SPAR-HT: A PMem-Optimized Hash Table for Efficient Joins

Sudip Chatterjee[1][0009-0008-4523-1941], Suprio Ray[1][0000-0003-0681-9685], Ian Finlay[2][0009-0009-8686-2448], Calisto Zuzarte[2][0009-0006-1440-6775], and Mark Stoodley[2][0009-0005-2108-8181]

[1] University of New Brunswick, Canada
{sudip.chatterjee,sray}@unb.ca
[2] IBM Canada
{finlay,calisto,mstoodle}@ca.ibm.com

Abstract. Join processing remains a fundamental bottleneck in modern analytical database systems, particularly when hash tables must be repeatedly constructed for recurring join workloads. While most systems build hash tables on the fly, many real-world analytical workloads entail executing joins on the same columns, making them strong candidates for persistent hash indexes. The emergence of byte-addressable persistent memory (PMem) enables database systems to store hash indexes directly in non-volatile memory and reuse them across queries, eliminating repeated build costs while reducing recovery overhead.

In this paper, we investigate persistent memory–resident hash indexes for accelerating join processing in OLAP workloads operating on largely static datasets. We propose `SPAR-HT`: a ***S****calable* ***P****Mem* ***A****ware,* ***R****esource-optimized* ***H****ash* ***T****able*, a persistent memory–optimized hash table designed for high-concurrency analytical environments. `SPAR-HT` leverages the read-mostly nature of analytical workloads and adopts a *rebuild-on-significant-change* strategy that simplifies maintenance while achieving near O(1) lookup performance. Through a comprehensive experimental evaluation against state-of-the-art PMem-aware hashing techniques, including CCEH, Level Hashing, and Dash, we demonstrate that `SPAR-HT` achieves up to 2×, 5×, and 13× higher performance compared to Dash, CCEH, and Level Hashing, respectively, under low-selectivity workloads. These results highlight the effectiveness of `SPAR-HT` and demonstrate the potential of PMem-native indexing structures for accelerating join-intensive analytical workloads.

Keywords: Hash table · Persistence memory · Index · Hash Join.

1 Introduction

Hash tables play a central role in Online Analytical Processing (OLAP) systems due to their near constant-time lookup performance, making them particularly effective for join processing and key-based access. In typical hash-join execution, hash tables are constructed on the fly. However, many workloads repeatedly

join on the same columns. Real-world analytical workloads exhibit significant redundancy; for instance, 80% of queries in half of all Amazon Redshift clusters are exact repeats [14]. Therefore, prebuilding and persisting hash tables for these frequently used attributes can substantially reduce join latency. Emerging byte-addressable persistent memory (PMem) technology offers a promising alternative by providing near-DRAM access latency while supporting significantly larger storage capacities. This capability enables database systems to maintain larger indexing structures directly in PMem. Leveraging PMem in this way offers a promising opportunity to improve overall hash-join performance.

Several hash tables have been specifically designed to exploit the characteristics of persistent memory. However, these designs have largely been evaluated in the context of indexing transactional workloads, focusing on insert, lookup, and update performance. Their effectiveness for accelerating join processing on persistent memory has not been systematically studied. In particular, it remains unclear how PMem-optimized hash tables behave under the access patterns and concurrency characteristics typical of hash-join execution. In this work, we address this gap by conducting a comprehensive evaluation of leading persistent memory hash tables under join workloads.

Persistent-memory hash indexing approaches, such as CCEH [11], Level Hashing [19], and Dash [10], use bucket- or segment-based layouts to enable fast point lookups. CCEH relies on a directory-driven extendible hashing scheme, while Level Hashing and Dash scale via two-level buckets or PMem-optimized segments, with Dash additionally supporting NUMA-aware segmented concurrency for highly parallel joins. Beyond their structural differences, modern PMem-aware hash tables can accelerate hash join execution by persisting hash tables across queries, reducing the cost of repeated construction and probe lookups. However, these PMem-aware hash indexes suffer from several limitations.

Level Hashing's performance degrades as the load factor increases because its log-free update mechanism depends on locating vacant slots; as the table becomes full, it falls back to expensive logging, and although it supports in-place scaling, it still incurs significant NVM write overhead. Similarly, CCEH introduces structural complexity through its three-level organization—directory, segments, and buckets, where large segment sizes can lead to high latency during splits due to extensive cacheline flushes. Like other extendible hashing variants, CCEH also requires an additional directory access for each hash table lookup. Although it places buckets within a single cacheline to reduce memory accesses, it still relies on linear probing to resolve collisions. Moreover, these hash tables tend to consume considerable memory while trying to optimize performance. CCEH often exhibits the most unused space because its segment-based splitting may trigger a rehash when only a single bucket overflows, leaving many other buckets within the segment under-utilized. Level Hashing maintains a relatively balanced occupancy but still requires some reserved empty space to operate efficiently. In contrast, Dash minimizes unused space by leveraging stash buckets and fingerprinting, enabling load factors exceeding 90% while maintaining high performance on real persistent memory hardware.

To address the limitations of existing PMem-aware hash indexes we introduce `SPAR-HT`, a PMem-aware hybrid hash table designed to efficiently exploit both persistent memory and modern CPU architectures. It is built on top of CAMEL-HT [3]. Our design places lightweight metadata in DRAM to enable fast access, while storing key–value (or address of the row) pairs in PMem to ensure persistence and reuse. To efficiently support incremental updates, we incorporate a small in-memory hash table that buffers updates before integration with the persistent structure. Furthermore, `SPAR-HT` leverages software prefetching and SIMD instructions to better utilize CPU parallelism and cache locality. Through this design, `SPAR-HT` achieves up to 13.4×, 4.9×, and 2× higher probing performance than Level Hashing, CCEH, and Dash, respectively, when probing 132M records against a 50M build table using 16 threads.

