a Bloom filter check. Afterwards, we can produce all matches by iterating over the collision list. In chaining hash tables, this requires traversing a linked list with expensive dependent loads of the next pointers. In contrast, our unchained table determines the range of collisions from a neighboring slot in the directory, allowing efficient iteration over the collision list.

3.3 Parallel Build

For chaining hash tables, Leis et al. [17] describe a way to efficiently build the hash directory in parallel. Our proposed duplicate storage in an adjacency array, however, needs slightly more synchronization, which we achieve through partitioning. In a first step to build the hash table, we collect all tuples of the build side of the join and hash partition them. Similar to Richter et al. [24], we use a slab allocator for this initial tuple storage, which keeps them dense in memory. However, for the adjacency array that we use as final tuple storage, we use a contiguous block of memory. In the final storage, the tuples are ordered based on their hash values; all tuples that correspond to an entry in the directory are stored in a contiguous block of memory, and adjacent entries in the directory point to adjacent blocks of tuples, in hash order. To determine the distribution of tuples onto the final storage, we count the number of tuples per directory entry. Afterwards, the tuples are written to the final storage, while simultaneously updating the directory.

3.3.1 Tuple Collection. Within the execution plan of a database query, a hash join sits atop arbitrary operators that produce data that is input into the join. This data is produced as a stream of tuples from various threads. The total amount of tuples is not known beforehand, and estimates thereof may not be accurate. Tuple collection must deal with this uncertainty and be able to handle a varying number of concurrently produced tuples, materialize them, and hash partition them before the construction of the directory.

The tuple collection is often bottlenecked by the memory allocator as many individual tuples need to be allocated and materialized concurrently. To avoid this bottleneck, we use a slab allocator that allocates memory in large chunks and then hands out memory from these chunks to individual tuples. This reduces the number of system calls and the contention within the global memory allocator. The memory is then freed in one go after the build is done.

The bump allocation strategy is complicated by the partitioning of tuples, as we want tuples within the same partition to be mostly

```
1 GlobalAllocator& level1;
2 BumpAlloc level2, level3[numPartitions];
3 size_t counts[numPartitions];
4 void consume(T tuple) {
5   u64 part = tuple.hash >> (64 - log2(numPartitions));
6   if level3[part].freeSpace() < sizeof(tuple):
7     if level2.freeSpace() < sizeof(BumpAlloc):
8       level2.addSpace(level1.allocate<LargeChunk>());
9     level3[part].addSpace(level2.allocate<SmallChunk>());
10  *level3[part]->allocate<T>() = tuple;
11  counts[part] += 1;
12 }
```

Figure 9: Thread-local three-level bump allocation logic for collecting tuples. The highest bits of the hash are used to determine the partition. The counts array is used to keep track of the number of tuples per partition.

contiguous in memory, to make later iterations over tuples within individual partitions efficient. To achieve this, we use a three-level bump allocator. The first level allocates memory in chunks for each thread, the second level allocates smaller chunks per partitions, and the third layer allocates individual tuples from the small chunks. Pseudocode for this is shown in Figure 9.

If the number of partitions is too high, the tuple collection might start incurring expensive TLB misses. Our multilevel allocation scheme avoids this problem, as the small chunks often share the same memory page, which becomes even more likely with 2 MB or 1 GB hugepages. This provides a similar effect as software write-combine buffers [26] without additional costs.

After all the tuples are collected, the partitions need to be exchanged among the threads. This is done by utilizing the internal structures of the bump allocators. The bump allocators store their memory chunks in a linked list. Before a thread processes a partition, it merges all linked lists of chunks corresponding to the partition from all threads. Afterwards, the tuples can be iterated over efficiently as a single chunked list.

3.3.2 Tuple Counting. After the tuple collection is done, we need to construct the directory and copy the tuples over to the final compact tuple storage. To copy tuples over to their final location, we need to first determine the ranges in which tuples that share the same hash prefix will be stored. This is done by counting the number of tuples per directory entry. The counts in the directory are then post-processed with an exclusive prefix sum to determine the ranges in which tuples will be stored. The ranges are then used to copy tuples to their corresponding location in the final storage.

