RABIT: Efficient Range Queries with Bitmap Indexing

JUNCHANG WANG, Nanjing University of Posts and Telecommunications, China
FU XIAO, Nanjing University of Posts and Telecommunications, China
MANOS ATHANASSOULIS, Boston University, USA

Range queries (RQ) are crucial for analytical workloads, with indexing support being essential to minimize storage accesses. However, indexing support for RQ faces several challenges. Existing tree-based indexes have suboptimal RQ performance and memory consumption when long-running RQs and short-lived updates coexist. Bitmap indexes show promise in overcoming these challenges because of their small size and their succinct and readily available query result; however, they have inherent limitations: they primarily target read-only, low-cardinality attributes.

In this paper, we propose **Ra**nge Queries with **Bit**map Indexing (RABIT), a solution that addresses these shortcomings. Our design relies on three principles. First, we propose Group Encoding (GE), a novel encoding scheme that provides fast RQs and real-time updates while maintaining high compressibility. Second, we propose an efficient bitvector merging mechanism for GE. Depending on the bit density of each bitvector, we merge it in either its compressed or decompressed form, leveraging SIMD instructions when beneficial. Third, we propose a multi-layer update framework that enables lightweight multi-versioning and native index-only scans, while retaining single-versioned bitvectors, significantly reducing memory usage. Putting everything together, RABIT provides efficient point and range queries on attributes with any cardinality in tables ranging from read-only to frequently updated, unlocking the use of bitmap indexing as a general-purpose secondary index. We demonstrate that RABIT accelerates key DBMS operators (Scan, Join, and Aggregation), achieving substantial performance gains. In a row-store DBMS under HTAP workloads, RABIT offers up to 2.2× faster RQs, 530× faster updates, and 118× smaller footprint than tree indexes. In columnar DuckDB, RABIT accelerates TPC-H queries by up to 14.8×.

CCS Concepts: • **Information systems** → **Unidimensional range search**; **Point lookups**; **Data structures**; *Data management systems.*

Additional Key Words and Phrases: Bitmap Indexes, Updatable Bitmap Indexes, Range Queries, Bitmap Encoding, Secondary Indexes

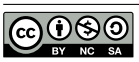
ACM Reference Format:
Junchang Wang, Fu Xiao, and Manos Athanassoulis. 2025. RABIT: Efficient Range Queries with Bitmap Indexing. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 354 (December 2025), 28 pages. https://doi.org/10.1145/3769819

1 Introduction

Range Queries (RQs) are essential for analytical queries in DBMSs [37, 52, 70]. For example, over 60% of transactions in HTAP benchmarks [11, 53] and TPC-H benchmark [57] involve RQs. RQs can be executed using either a full table scan or an index scan [28]. The latter is preferable because it minimizes storage accesses [20] and enables index-only scans [51] on modern DBMSs.

Tree Indexes. Traditional synchronization for tree indexes is either pessimistic, using latches to serialize RQs and updates [36], or optimistic, allowing latch-free RQs with post-validation and

Authors' Contact Information: Junchang Wang, Nanjing University of Posts and Telecommunications, China, wangjc@njupt.edu.cn; Fu Xiao, Nanjing University of Posts and Telecommunications, China, xiaof@njupt.edu.cn; Manos Athanassoulis, Boston University, USA, mathan@bu.edu.



ACM 2836-6573/2025/12-ART354
https://doi.org/10.1145/3769819

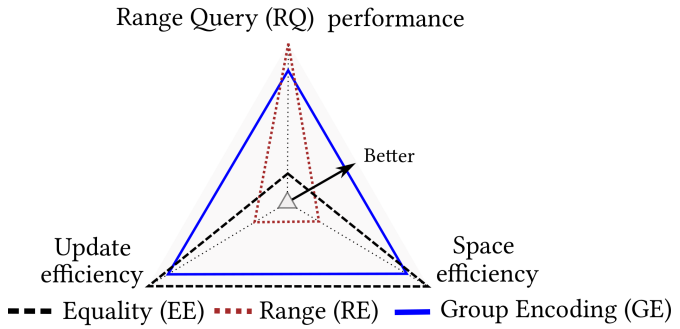


Fig. 1. RABIT introduces a novel encoding scheme, *Group Encoding (GE)*, that enables fast range queries, real-time updates, and significantly reduced memory usage. Combined with efficient bitvector merging and update mechanisms, RABIT is the first bitmap index suited for RQs on frequently updated, high-cardinality attributes in HTAP DBMSs.