In summary, this paper makes the following contributions:

- **PMem-optimized `SPAR-HT` (section 3):** We present a persistent-memory-optimized hash table, `SPAR-HT`, which follows a "*rebuild-on-significant-change*" strategy, tailored for read-mostly analytical workloads. Furthermore, we also support bulk insertion of new records after initial load.
- **Rehash-free hybrid DRAM–PMem design (section 4):** We propose a hybrid design that leverages the complementary strengths of DRAM and persistent memory (PMem). Our approach eliminates costly rehashing through an elegant design and further exploits modern hardware features, such as software prefetching and SIMD, to improve performance.
- **Comprehensive performance comparison (section 6):** We compare `SPAR-HT` with existing PMem hash tables, including CCEH, Level Hashing, and Dash for join operation on building time, probing time, cache utilization, and memory efficiency in multi-threaded environments on PMem (Intel Optane).

The remainder of this paper is organized as follows. Section 2 provides background. Sections 3 and 4 present the design principles of `SPAR-HT` and describe how it combines the benefits of DRAM and PMem within a unified architecture. Sections 5 and 6 present the experimental evaluation. Section 7 discusses the results, and Section 8 concludes the paper.

2 Background

We start by giving a summary of PMem and its features that are important for data management. Next, we give an overview of hash join and highlight cutting-edge persistent memory optimized hash tables and indexing structures. We then introduce `SPAR-HT`.

2.1 Persistent memory

Intel Optane DC Persistent Memory represents the first commercially available PMem technology that combines high capacity with byte-addressability and

near-DRAM latency. Its unique characteristics have led to increasing adoption in large-scale data management and query processing systems in recent years.

PMems bandwidth is similar to DRAM and high capacity (256/512 GB per DIMM) at a price lower than DRAM. The core of Intel's PMem modules lies the three-dimensional (3D) XPoint memory medium, which is inherently non-volatile and retains data even in the absence of power. Communication with Optane DIMMs is orchestrated through the processor's integrated memory controller (iMC), which employs the DDR-T interface, a 64-byte asynchronous extension of DDR designed for persistent memory [1,17]. The iMC manages separate Read Pending Queues (RPQs) and Write Pending Queues (WPQs) for each Optane DCPMM (Data Center Persistent Memory Module). Crucially, only the WPQs reside within the Asynchronous DRAM Refresh (ADR) domain [7]. The internal module controller on the Optane DIMM manages read and write operations to the persistent media at a 256-byte granularity. Since PMem persists data only after a cache-line flush (CLFLUSH, CLFLUSHOPT, or CLWB) [8] and provides just 8-byte failure atomicity, software must use memory fences and these flush/writeback instructions to prevent partial updates after a crash.

PMem operate on two modes: *Memory mode* and *App direct mode*. In Memory Mode, DRAM serves as a transparent cache for frequently accessed data, while DCPMMs supply a large but volatile memory capacity that is lost on power failure. In App Direct Mode, by contrast, the operating system and applications are explicitly aware of the separate DRAM and PMem regions, enabling software to issue direct load/store operations to DCPMMs and exploit their persistence [7]. Data persistence to PM is achieved when a cache line flush instruction, such as clflush, clflushopt, or clwb, is executed.

PMem has asymmetric read and write latency, with writes being slower. It has a read latency of approximately 300 ns, which is approximately four times that of DRAM. In comparison to DRAM, it has around 3×/8× slower sequential/random read bandwidth. The numbers for sequential/random writing are around 11×/14× [10].

2.2 Hash join

In a typical hash join involving tables R and S, |R| and |S| denote the cardinalities of the build and probe relations, respectively, with $|R| < |S|$. During the build phase, keys from |R| are hashed to locate target slots, where key–value pairs are inserted. In the probe phase, keys from |S| are hashed to identify candidate buckets, and matches are found via targeted search in the linked lists or arrays. Hash join methods are characterized as *hardware-oblivious* [2] or *hardware-conscious* [13], depending on their interaction with system architecture. Hardware-oblivious techniques promote portability and simplicity by using simple designs such as the build-probe hash join [2] (no partition join) that do not explicitly utilize hardware characteristics. In contrast, hardware-conscious techniques optimize architectural features including cache hierarchies, SIMD [6] support, and NUMA [6] effects. Radix partitioning is a good example for improving cache efficiency and memory locality.

2.3 Hash Tables and Indexes

A number of state-of-the-art hash tables and index structures have been specifically designed for persistent memory. In this work, we focus on Level Hash [19], CCEH [11], and Dash [10] as representative PMem-optimized hash tables. Based on their resizing strategies, persistent-memory hash tables can be broadly categorized into two groups: *Level Hashing*, which organizes data across multiple levels to enable cost-efficient incremental resizing, and *Extensible Hashing*, which supports dynamic growth and shrinkage without requiring full rehashing of the dataset [15]. These index structures are fundamentally hardware-aware, as their designs explicitly exploit persistent memory characteristics through write-efficient updates and cache-line–granularity optimizations.

Level Hashing: Level hashing [19] organizes data across two levels: a top level and a bottom level. Items are first inserted into the top level using cuckoo hashing [12], and overflow entries are placed in the bottom level. When the bottom level becomes full, it is resized to twice the size of the current top level, and only bottom entries are rehashed, reducing write amplification on persistent memory. This design, along with carefully ordered, failure-atomic updates, ensures correctness under crashes. After resizing, the expanded region becomes the new top level, while the former top level is demoted to the bottom, allowing the structure to grow incrementally without full-table rehashing.

CCEH: Cacheline-Conscious Extendible Hashing (CCEH) [11] is a PMem-optimized dynamic hash table that extends the classical extendible hashing [5] architecture. Instead of the traditional two-level design, CCEH introduces a three-tier structure consisting of a global directory, segments, and fixed-size buckets. The global directory, indexed by the least significant bits of the hash value, points to segments, which serve as the fundamental units of allocation and splitting. Each segment contains multiple buckets, and these buckets are sized to fit exactly within a single 64-byte CPU cache line. This cacheline-aware layout minimizes cache-line fetches during lookups, thereby improving performance.

Dash: Dash [10] incorporates principles from Extendible Hashing and Linear Hashing [9] while optimizing their behavior for PMem characteristics. Dash (all our experiments use the extendible-hashing variant of Dash; thus, Dash refers to Dash-EH throughout this paper) builds on several design elements from CCEH but adapts the bucket size to match the 256-byte XPLine of Intel Optane DCPMM, improving spatial locality. Each directory entry references a segment composed of 64 regular buckets and two stash buckets for overflow handling, all sharing the same internal layout. Every bucket is divided into a 224-byte record region that holds pointers to variable-length key-value items. The remaining 32-byte metadata region is used to optimize probing and sustain high load factors.

