Memory-Efficient Hash Joins

R. Barber G. Lohman I. Pandis V. Raman R. Sidle
IBM Research – Almaden

G. Attaluri N. Chainani S. Lightstone D. Sharpe
IBM Software Group

ABSTRACT

We present new hash tables for joins, and a hash join based on them, that consumes far less memory and is usually faster than recently published in-memory joins. Our hash join is not restricted to outer tables that fit wholly in memory. Key to this hash join is a new concise hash table (CHT), a linear probing hash table that has 100% fill factor, and uses a sparse bitmap with embedded population counts to almost entirely avoid collisions. This bitmap also serves as a Bloom filter for use in multi-table joins.

We study the random access characteristics of hash joins, and renew the case for non-partitioned hash joins. We introduce a variant of partitioned joins in which only the build is partitioned, but the probe is not, as this is more efficient for large outer tables than traditional partitioned joins. This also avoids partitioning costs during the probe, while at the same time allowing parallel build without latching overheads. Additionally, we present a variant of CHT, called a concise array table (CAT), that can be used when the key domain is moderately dense. CAT is collision-free and avoids storing join keys in the hash table.

We perform a detailed comparison of CHT and CAT against leading in-memory hash joins. Our experiments show that we can reduce the memory usage by one to three orders of magnitude, while also being competitive in performance.

1. INTRODUCTION

Joins are an enduring performance challenge for query processors. In recent years, inspired by the trend of cheaper and larger main memories, there has been a surge of advances on *in-memory joins*, e.g., [18, 9, 11, 8, 4, 6, 5]. An "in-memory join" typically means one in which the input tables, plus any intermediate data structures, completely fit in memory.

The performance of these modern in-memory joins is truly impressive. For example, we find that a join between a 100 million-row table and a 1 billion-row table runs at about 5 ns per tuple for many of the above variants. However, in our product experience, a fully in-memory join has limited applicability, because some input table or intermediate state will eventually exceed the available memory.

For hash joins, the focus of this paper, the main challenge for in-memory joins is the random accesses to DRAM involved in probing the hash table from the *"build"*, or *"inner"*, table with tuples from the *"probe"*, or *"outer"*, table. To minimize these random access costs, in-memory joins usually partition both the inner and outer tables into pieces that will fit well into the L2 or L3 cache, and do joins within each partition. Such a partitioned in-memory hash join has three steps:

- Partition the outer: Scan the outer and write out tuples to partitions, based upon values of the join columns.
- Partition the inner in a similar fashion, based upon the inner's join column(s).
- Join, partition by partition. This involves reading each inner partition, building a hash table on it, and probing it with tuples from the corresponding outer partition.

While this process improves cache behavior, it requires *two reads and one write* of the outer: to read it in, write out partitions, and then read each partition to probe the hash table. Even more scans may be needed if the partitioning is done in phases to reduce TLB misses [18]. When the outer – typically a large *fact* table – exceeds memory, these multiple scans become very expensive.

While DRAM is cheap and getting cheaper, it is rarely over-provisioned and unused. In many customer installations, storage is over-provisioned, but DRAM is still carefully budgeted, and the workload quickly grows to consume available memory. Moreover, a DBMS never runs just one join in isolation:

- Queries typically join the fact with many dimensions, on different columns. Using $O(|outer|)$ memory per join is expensive and sometimes impossible.
- DBMSs typically run many concurrent queries, and are expected to handle *ad hoc* workloads. So even if the machine has plenty of DRAM, a given query's memory budget will only be a modest fraction of that amount.

Even assuming that the inner will fit in memory is questionable. The inner's hash table has a significant effect on the inner's memory consumption, and can make the difference between a join whose inner fits in memory and one that does not.

1.1 Concise Hash Joins and Hash Tables

In this paper, we present a join that dramatically cuts down on the memory usage, by up to 3 orders of magnitude, compared to the leading in-memory hash joins from Balkesen

Proceedings of the VLDB Endowment, Vol. 8, No. 4

et al. [6]. Further, our memory usage is mostly insensitive to the outer's cardinality. Our join is also faster in most cases than this prior work, assuming that all the joins fully run in memory. We argue that our techniques provide users a more efficient, cost-effective, and robust configuration than prior work.

We achieve this reduction in stages. First, we cut the memory needed for data structures holding the join outer by scanning it just once. Second, we use a new concise hash table data structure for the join inner. Third, we can in some cases further compact the inner hash table using a compact array that avoids storing keys.

1.1.1 Scan the outer only once

We scan the outer in a pipelined fashion, probing the inner and outputting results as we go. We primarily achieve this using a join that does not partition the outer. To handle extremely large inners, we also describe a technique for partitioning the outer in a pipelined fashion (Section 3.2).

Historically, random access was viewed as the villain, and so partitioning was necessary for a fast join. This has been questioned by Blanas et al. [8], who argue for fully non-partitioned joins. But the authors of [6] challenge this by adding partitioning and hardware-specific optimizations to the hash table of [8] to achieve the currently fastest in-memory hash joins. We second the claims of [8], but *only* on the probe side. We observe that hash joins do *data-independent* random access (DIRA), which is not that vulnerable to access latencies, and hence probing with a non-partitioned outer works well.

However, the build side must also be considered. Traditionally, the build side of non-partitioned joins have suffered significant contention when multiple threads insert into the same hash table. In contrast, with partitioned inners, each thread builds an independent hash table for each partition, eliminating contention.

To avoid contention, our hash tables are designed to support *partitioned build and non-partitioned probe*. Our join uses a single physical hash table that is logically split by partition. During the inner scan, we do partitioning as usual. Then, each thread picks one or more partitions and builds the portion of the hash table corresponding to those partitions.

1.1.2 Concise Hash Tables

A join hash table must store both join keys and *payloads* (columns referenced later in that query). Implementations often double this raw size to handle collisions. For linear probing, this overhead arises from the fill factor. For chaining, the overhead comes from memory fragmentation and pointer storage. Many hash tables also round up the number of slots to a power of two.

We introduce the *Concise Hash Table (CHT)*, a linear-probing hash table that has almost no collisions and still gets 100% occupancy in its array of (key, payload). A CHT consists of three pieces: (a) a bitmap indicating the bucket occupancy of a very sparsely-filled linear probing hash table (this hash table is virtual and never built); (b) a dense array of (key, payload) pairs, holding the non-empty buckets of the virtual hash table; and (c) an overflow hash table.

Together, these structures compactly represent a large, sparse hash table that has very few collisions (often $< 1\%$).

CHT embeds precomputed population counts into its bitmap to speed up lookups, which must map hash values to positions in the (key, payload) array. These lookups have a pure DIRA pattern – each lookup can be issued before the previous ones finish. Doing N lookups into a CHT involves two DIRA rounds of N accesses each – first into the bitmap, then into the array. Compared to standard hash tables, the first round is extra, but the bitmap is much smaller than the array and usually fits higher in the cache hierarchy. Moreover, we can reuse the bitmap lookup as a Bloom filter lookup for the join, to avoid hash table probes for outer tuples that will not find a match.

Due to its dense array, CHT *does not support* point inserts, but join hash tables need only handle bulk inserts during build. We describe how this is done, without building the virtual hash table, in Section 2.2.

1.1.3 Concise Array Tables (CAT)

When the domain of the join key is not too sparse, we can do even better than CHT, in both memory consumption and probe speed, using a *concise array table* (CAT). This is often the case for numerically-typed (integer, date, decimal) keys.

The idea behind CAT is to map the key domain directly onto the bitmap, and have an array of payloads alone. As with CHT, this array has entries only for the occupied bits of the bitmap. A CAT is entirely collision-free, and so need not store join keys in the hash table, yet still has nearly 100% fill factor. We show experimentally that CAT can be exploited even for mildly dense keys, e.g., when the number of distinct keys is only 1% of the range.

We also present optimizations to deal with common ways in which key domains are not dense, including a technique for combining CAT for most of the keys with a regular hash table for the others, and a bijective hash scrambling technique to deal with skewed key distributions.

1.2 Other Contributions of this Paper

We have implemented CHT and CAT in Blink3, a column-store query processor being developed at IBM Almaden. This is based upon, but distinct from, DB2 v10.5 with BLU Acceleration [20], which uses a variant of CHT and CAT. In Blink3, inner tables are first scanned, local predicates applied, and then built into hash tables. Then the outer table is scanned and pipelined (in batches) through a series of joins with inner (build) tables. We do late materializing scans: each column is fetched only when needed.

