# CAMEL Hash Table: Striking a Balance Between CPU and Memory Efficiency in Main-Memory Hash Join

Sudip Chatterjee
Xiaozheng Zhang
Suprio Ray
sudip.chatterjee@unb.ca
xz.zhang@unb.ca
sray@unb.ca
University of New Brunswick, Canada

Ian Finlay
Calisto Zuzarte
Mark Stoodley
finlay@ca.ibm.com
calisto@ca.ibm.com
mstoodle@ca.ibm.com
IBM, Canada

## Abstract

Many hash tables used in hash joins incur substantial memory consumption to guarantee constant-time lookup performance. Hash joins, which rely on these tables for probe operations, have become increasingly memory-intensive due to the growing volume of data in modern workloads. As a result, memory-efficient hash table designs are critical for maintaining overall query performance. To address this challenge, the Concise Hash Table (CHT) was introduced as a space-efficient alternative that supports hash joins while utilizing available memory more effectively. However, our experimental evaluation reveals that the performance of CHT-based hash joins degrades with increasing data volumes and large results, limiting its scalability in real-world scenarios.

Modern processors attempt to hide cache and translation lookaside buffer (TLB) misses by employing out-of-order execution. However, hash table lookups are highly unpredictable, which can result in severe cache miss rates and low performance. Moreover, cache sizes in modern processors have not grown commensurately with increasing data volumes, making data prefetching and efficient cache utilization increasingly important. In this work, we propose a novel hash table, **`CAMEL-HT`**: a *<u>C</u>ache <u>A</u>ware, <u>M</u>emory <u>E</u>fficient, <u>L</u>ightweight* hash table that delivers high performance while consuming significantly less memory. Our experiments demonstrate that `CAMEL-HT` improves hash join performance by 2× over CHT with only a marginal increase in memory usage. `CAMEL-HT` consistently achieves 3× better memory efficiency compared to VIP Hashing, a state-of-the-art pointer chasing hash table, while also outperforming it in several scenarios in terms of execution time. `CAMEL-HT` matches the performance of the state-of-the-art hash table designs while achieving memory efficiency comparable to the most compact hash table approaches.

## Keywords

Hash Join, In-Memory, Cache Aware, Memory Efficient, Compact Hash Table, CAMEL-HT

## 1 Introduction

Hash tables [11] are foundational data structures designed to efficiently manage key-value pairs, offering constant-time lookups, low computational overhead, and favorable memory access patterns. These properties make them particularly well-suited for high-performance data processing in modern database systems.

*EDBT '26, Tampere (Finland)*

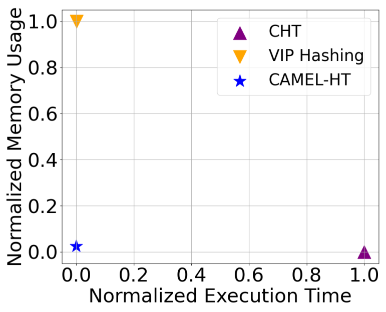


**Figure 1: Normalized execution time and memory usage (min–max scaling) for Concise Hash Table (CHT), `CAMEL-HT`, and VIP Hashing on the TPC-H dataset (scale factor 10)**

They are particularly relevant for in-memory indexing [29, 37], hash joins [1, 3, 4, 7, 10, 14, 16, 31], and aggregation operations [24, 27], where fast key-based access and efficient handling of large datasets are essential for performance.

The performance of the hash-join operator is heavily influenced by the underlying hash table design. Numerous hash-table architectures have been proposed over the years to optimize this operation. In general, these designs can be categorized based on their collision resolution strategies into two primary classes: *separate chaining* [11] and *open addressing* [11]. In separate chaining, each hash bucket maintains a linked list to handle collisions by storing all entries mapping to the same index. Open addressing, on the other hand, stores all entries in a contiguous array and resolves collisions through probing strategies, such as linear or quadratic probing, to locate alternative empty slots.

Due to the flexibility of separate chaining, it is a popular choice for many existing hash join implementations [3, 4, 7, 14, 31]. Moreover, separate chaining with embedded Bloom filters [8] enables both efficient probing and parallel construction [21]. However, it has been shown that separate chaining hash table achieves poor performance with skewed datasets [23], which are prevalent in real-world datasets [12]. Birler et al. [6] demonstrate that lookup time grows significantly in chained hash tables compared to unchained designs. Moreover, dataset skew may arise in the result set of a join operation, which is then joined with another intermediate result set or a table. Besides data skew, the sort order of the data also impacts the performance of the hash join. Recently, the VIP Hashing [18] approach was proposed that adapts to data skew by moving popular keys to the front of the linked list in a separate chain-based structure. However, being a separate chaining approach, VIP Hashing is not memory efficient. As shown in Figure 1, while VIP Hashing achieves good performance in terms of execution time, it exhibits considerably higher memory consumption compared to other hash table implementations.

Although main memory is becoming cheaper, it remains one of the most constrained resources in production environments, where concurrent queries compete for limited memory. Hence, efficient memory usage is critical for hash joins, yet research on memory-efficient hash join algorithms remains limited. To that end, Concise Hash Table (CHT) [5], which achieves a 100% fill factor for the hash table, is arguably among the state-of-the-art in terms of memory efficiency. CHT is an open addressing hash table that leverages several techniques to achieve a 100% fill factor. Hence, adopted for in-memory hash-joins by a few databases, including IBM Db2. As shown in Figure 1, execution time and memory usage are normalized using min–max scaling to the [0, 1] range for direct comparison across hash tables. While CHT is efficient in terms of memory usage, it falls short in execution time performance compared to VIP Hashing, making it less suitable for scenarios where CPU performance is a critical factor.

Our research aims to address the limitations of existing hash table approaches for hash join and achieve both CPU and memory efficiency. Separate chaining is not suitable for modern processors, as it is not cache-friendly due to its pointer-chasing nature. Also, a large contiguous array used in open addressing approaches often cannot be fully accommodated by the on-chip cache hierarchy. This work explores a middle ground by grouping elements with similar hashed keys into small, cache-resident arrays per bucket. Building on this idea, we propose CAMEL-HT, a CPU-optimized, memory-efficient, lightweight hash table that leverages these “lean” data structure to exploit cache locality during both build and probe phases. Similar to our approach, recently Birler et al. proposed a dense hash table [6] that achieves a 65% fill rate, whereas our design attains a significantly higher 99% (§5.7) fill rate.

The second contribution of this paper explores the prefetching approaches taken by the software prefetcher and propose a new one. Modern processors support multiple concurrent cache and TLB misses and provide explicit prefetch instructions to exploit memory-level parallelism. In database engines, state-of-the-art prefetching techniques like group prefetching [9], software pipelining [9], and asynchronous memory access chaining (AMAC) [20] improve performance by overlapping memory accesses. In this paper, we propose a prefetching approach that we call *Next Expected Use (NEU)* to minimize the number of software prefetch instructions. It uses a small buffer to store the potential key that could be accessed next.

We evaluate the performance and characteristics of CAMEL-HT in the context of in-memory hash joins, using both microbenchmarks and standard workloads such as TPC-H. We perform a detailed comparative analysis, evaluating our approach against VIP Hashing and CHT. We employ a single-threaded execution model consistent with the approach utilized in VIP Hashing during our experiments. CAMEL-HT achieves 2.24×, and 1.42× improvement in execution time over CHT, and VIP hashing, respectively. It has a comparable memory footprint to CHT while being 3× more memory-efficient than VIP Hashing. As shown in Figure 1, CAMEL-HT stands out as one of the most efficient hash tables in terms of both memory usage and execution time. However, it is important to note that CAMEL-HT currently does not support mixed read/write operations.

Our contributions are as follows:

(1) ***CAMEL-HT*** (§3) — We developed CAMEL-HT, a processor, memory, and cache-efficient hash table that matches the minimal memory footprint of the Compact Hash Table (CHT), while delivering significantly better performance across a range of workloads due to its optimized data layout and reduced access overhead.
(2) ***Next Expected Use*** (§3.3,§4.2) — We proposed Next Expected Use (NEU), a speculative cache prefetching approach for array-based structures that builds upon AMAC, a state-of-the-art prefetching technique.
(3) ***Insertion optimization during build phase*** (§4.1) — We developed an efficient hash table construction mechanism that enables constant-time insertions by eliminating the overhead of linear probing for locating vacant slots during insertion.
(4) ***Hash join evaluation*** (§5) — We conducted a systematic evaluation of CAMEL-HT within the framework of in-memory hash joins, demonstrating its superior efficiency and performance under realistic workload conditions. Although most of the experiments are conducted using N:1 joins, additional experiments show that CAMEL-HT naturally extends to M:N joins, highlighting its broader applicability (§5.6.2).

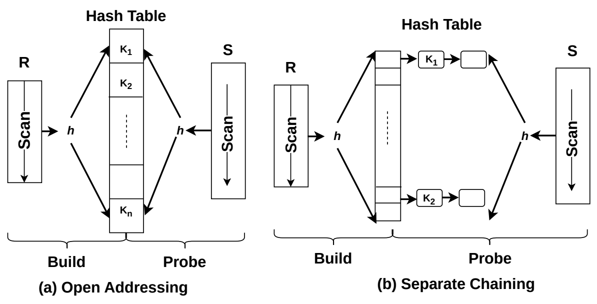


**Figure 2: An illustration of a canonical hash join**

## 2 Background and Related Work

Traditional query execution models, designed for disk-based systems, are being reevaluated as entire databases now reside in main memory. This shift demands rethinking core operators like hash joins. In this work, we investigate how to redesign hash tables for improved CPU efficiency and memory performance in modern in-memory settings.