3 `SPAR-HT` Architecture: Design and Implementation

Many modern analytical systems operate under read-mostly workloads, where datasets remain largely static for long periods. In such scenarios, traditional

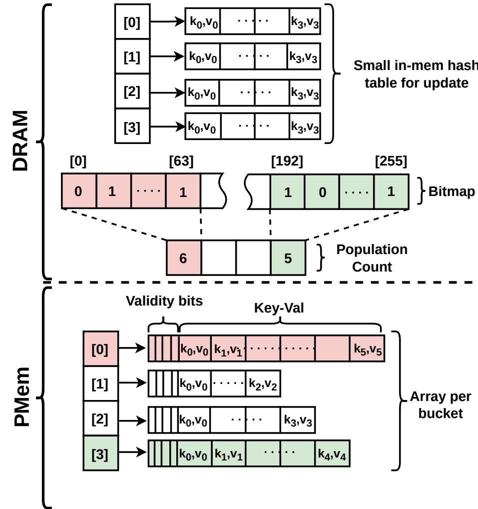

Fig. 1: Detailed Architecture of the `SPAR-HT` Framework

indexing structures designed for frequent updates often introduce unnecessary overhead. To address the limitations of traditional and dynamic hashing under certain workloads, we propose `SPAR-HT`, inspired by our previously proposed CAMEL-HT [3]. CAMEL-HT was originally designed as an in-memory hash table for hash join operations without update functionality and is not PMem-aware. In this work, we redesigned the system to leverage persistent memory (PMem) and added support for updates. Our architecture adopts a hybrid layout, storing metadata in DRAM and placing the main hash table in PMem. The proposed design mitigates several drawbacks commonly observed in dynamic PMem hash tables, including excessive pointer chasing during insert and probe operations, frequent memory allocations, and costly rehashing. Notably, it is inherently rehash-free, which simplifies the design and improves predictability for static workloads.

As shown in Figure 1, our design follows a hybrid architecture in which a compact bitmap is maintained in DRAM, while the actual hash table storing the key–value pairs resides in PMem. Since the bitmap is small, keeping it in DRAM enables fast existence checks and reduces expensive PMem accesses during probing.

To construct the hash table, the dimension table is scanned twice during the build phase. The first pass constructs the bitmap and determines the exact number of elements per bucket (*population count*), enabling precise memory allocation. In the second pass, the tuples are inserted into the hash table and persisted to PMem. Since Intel Optane persistent memory operates at a 256-byte access granularity, we adopt a persistence strategy similar to prior designs such as Dash. Specifically, data is persisted in 256-byte aligned segments, which reduces write amplification and improves write efficiency on PMem. Despite performing two iterations over the build data, the elimination of costly rehashing and PMem rewrite operations results in superior performance during the build phase. Additionally, we set validity bits for each item during insertion into PMem to ensure durability and facilitate recovery when needed.

During the probe phase, we further optimize performance by incorporating software prefetching and SIMD-based probing. These hardware-conscious optimizations improve cache utilization and parallelism during lookup operations. Together, these design choices significantly improve both build and probe performance compared to state-of-the-art persistent memory hash tables.

To support updates, we maintain a simpler in-memory data structure that is approximately one-quarter the size of the PMem-resident hash table. Newly arriving records are first inserted into this DRAM-resident structure, which acts as a delta store for small batches of updates. During query processing, probes are performed on both the in-memory delta store and the PMem-resident table to ensure correctness. When system resources permit, the accumulated updates in DRAM are persisted to PMem. This design enables efficient handling of small incremental updates without degrading query performance.

During persistence, the bitmap and the corresponding population count are updated to reflect the newly inserted entries. If a bucket receives additional records, a new array is allocated to accommodate the updated contents. The existing entries are copied to the new array, the new records are inserted, and the bucket pointer is updated to reference the new array. This process preserves the compact layout of the bucket while ensuring consistency and efficient access.

4 `SPAR-HT` Operational Workflow: Building the Index and Executing Probes

This section outlines the detailed build and probe steps for `SPAR-HT`. Our design optimizes both stages through structured multi-step processes, unlike the general hash join technique discussed previously. Three essential processes make up the build phase, which effectively builds the internal hash table and guarantees memory and cache efficiency. In a similar vein, the probe phase consists of two separate processes designed to find and confirm matching entries fast and with the least amount of overhead. In order to improve join performance on large-scale datasets, these methods are intended to take use of hardware features and data access patterns. The following SQL query, which involves a join between two tables R and S, is used in all ensuing discussions.

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

4.1 Build Phase

Many hash tables suffer from high resizing costs and over-allocation of memory in order to maintain constant-time lookups. To improve persistent-memory efficiency, structures such as CCEH and Level Hashing address this challenge by trading off memory utilization. In contrast, our approach completely avoids rehashing by scanning the dimension data twice during the build phase. The first pass determines the exact storage requirements, enabling precise memory allocation without overflow and allowing us to use a contiguous array instead of

Algorithm 1: Build the hash table

```
input  : (K_1, V_1), ... (K_N, V_N)
output: Hash table
1  bits[] ← 0;
2  prefixCount[] ← 0;
3  atomic_index ← 0;
4  flush_bytes ← 256;
5  flush_elems ← flush_bytes/(sizeof(key) + sizeof(value));
   /* Step 1: Calculate the prefix population count                    */
6  for each key K[i] in thread chunk do
7      bit_pos = hash(K[i]);
8      bucket = bit_pos/64;
9      Automic(updatebits[bucket]);
10     ++LocalCount_t[bucket];
11 After all threads finish, merge counts to compute prefixCount[bucket];
   /* Step 2: Precise memory allocation                                */
12 Partition buckets among threads;
13 for each bucket b in thread range do
14     Allocate(VariableArray) ← size(prefixCount[b]);
   /* Step 3: Insert the data into SPAR-HT and persist                 */
15 Partition keys among threads;
16 for each key K[i] in thread chunk do
17     bucket = hash(K[i])/64;
18     pos = atomic_index[bucket].fetch_add(1);
19     keyArr[pos] = K[i];
20     valArr[pos] = V[i];
21     Update validity bits;
22     if pos completes a 256B chunk then
23         pmemobj_persist(keyArr + chunk_start, flush_bytes);
24 After thread finishes, flush any remaining partial chunk;
```

linked structures. The second pass inserts the data and persists it into persistent memory. Accordingly, the build phase of `SPAR-HT` consists of three distinct steps as shown in Figure 2, described below.

