CUBIT: Concurrent Updatable Bitmap Indexing

Junchang Wang
Nanjing University of Posts and Telecommunications
wangjc@njupt.edu.cn

Manos Athanassoulis
Boston University
mathan@bu.edu

ABSTRACT

In-memory bitmap indexes are widely used for read-intensive analytical workloads because they are clustered and offer efficient reads with a small memory footprint. However, they are generally inefficient to update. As analytical applications are increasingly fused with transactional applications, leading to the emergence of hybrid transactional/analytical processing (HTAP), it is desirable that bitmap indexes support efficient concurrent real-time updates. In this paper, we propose **C**oncurrent **U**pdatable **Bit**map indexing (CUBIT) that offers efficient real-time updates that scale with the number of CPU cores used and do not interfere with queries. Our design relies on three principles. First, we employ a horizontal bitwise representation of updated bits, which enables efficient atomic updates without locking entire bitvectors. Second, we propose a lightweight snapshotting mechanism that allows queries to run on separate snapshots and provides a wait-free progress guarantee. Third, we consolidate updates in a latch-free manner, providing a strong progress guarantee. Our evaluation shows that CUBIT offers 3–16× higher throughput and 3–220× lower latency than state-of-the-art updatable bitmap indexes. CUBIT's update-friendly nature widens the applicability of bitmap indexing. Experimenting with OLAP workloads with standard, batched updates shows that CUBIT overcomes the maintenance downtime and outperforms DuckDB by 1.2–2.7× on TPC-H. For HTAP workloads with real-time updates, CUBIT achieves 2–11× performance improvement over the state-of-the-art approaches.

PVLDB Reference Format:
Junchang Wang and Manos Athanassoulis. CUBIT: Concurrent Updatable Bitmap Indexing. PVLDB, 18(2): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/junchangwang/CUBIT.

1 INTRODUCTION

Access Path Selection. When a query targets columns with secondary indexes, Database Management Systems (DBMSs) evaluate the available *access paths* based on characteristics like selectivity to choose between an index scan and a full sequential scan [26]. Historically, indexes are tree-based structures like B^+-trees [29, 36, 57] and tries [31, 39]. It is widely accepted that trees are valuable only for extremely selective queries; otherwise, scans are better [19, 53].

Proceedings of the VLDB Endowment, Vol. 18, No. 2 ISSN 2150-8097.
doi:XX.XX/XXX.XX

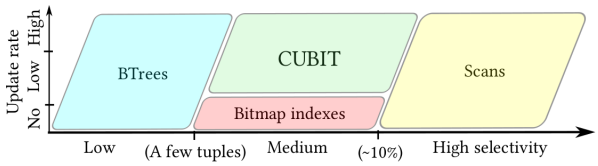


Figure 1: In the presence of both queries and updates, access path selection between tree-based indexing, bitmap indexing, and sequential scans depends on selectivity and update rate. Unlike prior bitmap indexes that were generally used for read-only workloads, our solution, CUBIT, enables bitmap indexing for higher update rates: OLAP with batched updates (§6.4) and HTAP with real-time updates (§6.5).

Bitmap Indexes. For read-only workloads, bitmap indexes are a great alternative, especially when the indexed attribute has low *cardinality* – often quantified as c/n, where c is the ratio of the number of distinct column values and n the number of rows in the table [47]. In its basic form, a bitmap index targets one attribute (of c unique values). The most common design, termed *equality encoding*, associates each value of the attribute's domain with a *bitvector*, which contains n bits, one for each row [7]. The conceptual $c \times n$ bit matrix that the bitvectors form allows queries to be quickly answered using efficient bitwise instructions [3, 7, 59].

Bitmap Index as a Secondary Index. Besides their fast query performance, bitmap indexes offer intrinsic benefits when used as secondary indexes in DBMSs. First, they have a *small memory footprint* because, apart from the bit-matrix and a mapping from the key domain to the corresponding bitvectors (columns of the bit-matrix), they do not store extra metadata like keys and pointers to tuples. Second, they are *clustered* in the sense that the order of the entries in the generated bitvector follows the physical order of the tuples. This significantly boosts query performance over traditional secondary indexes, which are generally unclustered. Third, bitmap indexes are *composition-friendly*, that is, multiple bitmap indexes on different attributes can be composed on demand, which outperforms multi-column tree indexes for large datasets (more details in §2). As an example, using a row-based DBMS for analytical queries, our bitmap index reduces memory consumption by 92% and boosts query performance by 1.7–2.6× when compared with tree-based multi-column indexes (more details in §6.4.1). In addition, for moderate selectivity, bitmap indexes incur less CPU cache and TLB misses than scans. For example, DuckDB [48] (a column-based OLAP DBMS) with our bitmap index achieves a ~2× performance improvement over its highly optimized scan, and this trend continues until the selectivity reaches up to 10% (§6.4.2).

Challenges with Bitmap Indexes. Despite their benefits, bitmap index designs do not typically target update-intensive workloads [47, 50]. To support updates, DBMSs either drop existing bitmap indexes and then rebuild them from scratch, or perform expensive batched updates that cause temporary inefficiencies [37, 47], while

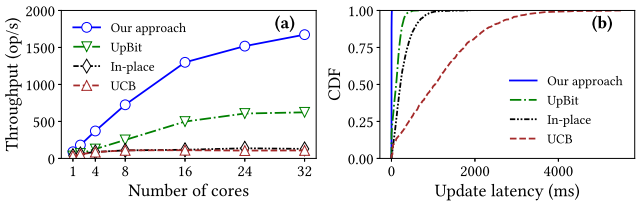


Figure 2: (a) Existing updatable bitmap indexes do not scale on multicores, and (b) have long tail latency.

some systems offer more efficient updates using multi-versioning and segmentation (e.g., Oracle which is closed-source). Overall, bitmap indexes target read-mostly workloads, as shown in Fig. 1.

To address this challenge, recently proposed *updatable bitmap indexes* [3, 6] support real-time Update, Delete, and Insert operations (henceforth, *UDI*s in short) using additional bitvectors to capture incoming updates in an *out-of-place* manner. In particular, UCB [6] uses a single bitvector to invalidate rows, while UpBit [3], the state-of-the-art solution, associates one new bitvector with each value of the domain to serve as a buffer for UDIs (§2).

When integrating these approaches into modern DBMSs, we found the following three drawbacks. (1) Prior updatable bitmap indexes were designed to run sequentially. When used on multicores, in-progress UDIs block concurrent queries that may fail to meet performance requirements (e.g., from a Service-Level Agreement). (2) UDIs on bitmap indexes flip bits in bitvectors that are commonly compressed using Run-Length-Encoding-based techniques like WAH [60]. To do this, one has to decode each bitvector, flip the corresponding bits, and re-encode. This *decode-flip-encode* procedure is time-consuming with large datasets. (3) Skewed datasets raise even higher contention around a few hot-spot bitvectors.

To demonstrate these performance issues, we parallelize and experiment with three bitmap index designs that support updates, *In-place* [53], *UCB* [6], and *UpBit* [3], by using a 48-core system (see §6 for experimental setup). Figure 2a shows that In-place and UCB scale only up to 4 CPU cores, while UpBit scales better but plateaus at 16 cores. Moreover, because of the increased contention at high concurrency levels, these bitmap indexes suffer from surprisingly long tail latency. Figure 2b shows that with 16 cores, their UDI latency can reach up to 6 seconds (see §6.1 for queries).

Our Approach. To solve the aforementioned drawbacks, we propose CUBIT, a new design for **C**oncurrent **U**pdatable **Bit**map indexing. CUBIT offers real-time updates and scales with the number of CPU cores, as shown in Figure 2 (blue lines).

From a bird's-eye view, CUBIT performs UDIs on compressed value bitvectors (VB) in an out-of-place manner, leverages multi-versioning for wait-free queries [21], and adopts latch-free techniques for real-time UDIs. Overall, CUBIT's design encompasses:

(A) Horizontal Update Deltas (HUD). We introduce HUD, a horizontal representation of the bits flipped by each UDI (§4.1). Contrary to prior work that buffers UDIs in additional bitvectors, we organize out-of-place updates per row of the bit-matrix in contiguous memory, concentrating all bit-flips to a single place (instead of multiple bitvectors). HUDs enable atomic UDIs without locking entire VBs, resolving the major update bottleneck for bitmap indexes (§3).

(B) Lightweight Snapshots. We develop a lightweight snapshotting mechanism for bitmap indexes (§4.5) using compact deltas (§4.2), batched merges (§4.3), and segmentation (§4.4). CUBIT allows queries and UDIs to work on *different index snapshots* and guarantees that an analytical query can be completed in a finite number of steps, even when UDIs are in progress. In contrast, the atomicity granularity of tree-based indexes (e.g., Bw-Tree [36] and ART [31]) is smaller, and their range queries could be interrupted by UDIs.

(C) Scalable Synchronization. Concurrency control mechanisms like two-phase locking (2PL) and multi-version concurrency control (MVCC) use latches to serialize UDIs on the same portion of data. However, updatable bitmap indexes face high contention because (1) UDIs lock bitvectors (which are typically significantly fewer than records or pages that are locked when using other indexes), (2) and skewed UDIs concentrate around a few hot-spots bitvectors leading to even higher contention (further discussed in §3). To meet the time constraints for indexing, we address this issue by developing a *consolidation-aware* [23] and *latch-free* [22] UDI mechanism (§4.7). Rather than competing with each other, CUBIT's UDIs work synergistically to reduce contention, thus making CUBIT scalable even with high UDI ratios and skewed datasets.

Broader Applicability. Being update-friendly, CUBIT widens the applicability of bitmap indexing by enabling DBMSs to maintain bitmap indexes on frequently updated attributes, in a variety of use cases, including OLAP with batched updates (§6.4) and HTAP with real-time updates (§6.5), the green area shown in Figure 1. CUBIT instances not only reduce the amount of data read from storage for the Scan operator, but also provide sufficient information for the Join and Aggregation operators, eliminating the need to build the costly intermediate data structures (e.g., hash table for joins).

Contributions. Our work offers the following contributions.