### 2.1 Hash Join

Hash join algorithms are typically categorized into *hardware-oblivious* [7] and *hardware-conscious* [33] approaches, reflecting how they interact with system architecture. Hardware-oblivious methods aim to provide platform independence and ease of implementation. They rely on simple designs, such as the classic build-probe hash join[7], without explicitly leveraging hardware-specific features. In contrast, hardware-conscious techniques are meticulously engineered to exploit the underlying architectural characteristics, including cache hierarchies, SIMD [17] capabilities, and NUMA [17] effects. Radix partitioning is a prominent example of this class, designed to improve cache utilization and memory locality [3, 19]. Most existing research on hash joins has focused primarily on performance efficiency, often overlooking memory efficiency. Despite the importance of balancing both, limited research has addressed the design of hash joins that achieve this dual objective. A notable exception is the Concise Hash Table (CHT), which explicitly targets both memory and performance efficiency. In this paper, we revisit the canonical hash join, as adopted in VIP Hashing [18] and illustrated in (Figure 2) — which comprises a build and a probe phase. Notably, this design can be parallelized to resemble the no-partitioning join approach, as demonstrated by Balkesen et al. [3].

In a typical hash join, $|R|$ and $|S|$ denote the cardinalities of the build-side and probe-side relations respectively, where $|R| < |S|$. During the build phase, each join key from relation $|R|$ is hashed to determine the target slot in the hash table, into which the corresponding key-value pair is inserted. In the subsequent probe phase, the same hash function is applied to each key in $|S|$ to locate the candidate bucket, followed by a localized search—typically implemented using a linked list or an array—to identify matching entries. The performance of this process critically depends on the quality of the hash function and appropriate bucket sizing, which together can provide near-constant time access under typical workload conditions.

## 2.2 Hash table

Hash tables are widely used in database systems due to their potential to offer constant-time lookups under ideal conditions. Classic designs aim to distribute keys uniformly across buckets to avoid collisions. High load factors improve space occupancy but often lead to elevated collision rates. In separate chaining, this manifests as longer linked lists and intensive pointer chasing, which significantly hinders performance on modern CPUs due to poor cache locality and increased memory indirection. Open addressing techniques, while avoiding the pointer chasing problem [22], degrade performance under high load factors.

Recent work has explored optimizations along both axes of memory efficiency and performance. The Concise Hash Table (CHT) [5] adopts an open-addressing scheme that minimizes memory usage and enhances cache efficiency by reducing pointer overhead. On the other hand, VIP Hash Table exemplifies a performance-driven approach that relies on pointer chaining. It adaptively reorders linked-list entries by promoting frequently accessed (hot) keys toward the front, thereby reducing average lookup time. However, this learning-based optimization introduces runtime overhead, particularly during phases of workload transition. These designs exemplify the trade-offs between spatial locality, memory consumption, and adaptive behavior, motivating our investigation into models that can better balance these competing factors.

CHT addresses these issues by storing elements in a compact array, eliminating the memory inefficiencies of pointer-based designs. It is an open-addressing hash table optimized for high memory efficiency by achieving a 100% fill factor, making it highly suitable for in-memory database systems. It avoids pointer-chasing by storing elements in a compact array and pre-determining the table size during the build phase. Using a sparse *bitmap*, CHT significantly reduces collisions and minimizes linear probing during insertion. However, under skewed workloads, hot keys may incur longer probe sequences, which can degrade performance.

VIP Hashing [18], a recently proposed separate-chaining-based hash table design, targets the challenge of expensive lookups under skewed data distributions. VIP Hashing learns the frequency of access and adapts the hash table on the fly. Frequently accessed data are moved to the front, reducing the number of hops during lookup. Since this learning process may become an overhead, it uses a dynamic "switch" to stop moving the frequent elements after the learning phase. While VIP Hashing exhibits robust performance, it incurs significant memory overhead due to the maintenance of numerous statistics and being a separate chaining approach.

## 2.3 Software prefetching

Software prefetching [26] aims to improve performance by loading data into the cache before its use, thus reducing latency. However, predicting future data access is difficult when access patterns are random, as is the case with hash table lookups. This randomness makes applying software prefetching to hash joins particularly challenging. Despite this, research has explored ways to enhance hash join performance using software prefetching.

Group prefetching [9] and software pipeline prefetching [9] are two major prefetching techniques used in hash tables. Both approaches try to exploit inter-tuple parallelism to overcome cache misses while processing one tuple and avoid latencies for other tuples. While group prefetching hides the latencies in a group, software pipeline prefetching hides latencies across iterations by combining different stages of code into an iteration.

Although these two approaches may improve performance when compared to no prefetching, they have some limitations. If the pointer difference exceeds or is smaller than the group size or number of pipeline stages, extra computations are introduced. To overcome these shortcomings, Asynchronous Memory-Access Chaining (AMAC) [20], a software prefetching technique, was proposed for pointer-chasing structures, such as hash tables and trees. It hides memory latency by exploiting inter-lookup parallelism. Unlike traditional prefetching, AMAC tracks each lookup independently, adapting to irregular and complex access patterns while overlapping computation with memory operations. Despite its advantages on earlier architectures, AMAC requires explicit software management and often underperforms on modern CPUs with adaptive hardware prefetchers. Our empirical results further confirm this limitation, as implementing AMAC on CHT led to performance degradation. Menon et al. [25] introduced *relaxed operator fusion*, which combines SIMD vectorization and software prefetching(*group prefetching*) to reduce memory latency and pipeline stalls, a promising avenue for future optimization in our system.

## 3 CAMEL hash table design

Hash table designers mostly focus on lookup performance, as the access pattern is random. However, as the size of the build table increases, the build phase can dominate the overall execution time, often consuming a significant percentage of the total processing time. Therefore, we focus on both building and probing in our proposed data structure. Hash tables either use linear data structures like an array or pointer chasing data structures like a linked list. Both have some advantages and disadvantages. We take the best of both architectures and propose a hybrid model, `CAMEL-HT`, where related data can be stored in a small array. Each bucket points to an array where each cell holds an 8-byte key and an 8-byte value or a pointer to a tuple, as depicted in Figure 3. While keys of different sizes (e.g., 4B, 16B, or 32B) are possible, our current work does not address this case. As future work, we plan to incorporate 2-byte fingerprints with an 8-byte tuple address, enabling a lightweight match during probing, followed by a full key verification through the pointer. This extension would further reduce the overall size of the hash table.

`CAMEL-HT` uses a bitmap [34] to filter out the probe data during lookup operations and a helper data structure to determine the number of elements per bucket during the build phase. Additionally, we incorporate a software prefetching technique, called Next Expected Use (NEU) prefetching, to enhance lookup performance. Our NEU approach adapts AMAC to make it more suitable for our processing during the probe phase. The key considerations behind the design of CAMEL-HT are mentioned below.

- We ensure that CAMEL-HT uses a minimum amount of memory to avoid unnecessary overhead. This involves structuring the data efficiently, eliminating unnecessary allocations, and minimizing the memory footprint while maintaining optimal throughput.
- To be highly efficient in cache usage and processor performance, we organize the data in a way that maximizes cache locality. Additionally, we ensure that CAMEL-HT minimizes the number of CPU cycles required for common operations, such as searching, and inserting.

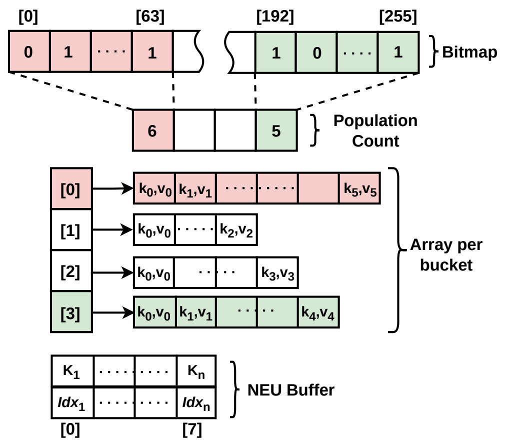


**Figure 3: High-level architecture of CAMEL-HT. The bitmap segment (0–63 and so on) represents a 64-bit integer used for filtering and determining the population count. Population count captures the number of set bits and collisions, indicating the number of key-value pairs per bucket. The red and green colors highlights related entities.**

The major components of the CAMEL-HT are described next.

## 3.1 Bitmap

We leverage a bitmap [34] for two distinct but complementary roles in our hash table design (Figure 3). The bitmap is implemented as an array of 64-bit integers, where each integer compactly encodes the occupancy of 64 positions using individual bits. We use bitmask operations to set or check specific bits within each integer (Figure 4), enabling a space-efficient representation of the bitmap. During the build phase, we set the corresponding bit before inserting a key into its target bucket, allowing accurate estimation of the cardinality per bucket. During the lookup phase, the bitmap acts as a fast pre-filter to bypass buckets known to be empty, thereby minimizing unnecessary memory accesses and improving probe performance.