Step 1: Bitmap creation and popularity count — To minimize synchronization overhead, the input keys are partitioned across multiple threads. Each thread maintains a thread-local counter array (algorithm 1, lines 6–10), eliminating contention on shared counters during the counting phase. The hash function calculates a bitmap position for each key; the thread updates its local bucket counter while the matching bit is atomically set in the shared bitmap. The final bucket cardinalities (*population_ count* or *prefix_ count*) are obtained by aggregating the local counts once all threads have finished.

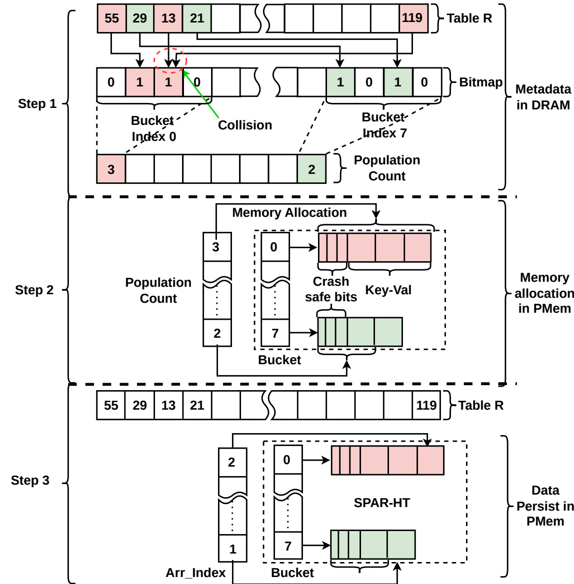

Fig. 2: Build phases of the SPAR-HT Framework

Step 2: Memory allocation — As shown in algorithm 1 (lines 12–14), storage allocation is parallelized by partitioning buckets among threads. Each thread allocates storage for its assigned range independently. The previously calculated prefix_count is used to establish the allocation size for each bucket. The variable-sized bucket arrays are allocated concurrently in this step to eliminate thread contention.

Step 3: Insert & persist — In the second iteration of the dimension table, the input keys are divided among the threads, and each thread simultaneously puts its designated keys into the appropriate buckets. The hash function selects the target bucket for each key, and an atomic counter designates a distinct location in the bucket's array. The corresponding validity bit is set to indicate that the entry is active, and the key–value pair is written to the designated contiguous array, as shown in algorithm 1 (lines 15–21). To optimize persistence and reduce write amplification, the implementation flushes each 256B chunk of a bucket incrementally into persistent memory as it is filled, aligning with the underlying Intel Optane XPLine granularity [16], described in algorithm 1 (lines 22-23). Any remaining partial chunks are flushed at the end, ensuring all inserted data is fully durable without requiring a single large flush after the entire insertion phase.

4.2 Probe phase

The probe phase comprises two steps. First, the in-memory hash table is checked; if a match is found, the search returns true. Otherwise, a bitmap-based prefiltering step is applied to prune probe keys before accessing the PMem hash table. Specifically, if the corresponding bit in the bitmap for a hashed probe key is not set, the key can be immediately discarded without probing the hash table. This early filtering step significantly reduces unnecessary memory accesses.

In the second step, when the bitmap indicates a potential match (i.e., the corresponding bit is set), the algorithm proceeds to search the corresponding bucket

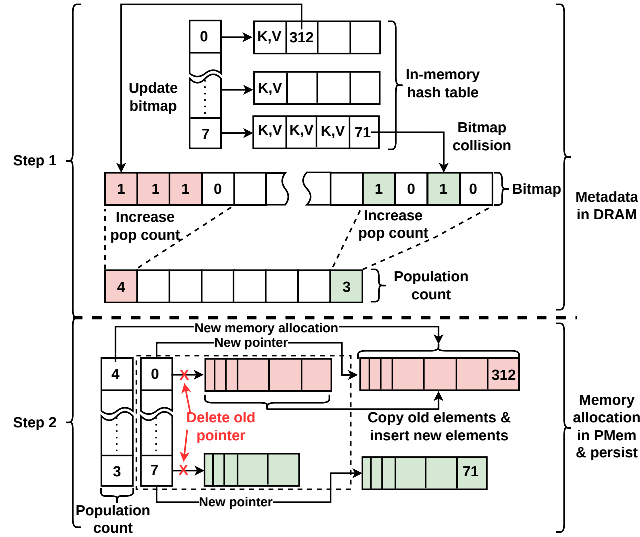

Fig. 3: Second phase of incremental insert of the `SPAR-HT`

in the hash table. This lookup is accelerated using SIMD-based comparisons, enabling multiple keys within the bucket to be checked in parallel. Furthermore, to reduce memory access latency, software prefetching is employed to load the next set of probe elements while the current SIMD operations execute.

We describe these steps in detail below.