- We parallelize state-of-the-art bitmap indexes and analyze their bottlenecks. Based on the insights gained from this effort, we propose CUBIT, a concurrent in-memory updatable bitmap index that combines (i) a horizontal bitwise representation of updates, (ii) lightweight snapshotting, and (iii) a latch-free design.
- We extensively evaluate CUBIT using synthetic workloads and industry-grade benchmarks, showing that it offers 3–16× higher throughput than the baselines at different concurrency levels, with 4–13× lower query latency and 3–220× lower UDI latency.
- CUBIT expands the applicability of bitmap indexing by enabling bitmap indexes on frequently updated attributes. We demonstrate this by integrating CUBIT in both DBx1000 [62] and DuckDB [48], using both OLAP and HTAP workloads.
  - Our evaluation in DBx1000 on TPC-H with standard refresh operations [54] shows that CUBIT (1) does not introduce any maintenance overhead, and (2) compared to state-of-the-art indexes (ART, Bw-Tree, and B$^+$-Tree), provides 1.7–2.2× higher throughput with up to 92% lower memory footprint.
  - Our evaluation shows that the CUBIT-powered query engine achieves a 1.2–2.7× query performance improvement on 12 out of 22 TPC-H queries compared to optimized native approaches in DuckDB.
  - Experimenting with CH-benCHmark [11] with real-time updates shows that CUBIT provides 2.1–11.2× higher throughput than DBx1000 (with indexes and scans).
- Overall, our results show that CUBIT is a promising indexing candidate for selective queries on workloads with updates.

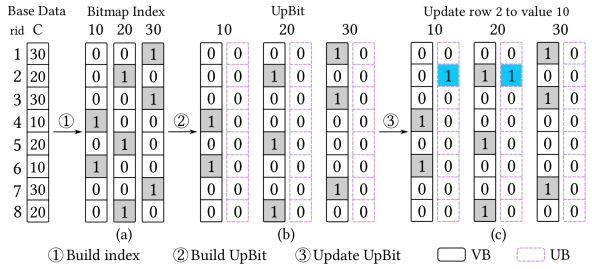


Figure 3: (a) A classic bitmap index. (b) The state-of-the-art updatable bitmap index UpBit [3] that associates a UB with each VB. (c) UpBit's UDIs update highly-compressible UBs.

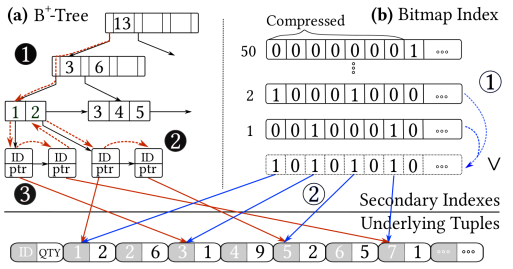


Figure 4: Queries of the form "*QTY < 3*" by using either a B$^+$-tree or a bitmap index. The bitmap index is clustered, composition-friendly, and space-efficient.

2 BACKGROUND ON BITMAP INDEXES

Bitmap Indexes Basics. Modern bitmap index designs [44, 45, 59] introduce a *bit-matrix* structure consisting of several bitvectors, one for each distinct domain value. The k^{th} bit of each bitvector is set to 1 if the k^{th} row of the attribute is equal to the corresponding value; otherwise, it is set to 0. Figure 3a shows an uncompressed bitmap index for an attribute with three different values. This bit-matrix structure fits static analytical workloads; an equality query simply reads the corresponding bitvector, and a range query performs a bitwise OR between the corresponding bitvectors.

Keeping Bitvectors Small. Bitvectors contain a lot of 0s, thus being amenable to compression. One class of compressing techniques uses Run-Length Encoding (RLE) [1, 10, 12, 18, 60]. In particular, Word-Aligned Hybrid (WAH) [60] splits the original bit-string into 31-bit words. A compressed bitvector contains two types of words: *fill* and *literal words*. The first encodes long sequences of 0s or 1s using RLE. The latter is for segments blended with 0s and 1s. For each word, the most significant bit is used to indicate its type (*fill* or *literal*). A literal word uses the remaining 31 bits to store the original bit-pattern, while a fill word uses the second most significant bit to indicate if the fill is all 0s or 1s, and the remaining bits to keep track of the number of consecutive fill words. In our work, we use the widely used open-source implementation of WAH [3, 6, 59].

Updatable Bitmap Indexes. Only a handful of bitmap indexes are update-friendly because of the costly *decode-flip-encode* procedure [3, 6, 53]. The most straightforward approach, denoted *In-place* [53], directly updates the underlying bit-matrix. For example, to update the n^{th} row from value v_1 to v_2, *In-place* decodes-flips-encodes both bitvectors for v_1 and v_2. To reduce this cost, *UCB* [6] introduces an additional compressed *existence bitvector* (EB) that indicates the validity of each row. Initially, all bits in EB are set to 1s. A delete operation sets the corresponding bit in EB to 0. An update operation appends the new value to the tail of the bitvector and maps the old row ID to the new one. The efficiency of UCB is predicated on EB being highly compressible and value bitvectors being immutable. However, their performance deteriorates sharply as UDIs accumulate and EB becomes less compressible [3].

UpBit. To address the above challenges, UpBit [3] maintains for every value in the domain, *val*, a value bitvector (VB) and an extra *update bitvector* (UB) to keep track of updates to VB (Figure 3b). Both VBs and UBs are compressed. To update the 2^{nd} row from value 20 to 10, UpBit flips the 2^{nd} bits of the UBs of values 10 and 20 (Figure 3c). Similarly, in order to delete the n^{th} row, UpBit retrieves the current value of this row and then flips the n^{th} bit of the corresponding UB. In order to insert a new entry, UpBit appends 1 at the tail of the corresponding UB and increments the global variable *N_ROWS* [3]. The core idea of UpBit is to perform UDIs on UBs. Because UBs are sparse and, thus, highly compressible, updating them is inexpensive. In several cases, it can work directly on compressed bitvectors, and when this is not possible, the decode-flip-encode cycle is lightweight. A query on value *val* performs a bitwise *XOR* on the corresponding <VB, UB> pair to retrieve the up-to-date bitvector, and a range query performs bitwise *OR* among the resulting bitvectors. As 1s in UBs accumulate, UpBit merges <VB, UB> pairs opportunistically (at query time) and generates new versions of VBs along with empty, highly compressible UBs [3].

Bitmap Indexes vs. Tree Indexes. To motivate our goal of widening the applicability of bitmap indexes, we now compare them with tree-based indexes, as shown in Figure 4. Consider a table with the primary-key attribute *ID* and a non-primary-key attribute *Quantity* (QTY for short) that spans the range of integers [1, 50] and that the underlying tuples are fixed-length (at the bottom of the figure). A B$^+$-tree secondary index on attribute *QTY* can accelerate range queries as shown in Figure 4a. We use the QTY values as the index keys and associate each leaf pointer with a linked list in case of duplicates. Each node in the list contains a pointer to the corresponding tuple and the *ID* to keep the list ordered. For a range query of the form *"QTY < 3"*, we ❶ traverse down the tree to the leaves, ❷ visit the linked lists, and ❸ access underlying tuples (red arrows in Figure 4), avoiding full scans.

On the other hand, Figure 4b illustrates a bitmap index on the same attribute. The bitmap consists of 50 compressed bitvectors, which, for ease of presentation, are laid out horizontally. In order to answer the same query *"QTY < 3"*, we ① perform a bitwise *OR* operation between the two bitvectors of values 1 and 2 to generate an on-the-fly resulting bitvector, and ② access the underlying fixed-length tuples sequentially in one pass (blue arrows).

The Benefits of Bitmap Indexing shown in Figure 4 are:

Clustered. Bitmap indexes are clustered [53] in the sense that the order of the entries in the generated bitvector follows the physical order of the tuples (blue arrows in Figure 4). Therefore, queries on bitmap indexes read data pages sequentially and only once. This addresses one significant performance drawback of existing secondary indexes that are typically unclustered (red arrows).

Small Memory Footprint. Being clustered, a bitmap index can efficiently transform the 1s in the resulting bitvector to row IDs, thus minimizing metadata size. The memory consumption of a bitmap

index is mainly due to its bitvectors, which can be highly compressed (§2). This addresses a significant space overhead of existing secondary indexes: tree nodes store metadata like pointers to tuples and, in some cases, primary key values, as shown in Figure 4.

Composition-friendly. In order to support multi-column indexes covering multiple attributes (e.g., *QTY* and an additional *Date*), bitmap indexes create an instance for each attribute and apply bitwise ANDs/ORs among the corresponding bitvectors in both instances. In contrast, tree-based indexes commonly build a single index for multiple attributes by using composite search keys (e.g., <*QTY*, *Date*>) [53]. In this paper, we demonstrate that with large datasets, composition-based bitmap indexes are faster than tree-based indexes. The reason is that bitmap indexes have a smaller memory footprint, and queries mainly perform sequential memory access to bitvectors, incurring fewer TLB and LLC misses (§6.4.1).

3 WHY BITMAP INDEXES DO NOT SCALE

Prior updatable bitmap indexes are single-threaded and do not allow UDIs and queries to execute simultaneously. We now discuss how to parallelize these designs for modern multi-cores and analyze their performance bottlenecks to motivate CUBIT's design decisions.

Bitmap Index Parallelization. UpBit can be parallelized using a fine-grained synchronization mechanism, as illustrated in Figure 5a. Specifically, the <VB, UB> pair of every value v is protected by a reader-writer latch, denoted $latch_v$. Global variables like *N_ROWS* are protected by a global latch $latch_g$. Update and delete operations first acquire the $latch_v$ of all values in shared mode to retrieve the current value of the specified row. Then, they upgrade $latch_v$ of the corresponding bitvectors to exclusive mode in order to flip the necessary bits. An insert acquires $latch_g$ and the corresponding $latch_v$ in exclusive mode. Consequently, a query operation acquires $latch_g$ and the corresponding $latch_v$ in shared mode.

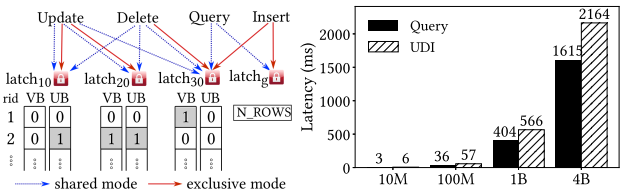
Parallelizing other updatable bitmap indexes involves coarse-grained global latches, and thus, the parallelized algorithms suffer from higher contention. We refer interested readers to the extended version of this paper for details [56].

Challenges. Parallelized updatable bitmap indexes scale poorly and incur high tail latency (§6.1) due to the following three challenges.

(C1) *High Contention.* Queries and UDIs access the same bitvectors simultaneously, leading to contention (Figure 5a). Each UpBit operation accesses two or more memory locations and thus acquires multiple latches in shared (blue dashed) and/or exclusive mode (red solid). For high concurrency, this leads to long chains, where each operation waits for the preceding one to complete.

(C2) *Long Critical Sections.* A query typically involves decoding and evaluating a bitvector, all under the protection of a latch. As a result, latches on bitvectors are *heavyweight* because their critical sections can last milliseconds or even seconds, which is orders of magnitude longer than typical critical sections (e.g., append to the tail of a list). Therefore, with large datasets, each query takes up to seconds and incurs severe UDI delays, as shown in Figure 5b. Further, UDIs can incur severe query delays because of the expensive *decode-flip-encode* procedure.