The size of the bitmap is a critical design choice in our architecture. We allocate the bitmap to be $k$ times larger than the number of input items, where $k$ is a configurable parameter (typically a power of two such as 4 or 8). To further optimize bit-level operations, the total bitmap size is rounded up to the next power

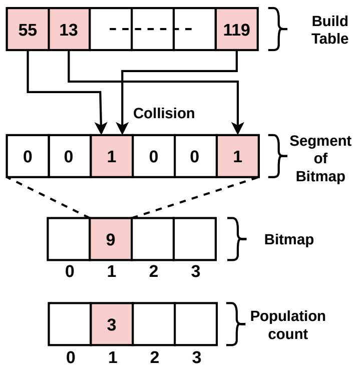


**Figure 4: Example of bitmap and population count. Colliding items (e.g., 55 and 119) increment the population count (e.g., to 3), while the bitmap simply reflects the set bit (e.g., decimal 9).**

of two [15, 30], enabling efficient bitmasking and alignment. The bitmap is divided into segments, with each segment implemented as a 64-bit integer. Let $n$ denote the number of input items. The total number of bits in the bitmap, denoted by $B$, is determined by:

$$B = k \times 2^{\lceil \log_2(n) \rceil} \quad \text{where } k \in \{2^i \mid i \in \mathbb{N},\ k \geq 1\}$$

Each 64-bit word (segment) is used to represent one bucket. Thus, the total number of buckets $b$ is:

$$b = \frac{B}{64} = \frac{k \times 2^{\lceil \log_2(n) \rceil}}{64}$$

Assuming a well-distributed hash (such as, MUM Hash [28]) function, the expected number of keys ($\mu$) per bucket is:

$$\mu = \frac{n}{b} = \frac{n \times 64}{k \times 2^{\lceil \log_2(n) \rceil}}$$

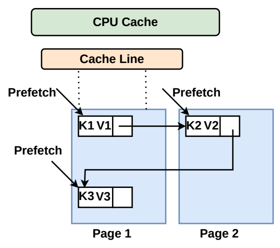
This configuration ensures that each bucket receives a moderate number of keys on average, which allows accurate per-bucket tracking via a population count array. The use of power-of-two alignment also facilitates efficient bitmasking for segment indexing. Overall, this approach strikes a balance between memory efficiency and predictable performance, especially under varying input sizes and data distributions.

## 3.2 Population count

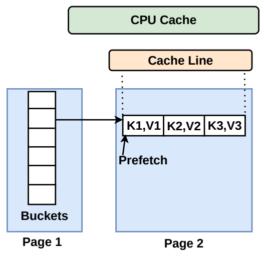
We maintain a separate population count array to track the number of keys assigned to each bucket as illustrated in Figure 3. Each bucket is represented by a 64-bit integer in the bitmap, where each set bit indicates the presence of a key. During the build phase, for every key insertion, we set the appropriate bit in the corresponding 64-bit integer using a bitmask and simultaneously increment the associated counter in the population count array. This approach ensures accurate tracking of the number of keys per bucket, even in the presence of hash collisions. The population count array is then used to allocate a dynamic array of appropriate size for each bucket. We do not employ linear probing within the bitmap; instead, all keys mapped to the same bucket are stored in its corresponding array.

## 3.3 Next Expected Use (NEU) prefetching

Cache prefetching is a well-established technique for mitigating cache miss rates and associated latency, significantly improving



**(a) Asynchronous Memory Access Chaining (AMAC)**



**(b) Next Expected Use Prefetching (NEU)**

**Figure 5: Comparison of software prefetching in AMAC and our cache-conscious NEU strategy.**

the performance of sequential data access compared to random access. Modern CPU architectures are highly optimized for spatial locality[13], where contiguous memory access patterns significantly outperform pointer-chasing structures in terms of cache efficiency. Linked lists, due to their inherently non-contiguous memory layout, suffer from poor spatial locality and frequent cache misses during traversal. In contrast, small arrays benefit from contiguous allocation, enabling hardware prefetchers to load multiple adjacent elements into cache with a single memory access. This efficient utilization of cache lines leads to lower memory latency and improved throughput.

To leverage these hardware characteristics, our architecture uses compact arrays to store key–value pairs within each bucket, ensuring both precise control over memory layout and enhanced cache performance, as illustrated in Figure 3. We evaluated AMAC [20], a state-of-the-art software prefetching technique designed for hash tables with separate chaining. AMAC issues fine-grained prefetch instructions for individual nodes in a linked list as illustrated in Figure 5a. However, in our array-based design, such element-level prefetching becomes counterproductive. Since adjacent elements are already stored within the same cache line, prefetching them individually leads to redundancy, increased instruction overhead, and potential saturation of the prefetch queue, which can cause cache pollution and performance degradation.

In contrast, our hash table design (CAMEL-HT) organizes elements within small, contiguous arrays, naturally supporting efficient spatial locality. A single prefetch instruction is often sufficient to bring an entire group of relevant elements into cache, as illustrated in Figure 5b. This reduces the number of required prefetches, improves cache utilization, and minimizes unnecessary memory traffic. By aligning data layout with cache line boundaries and avoiding fine-grained prefetching, CAMEL-HT achieves superior memory efficiency and performance compared to techniques like AMAC.

As illustrated in Figure 3, we introduce the NEU Buffer mechanism that temporarily stores probe keys along with their computed bucket indices after they pass through the bitmap filter. Rather than immediately probing the hash table, we first issue a software prefetch for the first element of the array corresponding to the computed bucket index. This prefetch loads an entire cache line into the cache, which includes multiple elements. When comparing the probe key against entries in the hash table, even if the first element in the bucket array does not match, the remaining elements are already in cache, enabling subsequent comparisons to proceed with minimal latency.

Probe keys from table S are buffered in the NEU Buffer until it reaches the maximum capacity. Once full, we begin processing the oldest buffered entry, under the assumption that the prefetched array is now resident in the cache. This strategy improves temporal locality[13] and reduces cache miss penalties by decoupling key arrival from immediate verification. The NEU Buffer aligns memory access with actual computation, resulting in lower latency and improved throughput. Empirical results across diverse data volumes and distributions confirm the robustness of this approach and its ability to adaptively optimize performance in varied workload scenarios.

## 4 Hash join with CAMEL-HT

In this section, we describe the fine-grained operations of the build and probe phases specific to CAMEL-HT. Unlike the general hash join algorithm previously discussed, our design introduces structured multi-step processes to optimize both phases. The build phase (§4.1) consists of *three* key steps that efficiently construct the internal hash table, ensuring the efficiency of memory and cache. Similarly, the probe phase (§4.2) involves *three* distinct steps tailored to quickly locate and verify matching entries with minimal overhead. These steps are designed to exploit hardware characteristics and data access patterns, ultimately improving join performance on large-scale datasets. In all subsequent discussions, we use the following SQL query, involving a join between two tables R and S.

```sql
SELECT SUM(r.val) FROM R r, S s
WHERE r.key = s.key;
```

### 4.1 Build phase

Many hash tables suffer from over-allocation or under-allocation, leading to costly resizing operations. Our build step eliminates this overhead through a three-step process: the first step determines the exact hash table size, the second step allocates the required memory, and the final step builds the table. This approach avoids resizing altogether and ensures predictable performance. To address this, we employ a bitmap-based approach that enables us to compute the exact cardinality of each bucket using population count (§3.2) operations. This allows for precise memory allocation, avoiding the overhead and cache inefficiencies associated with pointer-based structures such as linked lists. Instead, we allocate a contiguous array for each bucket, improving spatial locality and enabling faster access during the probe phase. The build phase of CAMEL-HT proceeds in three distinct steps, which are detailed below.

*4.1.1 Step 1: Bitmap creation and popularity count —* To balance key dispersion and cache efficiency, we choose the bitmap size as the next power of two of the build table size, and scale it by a factor of 2 to enhance sparsity. This power-of-two sizing enables efficient bit masking and avoids costly modulo operations. Larger bitmap sizes (e.g., 8× or more) degrade performance due to increased memory consumption, lower fill factor, and higher allocation overhead from numerous small arrays. Empirical results confirm that a 4× ratio provides an optimal trade-off between distribution quality and memory efficiency.

We represent the bitmap using a compact array of 64-bit integers as depicted in Figure 4, reducing its footprint to $\frac{1}{64}$th of the original size and improving cache locality. Each set bit in the bitmap corresponds to a key insertion and simultaneously increments a population counter that tracks collisions per 64-bit word. This count monitors the number of key-value pairs per bucket, enabling precise pre-allocation of fixed-size arrays without the need for dynamic resizing or using linked structures, resulting in improved space utilization and minimal memory overhead.

As depicted in Figure 6, 4-bit words are employed to store the bit positions. In this figure, we illustrate an example, where the topmost array represents the join column of the build table R. The elements of this array are used to build the hash table. When item 119 is about to be inserted into the hash table, a collision is observed since key 13 is already inserted in the same destination location. To resolve this, however, we do not employ linear probing to identify the next available bit position. Instead, we increment the population count to update the corresponding bucket's occupancy.

Algorithm 1 outlines the bitmap creation and population counting process. Two arrays are maintained: one for the bitmap and another for tracking population counts (Lines 1–2). Each key from the build table (Line 7) is hashed using the MUM (Multiply and Mix) hash [28] function (Line 8), producing a uniformly distributed hash value. This value determines both the bucket index and the bit position within a 64-bit word. The relevant bit is set via bitwise masking (Line 11), and the population count for that bucket is incremented (Line 12), independent of collisions. This design enables accurate bucket sizing while avoiding the overhead of linear probing.

```
struct camel_ht{
    uint64_t key=0;
    uint64_t val=0;
};
```

This phase enables precise memory allocation when the full input is known. However, if the data originates from an upstream operator, materialization may be necessary. To avoid this, we leverage statistics maintained by many database systems to estimate intermediate cardinality. To mitigate estimation errors, we allocate a bitmap based on the next power of two of the estimated cardinality, which inherently provides buffer space for additional elements. For example, an estimated cardinality of 100 leads to an allocation of 4×128 entries in a bitmap, allowing ample room for several additional elements without requiring reallocation.