In the case of multi-table join queries, we reclaim the extra cost of probing the CHT (or CAT) bitmap as a Bloom filter lookup. Bloom filters are a powerful method to eliminate non-matching join outers early, and were suggested as far back as $R^\star$ [16]. We use the CHT or CAT bitmap itself as the Bloom filter, and apply these filters for all joins first, before probing the join hash tables. Thus, outer rows get the cumulative filtering effect of all joins.

We present detailed performance numbers for hash joins using CHT and CAT in Blink3, including comparisons with partitioned and non-partitioned joins from the driver of [6].

1.3 Data-Independent Random Access

Before diving into CHT, we do a performance analysis of data-independent random access (DIRA). This is crucial to decide whether outer partitioning is needed.

Abstractly, hash table probing is just a *gather* that does repeated random accesses into an array of buckets. This is DIRA, since the result of one lookup is not needed before

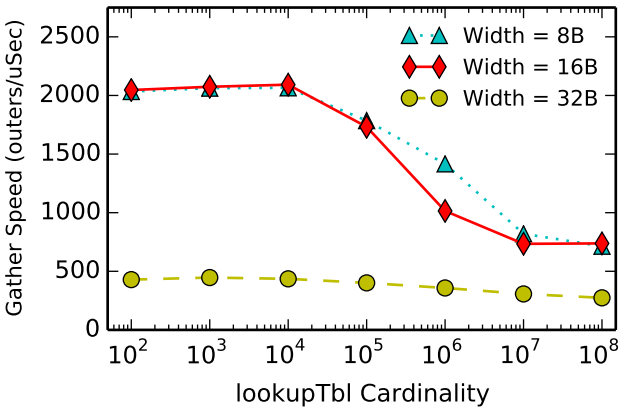


Figure 1: Speed of gather for various `lookupTbl` cardinalities and widths.

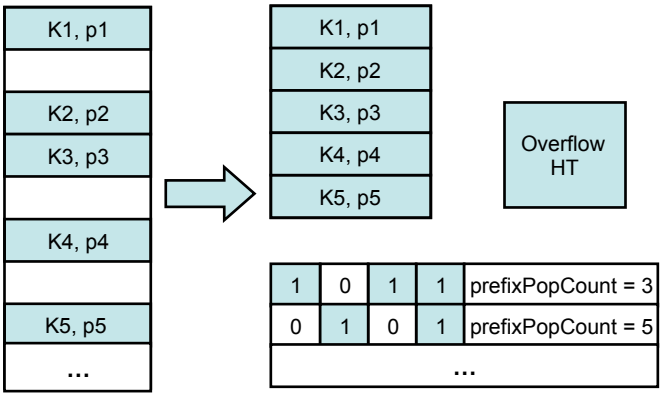


Figure 2: A CHT with 5 keys. On the left is the logical hash table with 8 slots, and on the right is what is stored physically.

issuing the next. So, theoretically, memory *latency* should not matter; only *throughput* should matter. To see how gather performs, we ran the following multithreaded micro-benchmark on a 2-socket Intel E5-2665 machine with 8 cores running at 2.4 GHz:

```
1.  for (i=0; i<N; ++i)
2.      op_thd[i] ← lookupTbl[perm_thd[i]];
```

Each thread scans a thread-local input array, $\texttt{perm}_{\texttt{thd}}$, and uses those values as offsets into a *shared* lookup table, `lookupTbl`, writing out to a thread-local array, $\texttt{op}_{\texttt{thd}}$ (each thread mallocs `perm` and `op` separately, thereby affinitizing them to the local socket). Array $\texttt{perm}_{\texttt{thd}}$ contains uniformly-generated random numbers, thus this does random accesses. However, these are DIRA accesses because the lookup results, $\texttt{op}_{\texttt{thd}}$, are not needed until *after* the for-loop. This allows the processor to prefetch these lookups.

Figure 1 plots the lookup speed as we vary the cardinality of `lookupTbl`, the region being probed. We compare three widths for the `lookupTbl` entries: 8, 16, and 32 bytes. The outer cardinality, N, is set to 2 billion (all threads combined). `perm` is always an array of 8-byte integers. All results use 32 threads, which gave the best speed.

Observe that modern machines do well at DIRA. At inner size 1E2, `lookupTbl` is ≈ 1KB, so fits well in L1 cache. But at size 1E8, the `lookupTbl` is much larger that the L3 cache. Yet the gather speed worsens by less than 2.7X across this range. In contrast, the latency to DRAM is usually 100X worse than the latency to L1 cache. This explains the appeal of non-partitioned joins such as [8]: we need very wide partitioning to speed up probes, and the partitioning cost can offset the gains.

0-byte payload

We also see in the plot that the random-access unit matters, and that widening the `lookupTbl` hurts the lookup speed. The 8-byte `lookupTbl` width is particularly significant. If we view `lookupTbl` as a (trivial) join hash table, each entry of `lookupTbl` has to have both a join key and a join payload. Since we use an 8-byte `perm` array, an 8-byte `lookupTbl` implies a 0-byte payload. Such empty payload joins occur fairly often due to existential sub-queries and antijoins.

But notice that the lookup speed is not that much better for the 8-byte width. We will see in Section 4 that CAT does particularly well for such joins, because it avoids storing the key entirely, so it needs no `lookupTbl`.

2. CONCISE HASH TABLES (CHT)

A concise hash table (CHT) is a linear probing hash table in which we compress out all the empty buckets. As Figure 2 shows, a CHT consists of three pieces:

- An *array* of the non-empty keys and payloads.
- A *bitmap* sized to the uncompressed hash table size (8 in the example), indicating for each bucket whether it is occupied or not.
- An *overflow* hash table, explained later.

The bitmap is implemented as an array of 64-bit words, but in each word only 32 bits are used for the actual bitmap (in the figure we use 8-bit words for illustration). The rest is a pre-computed 32-bit prefix population count that marks the number of 1-bits up to that word[1]. The population count is used when probing into a CHT. The pseudo-code below describes this process:

```
1. bkt = hash(key) % |bitmap|;
2. Check bkt'th bit of bitmap;
3. if not set,
4.   the key is not present in the CHT
5. else
     // Find # of '1' bits, up to bkt'th bit
6.   word = bitmapWords[bkt/32];
     bitsUptoBkt = word.bitmap & ~((~0)>>(bkt%32));
    // prefixPopCount: # of '1' bits in prior words
7.   pos =
       word.prefixPopCount + popCount(bitsUptoBkt);
8. search in array for key, at positions
     pos, pos+1, .. pos+Threshold-1
9. if not found, search in OverflowHT
```

During a probe, when we hash to a bucket *bkt*, we look up the bitmap word at offset *bkt*/32. The prefix count for this word is added to the population count within that word

[1] 96-bit words (64 for the bitmap plus 32 for the count), will compact further, but cause extra cache misses.

to find the offset into the key-payloads array. By in-lining the prefix count within the bitmap in this way, we avoid any extra cache-line accesses. We then search in the array from that offset up to some threshold maximum number of linear probes. Beyond that, we look in an overflow hash table. The overflow can be any hash table – we use a simple linear probing hash table.

When we do joins (Section 3), the first lookup into the bitmap is not really an extra random access, because we reuse it as the Bloom filter lookup for the join as well.

A CHT offers two benefits over conventional hash tables. First, the memory space it uses is much smaller than a regular hash table (both linear probing and chaining variants), because the array of keys and payloads has 100% fill factor. With the CAT variant (presented in Section 2.4), the savings are even greater. For hash joins, this means the inner hash table is more likely to fit in memory.

Second, the probe phase of a join does *batch* lookups. With CHT, these can be done efficiently in a DIRA pattern. Lookup into a CHT involves a lookup into the bitmap, followed by a linear probe against the array. But doing this one key at a time results in a *data-dependent* random access, because for each key the index into the array is determined only after we look up the bitmap. Even if we batch this lookup across many keys, we are accessing a *variable* number of array positions for each key. So it is difficult to pre-compute array positions before the array access.

We convert these into DIRA accesses by making collisions both *rare* and *bounded*:

- We make the CHT bitmap *very sparse*. This is affordable because we spend only 2 bits per bucket.
- We eliminate linear probes longer than a threshold by spilling such keys to an overflow hash table.