possible restarts on conflicts [33]. Both approaches incur high synchronization overhead in HTAP DBMSs, which must support atomic long-running RQs along with short-lived transactional updates [29]. To improve parallelism, recent research proposed MV tree indexes [41, 48, 61, 68], which face key limitations: (1) MV trees often adopt fine-grained versioning (e.g., per node or path), leading to high storage overhead. (2) MV trees commonly introduce indirections (e.g., through versioned nodes), increasing cache misses. Eliminating these indirections remains an open challenge [61, 68]. (3) Queries typically return ID lists, which must often be converted into bitvectors for efficient downstream processing [48, 53].

Bitmap Indexing. In this paper, we propose bitmap-based secondary indexes for RQs, which fundamentally differ from traditional tree-based indexes and offer the potential to overcome their inherent limitations. A bitmap index targets an attribute A with c distinct values and associates each value with an array of n bits, denoted *bitvector*, where each bit corresponds to a tuple in the dataset. Conceptually, these bitvectors form a $c \times n$ *bit matrix*, where the i^{th} bits of all bitvectors collectively encode the i^{th} entry of A using various *encoding schemes*. In the basic *equality encoding (EE)* scheme [8], the i^{th} bit of the corresponding bitvector is set to 1, while all other bits in the same row are set to 0. In the more advanced *range encoding (RE)* scheme, which offers optimal RQ performance [8, 63], the i^{th} bits of the bitvectors corresponding to values smaller (or greater) than the value of the i^{th} entry are set to 1. This allows to quickly find all entries that are below a specific value simply by accessing a single bitvector. In addition to having inherent advantages – including being space efficient [58], clustered [51], and composition-friendly for `WHERE` clauses with multi-column predicates [47] – bitmap indexes increasingly benefit from modern hardware advancements (e.g., wider SIMD instructions and larger CPU caches), making them a promising candidate for RQs on increasingly large datasets.

Challenges. Despite their advantages, bitmap indexes have traditionally been used for read-only attributes with a small number of distinct values (*column cardinality*). For instance, Oracle database manuals suggest that "[DBAs] should keep in mind that bitmap indexes are usually easier to destroy and re-create than to maintain" [46], and Sybase IQ's bitmap indexes are "recommended for up to 1,000 distinct values" [56]. These limitations stem from the inherent constraints of the two dominant bitmap encoding schemes (EE and RE) when applied to RQs, particularly in HTAP workloads. EE provides high compressibility and supports efficient updates [3, 58], but suffers from low RQ performance due to the cost of merging many compressed bitvectors for high-cardinality attributes. RE offers strong RQ performance but introduces significantly more 1s, reducing compressibility and increasing storage update costs severely (§3). A few DBMSs like Oracle have partially addressed this

by enabling bitmap indexes for medium-cardinality attributes [47], but their proprietary designs leave these challenges unresolved for the broader database community.

Our Approach. In this paper, we propose RABIT, a novel bitmap indexing design that addresses the core challenges of supporting RQs under HTAP workloads. As illustrated in Figure 1, RABIT introduces a novel encoding scheme GE. Combined with efficient merging and update mechanisms for GE, RABIT supports fast, wait-free RQs and real-time updates with minimum memory overhead on attributes with any cardinality in tables ranging from read-only to frequently updated. Our design introduces the following key components.

Group Encoding (GE). At the core of RABIT is a new encoding scheme, named *Group Encoding (GE)*, designed to replace RE for efficient RQs on frequently updated, high-cardinality attributes (§4.1). GE eliminates RE's encoding redundancy (i.e., excessive 1s per row) via a more compact design: each distinct value is associated with a *point bitvector*, while each group of contiguous values shares a *cumulative bitvector* that stores the logical sum (bitwise OR) of all point bitvectors in the group. This design offers three key advantages. (1) Point bitvectors are highly compressible, making GE as space-efficient as EE and orders of magnitude smaller than RE. (2) Each *Update*, *Delete*, and *Insert* operation (hereafter, *UDI*) flips at most four bits, matching EE's update efficiency and outperforming RE by orders of magnitude. (3) RQs sum a few cumulative bitvectors and point bitvectors from boundary groups, achieving RE-like speed with far fewer ORs than EE.