*4.1.2 Step 2: Memory allocation —* In the second step of the build phase, we allocate a contiguous array for each bucket based on its corresponding population count computed from the bitmap. This ensures that each bucket receives an array sized exactly according to the number of keys mapped to it, avoiding both over-allocation and costly resizing. As illustrated in Step 2 of Figure 6, if the population count for the bucket index 0 is 3, an array of size 3 is allocated accordingly. This adaptive allocation strategy enables efficient memory utilization, removes the requirement for re-hashing, and improves performance.

*4.1.3 Step 3: Build —* During the build phase, each key from the build table is hashed and directly mapped to its corresponding bucket within a pre-allocated array. An auxiliary array index (*arr_idx*) is maintained to track the next available slot for each bucket, enabling constant-time insertion without requiring an additional scan to locate the insertion position. As illustrated in Figure 6 (Step 3), this index ensures that each key-value pair is inserted at the correct location, effectively eliminating the need for linear probing or pointer chasing. This design significantly improves the performance of the build step by reducing computational overhead and enhancing cache efficiency. Furthermore, to minimize memory access latency, we incorporate software prefetching to proactively load relevant data into the cache, thereby further accelerating the build process.

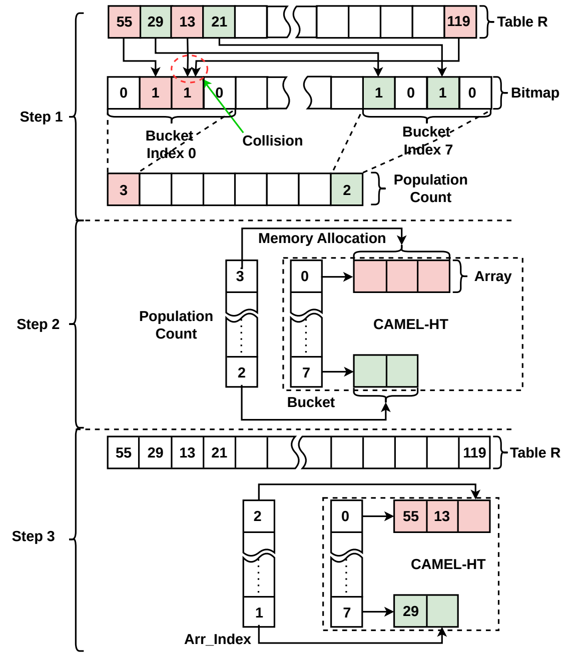


**Figure 6: Data flow during Build phase**

**Algorithm 1:** Bitmap creation and updating pop count

```
   Data: $build\_keys[\,] \geq 0$
   Result: $bits[\,], pop\_count[\,]$
1  $bits[\,] \leftarrow 0$;
2  $pop\_count[\,] \leftarrow 0$;
3  $i \leftarrow 0$;
4  $bit\_pos \leftarrow 0$;
5  $bkt\_idx \leftarrow 0$;
6  $bit\_in\_int \leftarrow 0$;
7  while $i < build\_keys.size()$ do
8      $bit\_pos = hash(build\_keys[i])$ ;
9      $bkt\_idx = bit\_pos/64$ ;
10     $bit\_in\_int = bit\_pos\%64$ ;
11     $bits[bkt\_idx] \mathrel{|}= (1ULL << bit\_in\_int)$;
12     $++pop\_count[bkt\_idx]$ ;
13     $++i$;
```

**Algorithm 2:** Build the hash table

```
   Data: $build\_keys[\,] \geq 0$
   Result: $camel\_ht[\,]$
1  $bkt\_idx[\,] \leftarrow 0$;
2  $arr\_idx[\,] \leftarrow 0$;
3  $N \leftarrow build\_keys.size()$;
4  for $i \leftarrow 0$ to $N - 1$ do
5      for $j \leftarrow 0$ to $9$ do
6          $bit\_pos \leftarrow hash(build\_keys[i + j])$;
7          $bkt\_idx[j] \leftarrow bit\_pos/64$;
8          $\_\_builtin\_prefetch(\&camel\_ht[bkt\_idx[j]], 0, 3)$;
9          $\_\_builtin\_prefetch(\&arr\_index[bkt\_idx[j]], 0, 3)$;
10     for $j \leftarrow 0$ to $9$ do
11         $camel\_ht[bkt\_idx[j]][arr\_idx[bkt\_idx[j]]].key =$
             $build\_keys[i + j]$;
12         $camel\_ht[bkt\_idx[j]][arr\_idx[bkt\_idx[j]]].val =$
             $build\_value[i + j]$;
13         $++arr\_idx[bkt\_idx[j]]$;
14     $i \mathrel{+}= 10$;
```

As described in Algorithm 2, the insertion process utilizes pre-computed array indices to enable constant-time insertion. Each hash bucket maintains an index (line 2) pointing to the next available slot, eliminating the need for iteration during insertion. To further improve performance, we incorporate software pipeline prefetching. After computing the hash of a build key, the system identifies the corresponding bucket and issues prefetch instructions for both the target array segment (line 8) and its position index (line 9). We prefetch the first 10 entries and insert them in-place (lines 10–13), ensuring minimal memory access latency. This combination of direct indexing and prefetching yields up to a 50% speedup in the build phase.

**Algorithm 3:** Probing using NEU prefetching

```
   Data: $probe\_keys[\,] \geq 0$
   Output: Probe result
1  initialize Probe CAMEL-HT module;
2  $i \leftarrow 0$;
3  $buff\_ptr \leftarrow 0$;
4  $crnt\_ptr \leftarrow 0$;
5  $N \leftarrow probe\_keys.size()$;
   /* Step 1: Invoke prefetch instruction and store into NEU
      buffer until it is full                                  */
6  while $buff\_ptr < BUFFER\_SIZE$ do
7      $bit\_pos \leftarrow hash(probe\_keys[i])$ ;
8      $bkt\_idx \leftarrow bit\_pos/64$ ;
9      $set\_bit \leftarrow bit\_pos\%64$;
10     if $set\_bit == 1$ then
11         $\_\_builtin\_prefetch(\&camel\_ht[bkt\_idx][0], 0, 3)$;
12         $prefetch\_ht\_index[buff\_ptr] \leftarrow bkt\_index$;
13         $prefetch\_pk\_index[buff\_ptr] \leftarrow probe\_keys[i]$;
14         $++buff\_ptr$;
15     $++i$;
   /* Step 2: Match and Prefetch                               */
16 while $i < N$ do
       /* Step 2.1: Pull Probe Key from the NEU Buffer and
          match with CAMEL-HT                                  */
17     $bkt\_idx \leftarrow prefetch\_ht\_index[crnt\_ptr]$;
18     $key \leftarrow prefetch\_pk\_index[crnt\_ptr]$;
19     $j \leftarrow 0$;
20     while $j < pop\_count[bkt\_idx]$ do
21         if $key == camel\_ht[bkt\_idx][j].key$ then
22             $count+ = camel\_ht[bkt\_idx][j].val$;
23             $break$;
24         $++j$;
       /* Step 2.2: Invoke prefetch instruction for the next
          element from Probe Table                             */
25     while $i < N$ do
26         $bit\_pos \leftarrow hash(probe\_keys[i])$ ;
27         $bkt\_idx \leftarrow bit\_pos/64$ ;
28         $set\_bit \leftarrow bit\_pos\%64$;
29         if $set\_bit == 1$ then
30             $\_\_builtin\_prefetch(\&camel\_ht[bkt\_idx][0], 0, 3)$;
31             $prefetch\_ht\_index[crnt\_ptr] \leftarrow bkt\_idx$;
32             $prefetch\_pk\_index[crnt\_ptr] \leftarrow probe\_keys[i]$;
33             $++crnt\_ptr$;
34             $++i$;
35             $break$;
36         $++i$;
       /* Step 3: Iterate over unvisited elements in the NEU
          Buffer                                               */
```

## 4.2 Probe phase

The probe phase in CAMEL-HT, illustrated in Algorithm 3, consists of three meticulously designed steps to maximize performance. In the first step (§4.2.1), the NEU buffer is populated with candidate probe keys that pass the bitmap filter. The second step (§4.2.2) begins by selecting an element from the buffer and verifying its presence in the hash table. Prefetching is then triggered for the next qualifying element (i.e., a hashed probe key that satisfies the bitmap filter), ensuring it is fetched and stored in the NEU buffer ahead of processing. In the final step (§4.2.3), the buffer is iterated over to process any remaining unvisited elements, thereby completing the probe phase with improved cache efficiency and reduced latency.

*4.2.1 Step 1: Fill NEU buffer —* As we iterate over the probe table for each probe key, we apply the same hash function used during the build phase (i.e., MUM), and compute the corresponding bitmap position (lines 7–9). If the bit at this position is set (line 10), a software prefetch instruction is issued (line 11), and both the key and its computed bucket index are stored in the NEU (§3.3) buffer (lines 12–13) until the buffer reaches capacity. As illustrated in Figure 7, this process begins by hashing each probe key from table S to determine its bit position and associated bucket. For example, the first key (value 21) maps to bucket index 7. Since the bit at the computed position is set, this suggests a potential match, prompting a prefetch and temporary storage of the key and bucket index in the NEU buffer. If the bit is unset, the key is definitively filtered out, avoiding any further lookup. This filtering and buffering continue until the NEU buffer is full.