The net result is that during a join, the hash table lookup for a batch of keys reduces to:

A) A DIRA access to the bitmap to fetch the words for the accessed hash positions, followed by arithmetic to calculate the positions in the array.

B) A DIRA access to the array to fetch threshold number of matching buckets for each key.

C) A final pass over the rare overflow keys to look up their key values from the overflow hash table.

2.1 Collisions and Overflow

For the bitmap, we use a fill factor of 1 in 8 – i.e., the logical linear probe HT whose bucket occupancy this bitmap tracks has a 1 in 8 fill factor. Even with this sparsity, the overhead due to the bitmap is only $8 \times 2 = 16$ bits per key, much less than with traditional chaining or linear probing (the extra factor of 2 comes from the population count).

We use classical multiplicative hash functions: for every 16-bit piece x_i of the input key, multiply by $(x_i + h_i)$, then take a final modulus of the prime $2^{31} - 1$, which can be done efficiently via standard methods. The h_i are hash function constants. This family is universal, and from balls-and-bins arguments we get the probability that a bucket has any collisions for n keys is $1 - ((8n-1)/(8n))^n - (1/8)((8n-1)/(8n))^{n-1}$ (the last two terms are the probability that a bucket has 0 and 1 keys hashing to it), which is, as $n \to \infty$, roughly $1 - (9/8)e^{-1/8} = 0.72\%$

Since the collision probability is this low, we do not need much linear probing. We use a linear probe threshold of 2, i.e., the third key hashing to the same bucket is placed in the overflow hash table. This prevents long collision sequences from running into neighboring buckets, causing them to overflow, too. The small threshold means that the buckets accessed in the second DIRA pass are typically in the same cache line (for each key). Overflows are expensive, but they generally occur only when there is a collision and the next bucket is also occupied, which occurs about 0.09% of the time.

The overflow hash table has another benefit – *robustness*. In our experience, every hash function fails (has excessive collisions) on some inputs, leading to hard-to-diagnose performance bugs. So we use a second hash function (from the same family, with different multiplier constants) for the overflow hash table. Keys spill to it only after excessive collisions on the primary hash table, so if the primary hash function is weak, the overflow can act as a backup.

2.2 Memory consumption during build

A challenge with CHT is that, during build, the final array position for a key is not known until all the keys have been inserted, because the position is given by the number of occupied buckets, up to the one holding the requested key.

So how do we build a CHT?

One way is to build an actual linear probe hash table, by allocating space for an array of |bitmap| keys, inserting keys and payloads into it with linear probing, then forming the bitmap based on which buckets are occupied, and finally compressing out the empty buckets. But for CHT we want a sparse bitmap. So it is expensive to allocate such a large array, only to throw it away after the bitmap is built.

2.2.1 N:1 joins with enforced PK constraint

Upon closer observation, we note that if the input keys are guaranteed to be unique (e.g., there is an enforced unique constraint), linear probing can be done directly on the bitmap itself. We do not need to insert any keys. So we can do a two-pass build:

A) Scan input keys, hash each one, and set bits in the CHT bitmap, with linear probing on collisions. If the threshold is exceeded, put the key in the overflow HT.

B) Compute the (cumulative) prefix population counts for the bitmap.

C) Scan input keys and payloads, hash again, and insert into a compacted array, using the prefix population count to find the insert positions.

2.2.2 Near N:1 and N:M joins

Enforced PK constraints are not that popular because they impose index maintenance cost during inserts. But queries sometimes care only for the first match for a join outer. This case is common with (not) existential sub-queries. Even for regular equijoin queries, where the SQL semantics requires all matches for each key, the join can be a "run time N:1 join" – a join that we determine is N:1 after building the hash table, because there are no duplicates (we have even seen cases in which the base table does have duplicate keys, but they are eliminated by local predicates). Another common situation is a *near* N:1 join, in which most keys have only 1

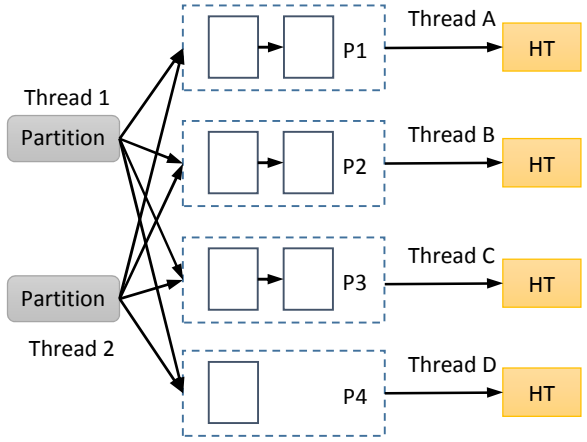


Figure 3: Building CHT in parallel by logically splitting into partitions, with each built independently.

or 2 matches, and only a small number of keys have many matches.

All of these situations are well handled by a CHT with its overflow hash table.

In the first scan (Step A of Section 2.2.1), we just set bits in the CHT bitmap. If there is a collision in this process, we treat it as a hash collision (we have no way to know it is not a duplicate key, because we are not storing keys during this scan), and do linear probing until the collision sequence grows beyond the threshold mentioned earlier. At that point, keys are inserted into the overflow hash table, so if the key repeats further, we can detect this is a duplicate, not a hash collision.

Steps B and C remain exactly as before: we compute prefix population counts on the CHT bitmap, and accordingly insert keys and payloads into the CHT array.

Now consider the situations discussed earlier:

- *(Not)Existential subquery:* Here duplicate keys should be eliminated, since the join does not need them. In the overflow hash table we eliminate duplicate keys early. But the first two occurrences of a key will occupy separate slots in the CHT, because they will take up two bits in the CHT bitmap during Step 1. We consider this 2X space overhead for such keys to be an acceptable cost (especially because these queries are often eligible for CAT, have no payload, and thus the (key, payload) array is not formed at all).
- *Run time N:1 join:* CHT is ideal for this situation, because there are no duplicate keys – any collisions in the bitmap are hash collisions, and thus the CHT strategy of treating them as hash collisions is perfect.
- *Near N:1 and N:M joins:* Keys that match to 1 or 2 payloads are handled well: all matches for a key will likely be stored consecutively in the CHT array, and thus accessed within a single cache line. Keys that match to many payloads go mostly to the overflow (all but the first two matches). So CHT doesn't directly handle these "high cardinality" keys, but any hash table good at N:M joins can be used for the overflow hash table.

2.3 Parallelism during build

As mentioned earlier, partitioning benefits build speed, because each thread can build a partition independently. Further, after partitioning we can size the hash table perfectly, and thus avoid having to resize it during build.

At the same time, we want to support non-partitioned probes, as discussed in the introduction. So in Blink3 we *always build a single physical hash table*. But, this physical hash table is built by a partitioned build process.

Figure 3 shows the build process. Each thread scans a subset of the input, and partitions it into thread-local structures. When a page worth of keys have accumulated for a partition, that page is added to a global linked list of pages of that partition, under a latch (only a pointer is copied under the latch). Since we want to form a single physical CHT, we partition on the most-significant $\log \text{NumPartitions}$ bits of hash(key).

After partitioning, a single physical CHT is formed. But *logically, this is split* into NumPartitions CHTs. The CHT bitmap is split uniformly into NumPartitions pieces. The CHT array (of (key,payload)) is split according to the number of keys found in each partition. Then, each thread scans the linked list of keys and payloads for a partition, and builds into the logical CHT for that partition (threads pick partitions in work-stealing fashion). We restrict the linear probing for each partition to operate within that partition alone (i.e., we cycle around within the partition itself, if needed).

Thus there is no synchronization during the build, except to compute the cumulative count of the number of keys up to each partition (for logically splitting the physical CHT).

Note that this logically partitioned hash table works well with either partitioned or non-partitioned probes.

2.4 Concise Array Table (CAT)

With CHT, we compressed out the empty hash table buckets, and bounded collisions with an overflow hash table. Our goal with concise array table (CAT) is to go even further, and *eliminate* collisions. Once we have a collision-free mapping from key to the hash table bucket, we can eliminate the keys from the hash table array as unneeded.

This makes CATs thinner than CHTs, so they pack better into cache lines. For example, consider a 4-byte key with a 16-byte payload. Only three of these 20-byte buckets will fit completely in a cache line – we can fit 3.2 on average, but then every fourth bucket will straddle a cache-line boundary. However, as a CAT, the bucket takes up only 16 bytes, and 4 buckets pack cleanly into a cache line.