(C3) *Hot Bitvectors Are a Bottleneck.* In practice, the distribution of indexed attributes is not always uniform, such that a few bitvectors have much more 1s than the remaining ones. As a result, those



(a) Parallelized UpBit. (b) Mean query and UDI latency.

Figure 5: Existing updatable bitmap indexes do not scale because (a) contention arises for high concurrency and (b) update costs increase with large datasets.

bitvectors (a) are less compressible and lead to a more expensive decode-flip-encode cycle, and (b) are more likely to be updated by UDIs, incurring even higher contention on the associated latches.

4 CUBIT DESIGN

We now discuss CUBIT's building blocks and API.

API. CUBIT complies with the standard specification for database indexes [53] and supports Query, Update, Delete, and Insert operations. The Query operation returns the result as a bitvector or a row ID list. Update retrieves the current value of the specified row (by checking all bitvectors) and updates that row to the new value. Insert appends the new value to the tail of the bitmap index. Further, an internal Merge operation propagates logged changes to VBs. CUBIT's operations are *atomic*.

Forward Progress. The Query operation is wait-free [21] with guaranteed completion, in contrast to state-of-the-art tree-based indexes where UDIs may interrupt overlapping range queries and force them to restart [36]. For UDIs, we implemented two versions: the basic version, CUBIT-lk, that employs a latch to synchronize concurrent UDIs, and a latch-free version, CUBIT-lf, that adopts the *helping mechanism* [2] and avoids UDIs blocking each other.

4.1 Horizontal Update Delta (HUD)

Similar to the basic bitmap indexes (§2), CUBIT associates each value of the indexed attribute to a compressed VB. It avoids expensive decode-flip-encode cycles for each UDI by storing update information in an out-of-place row-wise manner and merging it into VBs lazily. Note that, by row-wise, we refer to a row of the conceptual bit-matrix of the bitmap index. Our design, termed *Horizontal Update Deltas (HUDs)*, enables efficient atomic UDIs without locking entire bitvectors, and hence addresses the high contention challenge in prior designs (**C1** from §3). HUD is key for high concurrency, by offering an effective organization of UDI deltas.

Organization of HUD. Each HUD is conceptually a bit-array with a length equal to the cardinality of the domain. In the general case, it has only 0s (i.e., the row has not been updated), and 1s accumulate with updates. The number of 1s in a HUD is rarely more than two, as we discuss below. Therefore, we compact each HUD as a list of positions, <*row_id, n_ones, p1, p2, ...*>, where n_ones is the number of 1s in the raw HUD, followed by their positions.

Figure 6a illustrates a bitmap index initially equivalent to the one shown in Figure 3a. An update operation of the 2^{nd} row to value 10 would flip the VBs containing the old and the new value. Instead, CUBIT simply logs the delta information in the HUD <2, 2, 1, 2>. Queries fetching this HUD apply it to the bitmap index by flipping

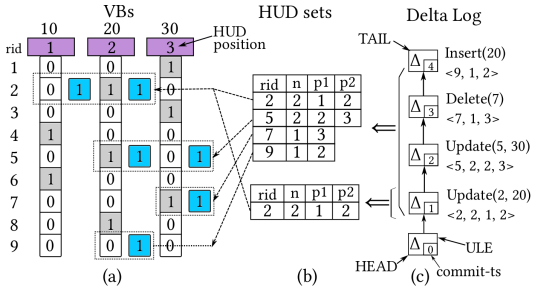


Figure 6: CUBIT records UDIs in the form of HUDs that are organized chronologically in the per-index Delta Log (c). Each query traverses a portion of the Delta Log and retrieves an on-demand HUD set (b). When applied to VBs, different HUD sets result in different snapshots of the index (a).

the 2^{nd} bits of both VBs of values 10 and 20. Similarly, updating the 5^{th} row to value 30, deleting the 7^{th} row, and inserting value 20 are executed by storing their delta information (Figure 6c), which will be applied to the queried VBs on demand.

Benefits. Using HUDs brings the following benefits. First, it enables lightweight atomic UDIs. By gathering the information of UDIs performed on each row, which was scattered among different VBs, UDIs do not lock entire bitvectors. Second, HUDs allow for efficient snapshotting for the underlying bit-matrix. HUDs maintain complete information on the completed UDIs, such that different HUD sets efficiently generate different snapshots of the index (§4.5).

HUD Has Small Size. A HUD, in most cases, has up to two bits set, as shown in Figure 6b. However, specific interleavings of updates and merges may result in longer HUDs, with very low probability. For example, when $cardinality = 128$, the probability that a HUD contains 2, 3, or 7 1s is $\frac{1}{2^7}$, $\frac{1}{2^{14}}$, and $\frac{1}{2^{42}}$. More details can be found in the extended version of this paper [56].

4.2 Delta Log

Organization. CUBIT stores the changes from UDIs in a log, called *Delta Log*, which consists of (linked) instances of *UDI log entries*, named *ULE* (Figure 6c). Each *ULE* entry contains the UDI's execution timestamp (*commit-ts*, in the bottom-right corner of each *ULE*). The corresponding operations and the generated HUDs are listed on the right-hand side of the *ULEs*. A UDI completes by appending its *ULE* at the tail of Delta Log. Two global pointers, *HEAD* and *TAIL*, point to the first and last *ULEs* in the list. The *commit-ts* values monotonically increase from HEAD to TAIL, allowing CUBIT to retrieve a HUD set by traversing the list in one pass. We initially insert a dummy ULE with time 0, ensuring the Delta Log is never empty and simplifying subsequent read operations. Appending HUDs to a log shortens UDIs' critical sections (**C2** from §3).

Example. As a concrete example, in Figure 6c, HEAD points to the dummy *ULE* that links to the *ULE* (with *commit-ts = 1*) corresponding to the update operation that changes the second row from value 10 to 20. The subsequent Update operation changes the value of the 5^{th} row to 30, by appending a *ULE* with a HUD <5, 2, 2, 3>. Then, a delete operation removes the 7^{th} row, and an insert operation appends value 20 at the tail of the index, by respectively appending their *ULEs*. By traversing *ULEs* in different timestamp ranges, queries can retrieve different HUD sets (Figure 6b) and thus different snapshots of the index (§4.5).

The exact implementation of Delta Log is orthogonal to CUBIT. Since the *commit-ts* values of *ULEs* monotonically increase, any data structure that implements the *list* abstract data type can be used. In this paper, we chose a singly linked list for ease of presentation.

4.3 Merging HUDs to VBs

In order to help queries and UDIs avoid traversing a long list of *ULEs* to retrieve the necessary HUD sets, CUBIT maintains multiple versions of each VB, previously merged with HUDs.

Multi-Versioning VBs. To this end, each value of the indexed domain is associated with a *version chain*, i.e., a linked list of VB instances, as shown in Figure 7a. Each VB instance contains its creation timestamp *commit-ts* (bottom-right corner of each VB instance), a pointer *start_delta* to the ULE where subsequent queries and UDIs may start retrieving (not already merged) HUDs, and a pointer, *prev*, linking to the previous version. A query operation traverses the corresponding version chain (starting from the newest version), chooses the one with the largest *commit-ts* that is less than or equal to the query's *start-ts*, and then scans the Delta Log from the *ULE* pointed to by the selected version's *start_delta*.

Figure 7 illustrates a merge on value 30, starting from the index shown in Figure 6. CUBIT first makes a private copy of the latest version of the bitvector of value 30 (the version with *commit-ts* = 0 in Figure 7a), retrieves the HUD set shown in Figure 7b, and then merges the HUDs for value 30 into the private copy by flipping the 5^{th} and 7^{th} bits. CUBIT then creates a new VB instance with an incremented *commit-ts* and a *prev* pointer referencing the old version of this VB. Its *start_delta* points to the newly-generated *ULE* to skip the merged *ULEs* and accelerate subsequent queries. For example, queries with *start-ts* > 5 on value 30 scan the Delta Log from *ULE* with *commit-ts* = 5, rather than from *HEAD*.

Synthetic ULE. Upon merging a VB with a subset of HUDs from the Delta Log (flipping the bits indicated by HUDs associated with this VB), a new VB version is produced. We now need to remove these specific entries from the Delta Log to ensure correctness. However, in-place updating existing *ULEs* would raise complicated synchronization issues. Instead, CUBIT appends a new *synthetic ULE* that contains the HUDs that have not yet been merged, invalidating previous HUDs for those rows. For example, Figure 7a shows the new version of VB for value 30 at timestamp 5. HUDs <5, 1, 2> and <7, 0, ∅> are generated and appended to the Delta Log, shown in Figure 7c, indicating that the 1s that were in the 5^{th} and 7^{th} row have been merged into the newly-generated VB. Note that the HUD <7, 0, ∅> can be omitted in practice. Each synthetic *ULE* corresponds to a successful merge operation, and each HUD in the synthetic *ULE* corresponds to one merged bit. Note that ULEs are truncated when all their HUDs are invalidated.

4.4 Segmented Bitvectors

Unlike prior update-friendly bitmap indexes, CUBIT uses a layout where each bitvector is divided into *multiple segments of fixed raw size* (bit count). Each segment is then independently compressed using standard techniques like WAH.

Benefits. Even though segmentation has been widely used in prior research on both bitmaps [34] and DBMSs [16, 25, 30], our mechanism is specifically designed for updatable bitmap indexes with the following unique features. (1) It limits the cost of flipping bits

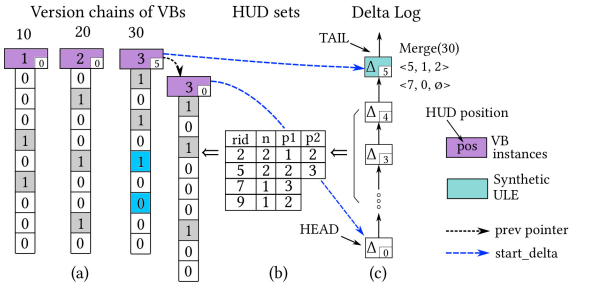


Figure 7: A Merge operation generates a new VB, reducing the cost of applying deltas for subsequent queries and UDIs.

of each UDI within one segment per bitvector (without any possible structure modification operations), hence reducing the cost of UDIs to a pre-defined threshold. (2) By compressing corresponding segments using the same mechanism, it optimizes one of the most expensive operations: logical bitwise operations between segments.

The combination of Delta Log (§4.2), merging (§4.3), and segmentation (§4.4) keep the critical sections of updating bitvectors short, overall addressing challenge **C2** from §3.