3.3.3 Copies. The final step of the build process is to copy the tuples to their final location in the tuple storage. After the counting, each directory entry contains the start of the range of tuples that share the same hash prefix. As we iterate over the tuples, we copy the tuples to the start, and then increment the start pointer. At the end, each directory entry will point to the end of the corresponding range, and thus the pointer of the previous entry can be used to determine the start. Additionally, a special entry `directory[-1]` is used to point to the very beginning of the tuple storage. This is accomplished by initially allocating an additional space for the table, setting this first entry to the start of the tuple storage, and

```
1 T partitionTuples[][];
2 T* tupleStorage;
3 void postProcessBuild(u64 partition, u64 prevCount) {
4   for (T tuple : partitionTuples[partition]) {
5     u64 slot = tuple.hash >> shift;
6     directory[slot] += sizeof(T) << 16;
7     directory[slot] |= computeTag(tuple.hash);
8   }
9   // prevCount is the total tuple count of previous partitions
10  u64 cur = tupleStorage + prevCount;
11  u64 k = 64 - shift;
12  u64 start = (partition << k) / numPartitions;
13  u64 end = ((partition + 1) << k) / numPartitions;
14  for (u64 i = start; i < end; ++i) {
15    u64 val = directory[i] >> 16;
16    directory[i] = (cur << 16) | ((u16)directory[i]);
17    cur += val;
18  }
19  for (T tuple : partitionTuples[partition]) {
20    u64 slot = tuple.hash >> shift;
21    T* target = directory[slot] >> 16;
22    *target = tuple;
23    directory[slot] += sizeof(T) << 16;
24  }
25 }
```

Figure 10: Count, exclusive prefix sum, and copy. The directory is used to count the number of tuples per hash prefix, then the counts are used to determine the ranges in which tuples will be stored. Finally, the tuples are copied to their final location. Care must be taken to handle the Bloom filters.

then shifting the pointer to the directory forward by one entry. Using a special entry avoids a potential branch in the probe. The pseudocode for the counting and copying is shown in Figure 10.

3.4 Handling Large Tuple Sizes

Copying the tuples remains relatively cheap as long as the tuples are small enough to fit into a cache line. Note that all tuples with size not greater than twice the CPU's vector width can be copied with just two load and two store instructions. This can be accomplished by using the largest vector width not larger than the tuple's size and then using one load/store pair for the start of the tuple and one load/store pair for the end. For example, if our tuple is 24 bytes, and the CPU's vector width is 16 bytes, our first load/store pair copies the bytes 0–15 and the second pair copies the bytes 8–23.

If tuples are very large, we need an alternative approach for build efficiency. In this case, we chain the tuples in a linked list instead of copying them to contiguous storage. This makes the build process somewhat more efficient, as we do not need to copy the tuples. However, we have to update linked list pointers within tuples, which, due to memory write amplification, result in entire cache lines containing the pointers being written back to memory. Still, for tuples larger than one cache line, linking can be worth it.

Linking tuples into directory entries is a relatively simple operation illustrated in Figure 11. The directory entry points to the last linked tuple, which in turn points to the previous tuple, building a linked list. This entry's pointer can be updated with a single XCHG, and the tag can be updated with an additional OR. These operations can be made thread-safe by using their atomic counterparts. Due to

```
1 void linkTuple(T& tuple) {
2   u64 slot = tuple.hash >> (64 - k);
3   u64 prevEntry;
4   xchg(directory[slot], prevEntry);
5   tuple.next = prevEntry >> 16;
6   u16 tag = computeTag(tuple.hash);
7   directory[slot] |= ((u16)prevEntry) | tag;
8 }
```

Figure 11: Linking a tuple into the corresponding directory entry. Replacing the XCHG and the OR with atomic counterparts would make the operation thread-safe.

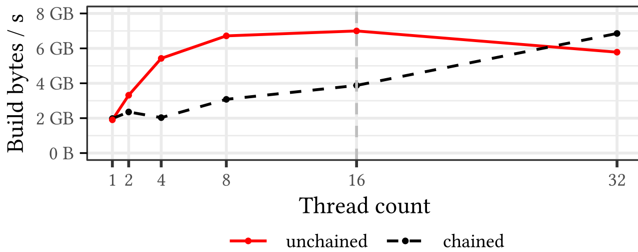

Figure 12: Hash table build in bytes per second with varying number of threads. Unchained scales better than chained up to 8 threads, at which point it hits peak performance. Chained scales worse but is able to better utilize hyperthreads, reaching peak performance at 32 threads.

the simplicity of the atomic variants, the initial partitioning stage can be avoided, increasing contention in the directory build but reducing the cost of initial collection.