2.4.1 Avoiding Collisions

Collision avoidance is based on the observation that DBAs usually design join keys to have dense domains. In fact, for uniqueness, it is a common practice to assign keys automatically via a serially increasing counter. Ideally, if the keys were integers from 1 to N, we could map them to positions 1 to N in the hash table bitmap. Even this cannot be an identity mapping because of skew; we address this further below. However, CAT can tolerate a fair amount of deviation from this ideal density and still be effective.

Local Predicates:

Even if the keys were originally dense, after applying local predicates the resulting keys will be non-dense. CAT will use 2N bits for the bitmap, where N is the range of the key values. But only |inner after predicates| entries will be used in the payloads array. Consider a hash table in which each bucket originally takes up 128 bits (e.g., an 8-byte key and 8-byte payload). Even if the predicates have a selectivity of

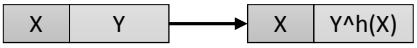
10%, the CAT bitmap only uses 20 bits per bucket, which is a small price to pay for eliminating the key (and collisions).
Non-Dense domains: Key domains often have outliers: values like MAXINT are used to indicate defaults, unknown, etc. Our general solution is to pick an approximate median value (during partitioning), and choose a range around it that is as large as will fit in the memory budget of that join. Keys within this range are handled via CAT, and other keys go to the overflow hash table. If this causes too high a fraction to overflow, we use CHT instead of CAT.
Multi-column keys: With composite keys, even if each part of the key is perfectly dense, their concatenation is anything but dense. Each part will be represented as a word-sized integer, so if the values range from 1 to N, we lose (word width) - $\log N$ bits between parts. We avoid this loss by tracking the second largest value seen, for each part of the composite key, during partitioning (on the build side, using the second largest avoids the MAXINT problem). Then when inserting into the CAT we concatenate the parts of each key in such a way that we remove those unused bits.

2.4.2 Avoiding Skew

The partitioning function in partitioning-based hash joins typically uses bits chosen from the hash value itself. Therefore, partitioning in hash joins relies on hashing to avoid skew. For CHT, we partition on the most significant bits of the hash value.

For CAT, if we use the key directly as the index into the bitmap, then skew becomes a major concern. If we partition on the most significant bits, and some local predicate filters out tuples with high key values, those partitions will have very few values. If we partition on the least significant bits, and this is a composite key, the partitioning bits are likely to come from the last component of the composite key. If a local predicate reduces the number of distinct values in that part, the partitioning will have huge skew, leading to poor multi-threaded scaling.

We solve this problem by applying a *bijective hash scrambling*, as shown below.



Suppose a key is w bits wide. We compute a $\log P$-bit hash on the most significant $w - \log P$ bits of the key, where P is the number of partitions. We XOR this hash onto the least significant $\log P$ bits. This simultaneously achieves:

- *Skew Avoidance:* we partition on those $\log P$ bits, so the partitioning is resilient to skew arising from local predicates.
- *No loss in density:* The resultant scrambled key is still w bits wide, so any density in the original key domain is preserved after scrambling as well.

3. FAST JOINS USING CHTS AND CATS

There is more to fast joins than just employing fast and concise hash tables. In this section we describe how we incorporate CHTs into a hash join and make it a part of a larger query plan.

The build side of hash joins in Blink3 is pretty much what we described in Section 2.3 and Figure 3. The inner side of the join – either a scan over a table, or the result of other operators – is pipelined into the CHT build in parallel.

In Blink3, the probe phase of a join query is executed with a multi-threaded query plan that has separate *scan*, *join*, and *output* operators (and *Bloom filter*, for multi-way joins), connected in a separate operator chain for each thread. What flows through the operator chain are work units, which correspond to 8192-tuple ranges of the input table, called *strides*. Each thread picks these strides in work-stealing fashion to balance work among threads. [20] describes the query plan in detail.

The scan operators scan one column each, fetching values only for rows that satisfy previously applied predicates (late materialization). Each column is stored in a separate file, and the scan operator reads them via the `read()` call, one stride at a time.

Scan results are pipelined to the *join* operator, which looks up strides of foreign keys[2] in the join hash table and pipelines results to the output operator.

We next discuss two improvements to this basic plan: early filtering of outer tuples that will not find join matches, and pipelined outer partitioning for very large inner tables.

3.1 Early filtering via the CHT bitmap

We saw in the introduction that the hash table payload seriously affects probe speed, and that 0-byte payload lookups were very fast. Some recent papers on joins assume that filtering is performed during the join, and so advocate *late materializing joins* [3]. That is, the join just maps each key to a row identifier (RID), and only at the end of all joins, just when the payload columns are needed, are the RIDs mapped to payloads.

In Blink3 we use the CHT (and CAT) bitmaps as Bloom filters to eliminate non-matching join outers before they enter the join. In the case of CAT, this is a non-lossy filter, because a CAT bitmap is collision-free. The bitmap takes up only two bits (one bit for the map, plus one on average for the prefix population count) per key, and so is cheaper to probe than the (key, payload) array.

Foreign keys are first sent to this bitmap as part of the first DIRA probe (recall the pseudo-code of Section 2; this is Step 1). If this bitmap lookup goes to a bit that is unset, this key is definitely not present in the CHT (or CAT). Otherwise, its PopCount yields an offset into the (key, payload) array where this key will likely be found.

For multi-way joins (where an outer is joined with multiple inners), we do this DIRA probe against the CHT bitmaps of all inners, before doing any join. Figure 5 shows an example with two joins. The cumulative filtering effect of both joins is applied before we look up into either CHT array. This provides robustness to poor join ordering, because join ordering only affects the number of probes into the bitmap, not (in a major way) the number of probes into the arrays.

3.2 Pipelined Partitioning

Joins in Blink3 generally do not partition the outer. But partitioned probes can be helpful for extremely large or wide inners. In these cases, we partition the outer into batches. This probe phase proceeds as shown in Figure 4. Each thread reads strides of tuples from the outer table (in work-stealing fashion). Since Blink3 is a column store, we do partitioning only on the foreign key columns, within each

[2] As a convenience, we use *foreign keys* as a short form to refer to the join columns of the outer table.

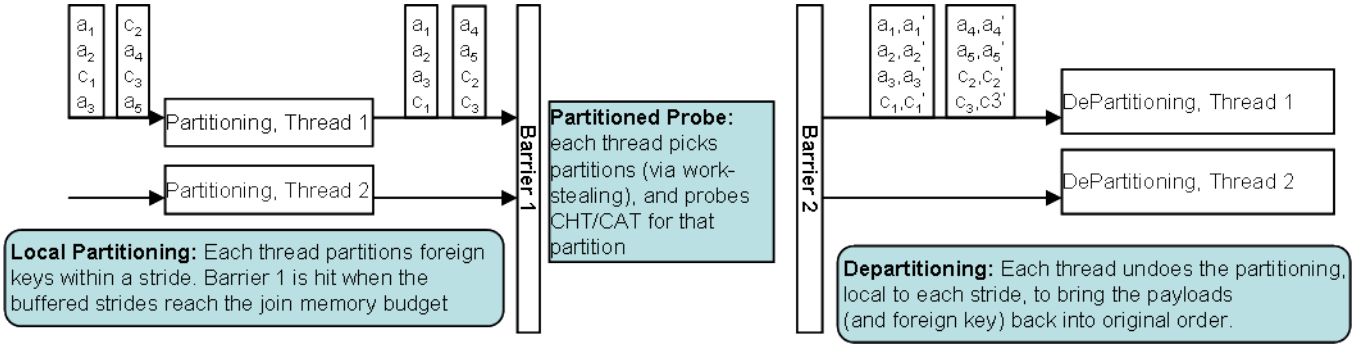


Figure 4: The pipelined partitioned join with local partitioning and barriers

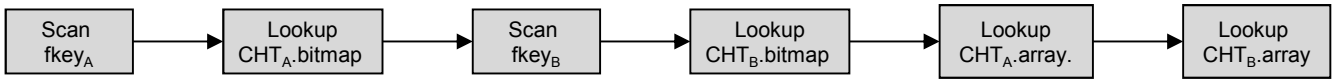


Figure 5: Execution plan for a query with two joins.

stride. At this point we have not even materialized other outer columns.