4.5 Snapshotting

Internally, CUBIT utilizes multi-versioning with timestamp ordering to allow queries and UDIs to work on different snapshots, so that queries are guaranteed to complete even when UDIs are in progress. To this end, CUBIT maintains a global *TIMESTAMP*, which is read at the beginning of every operation, denoted as *start-ts*, and is incremented to reflect changes to the bitmap index (i.e., when UDIs complete or merge operations succeed).

Multi-Versioning HUD sets. Given an operation's *start-ts*, we can retrieve a HUD set by (1) traversing Delta Log from HEAD to the *ULE* with the largest *commit-ts* less than or equal to the operation's *start-ts* and (2) collecting the HUDs in all of the *ULEs*. If HUDs for the same row appear in different *ULEs*, the one committed lately is used to reflect the latest updates. Traversing the Delta Log can be significantly accelerated by merging HUDs into VBs (§4.3) and moving HEAD forward.

Applying HUD Sets. Applying different HUD sets to the underlying bit-matrix results in different snapshots of the index. Take Figure 6b as an example. By traversing Delta Log, queries with *start-ts = 1* use the bottom HUD set with 1 HUD, while queries with *start-ts = 5* retrieve the top HUD set with 4 HUDs, capturing more UDIs. When a HUD set is large, CUBIT first orders the HUDs according to their row ids, and then invokes multiple threads, each of which updates a subset of the bitvector's segments (§4.4). By leveraging HUD sets, CUBIT offers lightweight snapshotting.

4.6 Index Operations

Queries. A query operation Q on the value *val* first retrieves a *start-ts* by reading the global *TIMESTAMP*, which essentially takes a snapshot of the bitmap index. It then uses the *start-ts* to search the version chain of the value *val*, and selects the bitvector with the largest *commit-ts* that is less than or equal to *start-ts*, which we refer to as the bitvector V. For the delta information, it traverses the Delta Log and collects the HUDs in *ULEs* with their *commit-ts* $\in$ (*V.commit-ts*, *Q.start-ts*]. The search starts from the *ULE* pointed to by the shortcut pointer *Q.start_delta*, which reduces the length of the traversal. The collected HUDs that do not affect value *val* are filtered out. If the result set is not empty, CUBIT makes a private copy of V, applys the delta information by flipping (in batch) the corresponding bits of the private copy, and then evaluates the resulting bitvector. If the number of bits flipped is larger than a pre-defined threshold, the query operation sends a merge request to the background maintenance threads which attempt to reuse the newly-generated bitvector and insert it into the version chain. If V is segmented (§4.4), CUBIT parallelized querying V using the available CPU cores at a nearly linear speedup.

UDIs. Updates and deletes use *start-ts* to retrieve the corresponding snapshot, similar to queries. Using this snapshot, they read the old value of the specified row by checking each VB along with the associated HUD set. This step is embarrassingly parallelized, by splitting the domain into ranges and assigning them to background threads. Inserts omit this step. A UDI logically flips the corresponding bits in the VBs. Specifically, an update operation on the n^{th} row would flip the n^{th} bit of the VBs corresponding to the old and the new value, a delete operation would flip the n^{th} bit of the deleted value, and an insert operation would set the last bit of the VB of the specified value *val*. CUBIT actually generates a HUD for each UDI and appends it at the tail of Delta Log.

4.7 Synchronization Mechanisms

The lightweight snapshotting mechanism (§4.5) allows queries to take snapshots of the whole index, such that CUBIT's queries are *atomic* and *wait-free*. On the other hand, UDIs must be atomic with respect to other concurrent operations. To this end, our basic algorithm employs a per-bitmap latch $latch_g$ to serialize the attempts to append *ULEs* at the tail of Delta Log and increment TIMESTAMP. Specifically, a UDI operation U first attempts to grab $latch_g$. Since U works on a snapshot of the bitmap index, other operations (including merge operations) may have appended new *ULEs* since U started. We thus check if there are any write-write conflicts between U's HUDs and the set of HUDs in *ULEs* with timestamps $\in$ (*U.start-ts*, *TAIL.commit-ts*]. Two HUDs conflict if they refer to the same row. If there is a conflict, U discards its HUD and restarts by reading with a new *start-ts*. Otherwise, U appends its *ULE* at the tail of Delta Log and increments TIMESTAMP by one. It then releases the latch. By incrementing the global TIMESTAMP, U becomes visible to other operations.

Overhead. Even though CUBIT employs $latch_g$, queries do not acquire it. Moreover, the work in the critical section of this latch is orders of magnitude lighter compared to that of the latches employed by other bitmap indexes (thus, addressing **C2** from §3). Our evaluation shows that CUBIT outperforms the alternatives in terms of throughput and (especially) tail latency (more details in §6.1).

Optimizations. Experimentally, we found that $latch_g$ can still raise high contention (see the full version of this paper [56] for an example with access skew). We propose two mechanisms to alleviate the pressure from hot bitvectors, addressing **C3** from §3.

Consolidation Array. When Delta Log's contention is high, a *consolidation array* approach [23, 42, 51] is beneficial. The basic idea is that when UDIs conflict, instead of busy-waiting, blocked UDIs consolidate their HUDs and delegate committing them in *ULEs* to subsequent ones. This allows a group of UDIs to consolidate into a single append operation, thus reducing contention.

Making CUBIT Latch-Free. We further introduce a *helping* mechanism [2, 41] that makes CUBIT latch-free. When UDIs and merges attempt to append *ULEs* to the log simultaneously (i.e., write-write conflicts), they first *help* the other complete and then retry, rather than competing with each other. Specifically, each UDI and merge operation records the old and new values of the variables to be updated in its *ULE* before appending it to the tail of Delta Log using a *CAS* instruction. Once this step O_1 succeeds (i.e., this UDI operation linearizes), the *ULE* becomes accessible to other threads via the next pointers of the *ULEs* in Delta Log. If another UDI or merge operation O_2 fails to append its *ULE*, it first assists O_1 by retrieving the old and new values and updating the variable to the new value using *CAS* instructions. This approach allows CUBIT to prevent UDIs from blocking each other, addressing the primary cause of long tail latency in UDIs. For implementation details and proof of correctness, we refer to the extended version of this paper [56].

5 CUBIT IMPLEMENTATION

We now present the implementation details of CUBIT.

ULE Pre-allocation. CUBIT pre-allocates an array of *ULEs*, each of which is 32-byte long. If a UDI wants a larger *ULE*, which is very unlikely (§4.1), it dynamically allocates a new *ULE*; otherwise, it fetches a pre-allocated one. This design benefits hardware prefetching because queries traverse Delta Log mostly sequentially.

Timestamp Allocation. Allocating timestamps is frequently a performance bottleneck for multi-versioning [61]. CUBIT addresses this issue from two angles. First, queries read but do not increment the global timestamp, dramatically reducing its contention. Further, we use a hardware-based mechanism for timestamp allocation [24] that provides a globally synchronized clock accessed as efficiently as reading hardware registers. Overall, in our experimentation, timestamp allocation is not a performance bottleneck.

Memory Reclamation. CUBIT's snapshotting provides higher concurrency levels, at the expense of increased memory footprint. To address this, we proactively reclaim retired VBs that are no longer visible to worker threads and *ULEs* once all of their HUDs have been merged into newly generated VBs.

Since VBs and *ULEs* use commit timestamps (*commit-ts*), it is natural for CUBIT to utilize an *epoch-based* memory reclamation mechanism [2, 40]. Specifically, a version A can be safely reclaimed, if (a) a newer version B has been inserted at the head of its version chain, (b) the global TIMESTAMP has become equal to or larger than B's commit-ts, and (c) each active worker thread has successfully performed at least one operation. The correctness of this approach stems from the fact that the proposed epoch-based mechanism guarantees that before reclaiming A, there is at least one system-wide *grace period* [40]. Therefore, subsequent operations will have *start-ts* larger than or equal to B's *commit-ts*, and their queries and UDIs will not access A anymore.

Similarly, a *ULE* can be safely reclaimed, if (a) it is no longer accessible starting from any *start_delta* pointers of all VBs, and (b) each active worker thread has successfully performed at least one operation. Once the head *ULE* has been reclaimed, the global pointer HEAD is moved forward by the maintenance thread.

Background Maintenance Threads. Memory reclamation is delegated to the background maintenance threads. These threads are in charge of periodically (a) detecting invisible VBs and *ULEs*, (b) detecting the grace period by utilizing a user-space implementation of RCU [13] that helps CUBIT avoid explicitly tracking every active worker thread [55], and (c) reclaiming retired objects.

Operations between Bitvectors. DBMSs need to perform logical AND/OR operations between a group of CUBIT bitvectors. We maintain the intermediate bitvector *IB* as compressed when its bit density is below 0.2% (highly compressible) to skip many 0-filled words in subsequent operations; otherwise, *IB* is maintained as decompressed. Meanwhile, when a bitvector to be merged has a bit density greater than 2% (barely compressible), we decompress it before merging with *IB*. This approach enables SIMD-based operations on 512-bit blocks, eliminating if-else branches and improving hardware prefetching.

6 EXPERIMENTAL EVALUATION

We demonstrate that CUBIT is fast, update-friendly, and scalable, and thus, it fits analytical queries on workloads with updates.

Methodology. We experiment using In-place [53], UCB [6], and UpBit [3] as our baseline updatable bitmap indexes (§2), and parallelize them using the most scalable strategies (§3). We refer to the parallelized versions as In-place$^+$, UCB$^+$, and UpBit$^+$. We assume that indexes are fully in-memory.

Implementation. CUBIT and the baselines are implemented as C++ programs. We compile the code with GCC 11.4 on Ubuntu 22.04, and use -O3 as our optimization level. The open-source *liburcu* [13] is used as the framework of safe memory reclamation. Our artifacts implementing CUBIT and our changes to DBx1000 and DuckDB are available at https://github.com/junchangwang/CUBIT.

Infrastructure. We experiment on a server with two Intel Xeon 5317 CPUs, each having 12 physical cores with Hyper-Threading running at 3.0GHz, and 18MB shared L3 cache. The system has 48 logical cores, 196GB DDR4 DRAM, and a 1TB SSD.

Benchmarking Framework. We spawn a group of *worker threads*, each of which executes the workload with the specified distribution of queries and UDIs. Crossing NUMA does not noticeably affect (with an additional 3% throughput overhead) bitmap indexes since they access bitvectors mostly sequentially. We spawn up to 32 worker threads, each bound to a logical core, leaving the other 16 logical cores to the operating system and background maintenance threads. Each experiment was repeated ten times, and the mean values were reported (standard deviation is less than 3%).

Workloads. We first use an in-house tool to generate integer data by varying three key properties: data set size, domain cardinality, and data distribution (uniform or Zipfian). We further test with industry-grade benchmarks, including the Berkeley Earth data [4], TPC-H [54], and CH-benCHmark [11].