Step 1: Bitmap search. — The approach first searches for the probe key in a small in-memory hash table. If the key is found, the lookup returns immediately. Otherwise, the method computes the hash of the key to identify the correspond-

Algorithm 2: Bitmap filtered lookup operation

input : $Key(k)$
output: True if key exists, otherwise False

1 **if** *find_dramHT(Key) == true* **then**
2 return true;
3 $pos \leftarrow hash(k)$;
4 $bucket_index(b) \leftarrow pos/64$;
5 $bit_position(i) \leftarrow pos\%64$;
6 **if** *bits[b][i] != 1* **then**
7 return false;
8 **else**
9 return ScanBucket(b, k);

Algorithm 3: Bucket search operation: **ScanBucket(b, k)**

```
input  : Bucket(b), Key(k)
output: True if key exists, otherwise False
1  Let A be the key array of size n stored in bucket b;
2  i ← 0;
3  w ← 4;
   // number of keys processed per SIMD vector
   /* Vectorized Scan (SIMD):                                  */
4  while (i + w) < n do
5      __builtin_prefetch(&keys[i + w], 0, 3)
6      Load w keys starting at position i;
7      Generate a bitmask of candidate matches;
8      while bitmask! = empty do
9          Extract the next candidate position p;
10         if validity bit of slot[p] == 1 then
11             return true;
12         Remove p from the mask;
13     i ← i + w;
   /* Scalar Scan (Remaining Entries):                         */
14 while i < n do
15     if validity bit of slot[i] == 1 && A[i] == k then
16         return true;
17     i ← i + 1;
18 return false;
```

ing bucket and bit position in a bitmap, and proceeds with the probe. As outlined in Algorithm 2, the bitmap is examined to determine whether the target bucket may contain the key quickly. If the corresponding bit is not set (Line 6), the probe terminates early. Otherwise, the search continues in persistent memory `SPAR-HT` (line 9).

Step 2: SIMD-accelerated bucket search. — As shown in Algorithm 3, the key array within the variable-length structure of the bucket is examined to determine whether the target key exists. Each entry includes a valid bit that indicates whether a valid key is present in the slot.

The approach leverages SIMD instructions to compare four keys in parallel when AVX2 is available. Specifically, given 64-bit keys and the use of 256-bit SIMD registers (e.g., __m256i), the method performs simultaneous comparisons to improve probe efficiency. The comparison produces a bitmask indicating possible matches as depicted in Algorithm 3 (lines 6-8). Before reporting success, the algorithm checks the validity of the matching bit for each candidate match to ensure the entry is legitimate. Software prefetching is employed to mitigate memory access latency (Line 5), enabling earlier retrieval of target data and

improving overall probe performance. Any remaining entries that the vectorized scan cannot process are handled using a scalar scan (lines 14-17).

If SIMD support is unavailable, the algorithm falls back to a scalar scan that checks each key sequentially while validating the corresponding bit. The function returns true if a valid matching key is found; otherwise, it returns false.

4.3 Incremental insert phase

Incremental insertion consists of two phases as shown in Figure 3. In the first phase, incoming records are inserted into an in-memory hash table, which occupies approximately one quarter of the space of the PMem-resident structure. When a new chunk of data arrives, the records are immediately inserted into this DRAM-resident structure, enabling significantly faster updates than directly modifying PMem. In the second phase, when no queries are running and system resources are available, the system iterates over the in-memory hash table and merges the updates into the PMem-resident structure. The second phase consists of two steps. In the first step, the corresponding bitmap and population count are updated, as outlined in Algorithm 4 (lines 1–6). In the second step, the updated population count is then compared with the current array capacity of the corresponding bucket cell. If the population exceeds the existing capacity, a larger array is allocated, existing entries are copied to the new array, and newly inserted elements are appended as described in lines 7–23.

This approach preserves the conciseness of the PMem hash table while enabling efficient incremental updates and maintaining high query performance.

5 Experiments

This section evaluates the performance of the `SPAR-HT`. We compare `SPAR-HT` to three modern concurrent PMem hash tables, Dash [10], CCEH [11], and Level hashing [19] from the Dash library [4]. The Dash-Extendible Hashing (Dash-EH) variation from the Dash-enabled hash tables was employed in our assessment. When compared to other Dash versions, Dash-EH provides speedier performance. Moreover, we describe the workload, datasets, experimental and hardware settings.

This section's objective is to respond to the following research questions:
1. How does `SPAR-HT`'s performance in join operations compare to that of other hash tables on PMem (section 6)?
2. In comparison to other hash tables, how does `SPAR-HT` scale as the number of threads increases (section 6)?
3. How does `SPAR-HT` compare with existing hash tables in terms of space efficiency (section 6.4)?
4. What is the cache efficiency of `SPAR-HT` in comparison to other hash tables (section 6.5)?

Workload and datasets: Our workload involves an equi-join between two relations, R and S, where R has unique primary keys and S has corresponding

Algorithm 4: Batch Update from DRAM Hash Table

```
input  : DRAM_HT(dramHT), Thread_count(T)
output: Insert incremental records in the HT
   /* Phase 1: Count incoming entries                                    */
1  foreach DRAM bucket d ∈ dramHT do
2      foreach key k ∈ d do
3          p ← hash(k, bitmap_size) ≫ 6;
4          bit ← bit_pos & 63;
5          bits[b] ← bits[b] | (1 ≪ bit);
6          newCount[p]++;
   /* Phase 2.1: Resize PMem buckets                                     */
7  foreach bucket b where newCount[b] > 0 do
8      prefixCount[b] ← prefixCount[b] + newCount[b];
9      oldSize ← size(b);
10     newSize ← prefixCount[b];
11     A ← allocate(newSize);
12     copy existing entries of b to A;
13     pmemBuckets[b] ← A;
14     persist(A);
15     atomic_index[b] ← newSize;
   /* Phase 2.2: Parallel insertion & persist                            */
16 foreach DRAM bucket d using T threads do
17     foreach (k, v) ∈ d do
18         p ← hash(k, bitmap_size) ≫ 6;
19         pos ← atomic_index[p].fetch_add(1);
20         insert (k, v) into bucket p at pos;
21         set validity bit(pos);
22         if 256B boundary reached then
23             persist inserted chunk;
24 foreach PMem bucket b do
25     persist remaining unflushed entries;
```

foreign keys referencing R. Although our experiments focus on PK-FK joins, SPAR-HT can support M:N joins as well. We run the following query:

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

In our experimental setup, we perform equi-join operations between two relations, denoted as R (build) and S (probe), where each tuple consists of an 8-byte key and an 8-byte value/address of the row. At a scale factor of 10, relation R contains 10 million tuples, while relation S contains 26 million tuples. To examine the impact of larger data volumes on persistent memory behavior, we further increase the scale factor to 50, resulting in 50 million tuples in R and 132 million

tuples in S. This scaling allows us to evaluate the performance of the evaluated join algorithms under higher memory pressure.