In Figure 12, we compare the time it takes to build the hash table with varying numbers of threads using the unchained hash table versus the chained hash table constructed with atomics and no partitioning. We find that the unchained build almost hits peak performance with 8 threads, while the chained build requires all 32 hyperthreads to reach the same level of performance.

3.5 Large Memory Allocation

In Section 3.3.1, we have discussed our memory allocation strategy for tuple collection. Here, we discuss issues with memory allocation for the hash table itself. As the hash table is a large contiguous block of memory, its memory management essentially consists of a single pair of MALLOC and FREE calls. However, as the allocated block can be huge, many subtle issues can arise.

In Linux, the MMAP system call is used to request a contiguous block of virtual memory from the operating system. After the MMAP call, the virtual memory is not yet backed by any physical memory. When the memory is accessed, the operating system will lazily allocate physical memory pages on demand and fill them with zeros. However, if the first access to a page is a read, the operating system will map the virtual page to a read-only zero page. Any subsequent writes will trigger the allocation of a new physical page, and the virtual page will be remapped to the new page. As the original mapping to the zero page might be cached in the TLBs of various CPU cores, the OS has to then issue a TLB shootdown. TLB shootdowns are very expensive [9], as they require the OS to send inter-processor interrupts. To avoid expensive TLB shootdowns, all initial operations with recently allocated pages should be writes. For example, a chaining hash table should prefer the XCHG instruction instead of a LOAD/CMPXCH pair to link tuples in directory entries.

A similar issue arises when the memory is freed as MUNMAP similarly requires a TLB shootdown. Thus, it is beneficial to execute MUNMAP operations later asynchoronously to not slow down critical queries. As Umbra is designed as a server database system, it can avoid these issues by reserving the entire memory upfront with a single MMAP call which is then managed internally. No memory is given back to the operating system until the database is shut down.

4 EVALUATION

We have implemented our unchained table in a microbenchmark and in the relational database Umbra. We ran our benchmarks on an AMD Ryzen Zen3 5950X machine with 16 cores, 32 threads, and 64 GB of RAM. The microbenchmarks in Figures 3, 5, and 12 use hash tables of 720720[1] tuples of size 32 bytes each. We additionally evaluated Umbra using all hardware threads on the relational benchmarks TPC-H SF {1, 10}, TPC-DS SF {1, 10}, and JOB [18]; graph benchmarks LDBC SNB BI [27] SF 10, and CE [7] (only the queries that produce less than 10^9 result tuples). All queries are repeated 10 times, and we report the median runtime.

We empirically evaluated the load factor of our hash tables, and the corresponding false positive rate for `couldContain` (cf. Figure 4). Given uniformly distributed input size, our table has a load factor of approximately 0.65 and, accounting for the distribution of chain lengths, a false positive rate of $1/168$.

To measure the relative performance of unchained (with build-partitioning) and chained (without partitioning as in Hyper [17] and DuckDB [23]) hash tables, we executed all 10312 queries from the aforementioned benchmarks and report the speedup for unchained for each query in Figure 13. For the vast majority of queries, both hash tables perform similarly and no significant speedup or slowdown can be observed. However, for tiny queries, the chained hash table reduces runtime by up to 30%, as the tables are small enough to fit into the caches for probing, and the build overhead of partitioning becomes visible. For large queries, in contrast, the unchained hash table design is beneficial, as it avoids atomic instructions for building the table and improves the memory access pattern for probes. Especially, for the graph workloads, the unchained design improves the performance by up to 20× on the expensive queries which run for more than 100 ms. For these queries, the hash tables contain more tuples with duplicate keys and chaining leads to one random memory access per tuple, which is expensive. Unchaining solves this problem by storing the tuples contiguously in memory.

We find that the biggest regressions for the unchained hash table are cases where the partition count is either too high or too low. dblp_cyclic_q9_06 and hetio_cyclic_q9_12, the two graph queries in the top left corner of Figure 13, are queries with huge build sides where individual partitions do not fit into the L1 cache. When individual partitions do not fit into lower level caches, this significantly slows down the copying of tuples. These queries run faster when the number of partitioned is increased, catching up to the

[1] 720720 is divisible by all numbers from 1 to 16, which helps with tests on multiplicity.