We use a fairly standard radix partitioning (compute histogram, then partition, as in [18]). Our strides are about 32K tuples, so we have not found TLB misses to be significant.

But we do no hash table probes at this point. Instead, we buffer up many outer strides, so that when we do the lookups, partition by partition, the hash table access has good temporal locality. This continues until the buffered strides reach the memory budget. At that point, all threads enter a barrier. Then each thread repeatedly picks a partition (via work-stealing) and does lookups for that partition.

We need a second barrier at the end, to *departition*. This brings join payloads back into the original tuple order, which is the order in which columns *other than the foreign key* are ordered, since we did not partition those other columns.

4. EVALUATION

We now evaluate the benefits of CHT and CAT, and our hash join using them. Our goal is to quantify the performance as well as the memory consumption.

We compare five hash joins. Two use **CHT** and **CAT**, implemented in Blink3 in a hash join with partitioned inner and non-partitioned outer. Two are the leading in-memory hash joins from [6]: **NPO** is a non-partitioned hash join and **PRHO** is a partitioned hash join. We use the code and driver graciously made available on the author's website (`http://www.systems.ethz.ch/node/334`). Those two hash joins use the chaining-based hash table of [8], enhanced by [6] to optimize it for the hardware. Last is **NPO-I**, which takes the NPO class from this codebase and integrates it into Blink3, to avoid materializing the join outers. We have also compared with Google SparseHash and DenseHash, and found them to be dominated on both speed and memory usage (these full results are available at [1]).

It must be noted that NPO and PRHO *do not perform a complete join*: the driver from [6] only computes a total count of the number of join matches (across all foreign keys), and neither accesses nor materializes join payloads. In contrast, in Blink3 the join operator materializes the join payloads and feeds them to an output operator that verifies that the join result is correct (for one tuple in every 8192 tuples).

Further, the driver for NPO and PRHO keeps the inner and outer table in contiguous in-memory arrays, and thus avoids the *read* system call as well as associated copies. Blink3 uses a scan operator to read input column values from files and to pass them to the join operator. All results are from warm runs, with both the inner and outer fitting in the file system cache (except in Section 4.4).

As a result, we see that NPO is about 2X faster than NPO-I. We believe that joins mostly do access join payloads, and thus NPO-I is a better measure of performance of the NPO hash table.

All but one experiment are run on a 2-socket machine with 8-core Intel Sandy Bridge-EP (E5-2665, 2.4 GHz) processors and 256GB of RAM, sufficient memory that there is no virtual memory paging, irrespective of memory usage. For the experiment with the outer streaming from disk, we use a 2-socket machine with 4-core Intel Xeon processors (X5570, 2.93 GHz), 64GB of RAM, and a Fusion-IO array of SSDs (ioExtreme Pro) that gets 545 MB/s sequential read bandwidth (measured via `dd`).

We have found that hyper-threading speeds up probes by hiding access latencies. This corroborates the results of [8]. But build speed is actually hurt by using more threads than cores, so for build in CHT and CAT, we use 16 threads, matching the number of physical cores. For NPO-I, we use only 8 threads, which gave it best performance. For the probe phase, we use 32 threads for all three algorithms. For NPO and PRHO, we use 32 threads uniformly, as the driver does not have distinct build and probe phases.

4.1 Savings in Memory Consumption

We first measure the memory usage of the various algorithms, using a 2-table equijoin query:

```
SELECT Inner.payload FROM Outer, Inner
WHERE Outer.fkey = Inner.key
```

The key is always an 8-byte integer. The payload is set to 8 bytes by default. Section 4.2 presents some results with 24-byte and 0-byte payloads.

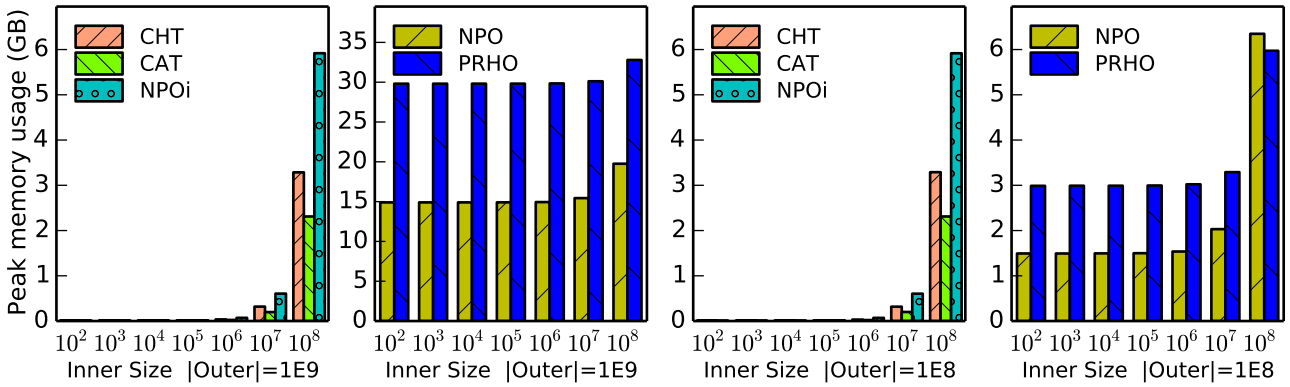


Figure 6: Memory usage for varying inner sizes (Table 1 has full numbers).

We use two cardinalities for the outer: 1E9 and 1E8 tuples. The inner cardinality is varied from 1E2 to 1E8 tuples. This range includes very small inners, in contrast to prior work [6, 8, 11] that mainly looks at large inners. In our experience with customer workloads, small inners are common, especially after applying predicates. In general, it is important for a join implementation to do well across a spectrum of cardinalities.

The values for the inner key are chosen as random (but unique) integers, with values in a range that is twice as large as the cardinality. This is to challenge CAT with non-dense key distributions. Each outer tuple finds exactly one match in the hash table. We choose foreign key values uniformly from among all the values in the inner. Section 4.5 studies variations, including even sparser key distributions, skew in the foreign key distribution, and selective joins.

We measure memory usage in terms of peak resident memory size (RSS), as measured by `ps`. We ran a script that issued "`ps -C command -o rss`" as each query ran, once every 100 ms, and recorded the largest value seen.

Dependence on Inner Size

Figure 6 plots the memory usage for the various joins, for each inner size, with an 8-byte payload. For small inners, the CHT, CAT, NPO-I memory usage is barely visible, but Table 1 has detailed numbers. For inners up to 1E7, CHT and CAT use $<$ 320MB (and $<$ 30MB up to 1E6 inners), in contrast to the 10s of GB that NPO and PRHO use. Even at 1E8 inner, CHT and CAT use 1/6th the memory of NPO, and 1/10th to 1/15th that of PRHO. The PRHO memory usage is dominated by the data structures used for partitioning the outer, so it varies little with the inner size.

Comparing to NPO-I: NPO's memory usage is high because it keeps the outer in an array. But this is not intrinsic to the NPO hash join: so NPO-I implements the NPO join, but with the outer streamed through. Still, NPO-I memory usage is 3X to 4X more than CHT's. This is due to two factors. First, the NPO hash table has a 16-byte overhead per bucket, for storing a pointer and a latch. Second, by design NPO is a chaining hash table, and there is significant space fragmentation from the overflow chain. At the 1E8 inner with 1E8 outer, the hash table and its overflow dominates the memory usage, so NPO-I is not that much better than NPO.

The RSS for CAT and CHT does increase with inner size, as we expect, since it is dominated by the inner. Table 1 also lists the exact hash table sizes for CAT and CHT; it is about half the RSS value. This is because we partition the inner, and thus have to hold in memory both the partitioned inner and the hash tables we are building. This can be avoided in low-memory environments by building hash tables on a few partitions at a time, or by spilling partition blocks.

We notice that the memory usage of CAT is always less than that of CHT. This is due to the keys that CHT stores.

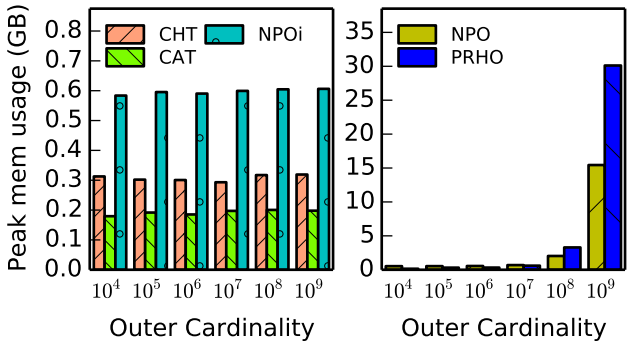


Figure 7: Memory usage for varying outer sizes (inner has 10M rows).