Prior experimental studies have largely overlooked the impact of data selectivity in the context of join operations. However, real-world workloads often exhibit varying selectivity levels in the build relation during probing. In this work, we systematically evaluate join performance across two input relations under different selectivity levels, ranging from 20% to 100%. In addition, we consider skewed access patterns by generating probe workloads following a Zipfian distribution. We also account for the effect of key order on join performance by fully shuffling the build and probe data, thereby eliminating any bias arising from presorted or clustered keys.

Experimental setting: The software was compiled using GCC 11.4.0 with the –O3 optimization level enabled. We vary the number of threads from 1 to 16 to evaluate the scalability of PMem-based hash tables. All experiments were conducted on a machine equipped with an Intel Xeon Gold 6248 processor running at 2.50 GHz, featuring 20 physical cores with two hardware threads per core (hyper-threading enabled). The system is configured with 370 GB of DDR4 main memory. Each core has private 32 KB L1 instruction and 32 KB L1 data caches, along with a 1 MB private L2 cache, while all cores share a 55 MB last-level (L3) cache. In addition, the system is equipped with 126 GB of Intel Optane persistent memory. PMem is accessed via fsdax on an ext4 filesystem and configured to use AppDirect mode. In fsdax mode, an Intel Optane namespace allows Linux file systems to bypass the kernel page cache, allowing for direct byte-addressable mapping with *mmap()* [18].

6 PMem benchmarks

In this section, we evaluate the total execution time of the join operation across different hash indexes while varying the number of threads from 1 to 16. Across both data volumes, the build phase dominates the overall execution time. When the build size is 10M, the probe size is 26M, and the selectivity is 20%, the build phase accounts for 95–98% of the total runtime, whereas the probe phase contributes only 2–5%. As selectivity increases to 100%, the probe time increases accordingly. We also observe that a substantial portion of the build time is spent on resizing operations. This trend remains consistent when the data volume is increased by a factor of five.

6.1 Smaller dataset

Build - `SPAR-HT` consistently achieves the lowest cost across all thread counts, as seen in Figure 4(a), suggesting noticeably improved efficiency when compared to the state-of-the-art PMem hash tables. During single-threaded hash table construction, `SPAR-HT` significantly outperforms existing persistent memory indexing structures, achieving a 2.98×, 5.18×, and 9.39× speedup over CCEH, Dash, and Level Hash, respectively. This performance indicates that the underlying architecture is highly optimized for modern hardware constraints, allowing

it to complete the same workload in a fraction of the time required by traditional persistent memory hash tables.

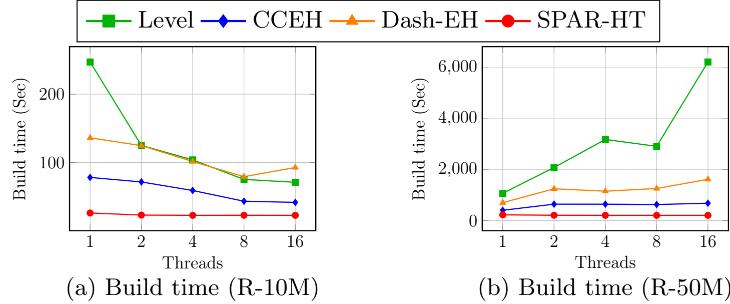

Fig. 4: Overall build time for 10M and 50M records across varying thread counts

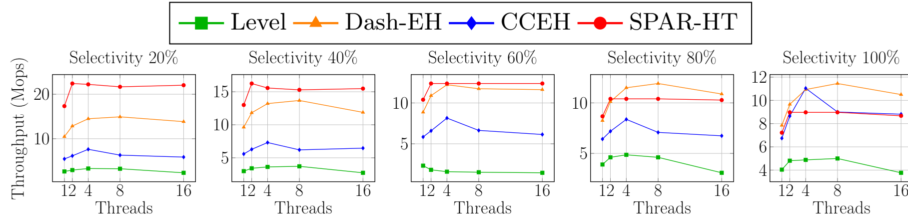

Fig. 5: Probe throughput of Level, CCEH, Dash-EH, and `SPAR-HT` over 26M records under varying thread counts and selectivity levels (higher is better)

Probe - `SPAR-HT` consistently achieves the maximum throughput throughout selectivity settings and thread counts, as illustrated in Figure 5. At 20%–60% selectivity, `SPAR-HT` significantly outperforms the other hash tables, achieving up to 3–4× higher throughput than CCEH and 6–8× higher throughput than Level Hashing as depicted in Table 1a, while maintaining strong scalability with increasing threads. Dash becomes more competitive at high selectivities (80%–100%), where many probes succeed, and can surpass `SPAR-HT` at higher thread counts (4 or more threads). Nonetheless, `SPAR-HT` still maintains superior or comparable throughput in most configurations, consistently demonstrating efficient lookup paths and better utilization of memory and CPU resources. In comparison, CCEH delivers moderate throughput, largely insensitive to selectivity, and Level Hashing experiences declining performance under high concurrency, reinforcing that `SPAR-HT` is the most robust and high-performing hash table overall.

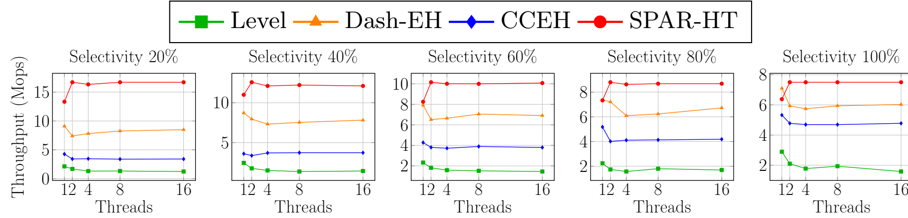

Fig. 6: Probe throughput of Level, CCEH, Dash-EH, and SPAR-HT over 132M records under varying thread counts and selectivity levels (higher is better)

6.2 Larger dataset

Build - The build results reveal (Figure 4(b)) significant differences across the evaluated hash tables as thread count increases. SPAR-HT consistently achieves the best performance, maintaining nearly constant insert time (228–210) across all threads. Compared to SPAR-HT, CCEH is about 2–3× slower, while Dash is roughly 4–8× slower depending on the thread count. Level Hashing performs the worst, becoming up to 30× slower at 16 threads due to severe scalability degradation. This behavior is mainly caused by PMem write-bandwidth saturation and persistence overheads, such as cache-line flushes and memory fences, which increasingly dominate large-scale build workloads for Level and Dash, while SPAR-HT remains largely unaffected due to its PMem efficient design.