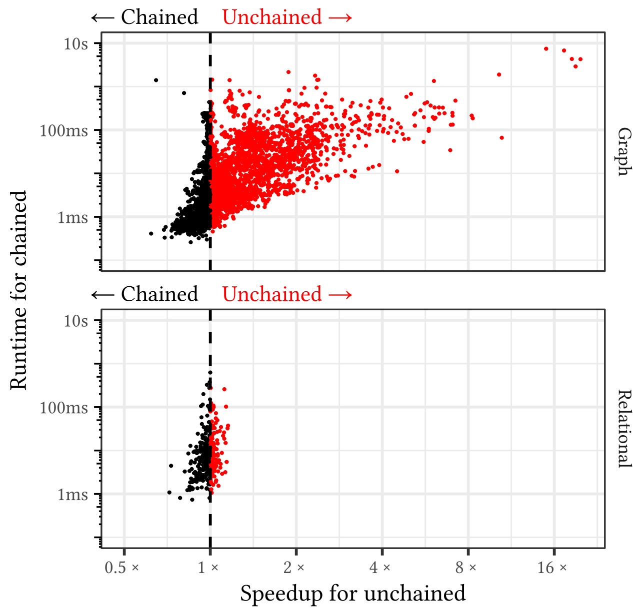

Figure 13: Chained vs. unchained hash tables. We measure the runtimes for executing all relational queries from TPC-H SF {1, 10}, TPC-DS SF {1, 10}, JOB; and graph queries from LDBC SNB BI SF 10, and CE and report the speedup of the fastest algorithm for each query.

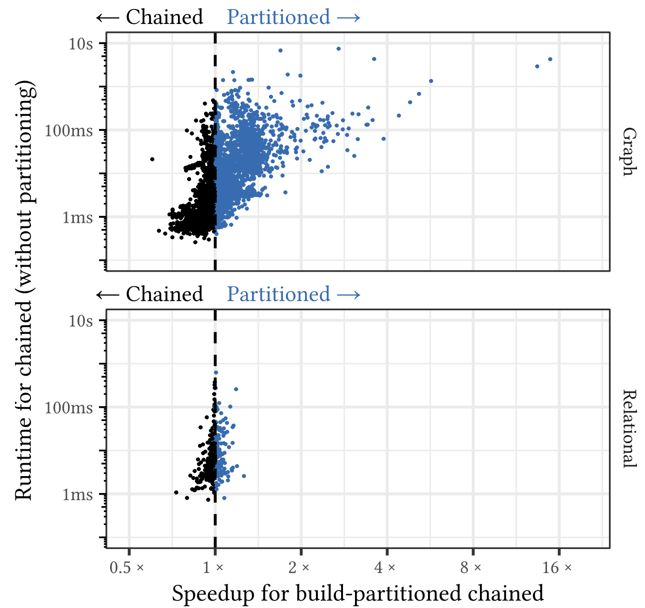

Figure 14: Chained without vs. with partitioning. We measure the runtimes for executing all relational queries from TPC-H SF {1, 10}, TPC-DS SF {1, 10}, JOB; and graph queries from LDBC SNB BI SF 10, and CE and report the speedup of the fastest algorithm for each query. Partitioning the build has a positive impact on performance.

chained hash table. yago_acyclic_Star_6_{42, 43}, the two graph queries in the bottom left corner of Figure 13, are queries with very small build sides where the partition count is too high. The fundamental problem is that our implementation sets the partition count once at the very beginning of the build. This initial partition count is based on heuristics such as cardinality estimates and the number of threads. These heuristics are not perfect and can lead to a suboptimal partition count. In the future, we plan to explore adaptive partitioning schemes that start with a low partition count and increase it as more tuples are collected.

To better demonstrate the individual impact of partitioned collection and subsequent copying of tuples, we implemented build-partitioned chaining in Umbra. We compare the performance of partitioned to chained without partitioning in Figure 14. We see that partitioning often improves locality by increasing the likelihood that tuples in the same chains are closer together in memory. Nonetheless, unchaining by copying tuples to contiguously storage is even more effective. We compare the performance of unchained to chained with build-partitioning in Figure 15. We find that unchaining improves the performance of graph queries by up to 10×.

Next, we measure how our hash tables perform on large workloads and compare them to state-of-the-art database systems. We use TPC-H SF 100 and 1000 running on a larger machine with two AMD EPYC 7713 CPUs with 128 cores, 256 threads and 256 GB of RAM. As competitors, we choose Hyper v0.0.18825 [13] and DuckDB v0.10.1 [23], two high-performance analytical query engines. Hyper and Umbra use query compilation, and DuckDB uses vectorization [5]. All systems utilize morsel-driven parallelism [17].