*4.2.2 Step 2: Lookup and Prefetch —* Next, an element is selected from the NEU buffer (lines 17–18) and compared against the entries in the corresponding bucket's array. If a match is found, the NEU buffer position is marked as free. As shown in Figure 7, the iterator (crnt_ptr) initially points to the 0th position of the NEU buffer (line 4), which contains the probe key and its associated bucket index. The array corresponding to this bucket index is then scanned for a match (lines 19–23). In the example, probe key 21 is found at the second position within the array for bucket index 7. If no match is found, the probe key is deemed a false positive, and the NEU buffer slot can be reused. The system then resumes scanning the probe table (line 25) for the next eligible key that passes the bitmap filter (line 29). Upon passing the filter, a prefetch instruction is issued (line 30), and the new probe key and its computed bucket index are inserted into the freed buffer slot (lines 31–32). For instance, item 119 from Table S replaces the 0th buffer position previously occupied by key 21. The iterator (crnt_ptr) is then advanced to the next position to continue the matching process within the hash table.

*4.2.3 Step 3: Lookup remaining —* The last stage of our algorithm is to iterate over the NEU buffer and check with CAMEL-HT for those items that have not been visited before.

## 5 Evaluation

This section provides a comprehensive overview of our experimental setup and contains a detailed performance evaluation of the proposed system. Initially, we present our datasets in §5.1. In §5.2, we outline the experimental setup. Then, §5.4, investigates the quantitative impact of integrating NEU prefetching into the CAMEL-HT. Next, §5.5 presents a comparative performance analysis of CAMEL-HT against other state-of-the-art hash table implementations. §5.6 explains how our system performs with TPC-H data. Finally, §5.7 highlights the compactness of the CAMEL-HT.

### 5.1 Datasets

Most of the existing datasets do not focus on selectivity while evaluating a hash join. However, selectivity plays a major role while evaluating a hash join performance. Another critical factor

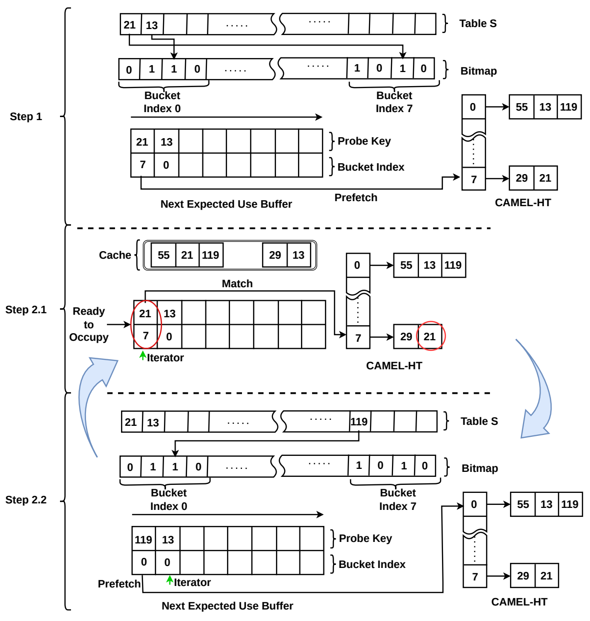


**Figure 7: Data flow during probe through NEU Buffer**

to consider is the order of keys. Prior research by Shanbhag et al. [32] emphasizes the significance of examining dataset shuffling in this context.

Table 1 provides details of how our datasets are generated. The Zipfian [36] distribution has been widely utilized in previous studies [2, 7] to statistically model skewness in popularity. All the data we use for the experiments are shuffled. We provide the workload distribution in Table 2. Build data size varies from 10M to 50M, however, the probe data is dependent upon the data distribution, i.e., Zipf. To evaluate performance under varying data distributions, we use different selectivities in our dataset. For example, a selectivity of 0.2 indicates that only 20% of the tuples in S have a key that matches a key in R. This implies that 80% of the S tuples will not find a match during the probe phase.

In addition to our custom microbenchmark, we use the TPC-H [35] dataset at scale factor 10 to evaluate the performance of our approach in realistic decision support workloads. Specifically, we focus on the *lineitem* and *orders* tables, which contain approximately 60 million and 15 million rows, respectively. These tables are representative of large-scale analytical queries involving joins and aggregations, enabling a comprehensive assessment of execution time, memory usage, and cache behavior.

| Dataset Name | Build Key Generation | Build to Probe Key Correlation | Parameters |
|---|---|---|---|
| Zipf 1:N | unique keys (shuffled) | relationship follows a Zipf distribution | Zipf skew=2.0 |

**Table 1: Dataset details**

## 5.2 Experimental setting

The experiments were carried out on two systems whose specifications are summarized in Table 3. To ensure consistent performance and minimize variability due to thread migration, all experiments were pinned to a specific core using the task set, following the methodology of previous work [18]. The software was compiled using GCC 11.4.0, with the -O3 optimization setting

| Distribution | Zipf | | |
|---|---|---|---|
| Key & Val Size | Key (8B) | | Val (8B) |
| Rows/Table | Scale Factor | Build(R) | Probe(S) |
| | 10 | 10M | 26M |
| | 20 | 20M | 52M |
| | 30 | 30M | 79M |
| | 40 | 40M | 105M |
| | 50 | 50M | 132M |

**Table 2: Workload characteristics**

activated to maximize efficiency across all machines. To ensure consistency in our comparison, we utilize the same hash function, specifically the MUM hash function [28], across all hash tables. To establish a baseline, we executed our initial and optimized `CAMEL-HT` implementations on Machine 1. A comparative performance analysis against other state-of-the-art hash tables was then conducted using our optimized version on Machine 2.

## 5.3 Impact of a Bitmap Filter

We conducted two experiments to analyze the impact of the bitmap filter, both without software prefetching. The first experiment used a bitmap filter, while the second did not. Our results, depicted in Figure 8, show that the bitmap filter significantly improves lookup performance when selectivity is low. However, its effectiveness decreases as selectivity increases. When selectivity reaches 100%, the version without the filter outperforms the one with it. This performance degradation is due to the added computational overhead of checking the bitmap before searching the hash table, which negates the benefits of the filter at high selectivities.

| Specification | Machine 1 (Xeon 5120) | Machine 2 (Xeon 6248) |
|---|---|---|
| CPU Model | Intel Xeon Gold 5120 @ 2.20 GHz | Intel Xeon Gold 6248 @ 2.50 GHz |
| Cores/Threads | 14 cores (2 threads/core), HT enabled | 20 cores (2 threads/core), HT enabled |
| Memory | 128 GB DDR4 | 32 GB DDR4 |
| L1 Cache | 32 KB (L1i) + 32 KB (L1d) per core | 32 KB (L1i) + 32 KB (L1d) per core |
| L2 Cache | 1 MB per core | 1 MB per core |
| L3 Cache | 19.25 MB shared | 27.5 MB shared |

**Table 3: System specifications of experimental platforms.**

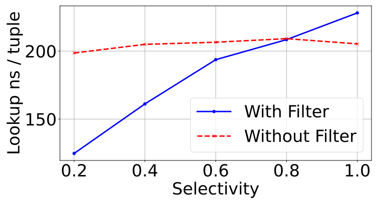


**Figure 8: Performance analysis of CAMEL- lookup with and without a bitmap filter (lower is better)**

## 5.4 Effect of Next Expected Use prefetching

To study the effect of NEU mechanism, we devise a single-threaded micro-benchmark on a baseline implementation of `CAMEL-HT` without NEU and an optimized version incorporating NEU-based software prefetching. We evaluate both versions across two dataset

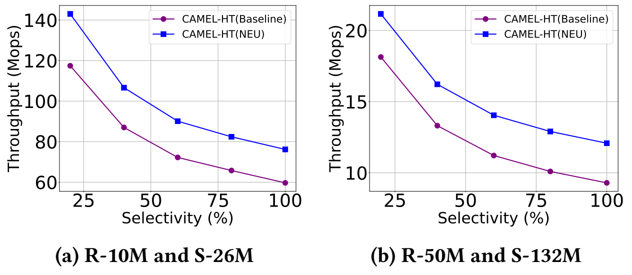


**Figure 9: Lookup throughput (in Mops) comparison between the baseline CAMEL-HT and the NEU-enabled CAMEL-HT across varying selectivity levels (20%–100%) for scale factors 10M and 50M.**

scales—10 million (10M) and 50 million (50M) tuples—and selectivity levels ranging from 20% to 100%, simulating a wide range of workload intensities. The evaluation focuses primarily on different critical performance indicators, such as: lookup throughput and Last-Level Cache (LLC) misses, CPU cycles, and overall execution time.

*5.4.1 Lookup throughput —* We report the lookup throughput of baseline `CAMEL-HT` and NEU-enabled `CAMEL-HT` across two dataset sizes (10M and 50M) and selectivity levels from 20% to 100%. As shown in Figure 9, NEU-enabled `CAMEL-HT` consistently delivers higher throughput than baseline `CAMEL-HT` across all configurations. At the 10M scale, throughput improvements range from 21.8% to 27.7%, while at 50M, gains range from 16.7% to 30.0%. These improvements become more pronounced with increasing selectivity, reflecting the NEU's effectiveness in reducing memory stalls under higher memory pressure.