Dependence on Outer Size

Our key argument against outer partitioning is that it makes the memory usage grow with the outer size. We now plot the memory usage as a function of outer cardinality, in Figure 7. We fix the inner cardinality to 1E7. Notice that PRHO and NPO have a very steep upward slope, due to the intermediate state needed for partitioning the outer. CHT, CAT, and NPO-I memory usage is mostly independent of the outer, suggesting that Blink3 is pipelining the outer tuples well.

4.2 Is there a Trade-off in Join Speed?

Next, we turn to join speed. Figure 8 plots the query run times (wall clock) for the different hash joins, for an 8-byte payload. For CHT, CAT and NPO-I, each bar has two parts: the lower part is the time for the build phase, and the upper part is that for the probe phase. The code of [6] does not separate build and probe timings.

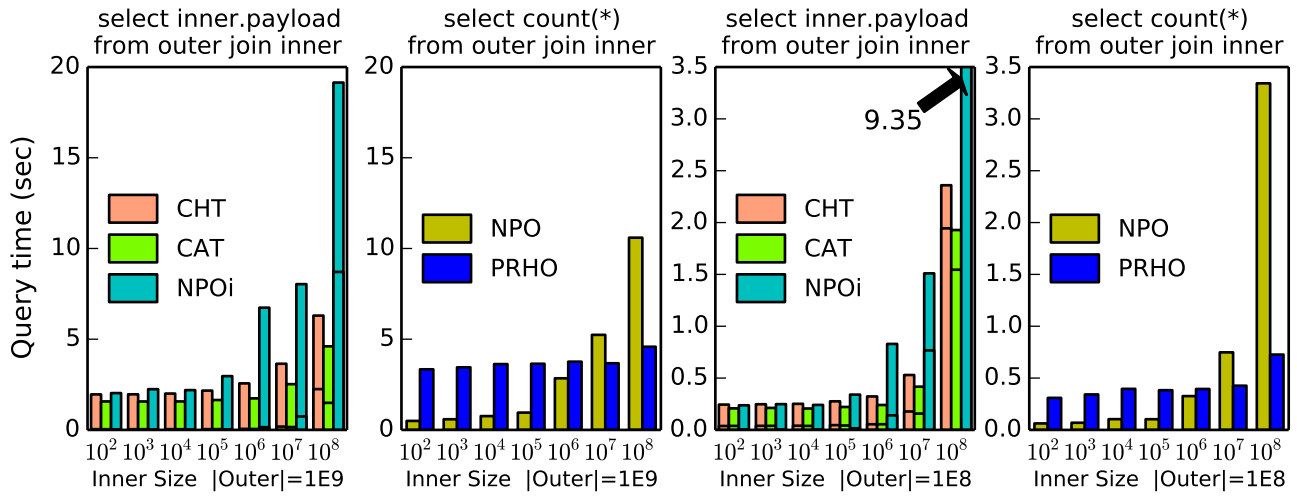


Figure 8: Query run time for varying inner sizes. For CHT, CAT, NPO-I we break the time into build (below the bar) and probe phases (above the bar).

As explained earlier, NPO and PRHO do not access payloads, so we plot those on the side (but to the same scale).

Dominance of Joins with Non-Partitioned Outer
Notice that the joins with non-partitioned outer – CHT, CAT, NPO-I and NPO – do very well, beating PRHO at all inner sizes except 1E8. PRHO has to partition the full outer, but at most inner sizes this is not an overall win. The PRHO times change little with the inner size, because the probes are always into small per-partition hash tables, and the inner size only affects the partitioning time, which is dominated by the outer, anyway.

The only case where PRHO wins is the 1E8 outer joined with 1E8 inner: here it is 2.6X faster than CAT. This is not primarily due to probe speed. We see that CAT and CHT build speed degrades for large inners, much more than probe speed. CHT and CAT do involve building an extra bitmap, but our profiling suggests this is not the bottleneck. Rather it is our inner partitioning and multi-threaded building that are slow. Since these two are not intrinsic to CHT or CAT, we hope to improve these in future, possibly applying some of the optimizations from PRHO.

Overall, these results validate that outer partitioning is not needed. Even in the case of the 1E8-1E8 join, PRHO will become slower if the join payloads need to be accessed, materialized, and fed to a subsequent operator.

CAT, CHT vs NPO-I and NPO
CAT and CHT beat NPO-I in every case, with larger savings for larger inners (eg, at 1E8 inner, CAT is $\approx$ 5X faster).

At inners of 1E6 and beyond, CAT is faster than NPO. This is despite CAT having to do one extra random access for the bitmap lookup. This validates our hypothesis that a hash table designed to use a DIRA pattern will not suffer much from random access. At smaller inners, NPO is the fastest. Blink3 has a minimum 1.5ns cost per outer row, which we attribute to the cost of accessing and materializing the join results, and to the `read()` system call.

CAT vs CHT
CAT is almost always faster than CHT. Blink3 uses CAT by default, whenever it can be applied. If the key is not of numerical type, or the range of values is too high, then the CAT bitmap can be very large. When this exceeds the memory budget, we fall back to CHT. We will see in Section 4.5 that even for sparse key distributions, the CAT memory usage is much smaller than that of PRHO or NPO.

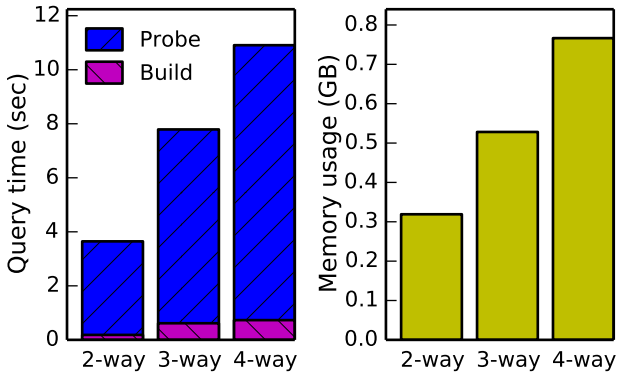


Figure 9: Query run time and memory usage for multi-way joins with CHT (1E9 outer, 1E7 inner).

Alternate payload sizes
We now repeat this query, with 0-byte and 24-byte payloads (only for CAT, CHT, since NPO and PRHO only support 8-byte key,payload and 4-byte key,payload). Table 1 lists the memory usage and join speed for these payload sizes. As we would expect, both memory usage and speed get worse for the wider payload. For the 0-byte payload, notice that CAT is especially effective (e.g., using 1/4th the memory and getting 7X the speed as compared to CAT with the 8-byte payload, for the 1E9 outer – 1E8 inner join). Here, CAT has neither key nor payload – it is just a bitmap.

4.3 Memory consumption for multi-way joins

One of our arguments for scanning the outer just once (and not partitioning it) was for multi-way joins: that it would be

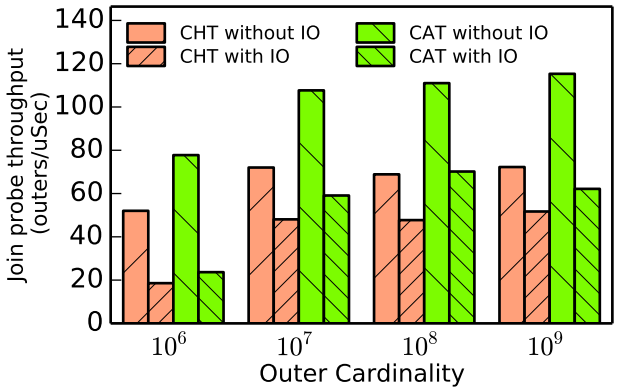


Figure 10: Effect of IO on probe speed (1E7 inner).