Figure 16 shows the performance of the three hash table designs for the TPC-H benchmark and the two competitors. We observe the best performance for our chained and unchained hash tables, which are 2× faster on average than a Robin Hood open addressing hash table. Our efficient probes and construction give the two designs a significant advantage over open addressing. Compared to the other systems, Umbra is 2× faster than Hyper and 6× faster than DuckDB at scale factor 100. Similarly, our design scales well to larger workloads, and we efficiently parallelize building the tables.

5 RELATED WORK

Join processing with hash tables is a core component of analytics in database systems and has been studied extensively. We provide a summary of selected techniques for hash tables in Table 1.

First parallel join implementations were based on radix partitioning [19], and later extended for hardware-consciousness [1, 2, 4]. Classical partitioning approaches, however, partition the probe side as well, leading too high overhead on the hot path, and thus is only useful in certain edge cases. In contrast, we only partition the build side to construct a non-partitioned table in parallel.

For non-partitioned hash tables, there are several papers using atomic instructions for chaining hash tables [15, 17, 24]. As we found in Section 4, chaining hash tables work well for build sides without duplicates and are decisively simple, however they are not robust against skew. Iterating the linked collision lists is especially costly, which makes low false-positive rate Bloom filters imperative.

For general-purpose hash tables, open addressing schemes are the predominant implementation technique. Modern variants use

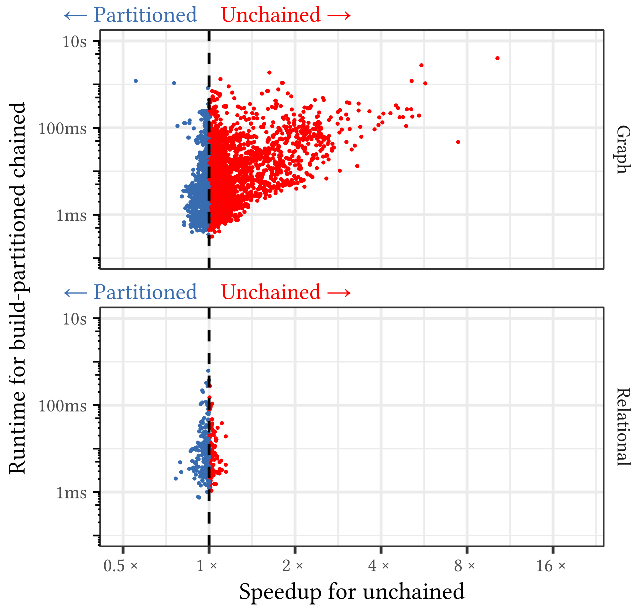

Figure 15: Build-partitioned chained vs. unchained. We measure the runtimes for executing all relational queries from TPC-H SF {1, 10}, TPC-DS SF {1, 10}, JOB; and graph queries from LDBC SNB BI SF 10, and CE and report the speedup of the fastest algorithm for each query. Unchaining further improves graph queries by improving locality.

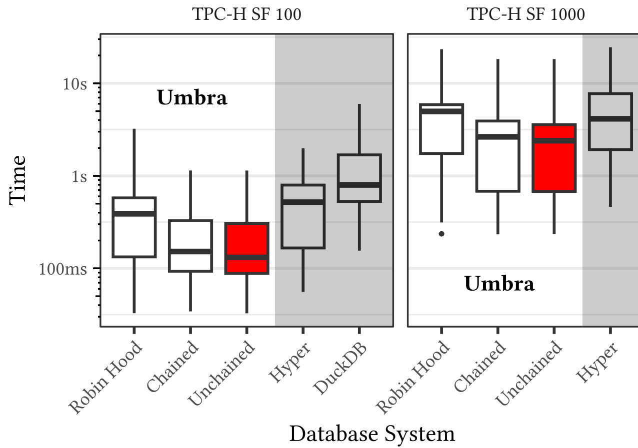

Figure 16: TPC-H performance for open addressing, chained, and unchained hash tables. We include the Hyper and DuckDB systems as reference. Note that DuckDB failed to execute the TPC-H SF 1000 workload within 24 hours.

cache-conscious schemes like hopscotch hashing [11], with similar techniques being used in Google's Swisstables [3], or Facebook's F14 Hash Table [6]. However, these implementations are not tuned for the selectivity that is typically present in join processing, and their more involved design results in a worse filtering fast path. In addition, they are not well-suited for multiset semantics, since storing duplicate values inline is prone to costly collisions, which makes them unsuitable for skewed workloads.