This upward trend in relative performance confirms that NEU's coordinated software prefetching becomes increasingly beneficial as the workload accesses a larger portion of the dataset. By preloading data closer to the CPU ahead of use, NEU helps sustain pipeline utilization and reduces idle processor cycles. Overall, the results underscore the practicality of NEU in scaling hash-based lookup performance under diverse query intensities.

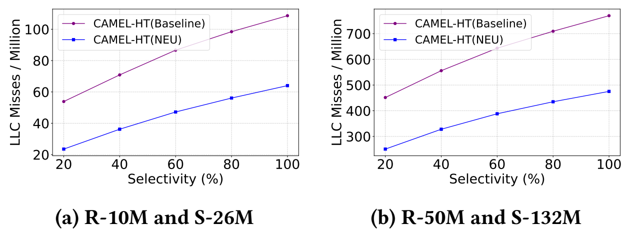


**Figure 10: LLC misses for baseline CAMEL-HT and NEU-enabled CAMEL-HT across varying selectivity levels (20%–100%) for two scale factors (10M and 50M).**

*5.4.2 Cache profile —* To gain deeper insight into the impact of NEU, we analyze Last-Level Cache (LLC) misses at selectivity levels of 20%, 60%, and 100%. As shown in Figure 10, the NEU-enabled `CAMEL-HT` consistently incurs significantly fewer cache misses compared to the baseline model, across both 10M and 50M dataset scales.

At 20% selectivity, LLC misses in NEU-enabled `CAMEL-HT` are reduced by 56.3% at 10M (from 53.9M to 23.5M) and 44.5% at 50M (from 451M to 251M). For 60% selectivity, the reductions are 45.5% (from 86.6M to 47.2M) and 39.7% (from 643M to 388M), respectively. At 100% selectivity, where memory access demand is highest, NEU-enabled `CAMEL-HT` still achieves substantial reductions—41.1% at 10M (from 108.6M to 64.0M) and 38.3% at 50M (from 769M to 475M).

These results confirm that NEU significantly improves cache efficiency, especially as selectivity increases and workloads become more memory-bound. The observed reduction in LLC misses directly contributes to the higher throughput of NEU-enabled `CAMEL-HT`, highlighting NEU's ability to prefetch relevant data more effectively and reduce memory latency under varying query intensities.

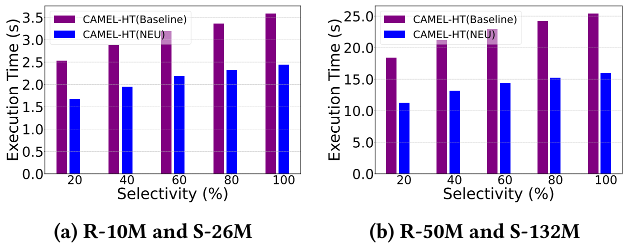


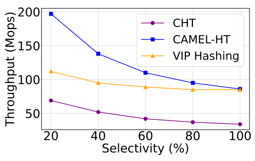
**Figure 11: Execution time comparison of baseline CAMEL-HT and NEU-enabled CAMEL-HT across varying selectivity levels (20% to 100%) for two dataset sizes: (a) 10 million and (b) 50 million records.**

*5.4.3 Overall Execution Time Analysis —* Figure 11 quantifies the performance benefits of our proposed `CAMEL-HT` architecture with NEU integration compared to the baseline `CAMEL-HT`. Across both 10M and 50M record datasets, NEU-enabled `CAMEL-HT` consistently outperforms baseline `CAMEL-HT` in execution time. At 10M records, NEU-enabled `CAMEL-HT` yields runtime reductions of 34.3%, 31.8%, and 31.8% for 20%, 60%, and 100% selectivities, respectively. These improvements become more pronounced at 50M records, reaching 38.7%, 37.2%, and 37.2% for the same selectivity levels. These results demonstrate NEU's effectiveness in optimizing memory access patterns, leading to significant latency reduction and consequently, substantial acceleration of query processing on large-scale data.

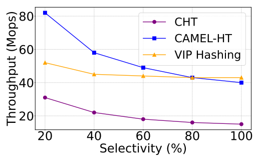
## 5.5 Performance comparison with other hash tables

This section presents a comprehensive evaluation of optimised NEU-enabled `CAMEL-HT` in the context of hash joins, benchmarking it against established hash table implementations, including the CHT and VIP Hashing. For the remainder of the evaluation, we refer to the optimized NEU-enabled version of `CAMEL-HT` as `CAMEL-HT`. We assess performance across several critical dimensions: lookup time (§5.5.1), memory footprint (§5.5.4), last-level cache (LLC) misses (§5.5.2), and execution time (§5.5.5). These metrics collectively highlight the computational cost, cache efficiency, and memory overhead of each approach. The comparison underscores `CAMEL-HT`'s effectiveness in terms of performance, resource utilization, and scalability relative to state-of-the-art hash table designs.

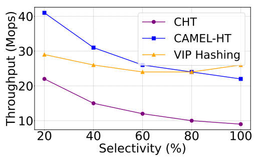
*5.5.1 Lookup throughput —* We evaluated the probe performance of three hash tables, CHT, `CAMEL-HT`, and VIP Hashing across varying selectivity levels (20% to 100%) while holding the



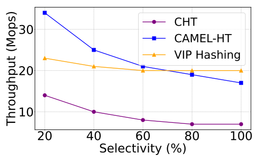
(a) Scale factor 10



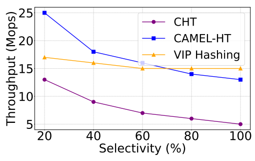
(b) Scale factor 20



(c) Scale factor 30

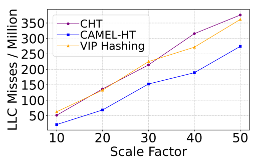


(d) Scale factor 40

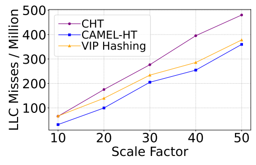


(e) Scale factor 50

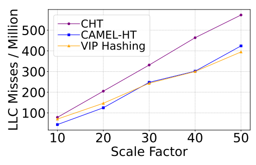
**Figure 12: The lookup throughput (Mops/sec) of CHT, CAMEL-HT, and VIP Hashing was evaluated across a range of scale factors and selectivities. CAMEL-HT exhibits the highest throughput in most configurations due to its cache-friendly design. Although VIP Hashing leverages hot-key optimization to surpass CAMEL-HT at higher selectivities and larger scale factors, CHT consistently shows the lowest throughput.**



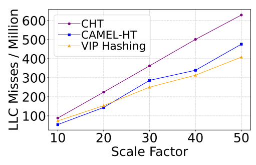
(a) 20% Selectivity



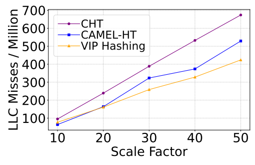
(b) 40% Selectivity



(c) 60% Selectivity

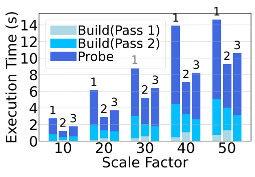


(d) 80% Selectivity

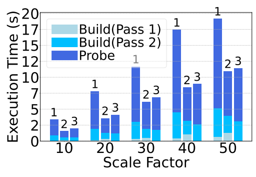


(e) 100% Selectivity

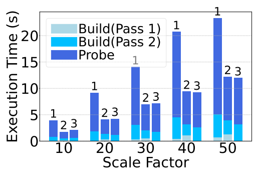
**Figure 13: Last Level Cache misses (LLC) for CHT, CAMEL-HT, and VIP Hashing across scale factors under selectivities of 20% to 100%. CAMEL-HT consistently reduces cache misses over CHT, while VIP Hashing outperforms CAMEL-HT at higher selectivities by leveraging hot-key reordering for better temporal locality.**



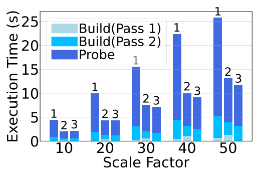
(a) 20% Selectivity



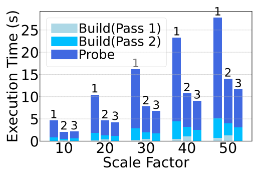
(b) 40% Selectivity



(c) 60% Selectivity



(d) 80% Selectivity



(e) 100% Selectivity

**Figure 14: Execution time (in sec) of CHT(1), CAMEL-HT(2), and VIP Hashing(3) across scale factors under selectivities of 20% to 100%. CAMEL-HT delivers the lowest execution time in most configurations due to its memory-efficient design, while VIP Hashing overtakes CAMEL-HT at high selectivities and large scales through hot-key reordering. CHT remains the slowest across all scenarios.**

build table size constant. We present Figures 12a to 12e corresponding to a fixed scale factor (10M to 50M) and different selectivities, to illustrate the throughput behavior. Across all scale factors, CAMEL-HT consistently achieves the highest throughput in low to moderate selectivity ranges. At the 10M scale ( Figure 12a), for example, at 20% selectivity, CAMEL-HT reaches 200 Mops/sec, outperforming VIP Hashing by 76% and CHT by 186%. As the data scale increases, throughput for all hash tables declines due to increased memory pressure, but CAMEL-HT maintains its advantage. At the 30M scale, CAMEL-HT maintains 41 Mops/sec at 20%, compared to CHT's 22 and VIP Hashing's 29 Mops/sec. These results highlight CAMEL-HT's efficiency in handling skewed data access patterns due to its optimized memory access and reduced cache misses, which are particularly beneficial under Zipf distributions.