Efficient Bitvector Merging. We propose a novel bitvector merging mechanism tailored for GE (§4.2). During query execution, an *intermediate bitvector* (*IB*) is maintained in decompressed form, with its *bit density* (the proportion of 1s) increasing as more operand bitvectors are ORed during an RQ. When the bit density of an operand bitvector B falls below a predefined threshold, which is common for point bitvectors on high-cardinality attributes, B remains compressed during merging. This allows efficient skipping over most of the decompressed *IB* and fast location and bit-flipping of the corresponding bits. Conversely, when B has higher bit density, typically with cumulative bitvectors, we decompress B before merging it with *IB*, enabling efficient use of SIMD instructions. We further extend state-of-the-art bitvector segmentation [3, 14, 35, 58] with a group-based design that performs logical operations at the segment group level, significantly reducing capacity cache misses [24]. As a result, our approach efficiently merges thousands of GE bitvectors.

Multi-Layer Update Architecture. RABIT adopts a multi-layer, out-of-place update architecture (§4.4). The first layer, a *per-RABIT-instance log*, compactly records UDI operations with versioning metadata, enabling UDIs to be applied atomically by appending to the log. Logged updates are periodically propagated to the second layer, *update buffers* associated with bitvector segments. When full, an update buffer is flushed into the corresponding bitvector. This architecture offers three key benefits. (1) RQs run independently of concurrent UDIs and can retrieve snapshots on demand, enabling index-only scans. (2) UDIs are lightweight, and a single background thread is sufficient to keep bitvectors up to date. (3) The majority of RABIT (bitvectors) remains single-versioned, avoiding the memory overhead typically associated with multi-versioning and making RABIT orders of magnitude more space-efficient than MV tree indexes [48, 61].

RABIT's Impact on Practical Systems. RABIT is the first bitmap index that supports efficient point and range queries, even on frequently updated, high-cardinality attributes. As a result, RABIT benefits HTAP systems, significantly accelerating their analytical queries. We demonstrate this by integrating it into DuckDB (§5.4) and an in-house DBMS with key components from PostgreSQL (§5.5), using an HTAP workload. Our evaluation shows that RABIT not only reduces columns (in column-stores) and pages (in row-stores) read from the storage for RQs, but also provides sufficient metadata for join and aggregation operations, eliminating the need to build intermediate data structures such as hash tables for joins.

Contributions. In summary, we make the following contributions.

- We propose *group encoding (GE)*, the first bitmap index encoding scheme for frequently updated, high-cardinality attributes. We also provide cost models to guide static parameter tuning for optimal RQ performance and space efficiency.
- We propose the first public mechanism for efficiently merging GE bitvectors across both low- and high-cardinality attributes.
- We propose a lightweight MV and update architecture for bitmap indexes, enabling wait-free RQs while maintaining a compact size through single-versioned bitvectors.
- These contributions collectively form RABIT, a bitmap indexing solution that offers efficient RQs. On synthetic workloads with varying UDI ratios and query ranges (§5.1), RABIT achieves 14×–52× (37×–118×) higher throughput than traditional EE (RE).
- We integrate RABIT into an in-house row-store DBMS. On HTAP workloads, RABIT outperforms state-of-the-art tree indexes (B^+-Tree, Bw-Tree, ART, and P-Tree) with 1.6×–2.2× faster RQs, 130×–530× faster updates, and 12.4×–118× better space efficiency (§5.4).
- We integrate RABIT into the columnar DuckDB to accelerate analytical queries. In evaluation, the RABIT-powered query engine outperforms DuckDB by 1.4×–14.8× on 12 out of 22 TPC-H queries with performance choke points (§5.5).