The 3D hash join [10] attempts to reduce the cost of collisions in probing the hash table. However, it incurs complexity and runtime overhead for simple data distributions. Our approach mitigates the cost of collisions by sizing the hash table proportionally to the number of tuples, instead of the number of distinct keys.

Prefetching [8, 12, 14] is an alternative approach to make the memory access in hash tables fast. Explicitly issuing prefetch instructions potentially avoids memory stalls by utilizing idle memory bandwidth to fetch data that will be needed later. In contrast, our approach implements the hash table lookup in very few instructions and exploits the out-of-order execution of modern CPUs that have a large reorder buffer. This way, the CPU automatically schedules the data prefetching without overhead for the hot path. Our approach, with hyperthreading, can hit memory bandwidth limits, at which point prefetching would not be useful.

6 CONCLUSION

Hash tables for join processing have a huge design space. Nonetheless, many workloads exhibit common characteristics that can be exploited when designing efficient hash tables. In this paper, we have presented the unchained hash table design that is optimized for both relational and graph processing. Our hash table design is based on the observation that joins are often selective, asymmetric, and have duplicates in their inputs. We have shown that unchained hash tables outperform state-of-the-art hash tables in a variety of workloads, including TPC-H, TPC-DS, JOB, LDBC, and CE.

REFERENCES

[1] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. IEEE Computer Society, 362–373.

[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 *SIGMOD Conference*. ACM, 168–180.

[3] Sam Benzaquen, Alkis Evlogimenos, Matt Kulukundis, and Roman Perepelitsa. 2018. Swiss Tables and `absl::Hash`. https://abseil.io/blog/20180927-swisstables.

[4] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD Conference*. ACM, 37–48.

[5] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. www.cidrdb.org, 225–237.

[6] Nathan Bronson and Xiao Shi. 2019. Open-sourcing F14 for faster, more memory-efficient hash tables. https://engineering.fb.com/2019/04/25/developer-tools/f14/.

[7] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Kenneth Salem. 2022. Accurate Summary-based Cardinality Estimation Through the Lens of Cardinality Estimation Graphs. *Proc. VLDB Endow.* 15, 8 (2022), 1533–1545. https://doi.org/10.14778/3529337.3529339

[8] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving hash join performance through prefetching. *ACM Trans. Database Syst.* 32, 3 (2007), 17.

[9] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p13-crotty.pdf

[10] Daniel Flachs, Magnus Müller, and Guido Moerkotte. 2022. The 3D Hash Join: Building On Non-Unique Join Attributes. In *CIDR*. www.cidrdb.org.

[11] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In *DISC (Lecture Notes in Computer Science, Vol. 5218)*. Springer, 350–364.

[12] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin J. Levandoski, and Gor V. Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.* 11, 11 (2018), 1702–1714.

[13] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206.

[14] Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *Proc. VLDB Endow.* 9, 4 (2015), 252–263.

[15] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2013. Massively Parallel NUMA-aware Hash Joins. In *IMDM@VLDB*. 1–12.

[16] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter A. Boncz. 2019. Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput. *Proc. VLDB Endow.* 12, 5 (2019), 502–515.

[17] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.

[18] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668. https://doi.org/10.1007/S00778-017-0480-7

[19] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2000. Optimizing database architecture for the new bottleneck: memory access. *VLDB J.* 9, 3 (2000), 231–246.

[20] Jan Mühlig and Jens Teubner. 2023. Micro Partitioning: Friendly to the Hardware and the Developer. In *DaMoN*. ACM, 27–34.

[21] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[22] Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient bloom filters. *ACM J. Exp. Algorithmics* 14 (2009). https://doi.org/10.1145/1498698.1594230

[23] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD Conference*. ACM, 1981–1984.

[24] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. *Proc. VLDB Endow.* 9, 3 (2015), 96–107.

[25] Tobias Schmidt, Maximilian Bandle, and Jana Giceva. 2021. A four-dimensional Analysis of Partitioned Approximate Filters. *Proc. VLDB Endow.* 14, 11 (2021), 2355–2368.

[26] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD Conference*. ACM, 1961–1976.

[27] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter A. Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.* 16, 4 (2022), 877–890. https://doi.org/10.14778/3574245.3574270