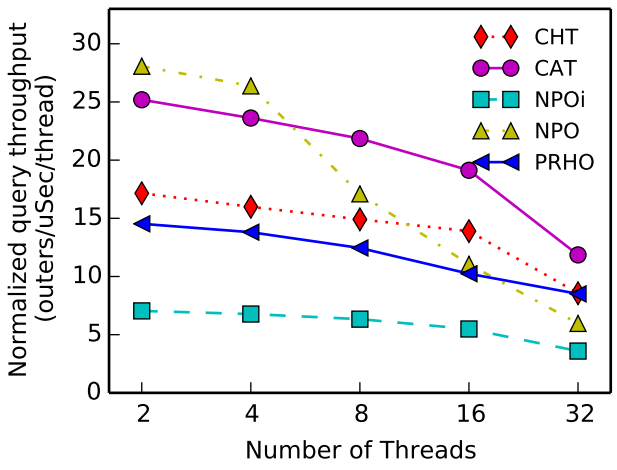


Figure 11: Threads scaling of CHT and CAT (1E9 outer, 1E7 inner)

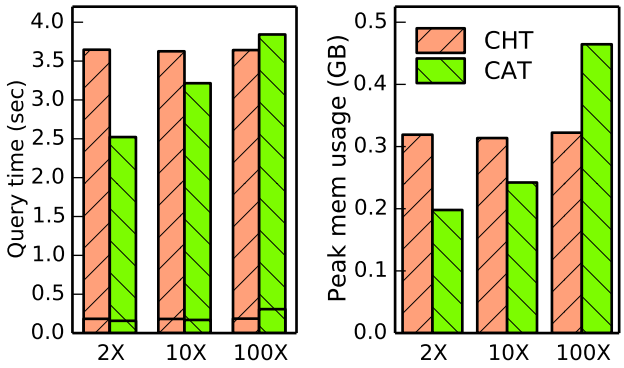


Figure 12: Query run time and memory usage at different key densities (1E9 outer, 1E7 inner).

prohibitively memory-intensive to partition the entire outer once per join. Our next experiment compares 2-way, 3-way, and 4-way joins. The driver of [6] only does 2-way joins, so we only study CHT here, to see how its performance fares as the number of joins in a query increases.

We join a 1E9 outer with one, two, or three 1E7 inners, on different foreign key columns. Figure 9 plots the query run time and the peak memory usage (from RSS) as we vary the number of dimensions being joined. Observe that the memory usage increases near-linearly with the number of dimensions, rising by only 200MB per join – an outer-partitioning scheme would consume much more per join. Query run time also increases linearly with the number of joins, since there is no filtration.

4.4 When the outer does not fit

So far, we have seen that non-partitioned outer joins do well, and have low memory usage. But our results have been on warm runs, on a machine with enough DRAM to hold the inner, outer, and all intermediate state in memory.

A natural question is, since we are pipelining the outer, can we read it from storage? Our next experiment forces the outer to be on disk and "cold", by explicitly dropping the file system cache before every run (via the Linux /proc/sys/vm/-drop_caches – we drop the page cache entries and i-nodes).

Then we rerun the speed experiment (only for CHT and CAT, because the input needs to be on files). As mentioned earlier, this uses a different machine, with Fusion-IO SSDs.

Figure 10 compares the time for the probe phase, with and without the dropping of the file system cache (we ignore build time because we expect the inner to almost always fit in memory). Our main observation is that our overall slowdown in going from outer-on-memory to outer-on-SSD is not that severe at all. This suggests that the focus on in-memory joins should shift away from trying to keep the outer in memory. Machines with modest memory will be cost-effective for join performance, especially in shared-nothing systems.

4.5 Sensitivity analysis

Our last set of experiments studies the sensitivity of these performance results to the number of threads used, and to the data distribution. We revert back to the 2x8-core Sandy Bridge machine for this experiment.

Multi-core Scaling

We first compare the join speed as we vary the number of threads. Figure 11 plots the query speed as we vary the number of threads used, for a 1E9 (outer) – 1E7 (inner) join, with 8-byte payloads (the numbers for other sizes were similar). We normalize the speed by the number of threads, so perfect scaling would be a horizontal line. For CAT, CHT and PRHO, the scaling is good until 16 threads, which matches the number of physical cores, but we do see some benefits beyond that as well.

Speed of CAT at different densities

Next we vary the density of the join key values. Until now, we chose the join keys as uniform random integers, in a range twice as large as the inner cardinality. Now, we consider joins where the range is 10X and 100X the inner cardinality. Figure 12 plots the speed and memory consuption of both CHT and CAT. We use a 1E9 outer and a 1E7 inner. As expected, CHT is little affected by sparser keys. CAT does slow down slightly, and its memory usage does go up, because its bitmap size is proportional to the range of key values. Even at 100X density, CAT consumes less than 500MB.

Effect of Skew: We now consider skewed foreign key distributions. We reran the query speed experiments on

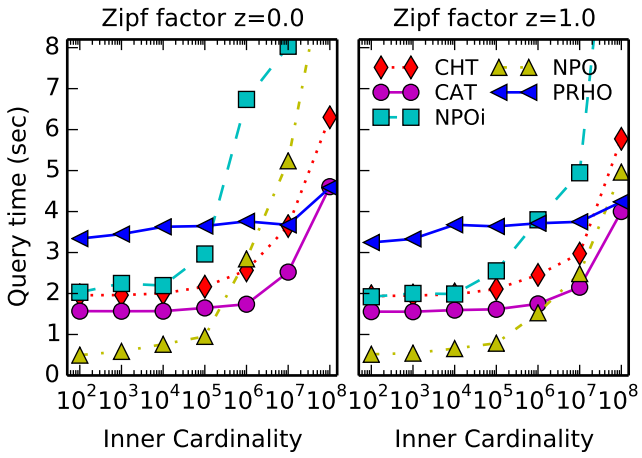


Figure 13: Beneficial effect of skew (1E9 outer).

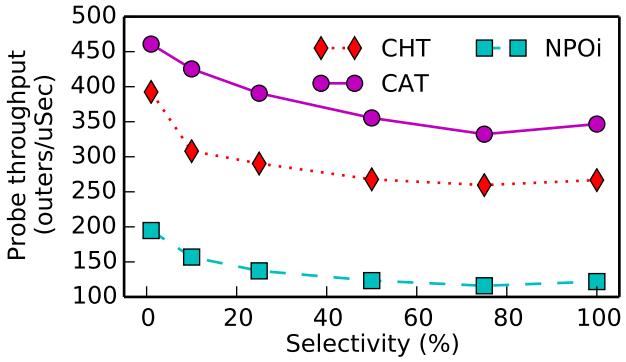


Figure 14: Join speed at different join selectivities.

data generated with a Zipfian distribution for the foreign key column Figure 13 compares the query speed for varying inner sizes, with and without skew. The main effect of skew is to make large inners behave more like small inners, because the hash table entries for frequent join keys fit well in cache.

Effect of Join Selectivity: So far in all our joins every outer tuple finds a match. Now we turn to selective joins, where the CHT and CAT bitmap is used to filter out foreign keys early, without visiting the array of keys and payloads. Figure 14 plots the probe time for CHT, CAT, and NPO-I as we vary the join selectivity from 0 to 100%. We observe a 40% speedup at 0% selectivity, compared to at 80% selectivity. Notice also that 100% selectivity yields slight speedup over 80% selectivity, due to better branch prediction.

5. RELATED WORK

Join processing is one of the most important data management operations. Hence, it is a very old and long studied topic. Hash joins were pioneered in the 1980s [12, 10], and have been implemented in almost all DBMSs since then, and undergone numerous improvements. For example, Microsoft SQL Server 2000 starts using an in-memory hash join and gradually transitions to grace hash join, and recursive hash join, depending on the size of the build input [2].

In-memory joins have seen much recent attention, especially with column stores. MonetDB pioneered this trend [17], introducing the radix partitioned join algorithm, and pointing out the potential of optimizing for cache and memory. This style of partitioning has been used in many subsequent papers, including [11] and [5].

Blanas et. al [8] were the first to observe that non-partitioned joins work just as fast as partitioned joins, especially on hyper-threaded machines, and that they are much simpler. CHT and CAT can be applied for both partitioned and non-partitioned joins. But in this paper we have pointed out additional challenges: for partitioned joins, the difficulty of fitting the entire outer in memory, and for non-partitioned joins: estimating the hash table size during builds (and the resize it introduces if mis-estimated), and the problem of build parallelism. We think our technique of partitioned build and non-partitioned probe can be applied to other hash table data structures as well, besides CHT and CAT.