2 Background on Bitmap Indexes

We now provide the necessary background on bitmap indexes, leaving the challenges of using them for RQs to the next section.

2.1 Bitmap Index and Encoding Schemes Basics

Bitmap Indexes. The first binary-encoding-based indexing structure [63] proposes assigning a bit array for each distinct value of an indexed attribute A, with each bit corresponding to a tuple. Subsequent bitmap index designs [43, 44, 64] highlight their unique organization in the form of a *bit-matrix*, where each row-wise bitvector encodes the corresponding entry in A, with the matrix physically stored in a columnar manner. In its basic form, called *equality encoding (EE)* [7], a bitmap index associates one bitvector with each distinct value of A, resulting in a total of C bitvectors, where C is the *cardinality* of the indexed attribute. 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 2a illustrates the projection on an attribute A with cardinality = 10, and Figure 2b the corresponding EE bitmap index. Since each bitvector contains a few 1s, the bitvectors are highly compressible. EE bitmap indexes are well-suited for point queries of the form "$A = x$" or "$A \neq x$". However, they require bitvector merging for RQs (e.g., "$A \leq x$" and "$x_1 \leq A \leq x_2$"), with a cost proportional to the number of bitvectors involved. Our evaluation shows that as cardinality increases, RQ performance using EE bitmap indexes degrades sharply (§5.1).

To address this issue, advanced encoding schemes like *range encoding (RE)* and *interval encoding (IE)* have been proposed [7, 8]. Both RE and IE encode each entry by using more bits to facilitate RQs. In this paper, we use RE as a representative example of these encodings. In RE, the bits of the bitvectors of values less than or equal to (or greater than, depending on the encoding formats) a given value *val* are set to 1s, as illustrated in Figure 2c. RE bitmap indexes offer excellent RQ performance because each query can be answered by reading only one or two bitvectors. However, this comes at the cost of reduced compressibility: the higher density of 1s makes the bitvectors difficult to compress, resulting in RE indexes multiple times larger than the base data [38]. Additionally, updates become significantly more expensive because each update must modify, on

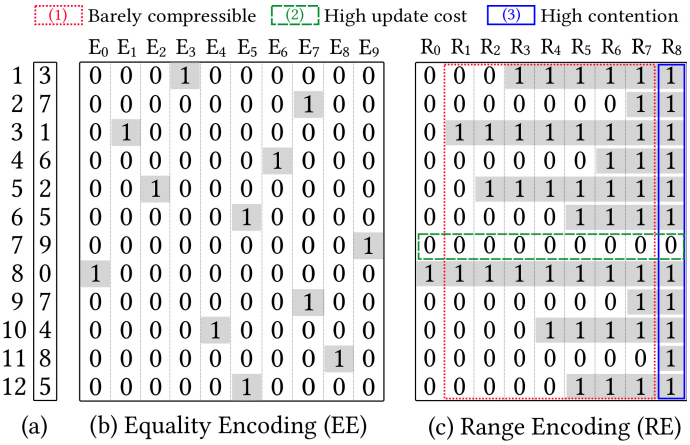


Fig. 2. (a) Attribute A with cardinality $C = 10$, (b) the corresponding EE bitmap index, and (c) the corresponding RE bitmap index with a "$R_v : A \leq v$" encoding scheme. RE bitmap indexes can only be used for read-only, low-cardinality attributes because (1) the bitvectors in the middle range are barely compressible, (2) each UDI flips bits in many bitvectors, incurring high update cost, and (3) the bitvectors on the right-hand side are updated frequently, leading to high contention.

average, $C/2$ bitvectors, and hence the state-of-the-art update process is not directly applicable. We discuss these challenges in more detail in §3.