6.1 CUBIT Scales with Increased Parallelism

We evaluate the scalability of CUBIT and the baselines by varying the number of worker threads. We simulate a typical use-case of updatable bitmap indexes [3], where we have 90% queries and 10% UDIs, with a dataset of 100 million entries and domain cardinality equal to 100, and tune all approaches for this workload. All bitmap indexes are tuned for this use-case. For CUBIT, the merging threshold is 16, the number of segments per bitvector is 1,000. Figure 8 compares the overall throughput, along with the response time of each type of operation. We make the following observations.

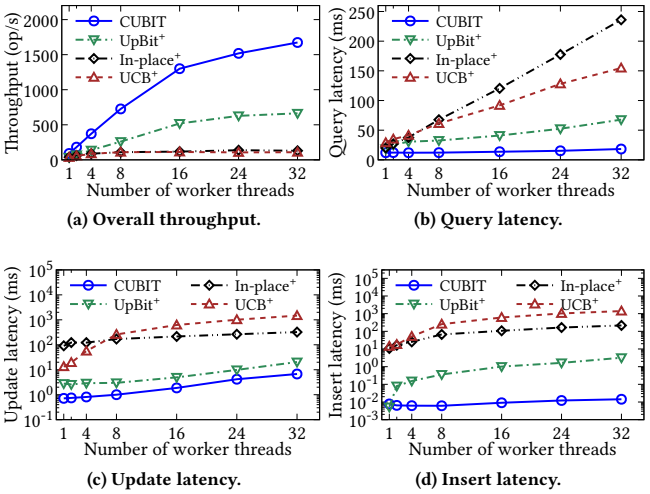


Figure 8: (a) CUBIT scales on multicore processors, and (b – d) its query and UDI latency is almost constant for high concurrency. The y-axis of (c) and (d) are in log scale.

CUBIT is Faster and Scales Better Than the Baselines. Figure 8a shows that In-place$^+$ and UCB$^+$, both of which are parallelized using global reader-writer latches, cannot scale with the number of worker threads. By utilizing fine-grained, per-bitvector reader-writer latches, UpBit$^+$ scales better but plateaus for 16 or more threads. This demonstrates that at high concurrency levels, the fine-grained locking mechanism in UpBit$^+$ becomes the bottleneck. The throughput of CUBIT, in sharp contrast, increases nearly linearly. With 32 threads, CUBIT has 2.7× (13×, 15.5×) higher throughput than UpBit$^+$ (In-place$^+$, UCB$^+$).

CUBIT Offers Fast and Scalable Read Performance. Figure 8b shows that CUBIT outperforms single-threaded UpBit for queries because merging a HUD set into a UB in CUBIT is more lightweight than merging a <VB, UB> pair in UpBit. For high concurrency, the query latency of both In-place$^+$ and UCB$^+$ increases sharply because of the increased contention on the global reader-writer latch. UpBit$^+$ distributes the contention to a group of fine-grained latches, improving query latency. CUBIT, however, outperforms all alternatives. In particular, with 32 threads, the mean latency of CUBIT is 3.9× (8.5×, 13.1×) faster than UpBit $^+$ (UCB$^+$, In-place$^+$).

CUBIT's UDIs are Fast. Figure 8c shows that for singled-threaded execution, CUBIT's updates outperform UpBit because CUBIT simply appends the update information to its Delta Log, while UpBit must decode-flip-encode the corresponding UBs. Irrespective of the number of worker threads, each update and delete operation of CUBIT takes less than 10 ms (Figure 8c), and each insert takes about 0.01 ms (Figure 8d). Note that since updates and deletes follow the same general trends, Figure 8c only shows results for updates. The mean UDI latency of CUBIT is 3.0×, 48.1×, and 220.4× faster than UpBit$^+$, In-place$^+$, and UCB$^+$, respectively.

6.2 Sensitivity Analysis

We now present a sensitivity analysis on domain cardinality, workload composition, and data size. The experimental setup is the same as in §6.1, with 16 worker threads and the best tuning for the baseline designs.

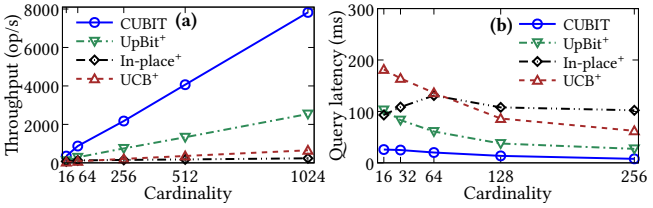


Figure 9: Varying domain cardinality does not noticeably affect CUBIT's (a) throughput and (b) query latency.

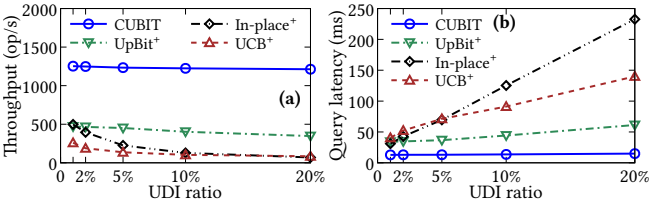


Figure 10: (a) Throughput and (b) latency of CUBIT remain with variable UDI ratio.

Impact of Domain Cardinality. Figure 9a shows that as the cardinality increases, the overall throughput of the tested approaches also increases. The reason is that a larger cardinality leads to more compressible bitvectors, hence a smaller memory footprint. Further, a larger cardinality eases the contention on each bitvector. This also leads to lower query and UDI latency, as shown in Figure 9b, which zooms to cardinality between 16 and 256. The only outlier is In-place$^+$ because of the locking contention to append 0s and 1s.

Impact of UDI Ratio. As the UDI ratio increases, the throughput of both In-place$^+$ and UCB$^+$ decreases sharply (Figure 10a). The main reason is the increased contention on the global latches that increases query latency (Figure 10b). UpBit's and CUBIT's throughput decreases as well. However, the performance loss is less than In-place$^+$ and UCB$^+$. UpBit$^+$ faces increased contention on the reader-writer latches protecting bitvectors, while CUBIT faces increased contention on the latch protecting Delta Log. Since the duration of the critical sections of CUBIT is orders of magnitude shorter than the baselines, CUBIT outperforms them in terms of both throughput and latency. Note that the performance trends in Figure 10 continues until the UDI ratio becomes larger than 80%.

Impact of Data Size. As the dataset size increases, the relative behavior of different indexes remains the same. Figure 11a shows the evaluation results with datasets containing 1B entries (cardinality = 100). Figure 11a looks almost identical to Figure 8a, which demonstrates that data size does not affect the performance trends and the relative behavior of different algorithms. In this evaluation, however, each bitvector contains more bits. Therefore, the absolute performance decreases nearly linearly as the dataset size increases.

Berkeley Earth Dataset. For a real-life application, we evaluate CUBIT and its competitors using the Berkeley Earth dataset. It is an open dataset for a climate study that contains measurements from 1.6 billion temperature reports, each of which contains information including temperature, time of measurement, and location. From the Berkeley Earth data, we extract a dataset containing 31 million entries with cardinality 144. Figure 11b shows that with 32 worker threads, CUBIT's throughput is about 2.6×, 11.5×, and 16.2× higher than that of UpBit$^+$, UCB$^+$, and In-place$^+$, respectively.

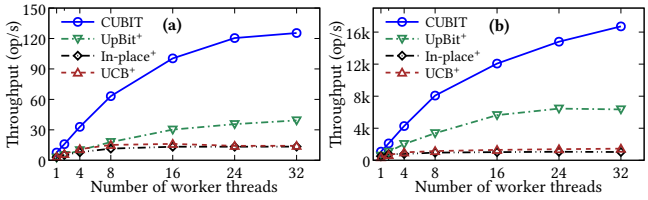


Figure 11: (a) When increasing the dataset size (1B entries), and (b) when querying real datasets (Berkeley Earth dataset with 31M tuples and cardinality of 114), the relative behavior of all approaches remains the same.

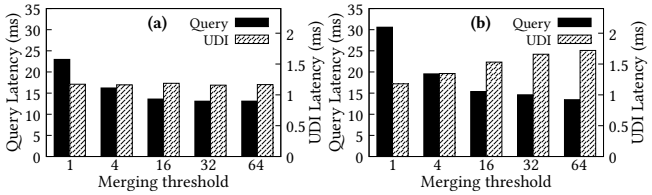


Figure 12: For workloads with (a) 10% UDIs and (b) 20% UDIs, CUBIT's query latency (left axis) decreases until the merging threshold becomes larger than 16, and UDI latency (right axis) slightly increases as a function of the merging threshold.

Impact of Data Skew. The distribution of data among bitvectors plays a key role in performance for two reasons. First, biased distributions may lead to few *target* bitvectors containing many more 1s than others, making them less compressible. Second, the target bitvectors face higher contention levels among concurrent UDIs. Our evaluation results show that for all bitmap indexes, UDI latency increases for skewed data because most of the UDIs involve a few bitvectors, leading to high contention on them. However, CUBIT remains the most stable design because the *helping mechanism* reduces the number of latches acquired, reducing CUBIT's P99 UDI latency (see extended version of this paper for details [56]).

6.3 Tuning CUBIT

We now discuss the impact of CUBIT's tuning knobs and show that static tuning delivers robust performance.

Merging HUDs into VBs. As UDIs accumulate, queries and UDIs need to traverse a long list in Delta Log. CUBIT thus merges HUDs into VBs, as described in §4.3. Figures 12a and 12b show the average query and UDI latency as we vary the merging threshold, for the workloads with 10% and 20% UDIs, respectively. In the general case, the query latency decreases as the merging threshold increases because of the reduced frequency of generating new VBs in query operations (§4.6). However, when the threshold is larger than 16, the query latency plateaus because of the increased cost of traversing the list. On the other hand, for both workloads, we observe a small increase in UDI latency as the merging threshold increases, because a UDI must traverse the Delta Log. This trend, however, is almost hidden because the update latency is dominated by the cost of retrieving the old value of the updated row. Therefore, we set the merging threshold to 16 in our experiments.

Number of Maintenance Threads. CUBIT offloads operations like garbage collection and merge operations to background maintenance threads. Thus, it is critical to know in advance how many maintenance threads are enough. Figure 13 shows the average query

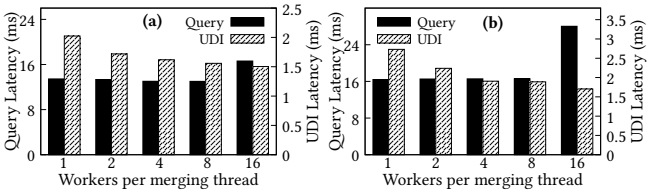


Figure 13: For workloads with (a) 10% UDIs and (b) 20% UDIs, CUBIT's query latency plateaus until the ratio of worker and maintenance threads becomes larger than 8. UDIs always benefit from an increased ratio.