Cuckoo hashing [19] is a powerful way to avoid hash table collision, and still get high fill factors. Like CHT, a cuckoo hash table probe also involves up to two random lookups. But both lookups are into a wide data structure (the array of keys and payloads), whereas for CAT and CHT the first lookup is into a bitmap. Splash tables [21] place multiple keys into the same hash bucket to optimize for cache line accesses. We believe this optimization is complementary to CHT, and intend to explore this in future.

C-Store [3] introduced late materialization of join payloads, which cuts the size of the join hash table. But one pays after the join, to fetch payload columns using RID – which will involve random page accesses and extra buffer pool fixing costs. We prefer to use Bloom filters, as suggested in $R^\star$ [16], via the CHT bitmap. This cuts the cuts the width needed for the data structure used for the initial filtering. A recent enhancement to the NPO hash table [14] uses 16 bits from the overflow pointer as a filter, to avoid following the pointer. We don't need this optimization with CHT because overflows are much rarer.

Begley et al [7] also study the problem of memory-constrained joins. Their block nested loop like solution is appealing in the case that the inner cannot fit in memory, and can complement our approach of non-partitioned outers.

This paper focuses on hash joins, but sort-merge join is a powerful alternative, with good recent progress (e.g., [4]). Generally, hash join allows better memory efficiency because we can keep the smaller table as the inner and stream the larger table. A careful comparison between sort and hash join is an important topic for future work on fast joins.

There is rich recent literature on hardware-conscious joins [18, 9, 11, 8, 4, 6, 5]. [4] proposes a partitioned join that minimizes random inter-socket reads, and [15] improves upon that with a NUMA-aware data shuffling stage. [13] presents a latch-free hash table design for scalable NUMA-aware build phase. We think that simplifying hash joins to a series of DIRA lookups will make hardware acceleration easier, because we can repeatedly use a gather primitive.

6. CONCLUSIONS

The availability of large, cheap memories has led to a renaissance in equijoin algorithms. The focus of the research literature has mostly been towards improving join speed, and less attention has been paid to memory consumption. We

Outer cardinality = 1E8

In-	Payload = 8 byte					Payload = 24 byte	Payload = 0 byte	
ner	CHT	CAT	NPOi	NPO	PRHO	CHT	CHT	CAT
1E2	.01/2.1, .04+.21	.01/1.3, .04+.17	.01/3.0, .00+.23	1.5, .06	3.0, .31	.02/3.7, .04+.23	.01/1.3, .02+.19	.01/5.0e-1, .02+.14
1E3	.01/1.8e1, .04+.21	.01/8.3, .04+.17	.01/2.4e1, .00+.25	1.5, .07	3.0, .34	.02/3.3e1, .05+.24	.01/9.8, .02+.18	.01/5.0e-1, .02+.14
1E4	.01/1.9e2, .04+.21	.01/8.6e1, .04+.17	.01/3.8e2, .00+.24	1.5, .10	3.0, .40	.02/3.4e2, .04+.26	.01/1.1e2, .02+.19	.01/8.0, .02+.14
1E5	.01/1.8e3, .05+.23	.01/8.5e2, .04+.18	.01/3.1e3, .02+.32	1.5, .10	3.0, .38	.02/3.4e3, .05+.28	.01/1.0e3, .03+.19	.01/6.4e1, .03+.14
1E6	.03/1.8e4, .06+.27	.02/8.3e3, .06+.19	.07/2.5e4, .14+.69	1.5, .33	3.0, .39	.07/3.3e4, .07+.37	.02/9.9e3, .04+.21	.01/5.1e2, .04+.14
1E7	.32/1.7e5, .18+.35	.20/8.6e4, .16+.26	.60/3.9e5, .77+.74	2.0, .75	3.3, .43	.79/3.3e5, .30+.50	.16/9.5e4, .11+.31	.03/8.2e3, .08+.15
1E8	3.3/1.8e6, 1.9+.42	2.3/8.5e5, 1.5+.38	5.9/3.1e6, 8.2+1.1	6.3, 3.3	6.0, .73	8.6/3.4e6, 3.7+.61	1.8/1.0e6, 1.5+.37	.82/6.6e4, .48+.23

Outer cardinality = 1E9

1E2	.01/2.1, .04+1.9	.01/1.3, .04+1.5	.01/3.0, .00+2.0	15, .50	30, 3.3	.02/3.7, .04+2.1	.01/1.3, .02+1.7	.01/5.0e-1, .02+1.3
1E3	.01/1.8e1, .04+1.9	.01/8.3, .04+1.5	.01/2.4e1, .00+2.2	15, .59	30, 3.4	.02/3.3e1, .05+2.2	.01/9.8, .02+1.7	.01/5.0e-1, .02+1.3
1E4	.01/1.9e2, .04+2.0	.01/8.6e1, .04+1.5	.01/3.8e2, .00+2.2	15, .76	30, 3.6	.02/3.4e2, .05+2.3	.01/1.1e2, .02+1.7	.01/8.0, .03+1.3
1E5	.02/1.8e3, .04+2.1	.01/8.5e2, .04+1.6	.01/3.1e3, .02+2.9	15, .95	30, 3.6	.03/3.4e3, .05+2.6	.01/1.0e3, .03+1.8	.01/6.4e1, .03+1.3
1E6	.04/1.8e4, .06+2.5	.03/8.3e3, .05+1.7	.07/2.5e4, .15+6.6	15, 2.8	30, 3.8	.08/3.3e4, .07+3.5	.02/9.9e3, .04+2.0	.01/5.1e2, .04+1.3
1E7	.32/1.7e5, .18+3.5	.20/8.6e4, .16+2.4	.61/3.9e5, .74+7.3	15, 5.2	30, 3.7	.80/3.3e5, .29+4.9	.16/9.5e4, .11+3.0	.03/8.2e3, .08+1.4
1E8	3.3/1.8e6, 2.2+4.1	2.3/8.5e5, 1.5+3.1	5.9/3.1e6, 8.7+10	20, 11	33, 4.6	8.6/3.4e6, 3.7+6.0	1.8/1.0e6, 1.5+3.7	.82/6.6e4, .49+2.0

Table 1: Detailed speed and memory usage measurements. Format: RSS in GBs/hash table size in KBs, Build time + Probe time (seconds). For NPO, PRHO, hash table size and build-probe split is not available. Hash table for NPO-I excludes the overflow chains. 24-byte data for CAT is similar to CHT.

have introduced two new concise hash tables, and equijoin algorithms using these tables, that significantly reduce memory consumption compared to leading in-memory join algorithms. At the same time, their join speed is still competitive.

Traditional partitioned joins that partition the entire outer, as well as non-partitioned joins that do no partitioning at build time, have serious limitations in terms of I/Os and parallelism, respectively. Instead, our equijoin scans the outer in pipelined fashion, and benefits from build-side partitioning even when the probe side is non-partitioned.

We hope this paper leads to more research at the boundary between fully in-memory and on-disk joins. We are excited by the potential of CHT and CAT, and would like to apply them in the future to tasks such as group-by. Graph databases are another area where is a lot of equijoin-like pointer chasing, and structures such as CAT can be useful.

7. REFERENCES

[1] Full experimental results. http://researcher.ibm.com/person/us-ravijay.

[2] Understanding hash joins. Microsoft TechNet SQL Server.

[3] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, 2007.

[4] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10), 2012.

[5] C. Balkesen, G. Alonso, J. Teubner, and T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 2014.

[6] C. Balkesen, J. Teubner, G. Alonso, and T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, 2013.

[7] S. K. Begley, Z. He, and Y.-P. P. Chen. Mcjoin: A memory-constrained join for column-store main-memory databases. In *SIGMOD*, 2012.

[8] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, 2011.

[9] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), 2007.

[10] D. J. DeWitt et al. Implementation techniques for main memory database systems. In *SIGMOD*, 1984.

[11] C. Kim et al. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PLVDB*, 2(2), 2009.

[12] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1), 1983.

[13] H. Lang et al. Massively parallel NUMA-aware hash joins. In *IMDM*, 2013.

[14] V. Leis et al. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[15] Y. Li et al. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[16] L. F. Mackert and G. M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *VLDB*, 1986.

[17] S. Manegold, P. Boncz, and M. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB Journal*, 9(3), 2000.

[18] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *VLDB*, 2000.

[19] R. Pagh and F. F. Rodler. Cuckoo hashing. In *ESA*, 2001.

[20] V. Raman et al. DB2 with BLU Acceleration: So much more than just a column store. *PVLDB*, 6, 2013.

[21] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, 2007.