2.2 Bitvector Compression

Bitvectors are commonly compressed to remove long runs of 0/1s by using variations of Run-Length Encoding (RLE) [1, 10, 14, 19, 66]. In particular, one popular compression algorithm is Word-Aligned Hybrid (WAH) [66] that splits the original bit-string into 31-bit (or 63-bit for the 64-bit variant) words that fall into two categories: words that only contain 1s or 0s and words that are both mixed. The former is encoded by using the run-length encoding (denoted *fill words*), and the latter is recorded literally (denoted *literal words*). After compression, for each word, the most significant bit indicates the word type (either *fill* or *literal*). A literal word uses the remaining 31 bits to verbatim store the original bit-string, while a fill word uses the second to the most significant bit to indicate if the original bit-string is all 0s or 1s, and the remaining bits to keep track of the number of words of the bit-string. When few *interrupt* bits interleave with long runs (e.g., a single 1 in a long run of 0s), CONCISE [10] and PLWAH [14] improve the compression ratio of WAH, by borrowing a few bits from a fill word to record a single interrupt bit before or after this fill word, saving the corresponding literal word. Similarly, Compax [19] introduces two new word types that borrow more bits from a fill word to record two interrupt bits and save two adjacent literal words. Another approach to addressing moderate interrupt bits is to shorten the word length. For example, Variable Length Compression (VLC) [12] allows choosing arbitrary word lengths for each bitvector, each encoded as in WAH. However, choosing appropriate word lengths is difficult; misaligned lengths across VLC bitvectors can severely degrade query performance.

Roaring bitmaps [35] divide the column data into fixed-size containers. Depending on the density and distribution of set bits, each container is encoded as a position array, an uncompressed bitmap, or a run-length encoded bitmap. While Roaring offers a better trade-off between compression and query performance, it involves structure modifications during updates (i.e., switching between array and bitmap containers). We thus adopt WAH [66] in RABIT.

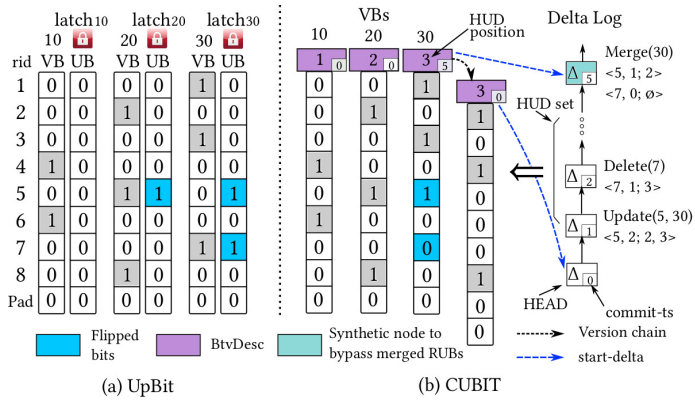


Fig. 3. State-of-the-art updatable bitmap index designs. (a) UpBit uses out-of-place updates to reduce the overheads of UDIs. (b) CUBIT further introduces multi-versioning to allow queries and UDIs to run in parallel. CUBIT only supports EE bitmap indexes, in which each UDI flips at most two bits.

2.3 Updatable Bitmap Indexes

Prior work on update-aware EE bitmap indexes [3, 6, 58] mainly leverages out-of-place updates [3] and multi-versioning techniques [58] to reduce the overhead of UDIs. However, these approaches face significant challenges when applied to RE bitmap indexes (§3).

UpBit adopts out-of-place UDIs to avoid frequent modifications to compressed bitvectors. For each *Value Bitvector* (VB), it maintains an associated *Update Bitvector* (UB) to track changes. Figure 3a illustrates the UpBit instance on an attribute with values 10, 20, and 30. Updating the 5^{th} row from value 20 to 30 involves flipping the 5^{th} bits in the UBs for both values. Similarly, *Delete(7)* flips the 7^{th} bit of the corresponding UB. Since UBs are sparse and highly compressible, bit flipping is lightweight, making UpBit's UDIs efficient. A point query *XORs* the corresponding <VB, UB> pair to reconstruct the up-to-date bitvector, and a range query *ORs* the resulting bitvectors. As UDIs accumulate (i.e., more 1s appear in UBs), UpBit periodically merges <VB, UB> pairs, generating new VBs and resetting UBs to empty [3]. UpBit uses a reader-writer latch to protect each <VB, UB> pair during concurrent queries and UDIs, but this design introduces high contention at scale [58].