Probe - We evaluated the probe throughput of four hash tables, Level, CCEH, Dash, and SPAR-HT, on a 132M-record probe dataset across thread counts (1–16) and selectivities from 20% to 100% as depicted in Figure 6. SPAR-HT consistently achieves the highest throughput, particularly at low and medium selectivities, delivering roughly 2× the throughput of Dash, 5× that of CCEH, and over 13× that of Level at 20% selectivity with 16 threads as shown in Table 1b. Dash provides moderate throughput and approaches SPAR-HT at higher selectivities, while CCEH shows stable but moderate performance, and Level remains the lowest due to cache inefficiencies. Overall, throughput decreases slightly as selectivity increases, and thread scalability is limited across all hash tables.

6.3 Update operation

To evaluate the efficiency of handling dynamic updates, Figure 7 compares the execution latency of SPAR-HT against Level Hashing, CCEH, and Dash for an incremental load of 5 million records (10% of the total dataset). CCEH was the most performant solution, executing the update operation in 43 sec. Level Hashing (66 sec) and Dash (84.42 sec) showed moderate performance loss, with 1.53× and 1.96× slower than the CCEH, respectively. SPAR-HT had the highest latency, requiring 117.48 sec to execute the same volume of records, resulting in a 2.73× increase in execution time versus CCEH. This discrepancy in performance suggests that CCEH gains from more cache-aware design and less element

(a) Probe size 26M

	SPAR-HT vs Level					SPAR-HT vs CCEH					SPAR-HT vs Dash-EH				
Sel.	1	2	4	8	16	1	2	4	8	16	1	2	4	8	16
20%	6.6	7.6	6.7	6.7	9.4	3.2	3.7	2.9	3.4	3.8	1.7	1.7	1.5	1.5	1.6
40%	4.4	4.8	4.3	4.1	5.7	2.3	2.6	2.1	2.5	2.4	1.4	1.4	1.2	1.1	1.3
60%	3.3	3.0	3.2	3.1	4.6	1.8	1.9	1.5	1.9	2.0	1.2	1.1	1.0	1.1	1.1
80%	2.2	2.3	2.1	2.3	3.3	1.4	1.5	1.2	1.5	1.5	1.1	1.0	0.9	0.9	0.9
100%	1.8	1.9	1.8	1.8	2.3	1.1	1.0	0.8	1.0	1.0	0.9	0.9	0.8	0.8	0.8

(b) Probe size 132M

	SPAR-HT vs Level					SPAR-HT vs CCEH					SPAR-HT vs Dash-EH				
Sel.	1	2	4	8	16	1	2	4	8	16	1	2	4	8	16
20%	6.3	9.8	12	13	13.4	3.1	4.9	4.7	4.9	4.9	1.5	2.3	2.1	2.0	2.0
40%	4.5	7.1	8.0	8.9	8.4	3.0	3.7	3.3	3.3	3.2	1.3	1.6	1.7	1.6	1.5
60%	3.5	5.6	6.3	6.6	6.9	1.9	2.7	2.7	2.6	2.7	1.0	1.6	1.5	1.4	1.5
80%	3.3	5.1	5.5	4.8	5.1	1.4	2.2	2.1	2.1	2.1	1.0	1.2	1.4	1.4	1.3
100%	2.2	3.6	4.2	3.9	4.7	1.2	1.6	1.6	1.6	1.6	0.9	1.3	1.3	1.3	1.2

Table 1: Relative speedup of SPAR-HT across varying selectivity and thread counts. Values > 1 indicate SPAR-HT is faster.

shifting. In contrast, in order to maintain conciseness, SPAR-HT must copy the old array to the new array and add the new members; nevertheless, this is an expensive PMem operation. Since the overhead of incremental updates is higher, the target application of SPAR-HT is more suitable for read-mostly workloads, which is typical in analytical applications.

6.4 Memory usage

Memory efficiency varies dramatically among the four models tested, with requirements ranging from 1.18 to 2.12 GB. CCEH exhibits the largest memory footprint (2.12 GB), primarily due to its multi-level directory structure and segment-based allocation. Although this design is cache-conscious, it can incur higher memory overhead under dynamic workloads. Dash and Level, on the other hand, use less data, with 1.24 GB and 1.95 GB, respectively. In particular, SPAR-HT delivers the highest optimized performance, using only 1.18 GB — a 44% reduction over CCEH. This implies that SPAR-HT uses a more compact data structure or has a higher load factor, allowing it to maintain competitive indexing capabilities while staying the most resource-efficient alternative for memory-constrained systems.

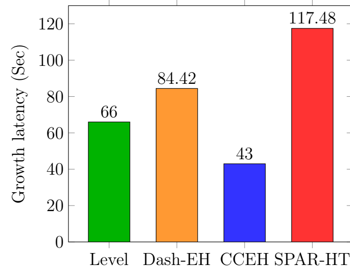

Fig. 7: Execution time for update of 5M records in the existing `SPAR-HT`

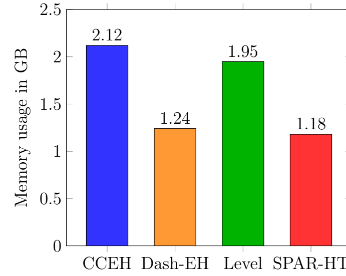

Fig. 8: Memory usage of four hash tables in PMem

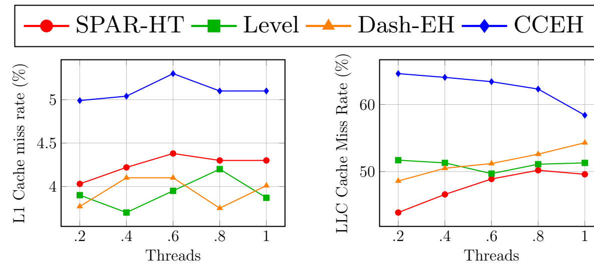

Fig. 9: Cache Performance of Hash Tables During Join Execution with 50M Build and 132M Probe Records (Lower is better)