VIP Hashing, however, exhibits a notable trend reversal at higher selectivity levels (≥80%), especially under larger scale factors. In these cases, VIP Hashing gradually outperforms CAMEL-HT —for instance, at 50M and 100% selectivity, VIP Hashing achieves 15 Mops/sec, surpassing CAMEL-HT's 13 Mops/sec. This is attributed to VIP Hashing's adaptive design that learns access patterns from previously seen probe keys, and shifts frequently accessed (hot) keys to the front, improving spatial locality under Zipf skew. In contrast, CHT performs consistently worst across all configurations, with steep throughput degradation as both scale and selectivity rise. These findings confirm that while CAMEL-HT is the best choice for general-purpose join processing under skewed data, VIP Hashing becomes increasingly effective in high-selectivity scenarios—especially where data reuse and key frequency skew are pronounced, which is a defining characteristic of Zipfian workloads.

*5.5.2 Cache profile —* Figure 13a through Figure 13e present the LLC miss counts for CHT, CAMEL-HT, and VIP Hashing across increasing selectivities from 20% to 100%. CAMEL-HT consistently demonstrates a strong performance advantage over the CHT across all selectivities and scale factors. For instance, at 20% selectivity, CAMEL-HT reduces LLC misses by more than half compared to CHT, maintaining this significant lead even as the scale factor grows. This highlights CAMEL-HT's efficient memory utilization and its capability to handle a wide range of workloads with lower cache pressure. However, as selectivity increases, the data shows that the gap between CAMEL-HT and CHT narrows, indicating that while CAMEL-HT excels in general efficiency, it does not fully exploit temporal locality inherent in highly selective queries.

Focusing on higher selectivities (60% and above), CAMEL-HT continues to offer a solid baseline but is eventually outperformed

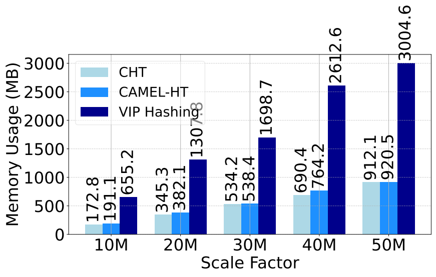


**Figure 15: Memory usage of the three Hash Tables. CAMEL-HT achieves a favorable balance between memory efficiency and performance, maintaining a compact footprint while avoiding the significant overhead observed in VIP Hashing. CHT uses the least memory but lacks the performance benefits of CAMEL-HT.**

by VIP Hashing's hot-key reordering optimization. The figures reveal that at large scale factors and high selectivities, VIP Hashing achieves up to 30–40% fewer LLC misses compared to CAMEL-HT. Nonetheless, CAMEL-HT's low memory footprint and consistent cache efficiency make it a highly practical solution for workloads where access patterns are more uniform or selectivity is moderate. These observations confirm that CAMEL-HT strikes a strong balance between memory efficiency and cache performance, serving as a robust foundation upon which more specialized techniques like VIP Hashing's workload-aware optimizations can build.

*5.5.3 Execution time —* To illustrate the performance scalability of the proposed hash tables under varying data volumes, we present five graphs—each corresponding to a fixed selectivity level (Figure 14)—while increasing the scale factor from 10M to 50M. Across the first three graphs (Figures 14a to 14c), CAMEL-HT delivers the lowest execution time overall, outperforming both CHT and VIP Hashing. The only exception occurs in Figure 14c at scale factors 40 and 50, where VIP Hashing slightly outperforms CAMEL-HT. For example, at 20% selectivity and 50M records, CAMEL-HT outperforms CHT by approximately 55% and VIP Hashing by around 30%, showcasing its robust efficiency across both build and probe phases. A similar trend is observed at 40% selectivity, where CAMEL-HT maintains a 43% performance gain over CHT and 3% over VIP Hashing at the largest scale. Even at 60% selectivity, CAMEL-HT retains a significant lead over CHT and achieves 3% or more improvement over VIP Hashing up to 30M SF. These graphs collectively highlight CAMEL-HT's superior scalability and its ability to maintain low memory access costs and high cache efficiency across increasing data volumes.

However, the last two graphs (Figure 14d and Figure 14e) reveal a shift in trend. As selectivity increases, VIP Hashing begins to close the gap with CAMEL-HT and even outperforms it in several scenarios. At 100% selectivity, CAMEL-HT achieves performance gains at SF 10 but begins to degrade at larger scales, with VIP Hashing outperforming it by about 8.7% at 20M and 20.5% at 50M records. This improvement is attributed to VIP Hashing's adaptive behavior, where it incrementally reorganizes its data layout based on past probe accesses. As probe selectivity increases, VIP Hashing effectively promotes hot keys to the front, enhancing temporal locality and reducing probe latency. Another key factor contributing to the performance degradation of CAMEL-HT is the two-pass iteration over the build array during the build phase. This overhead becomes increasingly significant as the data volume grows, leading to noticeable delays in execution time. While CAMEL-HT remains dominant in low to medium selectivity scenarios due to its architectural optimizations, VIP Hashing demonstrates its advantage in high-selectivity, high-volume settings, suggesting that learning-driven layout adaptation offers measurable benefits when the workload contains repeated access patterns.

*5.5.4 Memory profile —* Figure 15 presents the average memory usage of the three hash tables CHT, CAMEL-HT, and VIP Hashing across increasing data volumes from 10M to 50M. Memory usage is a critical factor for in-memory databases, where excessive memory consumption can limit scalability or reduce performance due to cache inefficiencies. As shown in the figure, CHT maintains the lowest memory footprint, scaling linearly from 173MB at 10M to 912MB at 50M. CAMEL-HT incurs slightly higher memory usage than CHT, typically 10–12% more, but remains well within practical bounds. In contrast, VIP Hashing requires substantially more memory, growing from 655MB to over 3GB, which is more than 3× the memory of CAMEL-HT at the largest scale.

CAMEL-HT, however, effectively balances both memory usage and performance, offering a more efficient solution without the significant memory overhead observed in VIP Hashing. While VIP Hashing offers better probe performance under high-selectivity workloads, *its high memory cost may make it unsuitable for many in-memory settings.* These results emphasize CAMEL-HT's suitability for memory-constrained environments, especially in systems that must serve large datasets entirely from RAM.

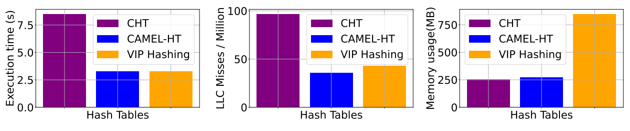
| Selectivity | CAMEL-HT vs CHT | | | | | CAMEL-HT vs VIP Hashing | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 10M | 20M | 30M | 40M | 50M | 10M | 20M | 30M | 40M | 50M |
| 0.2 | 2.22 | 2.11 | 1.69 | 1.96 | 1.57 | 1.42 | 1.27 | 1.22 | 1.15 | 1.13 |
| 0.4 | 2.18 | 2.17 | 1.89 | 2.07 | 1.75 | 1.28 | 1.15 | 1.11 | 1.06 | 1.04 |
| 0.6 | 2.24 | 2.24 | 2.01 | 2.20 | 1.91 | 1.18 | 1.04 | 1.02 | 0.98 | 0.97 |
| 0.8 | 2.23 | 2.29 | 2.05 | 2.21 | 1.95 | 1.09 | 0.98 | 0.94 | 0.90 | 0.89 |
| 1.0 | 2.21 | 2.26 | 2.07 | 2.17 | 1.98 | 1.01 | 0.92 | 0.86 | 0.84 | 0.82 |

**Table 4: Performance speedup of CAMEL-HT over CHT and VIP Hashing across varying selectivities and scale factors. Values $> 1$ indicate CAMEL-HT is faster.**

*5.5.5 Overall Analysis —* As shown in Table 4, CAMEL-HT consistently outperforms CHT across all selectivities and scale factors, achieving over 1.5× improvement in execution time in nearly every configuration. This performance gain demonstrates the significant efficiency of CAMEL-HT's design, especially under low-memory constraints. The highest speedup over CHT (up to 2.29×) is observed at 80% selectivity and 20M scale, indicating CAMEL-HT's robustness as data volume and query selectivity increase. Compared to VIP Hashing, CAMEL-HT generally performs better for lower selectivities (20%–60%), with speedups ranging between 1.02× and 1.42×. However, as selectivity increases (80%–100%), VIP Hashing begins to close the gap or even outperform CAMEL-HT, particularly at larger scales, owing to VIP Hashing's reordering strategy that moves hot keys to the front. This adaptive behavior of VIP Hashing gives it an edge in highly selective queries where access locality becomes critical. Overall, CAMEL-HT strikes an adequate balance between memory efficiency and performance, especially under moderate selectivities and scale.

## 5.6 Results with TPC-H dataset

*5.6.1 1:N Relationship —* We validated the broad applicability of CAMEL-HT by conducting an additional evaluation using the



(a) Execution time (b) LLC misses (c) Memory usage

**Figure 16: Performance comparison of CHT, CAMEL-HT, and VIP Hashing on TPC-H dataset (scale factor 10)**

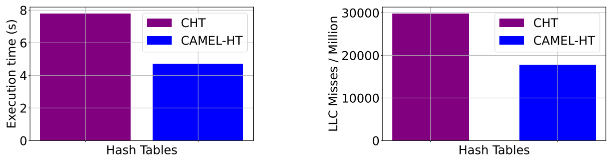
TPC-H dataset at a scale factor of 10, measuring execution time, memory consumption, Last Level Cache (LLC) misses, and CPU cycles. For this experiment, we executed an aggregate query to assess overall performance.

```
SELECT COUNT(*) FROM
ORDERS o, LINEITEM l WHERE
o.orderkey = l.orderkey;
```

As presented in Figure 16a, CAMEL-HT consistently outperforms both the CHT and VIP Hashing, delivering up to 2× speedup over CHT while achieving performance comparable to VIP Hashing.