CUBIT also uses out-of-place updates by representing each UDI as a delta of the row-wise bit-string in the bit matrix, termed *Horizontal Update Delta (HUD)*. Each HUD is compactly stored as a list of positions, *<row_id, n_ones; p1, p2, ...>*, where n_ones is the number of 1s in the HUD, followed by their positions. UDIs commit by appending their HUDs to the end of a per-CUBIT-instance log, called *Delta Log*. To facilitate concurrent queries and UDIs, each HUD includes a timestamp indicating the commit time of the update. Figure 3b illustrates CUBIT starting from the same bitmap index as in Figure 3a. Changing the 5^{th} tuple to 30 would flip bits in the VBs of values 20 and 30. Instead, CUBIT logs this update as a delta in the HUD <5,2;2,3>, indicating that the 5^{th} bits in the 2^{nd} and 3^{rd} VBs should be flipped since timestamp 1. Similarly, deleting the 7^{th} row is recorded by appending the HUD <7,1;3> with the timestamp set to 2. A query retrieves a *start timestamp* and then traverses the Delta Log, collecting HUDs committed before it started. Applying different HUD sets to VBs yields various snapshots of the index.

As UDIs accumulate, CUBIT periodically flushes HUDs into VBs, maintaining multiple VB versions as a linked list. In each version chain, a new VB instance B has a greater commit timestamp than its predecessor instance A and is equivalent to A merged with a HUD set. Each instance contains a shortcut pointer *start-delta*. Each flush operation inserts a *synthetic HUD* at the tail of the Delta Log and sets the shortcut pointer *start-delta* of B to it, shortening future query traversal. As

an example, in Figure 3b, a new version of the VB for value 30 is created at timestamp 5 and linked into its version chain. Subsequent queries with start-ts > 5 traverses the version chain of value 30, chooses the latest VB instance with commit-ts = 5, and then traverses the Delta Log via the shortcut pointer *start-trans*, effectively skipping the HUDs already incorporated into the new VB.

3 Challenges using Range Encoding

RE-based bitmap indexes offer optimal RQ performance [8]. However, they face significant challenges for indexed attributes with updates or high cardinality, as illustrated in Figure 2c.

(1) Compression Limitation. RLE-based compression mechanisms are primarily effective for sparse bitvectors. For example, WAH is effective only when the proportion of 1s in a bitvector (denoted *bit density*) is below 5% (or above 95%) [67]. Its successors, such as CONCISE and PLWAH, extend this limitation slightly to below 10% [14]. Other hybrid methods (e.g., Roaring [35]), which combine value-list encoding and RLE techniques, do not effectively support sparse bitvectors beyond the density limitations of WAH.

In RE bitmap indexes, most bitvectors have high bit density since each column value is encoded by $C/2$ 1s, where C (the cardinality) can reach millions in real-world datasets. This produces compression-unfriendly bitvectors in the middle range of the value domain, as shown in the red-dotted box in Figure 2c. Because their size grows nearly linearly with attribute cardinality, RE indexes are impractical for high-cardinality attributes.

(2) Increased UDI Cost. Each UDI to an RE bitmap index flips bits in approximately $C/2$ bitvectors, orders of magnitude more costly than updating its EE counterpart, where each UDI flips only one or two bits uniformly across all bitvectors. For example, Figure 2c illustrates that updating the 7^{th} row to value 0 requires flipping bits in the bitvectors from R_0 to R_8. Due to this substantial overhead, existing updatable bitmap indexes [3, 6, 58] are ill-suited for RE bitmap indexes, which are thus limited to read-only datasets.

(3) Hotspot Bitvectors. RE bitmap indexes contain hotspot bitvectors since each UDI has a very high probability ($(C-1)/C$) of updating the right-most bitvector, as illustrated by the blue solid box in Figure 2c. Similarly, the second right-most bitvector has a probability of $(C-2)/C$ of being updated, and so forth. Conventional latch-based synchronization strategies, which serialize queries and UDIs, significantly increase contention at these hotspots. Under high concurrency, this results in long waiting chains where queries and UDIs wait for preceding operations to complete, dramatically deteriorating P99 query latency [58].