6.5 Cache performance

Figure 9 reports the L1 and LLC cache miss rates across different workload settings. Overall, `SPAR-HT` consistently achieves the lowest LLC miss rate, ranging between 43.9% and 50.2%, indicating better last-level cache locality compared to the other approaches. While Level Hashing and Dash exhibit slightly lower L1 miss rates (around 3.7–4.1%), their LLC miss rates remain noticeably higher, reaching up to 54.3%. In contrast, CCEH suffers from the poorest cache behavior, with L1 miss rates exceeding 5% and LLC misses as high as 64.6%, suggesting inefficient memory access patterns and reduced spatial locality. Across all workload configurations, `SPAR-HT` maintains stable cache performance, demonstrating improved memory locality and more efficient cache utilization compared to the baselines.

7 Discussion

In our experiments, for the smaller dataset, CCEH, Level Hash, and Dash initially benefit from increased parallelism, with throughput rising as thread count grows, but performance starts to drop at high thread counts due to contention on shared metadata, synchronization overheads, and memory system pressure.

However, SPAR-HT shows linear scaling and does not degrade its performance. For the larger dataset, however, these hash tables fail to scale effectively except SPAR-HT, as the combined impact of random memory accesses, cache-line invalidations, and persistent memory durability enforcement (flushes and fences) overwhelms the benefits of parallelism, leading to limited throughput gains even at moderate concurrency levels. Even though SPAR-HT consistently maintains high throughput for build and probe operations, its performance is worse than others in incremental updates. Since our target workloads are read-mostly analytical workloads, we mainly highlight SPAR–HT's superior build and probe performance, emphasizing its robustness and optimization over existing hash tables. Furthermore, of all the PMem hash tables covered here, SPAR-HT has the most memory-efficient architecture.

8 Conclusion

We have introduced a cache-aware, and lightweight hybrid DRAM-PMem optimized hash table that is specifically designed to be reused while performing hash join operations in database query processing. We compare SPAR-HT against state-of-the-art PMem-aware hash tables, and the experiment results demonstrate that it consistently outperforms the build operation across all load factors and threads. While Dash performs slightly better than SPAR-HT under low data volume and high selectivity, SPAR-HT outperforms it in all other scenarios. Moreover, SPAR-HT consistently outperforms both Level Hashing and CCEH across all workloads and thread counts. Although SPAR-HT incurs higher growth latency, it is amortized in read-mostly analytical workloads with repeated join operations.

References

1. Akel, A., Caulfield, A.M., Mollov, T.I., Gupta, R.K., Swanson, S.: Onyx: A prototype phase change memory storage array. In: 3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11) (2011)
2. Blanas, S., Li, Y., Patel, J.M.: 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. pp. 37–48 (2011)
3. Chatterjee, S., Zhang, X., Ray, S., Finlay, I., Zuzarte, C., Stoodley, M.: Camel hash table: striking a balance between cpu and memory efficiency in main-memory hash join. In: EDBT (2026)
4. Dash Library. https://github.com/baotonglu/dash
5. Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R.: Extendible hashing — a fast access method for dynamic files. ACM Transactions on Database Systems (TODS) **4**(3), 315–344 (1979)
6. Hennessy, J.L., Patterson, D.A.: Computer architecture: a quantitative approach. Elsevier (2011)
7. Hu, D., Chen, Z., Wu, J., Sun, J., Chen, H.: Persistent memory hash indexes: An experimental evaluation. Proceedings of the VLDB Endowment **14**(5), 785–798 (2021)

8. Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y.J., Wang, Z., Xu, Y., Dulloor, S.R., et al.: Basic performance measurements of the intel optane dc persistent memory module. arXiv preprint arXiv:1903.05714 (2019)
9. Litwin, W.: Linear hashing: a new tool for file and table addressing. In: VLDB. vol. 80, pp. 1–3 (1980)
10. Lu, B., Hao, X., Wang, T., Lo, E.: Dash: Scalable hashing on persistent memory. arXiv preprint arXiv:2003.07302 (2020)
11. Nam, M., Cha, H., Choi, Y.r., Noh, S.H., Nam, B.: {Write-Optimized} dynamic hashing for persistent memory. In: 17th USENIX Conference on File and Storage Technologies (FAST 19). pp. 31–44 (2019)
12. Pagh, R., Rodler, F.F.: Cuckoo hashing. Journal of Algorithms **51**(2), 122–144 (2004)
13. Shatdal, A., Kant, C., Naughton, J.F.: Cache conscious algorithms for relational query processing. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences (1994)
14. Van Renen, A., Horn, D., Pfeil, P., Vaidya, K.E., Dong, W., Narayanaswamy, M., Liu, Z., Saxena, G., Kipf, A., Kraska, T.: Why tpc is not enough: An analysis of the amazon redshift fleet (2024)
15. Wang, C., Hu, J., Yang, T.Y., Liang, Y., Yang, M.C.: {SEPH}: Scalable, efficient, and predictable hashing on persistent memory. In: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). pp. 479–495 (2023)
16. Xiang, L., Zhao, X., Rao, J., Jiang, S., Jiang, H.: Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In: Proceedings of the Seventeenth European Conference on Computer Systems. pp. 488–505 (2022)
17. Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J., Swanson, S.: An empirical guide to the behavior and use of scalable persistent memory. In: 18th USENIX Conference on File and Storage Technologies (FAST 20). pp. 169–182 (2020)
18. Zhuge, Q., Zhang, H., Sha, E.H.M., Xu, R., Liu, J., Zhang, S.: Exploring efficient architectures on remote in-memory nvm over rdma. ACM Transactions on Embedded Computing Systems (TECS) **20**(5s), 1–20 (2021)
19. Zuo, P., Hua, Y., Wu, J.: {Write-Optimized} and {High-Performance} hashing index scheme for persistent memory. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). pp. 461–476 (2018)