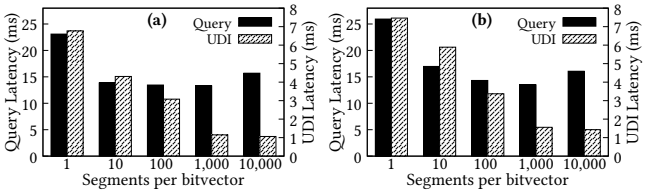


Figure 14: For workloads with (a) 10% UDIs and (b) 20% UDIs, CUBIT's query latency decreases with finer segmentation granularity until each segment becomes less than 5KB. UDIs always benefit from a finer granularity.

latency and UDI latency as we vary the ratio of worker threads and maintenance threads, for workloads with 10% (Figure 13a) and 20% (Figure 13b) updates. For both workloads, UDIs benefit from an increased ratio (fewer merging threads) because of a decreased contention on Delta Log. The query latency plateaus when the ratio is less than eight. However, when the ratio becomes larger than eight, query latency increases sharply because the maintenance threads become the bottleneck. We thus set the ratio as 4 and create one maintenance thread for every four worker threads.

Segments per Bitvector. CUBIT bitvector segmentation affects queries and UDIs in different ways. Figure 14 shows the average query and UDI latency for different numbers of segments of each bitvector (5MB in size), for workloads with 10% (Figure 14a) and 20% (Figure 14b) UDIs. UDIs always benefit from a finer segmentation granularity that limits the execution of a UDI within a smaller segment. However, the UDI latency plateaus when a bitvector is segmented into more than 1,000 segments (5KB-sized segments). On the other hand, a finer segmentation granularity allows a query to assign its tasks to a group of CPU cores (2 cores per query in this experiment) more evenly, significantly reducing query latency, as shown in Figure 14. For more than 1,000 segments, query latency increases due to the cost of traversing the segments, thus, we use 1,000 segments in our experiments.

Parallelism for Each Query. Using segmented bitvectors, queries can be embarrassingly parallelized. This leads to a nearly linear speedup as long as the system has available CPU cores. Figure 15 shows the average query and UDI latency as we vary the number of CPU cores for each query, for the experiments with 4 (Figure 15a) and 16 (Figure 15b) worker threads. Figure 15a shows that using less than 8 cores per query, latency increases sharply. However, for 16 cores (which means that CUBIT needs $4 \times 16 = 64$ cores in total), query and UDI latency increase sharply, as expected, since our server has 48 cores. Similarly, Figure 15b demonstrates that for 16 worker threads, the number of cores used by each query should

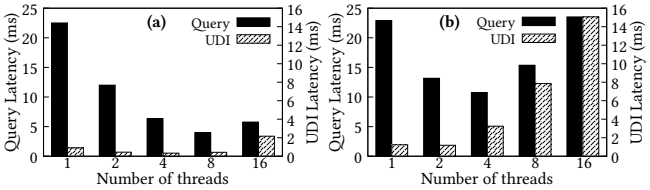


Figure 15: When there are (a) 4 and (b) 16 worker threads in the system, CUBIT's query and UDI latency decreases sharply as we assign more cores for each query, until the total required cores exceed the physical limit.

be less than 4 to not hurt query and UDI latency. Based on these, we use two cores per query since our setup has 32 worker threads.

6.4 CUBIT Benefits OLAP

We now demonstrate that CUBIT benefits OLAP with batched updates by experimenting with the TPC-H benchmark with the standard refresh operations on DBx1000 (§6.4.1) and DuckDB (§6.4.2).

TPC-H. We experiment with TPC-H with scale factor (SF) = 10 and refresh operations (RF1 and RF2). RF1 (RF2) inserts (deletes) 4,500 tuples in *LINEITEM* in batch, and then updates related indexes. The workload consists of 98% Q6, 1% RF1, and 1% RF2.

6.4.1 ***DBx1000 Integration.*** In order to compare CUBIT with the alternative indexes at different concurrency levels, we integrate CUBIT into DBx1000 [62], a row-based prototype DBMS.

Methodology. We implement Bw-Tree and ART based on their open and comparable implementations [57] and optimize their performance with our testbed. In particular, our ART uses optimized arrays, rather than linked lists, to handle duplicate keys. Overall, we compare: (1) a parallelized **Scan**, (2) a **Hash** index with bucket size $B = 64K$, (3) a **B$^+$-tree** index with a fixed fanout of 32, (4) **Bw-Tree** with a maximum inner node size of 64 [36] , (5) **ART** with a maximum node fanout of 256 [31], and (6) **CUBIT**. Since Q6 selects on *l_shipdate*, *l_discount*, and *l_quantity*, we built three CUBIT indexes. Other indexes are built using multi-column indexes over the three attributes (the *best-case* for the baselines). The composite search keys (*l_shipdate*, *l_discount*, and *l_quantity*) for these indexes are not unique, so for SF = 10, each index node contains, on average, 50 entries with the same key in the form of an array.

The Hash index is protected by fine-grained per-bucket reader-writer latches, and the B$^+$-Tree and ART employ optimistic lock coupling [32]. Both Bw-Tree and CUBIT offer latch-free queries. Apart from the Scan, each query first retrieves a set of tuple IDs through the indexes, and then fetches these tuples to calculate the query result. Tree-based indexing could ensure a better access locality by sorting the IDs first. This approach, however, requires CPU and memory resources. Similarly to prior work [26], we observed that sorting slows down tree-based indexing by ~6%.

Benefits. We observed the following benefits of using CUBIT.

(a) High Availability. CUBIT enables bitmap indexing for OLAP with batched updates, by overcoming the maintenance downtime [43]. Our evaluation results show that RF1 and RF2 do not noticeably affect concurrent queries.

(b) Low Memory Footprint. The hash index, ART, B$^+$-Tree, and Bw-Tree, respectively, use 1.92GB, 1.93GB, 1.96GB, and 1.97GB of memory, mainly because they must maintain keys and pointers to the tuples, which is about 1.44GB. In contrast, CUBIT size is only 156MB.

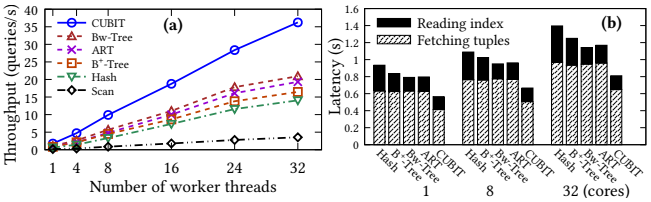


Figure 16: (a) At different concurrency levels, DBx1000 with CUBIT outperforms all baselines for TPC-H Q6. (b) CUBIT index access cost is always smaller. Moreover, CUBIT provides an ordered ID list, leading to faster tuple fetches.

(c) High Performance. Despite the batched updates, CUBIT enables faster execution for selective queries, and the throughput of DBx1000 with CUBIT scales linearly with the number of worker threads, as shown in Figure 16a. The scalability of DBx1000 with the baseline indexes suffers from random accesses and cache misses because the indexes are not clustered, while by construction, CUBIT always generates clustered base data accesses. In particular, using 32 threads in DBx1000, CUBIT provides 1.7×, 2.2×, 2.2×, 2.6×, and 10.3× higher throughput than Bw-Tree, ART, B$^+$-Tree, Hash index, and Scan, respectively. Scanning is the slowest approach because of its excessive data access and its high cache miss rate.

Table 1: Normalized last level cache misses and TLB misses of the two stages of each Q6. DBx1000 with CUBIT exhibits better spatial locality and experiences fewer misses.

Stages \ Indexes		Bw-Tree / CUBIT	ART / CUBIT	B$^+$-tree / CUBIT	Hash / CUBIT	Scan / CUBIT
Reading index	LLC	1.4	1.3	1.7	1.9	—
	TLB	1.4	1.5	2.3	2.2	—
Fetching tuples	LLC	5.4	5.5	6.3	5.8	342.7
	TLB	4.0	3.9	4.3	4.4	12.2

We inspect each indexing strategy by measuring the cost of the two stages of each Q6 – probing the index and fetching tuples. Figure 16b demonstrates that the benefits of using CUBIT are two-fold. On the one hand, probing CUBIT is faster for all concurrency levels. For example, for the serial execution (# worker thread = 1), it takes CUBIT 152ms to retrieve the tuple ID list, which is 1.1 – 3.0× faster than the alternative indexes. The main reason is that CUBIT has a smaller memory footprint, and a query fundamentally performs sequential memory access to each bitvector, exhibiting better spatial locality. This is demonstrated in Table 1 in which we list the normalized last level cache (LLC) and TLB misses of different stages of each Q6. The first two rows show that reading Bw-Tree incurs 1.4× and 1.4× more LLC and TLB misses than reading CUBIT. On the other hand, fetching tuples according to the ID list generated by CUBIT is faster. For example, for the serial execution version, it takes DBx1000 413ms to fetch the tuples by using the ID list provided by CUBIT, 1.5× faster than by using the ID lists generated by the alternative indexes, including Bw-Tree and ART. The main reason is that the tuple IDs provided by CUBIT are inherently ordered, while this is not generally the case for other indexes. Essentially, by virtue of its construction, CUBIT is a clustered secondary index since the retrieved IDs follow the same order as the tuples in the base data. In contrast, accessing data with the hash and tree-based indexes leads to more cache and TLB misses. This is shown in Table 1 (last two rows): using the ID list generated by Bw-Tree, the DBMS incurs 5.4× and 4.0× more

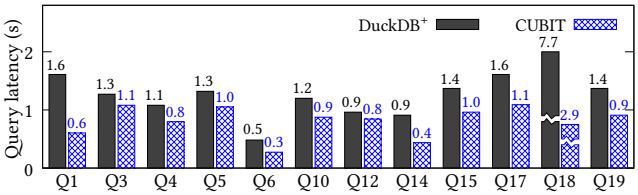


Figure 17: Being update-friendly, CUBIT can be built on the LINEITEM and ORDERS tables that are updated by refresh transactions. By replacing the column-based scan, perfect-hash-based aggregation, and hash-based join operators of DuckDB+, CUBIT-powered query engine outperforms for all of the queries.