In terms of memory usage (Figure 16c), CAMEL-HT incurs a bit higher memory overhead than CHT, while VIP Hashing demonstrates the highest memory footprint, consuming approximately 3× more memory than CAMEL-HT. The trends in LLC misses (fig. 16b) align with our observations on other datasets. In particular, CAMEL-HT exhibits the fewest LLC misses, followed by VIP Hashing. Although CAMEL-HT and VIP Hashing show similar CPU cycle counts, CAMEL-HT's overall efficiency is underscored by its superior cache behavior and execution time.

*5.6.2 N:M Relationship —* We further evaluate CAMEL-HT on the TPC-H (scale factor 10) benchmark by analyzing the many-to-many (N:M) relationship between PARTSUPP and LINEITEM, joined via the foreign key PARTKEY. This setting captures a realistic high-cardinality join scenario. VIP Hashing is excluded from this experiment as it lacks support for N:M joins.



(a) Execution time for different hash tables (b) LLC misses for different hash tables

**Figure 17: Performance comparison of CHT, and CAMEL-HT for many-to-many relationship on TPC-H dataset (scale factor 10)**

As depicted in Figure 17a, across evaluated workloads, CAMEL-HT achieves an average 1.7× speedup over the CHT, reduces execution time by 40%, and improves throughput by up to 70%, thereby demonstrating robust efficiency in handling many-to-many joins. We further evaluate cache efficiency by measuring last-level cache (LLC) misses. As depicted in Figure 17b, CAMEL-HT reduces LLC misses by approximately 1.7× compared to baseline, highlighting its effectiveness in improving cache utilization and allowing more memory efficient join processing.

## 5.7 Compactness and array length analysis

We evaluate the compactness of CAMEL-HT by analyzing the array lengths within its buckets, which play a crucial role in cache efficiency. As shown in Table 5, CAMEL-HT consistently achieves a high fill factor of 99.99%, indicating minimal wasted space and effective use of memory. The maximum array length is 38, while the average ranges from 9 to 14. These compact sizes help ensure that arrays fit within the CPU cache, enhancing cache locality and overall performance.

| Build size | Fill factor | Max arr length | Avg arr length |
|---|---|---|---|
| 10M | 99.99% | 30 | 9 |
| 20M | 99.99% | 28 | 9 |
| 30M | 99.99% | 38 | 14 |
| 40M | 99.99% | 30 | 9 |
| 50M | 99.99% | 35 | 11 |

**Table 5: Fill factor**

## 6 Conclusion & future work

In this paper, we present a novel, cache-aware, and lightweight hash table designed specifically to enhance hash join operations in database query processing. Our proposed approach introduces a hybrid data structure that seamlessly integrates *open addressing* with *separate chaining*. Additionally, we extend the AMAC prefetching strategy by introducing the *Next Expected Use* (NEU) software prefetching technique tailored for array-based data structures. We rigorously evaluate the performance of our hash table, CAMEL-HT, across a diverse set of workloads, demonstrating its superior cache utilization and high computational efficiency. We compare CAMEL-HT with state-of-the-art hash tables, including VIP Hashing, and CHT. A breakeven point is observed at 60% selectivity, below which CAMEL-HT surpasses VIP Hashing across all workloads, delivering superior performance in terms of lookup throughput and improved memory efficiency. On the TPC-H benchmark, CAMEL-HT outperforms CHT by 2× in query performance and achieves throughput comparable to VIP Hashing, demonstrating its efficiency in processing analytical workloads. Overall, CAMEL-HT achieves 2× faster execution times than CHT with similar memory usage and offers 3× higher memory efficiency while outperforming VIP Hashing in several scenarios.

Our current design is implemented in a single-threaded manner, but a natural future extension is to support multithreading. Parallelism can be introduced by applying mutex locks on individual indices of the bitmap and population count arrays, while bucket-level locking ensures safe concurrent insertions. This extension opens the door to scalable parallel performance, making the design suitable for modern multicore architectures. While CAMEL-HT exhibits strong performance under a fixed configuration, future work could explore adaptive tuning strategies that adjust internal parameters at runtime based on workload characteristics. Another potential direction for future work is to integrate dynamic hotkey placement strategies to improve lookup performance while preserving a low memory footprint.

## Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. We also thank Dr. Georgiy Krylov and Avinaba Mistry for their valuable feedback. We acknowledge the Digital Research Alliance of Canada for providing access to high-performance computing resources used in our experiments. This research was supported by NSERC Alliance Grant ALLRP 590659 - 23 and an IBM Advanced Studies Fellowship.

## Artifacts

The implementation details and source code related to this research are part of the IBM project and fall under a non-disclosure agreement (NDA). Due to confidentiality constraints and proprietary considerations, we are unable to publicly share the code at this time. However, we have provided algorithms, sufficient methodological details, and experimental results in the paper to ensure reproducibility. If further clarifications are required, we are open to discussing high-level concepts or methodologies within the permissible limits of our agreement.

## References

[1] Anastassia Ailamaki, David DeWitt, Mark Hill, and David Wood. 1999. *DBMSs on modern processors: Where does time go?* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.

[3] 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 *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 362–373.

[4] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2014. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2014), 1754–1766.

[5] Ronald Barber, Guy Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-efficient hash joins. *Proceedings of the VLDB Endowment* 8, 4 (2014), 353–364.

[6] Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2024. Simple, efficient, and robust hash tables for join processing. In *Proceedings of the 20th international workshop on data management on new hardware*. 1–9.

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

[8] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[9] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. 2007. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)* 32, 3 (2007), 17–es.

[10] Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. 2017. A study of main-memory hash joins on many-core processor: A case with intel knights landing architecture. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 657–666.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms* (fourth ed.). MIT Press. https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/

[12] Alain Crolotte and Ahmad Ghazal. 2011. Introducing skew into the TPC-H benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 137–145.

[13] Ulrich Drepper. 2007. What every programmer should know about memory. *Red Hat, Inc* 11, 2007 (2007), 2007.

[14] Erik Frøseth. 2019. Hash join in MySQL 8. *MySQL. Retrieved November* 13 (2019), 2019.

[15] Holmes He. 2021. Understanding the Memcached source code. *Retrieved January* 1 (2021), 2021.

[16] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment* 6, 9 (2013), 709–720.

[17] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture: A Quantitative Approach* (6 ed.). Elsevier.

[18] Aarati Kakaraparthy, Jignesh M. Patel, Brian Kroth, and Kwanghyun Park. 2022. VIP Hashing - Adapting to Skew in Popularity of Data on the Fly. *Proc. VLDB Endow.* 15, 10 (2022), 1978–1990.

[19] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389.

[20] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous memory access chaining. *Proceedings of the VLDB Endowment (PVLDB)* 9, 4 (2015), 252–263.

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

[22] Chi-Keung Luk and Todd C Mowry. 1996. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. 222–233.

[23] Puya Memarzia, Suprio Ray, and Virendra C Bhavsar. 2018. On Improving Data Skew Resilience In Main-memory Hash Joins. In *Proceedings of the 22nd International Database Engineering & Applications Symposium (IDEAS)*. 226–235.

[24] Puya Memarzia, Suprio Ray, and Virendra C Bhavsar. 2019. A Six-dimensional Analysis of In-memory Aggregation.. In *EDBT*. 289–300.

[25] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment* 11, 1 (2017), 1–13.

[26] Todd C Mowry, Monica S Lam, and Anoop Gupta. 1992. Design and evaluation of a compiler algorithm for prefetching. *ACM Sigplan Notices* 27, 9 (1992), 62–73.

[27] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-efficient aggregation: Hashing is sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1123–1136.

[28] MUM Hash [n. d.]. MUM Hash. https://github.com/vnmakarov/mum-hash.

[29] Sho Nakazono, Yutaro Bessho, Hideyuki Kawashima, and Tatsuhiro Nakamori. 2024. Griffin: Fast transactional database index with hash and b+-tree. In *2024 IEEE 20th International Conference on e-Science (e-Science)*. IEEE, 1–10.

[30] Kousik Nath. 2017. A little internal on Redis hash table implementation. *Retrieved June* 23 (2017), 2022.

[31] Anil Shanbhag, Holger Pirk, and Sam Madden. 2016. Locality-adaptive parallel hash joins using hardware transactional memory. In *International Workshop on In-Memory Data Management and Analytics*. Springer, 118–133.

[32] Anil Shanbhag, Holger Pirk, and Sam Madden. 2017. Locality-adaptive parallel hash joins using hardware transactional memory. Springer, 118–133.

[33] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. 1994. *Cache conscious algorithms for relational query processing*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[34] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. 2019. *Database System Concepts* (7 ed.). McGraw-Hill Education.

[35] Transaction Processing Performance Council. 2023. TPC Benchmark H (Decision Support). http://www.tpc.org/tpch/.

[36] Wikipedia. 2025. Zipf's law — Wikipedia, Wikimedia Foundation, Ltd. https://en.wikipedia.org/wiki/Zipf's_law Accessed: 2025-05-20.

[37] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.