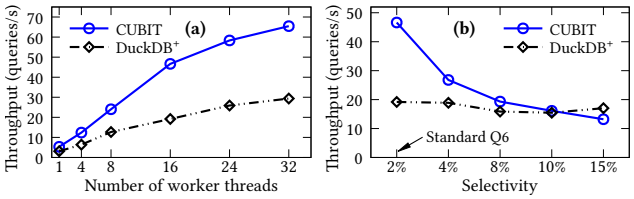


Figure 18: (a) At different concurrency levels, CUBIT-based probe outperforms the highly-optimized scan in DuckDB+ for TPC-H Q6. (b) This trend continues until the selectivity reaches 10%.

LLC misses and TLB misses than CUBIT. Finally, Table 1 shows that Scan performs poorly in row-based DBMSs because it accesses more data and incurs more cache misses.

6.4.2 ***DuckDB Integration.*** To demonstrate the usefulness of CUBIT, we integrate CUBIT into DuckDB [48], a column-based OLAP DBMS, that has been heavily optimized for analytical applications. We optimized as many choke points in DuckDB's query engine as possible [5, 15]. For example, by excluding certain group-by attributes that can be derived from the primary key, we achieved a 46% performance improvement in Q10. We optimize the Scan operator by (1) storing column values more compactly (e.g., by compressing *l_quantity* from 13 to 6 bits) with a storage layout inspired by BitWeaving's HBP [38], and (2) scanning columns using SIMD instructions, leading to a 36% lower latency for Q6. We refer to our optimized version as DuckDB+ and use it as the baseline.

CUBIT's update-friendly design allows us to maintain CUBIT instances for frequently updated attributes instead of repeatedly destroying and re-creating them [46, 47]. This not only reduces the amount of data read from storage for the *Scan* operator, but also provides sufficient information for the *Aggregation* and *Join* operators, eliminating the need to build intermediate data structures (e.g., hash table for Joins). We demonstrate this by using CUBIT to accelerate the Scan, Aggregation, and Join operators (the top three choke points in DBMS query engines [5, 15]) in DuckDB+. We use TPC-H Q6, Q1, and Q5 as illustrative examples.

Scan. By maintaining CUBIT instances on the attributes involved in the WHERE clauses, our CUBIT-powered Scan operator reads fewer columns. For example, in Q6, DuckDB+ sequentially scans four columns (*l_extendedprice*, *l_shipdate*, *l_discount*, and *l_quantity*), which accounts for 95% of the execution time. In contrast, our operator generates a resulting bitvector, allowing it to probe only two columns, effectively halving the data read from storage.

Aggregation. By maintaining CUBIT instances on the attributes used as group-by factors (e.g., *l_returnflag* and *l_linestatus* for Q1), the CUBIT-powered Aggregation operator avoids reading these columns from storage. Additionally, CUBIT eliminates the need to maintain data structures for aggregations by (1) determining the positions of matching tuples for each group-by category by *AND*ing bitvectors from CUBIT instances, and (2) calculating the aggregations for each category by reading the specified entries in one pass—a computation mode amenable to SIMD instructions.

Join. By maintaining a CUBIT instance on the join attribute (e.g., *l_orderkey* for Q5), the CUBIT-powered Join operator avoids scanning one column data in the fact table. Additionally, the CUBIT instance provides sufficient information to join two tables by using its bitvectors. For example, for the join in Q5, our operator retrieves the *orderkey* set from the ORDERS table, reads the corresponding bitvectors from the CUBIT instance, and performs logical ORs between them. Using the resulting bitvector, the operator probes the fact table and performs aggregations in one pass.

With our optimized operators, we experiment with 12 out of 22 TPC-H queries, including scan-intensive (Q6 and Q12), aggregation-heavy (Q1 and Q18), and join-dominant (Q3, Q4, Q5, Q10, Q14, Q15, Q17, and Q19) queries. We omit other queries that neither involve the fact table LINEITEM (e.g., Q2) nor present obvious choke points for query engines (e.g., Q16). Our evaluation results show that CUBIT-powered query engine is 1.2–2.7× faster than the native approaches in DuckDB+, shown in Figure 17.

We further explore the conditions under which maintaining CUBIT instances outperforms traditional column scanning by using Q6 as an example. In Q6, we create three CUBIT instances on the attributes *l_shipdate* (cardinality = 2,526), *l_discount* (cardinality = 11), and *l_quantity* (cardinality = 50). Our operator selects bitvectors for 365 days, 3 discounts, and 25 quantities, and performs ORs/ANDs between these 393 (365+3+25) bitvectors to compute the resulting bitvector (selectivity = 2%), which is maintained in segments (§4.4). We use an AVX-512-based mechanism [33] to convert the resulting bitvector to an ID list, which is used to probe the *l_extendedprice* and *l_discount* columns. We made the following observations:

CUBIT Scales for High Concurrency. Each bitvector segment covers a small range of physical pages, such that worker threads run in parallel, without any synchronization overhead, leading to near-ideal scalability (Figure 18a).

CUBIT Has Wide Applicability. The effectiveness of CUBIT makes it a potential replacement for Scan for several use cases. To demonstrate this, we use 16 cores but artificially increase the selectivity of Q6 by varying the combination of *l_shipdate* and *l_discount*. CUBIT outperforms Scans for up to 10% selectivity (Figure 18b).

6.5 CUBIT Benefits HTAP

CH-benCHmark. We use DBx1000 because of its scalability and implement the CH-benCHmark [11] that consists of a full version of the TPC-C benchmark and a set of TPC-H-equivalent analytical queries on the same tables. *T* threads run TPC-C transactions that heavily update tables, including the *ORDER-LINE* table, and the remaining *A* threads run CH-benCHmark analytical queries on these tables. We use an initial dataset of 100 warehouses, so *ORDER-LINE* contains about 2.5GB of data and 30M tuples. The TPC-C transactions insert ~10M new tuples during each trial.

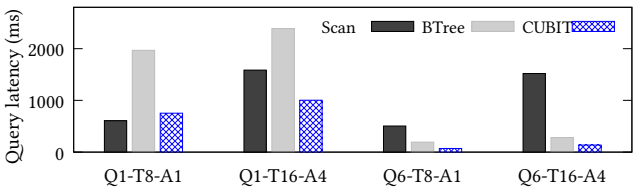


Figure 19: For HTAP DBMSs, CUBIT can speedup analytical queries with high (Q1) and low (Q6) selectivity, irrespective of concurrent updates. Higher contention (16 TPC-C threads instead of 8) and parallelism (4 analytical queries instead of 1) do not noticeably affect CUBIT's performance.

Indexes. We built one CUBIT instance for each attribute in the analytical queries (i.e., Q1 and Q6), and other indexes are built using multi-column indexes for each query, as in §6.4.1.

Concurrency Control. CUBIT can be used as a general secondary index when the DBMS has its concurrency control mechanism (e.g., 2PL). An independent lock manager is responsible for resolving conflicting data updates [35, 36] and, thus, we focus on the atomicity of CUBIT's operations. We also implement an MVCC mechanism, called *Post-Timestamping MVCC (PTMVCC)*, that is similar to Hekaton [14] but only increments its TIMESTAMP when transactions with updates successfully commit [27]. A transaction traverses the chains of tuples or speculatively reads other transactions' workspace to fetch the tuple versions visible to it. PTMVCC reduces the contention on updating TIMESTAMP [62] and provides a stronger progress guarantee for analytical queries that are never blocked nor aborted [27]. PTMVCC provides snapshot isolation. When using 16 transactional and 4 analytical threads, PTMVCC improves performance by 8% over the 2PL+CUBIT solution, primarily due to executing queries wait-free even during ongoing updates and eliminating tuple latching overhead. We thus use PTMVCC.

Selectivity. In our evaluation, we found that many attributes in CH-benCHmark cover a narrow scope, such that most queries select almost all tuples. We thus modified the propagated values and the query predicates to provide variable selectivity. For example, we set the values of the *ol_delivery_d* attribute in the *ORDER-LINE* table in the range of [1983, 2023), and the values of the *ol_quantity* attribute in the range of [1, 25000), both in a uniform distribution. As a consequence, each CH-benCHmark Q1 selects rows on years (16 out of 40) and delivery state (9 out of 10), leading to an average selectivity of $\frac{16}{40} \times \frac{9}{10} \approx 36\%$, and each Q6 selects rows on years (20 out of 40), quantities (1000 out of 25,000), and delivery state (9 out of 10), leading to an average selectivity of $\frac{20}{40} \times \frac{1}{25} \times \frac{9}{10} \approx 1.8\%$.

CUBIT Offers Fast Scalable HTAP Queries. Figure 19 shows the response time of the representative analytical queries (Q1 and Q6) at different concurrency levels (with 8 or 16 *Transactional* and 1 or 4 *Analytical* threads). Bw-Tree and ART show similar performance trends with B$^+$-Tree in our evaluation, so we denote them as *BTree*. We make the following observations. (1) For Q1, which has high selectivity (36%) and mainly performs aggregation, BTree indexing hurts analytical queries. However, CUBIT is comparable to Scan. (2) For Q6, which has low selectivity (1.8%) and spends more time scanning tuples, BTree indexing outperforms the scan. CUBIT further reduces the response time and achieves 2–11× performance improvement compared to the baselines, irrespective of concurrent TPC-C updates. (3) For both Q1 and Q6, increasing the number of analytical threads from 1 to 4 does not noticeably slow down CUBIT, but it hurts scan and increases the response time by 3× mainly because the cache miss rate increases by 2× (perf's *capacity misses*). In addition, increasing the number of TPC-C threads from 8 to 16 incurs higher contention and leads to increased query latency for BTree, while CUBIT is barely affected. Overall, CUBIT is a promising indexing candidate for analytical queries on HTAP.

7 RELATED WORK

Offering Faster Queries. Designs like TEB [28] and BinDex [37] focus on improving query performance by leveraging customized read-only data structures. Although recent research has improved update performance [49], updates remain costly. CUBIT inherently offers efficient updates and is orthogonal to these designs.

Compressing Bitvectors. Run-length encoding and its variants [1, 10, 34, 60] have been widely used to compress bitvectors. The latest research, Roaring [34], splits a bitvector into fixed-length containers that are recorded and compressed independently. These designs support updates in the granularity of bitvector (or container), which suffer from performance deterioration when naively used in updatable bitmap indexes. However, CUBIT provides a framework to atomically and efficiently update each bitvector. We select WAH [60] in CUBIT's implementation and leave utilizing Roaring, which contains structure modification operations (i.e., switching between an array and a bitmap container), as future work.

Bit-Packing. Bitmap indexes and bit-packed storage (e.g., SIMD-Scan [58], BitWeaving [38], and ByteSlice [17]) are closely related regarding space efficiency and fast data retrieval. Nevertheless, bit-packed storage is primarily optimized for fast scanning, making merging results from multiple scans typically expensive [38]. Another trend involves incorporating lightweight (e.g., Column Imprints [52] and Column Sketches [20]) and customized indexes (e.g., BinDex [37] and Cabin [9]) to accelerate scans. These research results highlight the necessity of maintaining indexes on bit-packed storage, and CUBIT is an effort toward a general solution.

Modeling Analysis. Prior research [7, 8, 45] has extensively examined bitmap indexes in terms of space-time tradeoffs. However, existing conclusions may not fully apply to CUBIT that focuses on updates. To analyze CUBIT, a new model that considers the impact of updates is necessary, which we leave as our future work.

8 CONCLUSION

We present CUBIT, a bitmap index that supports scalable, real-time updates, enabling a departure from the traditional approaches of updating bitmap indexes which are notably in a serial, batch mode. CUBIT expands the applicability of bitmap indexing by, for the first time, enabling the efficient maintenance of bitmap indexes on frequently updated attributes, thereby demonstrating the potential to accelerate a DBMS's critical operations including Scan, Join, and Aggregation. Experimenting with OLAP and HTAP workloads demonstrates that the CUBIT-powered query engine delivers remarkable performance improvement for analytical queries.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback, as well as Mark Callaghan, Wei Xi, and Weijian Guo for their valuable comments. This work is partially funded by NSF #2144547.

REFERENCES

[1] Gennady Antoshenkov. 1995. Byte-aligned Bitmap Compression. In *Proceedings of the Conference on Data Compression (DCC)*. 476–476. http://dl.acm.org/citation.cfm?id=874051.874730

[2] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing epoch-based reclamation for efficient range queries. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 14–27. https://doi.org/10.1145/3178487.3178489

[3] Manos Athanassoulis, Zheng Yan, and Stratos Idreos. 2016. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://dl.acm.org/citation.cfm?id=2915964

[4] Berkeley. 2016. Berkeley Earth Data. http://berkeleyearth.org/data/. [Online; accessed 1-May-2024].

[5] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Proceedings of the TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems (TPCTC)*. https://link.springer.com/chapter/10.1007/978-3-319-04936-6_5

[6] Guadalupe Canahuate, Michael Gibas, and Hakan Ferhatosmanoglu. 2007. Update Conscious Bitmap Indices. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. 15–25. https://doi.org/10.1109/SSDBM.2007.24

[7] Chee-Yong Chan and Yannis E. Ioannidis. 1998. Bitmap index design and evaluation. *ACM SIGMOD Record* 27, 2 (1998), 355–366. https://doi.org/10.1145/276305.276336

[8] Chee-Yong Chan and Yannis E. Ioannidis. 1999. An efficient bitmap encoding scheme for selection queries. *ACM SIGMOD Record* 28, 2 (1999), 215–226. https://doi.org/10.1145/304181.304201

[9] Yiyuan Chen and Shimin Chen. 2024. Cabin: A Compressed Adaptive Binned Scan Index. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 1 (2024), 57:1–57:26. https://doi.org/10.1145/3639312

[10] Alessandro Colantonio and Roberto Di Pietro. 2010. Concise: Compressed 'N' Composable Integer Set. *Inform. Process. Lett.* 110, 16 (2010), 644–650. https://doi.org/10.1016/j.ipl.2010.05.018

[11] Richard L. Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The mixed workload CH-benCHmark. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. 8. https://doi.org/10.1145/1988842.1988850

[12] François Deliège and Torben Bach Pedersen. 2010. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 228–239. https://doi.org/10.1145/1739041.1739071

[13] Mathieu Desnoyers. 2012. Userspace RCU. https://liburcu.org/. [Online; accessed 1-May-2024].

[14] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1243–1254. https://doi.org/10.1145/2463676.2463710

[15] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1206–1220. https://doi.org/10.14778/3389133.3389138

[16] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 313–324. https://doi.org/10.5441/002/EDBT.2019.28

[17] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 31–46. https://doi.org/10.1145/2723372.2747642

[18] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. 2010. Net-Fli: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. *Proceedings of the VLDB Endowment* 3, 2 (2010), 1382–1393. https://doi.org/10.14778/1920841.1921011

[19] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402. http://dx.doi.org/10.1561/1900000028

[20] Brian Hentschel, Michael S Kester, and Stratos Idreos. 2018. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 857–872. https://doi.org/10.1145/3183713.3196911

[21] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149. https://doi.org/10.1145/114005.102808

[22] Maurice Herlihy and Jeannette M Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[23] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 681–692. http://dl.acm.org/citation.cfm?id=1920841.1920928

[24] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. 2018. A scalable ordering primitive for multicore machines. In *Proceedings of the EuroSys Conference (EuroSys)*. 34:1–34:15. https://doi.org/10.1145/3190508.3190510

[25] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[26] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 715–730. https://doi.org/10.1145/3035918.3064049

[27] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. 2019. MV-RLU: Scaling Read-Log-Update with Multi-Versioning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 779–792. https://doi.org/10.1145/3297858.3304040

[28] Harald Lang, Alexander Beischl, Viktor Leis, Peter A Boncz, Thomas Neumann, and Alfons Kemper. 2020. Tree-Encoded Bitmaps. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 937–967. https://doi.org/10.1145/3318464.3380588

[29] Philip L Lehman and S Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670. https://doi.org/10.1145/319628.319663

[30] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 743–754. https://doi.org/10.1145/2588555.2610507

[31] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[32] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 3:1–3:8. https://doi.org/10.1145/2933349.2933352

[33] Daniel Lemire. 2022. Faster bitset decoding using Intel AVX-512. https://lemire.me/blog/2022/05/10/faster-bitset-decoding-using-intel-avx-512/. [Online; accessed 1-May-2024].

[34] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, FranÃ§ois Saint-Jacques, and Gregory Ssi Yan Kai. 2018. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience* 48, 4 (2018), 867–895. https://doi.org/10.1002/SPE.2560

[35] Justin J Levandoski, David B Lomet, Mohamed F Mokbel, and Kevin Zhao. 2011. Deuteronomy: Transaction Support for Cloud Data. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. 123–133. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper14.pdf

[36] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[37] Linwei Li, Kai Zhang, Jiading Guo, Wen He, Zhenying He, Yinan Jing, Weili Han, and X Sean Wang. 2020. BinDex: A Two-Layered Index for Fast and Robust Scans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 909–923. https://doi.org/10.1145/3318464.3380563

[38] Yinan Li and Jignesh M Patel. 2013. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 289–300. https://doi.org/10.1145/2463676.2465322

[39] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 183–196. https://doi.org/10.1145/2168836.2168855

[40] Paul E McKenney. 2014. What is RCU? https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html. [Online; accessed 1-May-2024].

[41] Maged M Michael and Michael L Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 267–275. https://doi.org/10.1145/248052.248106

[42] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. 2005. Using elimination to implement scalable and lock-free FIFO queues. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 253–262. https://doi.org/10.1145/1073970.1074013

[43] Elizabeth J O'Neil, Patrick E O'Neil, and Kesheng Wu. 2007. Bitmap Index Design Choices and Their Performance Implications. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*. 72–84. https://doi.org/10.1109/IDEAS.2007.4318091

[44] Patrick E. O'Neil. 1987. Model 204 Architecture and Performance. In *Proceedings of the International Workshop on High Performance Transaction Systems (HPTS)*. 40–59. http://dl.acm.org/citation.cfm?id=645575.658338https://link.springer.com/chapter/10.1007/3-540-51085-0_42

[45] Patrick E. O'Neil and Dallan Quass. 1997. Improved query performance with variant indexes. *ACM SIGMOD Record* 26, 2 (1997), 38–49. http://dl.acm.org/citation.cfm?id=253262.253268

[46] Oracle. 2007. Using Bitmap Indexes in Data Warehouses. *White Paper of Oracle 11g* (2007). https://docs.oracle.com/cd/E11882_01/server.112/e25554/indexes.htm#CIHGAFFF

[47] Oracle. 2024. Oracle23ai Database Concepts. https://docs.oracle.com/en/database/oracle/oracle-database/23/cncpt/indexes-and-index-organized-tables.html#GUID-B15C4817-7748-456D-9740-8B9628AF9F47. [Online; accessed 1-October-2024].

[48] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1981–1984. https://doi.org/10.1145/3299869.3320212

[49] Marcellus Prama Saputra, Eleni Tzirita Zacharatou, Serafeim Papadias, and Volker Markl. 2022. In-Place Updates in Tree-Encoded Bitmaps. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. 18:1–18:4. https://doi.org/10.1145/3538712.3538745

[50] Vivek Sharma. 2005. Bitmap Index vs. B-tree Index: Which and When? *Oracle White Paper* (2005).

[51] Nir Shavit and Dan Touitou. 1997. Elimination Trees and the Construction of Pools and Stacks. *Theory of Computing Systems* 30, 6 (1997), 645–670. https://doi.org/10.1007/S002240000072

[52] Lefteris Sidirourgos and Martin L. Kersten. 2013. Column Imprints: A Secondary Index Structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–904. http://dl.acm.org/citation.cfm?id=2463676.2465306

[53] Avi Silberschatz, Henry F Korth, and S Sudarshan. 2020. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company. https://www.db-book.com/

[54] TPC. 2021. TPC-H benchmark. *http://www.tpc.org/tpch/* (2021).

[55] Josh Triplett, Paul E McKenney, and Jonathan Walpole. 2011. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/usenixatc11/resizable-scalable-concurrent-hash-tables-relativistic-programming

[56] Junchang Wang and Manos Athanassoulis. 2024. CUBIT: Concurrent Updatable Bitmap Indexing (Extended Version). *CoRR* 2410.16929 (2024). https://arxiv.org/abs/2410.16929

[57] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 473–488. https://doi.org/10.1145/3183713.3196895

[58] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *Proceedings of the VLDB Endowment* 2, 1 (2009), 385–394. http://www.vldb.org/pvldb/2/vldb09-327.pdf

[59] Kesheng Wu, S Ahern, E W Bethel, J Chen, H Childs, E Cormier-Michel, C Geddes, J Gu, H Hagen, B Hamann, W Koegler, J Lauret, J Meredith, P Messmer, Ekow J. Otoo, V Perevoztchikov, A Poskanzer, O Rübel, Arie Shoshani, A Sim, Kurt Stockinger, G Weber, and W-M Zhang. 2009. FastBit: interactively searching massive data. *Journal of Physics: Conference Series* 180, 1 (2009), 012053. https://doi.org/10.1088/1742-6596/180/1/012053

[60] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2006. Optimizing Bitmap Indices with Efficient Compression. *ACM Transactions on Database Systems (TODS)* 31, 1 (2006), 1–38. https://doi.org/10.1145/1132863.1132864

[61] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792. https://doi.org/10.14778/3067421.3067427

[62] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 209–220. https://doi.org/10.14778/2735508.2735511