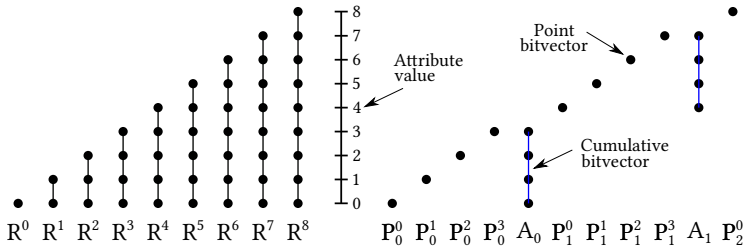
4 RABIT Design

We now discuss RABIT's semantics, design, and key building blocks.

Semantics. RABIT conforms to the standard database index specifications [51] and supports (point and range) Query, Update, Delete, and Insert operations. The Query operation returns a bitvector or an ID list. It guarantees wait-free progress for queries [27], ensuring their completion without interference from concurrent UDIs.

4.1 Group Encoding (GE)

Intuition. As discussed in §3, although RE is read-optimal (but not space-optimal) for RQs, it encounters further challenges when extending beyond read-only, low-cardinality use cases. The core issue is that each value is encoded using $\sim C/2$ bits distributed across multiple bitvectors, as illustrated in Figure 4a. We thus tailor RE by eliminating most redundant encoded information, retaining only what is essential for efficient point and range queries. This design enhances bitvector compressibility and limits the number of bitvectors each update must modify. Our new encoding



(a) Range Encoding (b) Group Encoding

Fig. 4. The set of values captured by each bitvector in (a) RE and (b) GE bitmap indexes. In RE, each value is encoded using on average $C/2$ bits, where C = cardinality. In contrast, GE requires only two bits per entry, addressing the challenges associated with RE bitmap indexes discussed in §3.

scheme, *Group Encoding (GE)*, thus significantly improves update efficiency and space utilization, while preserving the performance of RQs.

Preliminaries. Similar to prior work, we assume that the indexed attribute has a domain with cardinality C that has a unique order which is mapped w.l.o.g. to the integers $\{0, 1, \ldots C-1\}$.

Organization. Similar to EE, GE assigns a distinct *point bitvector* to each unique value, offering high compressibility for attributes with large cardinalities. Additionally, GE partitions the value domain of the indexed attribute into groups of size G, and maintains a *cumulative bitvector* that aggregates all point bitvectors in each group. Consequently, a GE bitmap index comprises C compressed point bitvectors and $\lceil C/G \rceil$ cumulative bitvectors, as illustrated in Figure 4b. Formally, we denote the j^{th} point bitvector in the i^{th} group as P_i^j, and the i^{th} cumulative bitvector as A_i. Thus, we have:

$$GE \coloneqq \{P_0^0, P_0^1, ..., P_0^{G-1}, A_0, P_1^0, ..., P_1^{G-1}, A_1, ..., P_{C/G-1}^0, ..., A_{C/G-1}\} \tag{1}$$

Additionally, a trailing group $[P_{C/G}^0, ..., A_{C/G}]$ may exist with group size less than G. Evaluating point queries with a GE bitmap index is as efficient as reading a single point bitvector, as follows:

$$\text{“}X = v\text{”} \coloneqq P_{\lfloor v/G \rfloor}^{v \bmod G} \tag{2}$$

For one-sided RQs of the form $X \leq v$, GE aggregates all cumulative bitvectors A_i with $i \leq \lfloor v/G \rfloor$. Additionally, if v crosses a group boundary and falls into the subsequent group, GE also sums the corresponding point bitvectors in the last group, as shown below:

$$\text{“}X \leq v\text{”} \coloneqq \begin{cases} \bigvee_{i=0}^{\lfloor v/G \rfloor} A_i & \text{If } (v+1) \bmod G = 0, \\ \left(\bigvee_{i=0}^{\lfloor (v-G)/G \rfloor} A_i \right) \bigvee \left(\bigvee_{i=0}^{v \bmod G} P_{\lfloor v/G \rfloor}^i \right) & \text{Otherwise.} \end{cases} \tag{3}$$

GE achieves similar performance for other predicates. Specifically, a predicate in the form of $X > v$ is evaluated by applying a bitwise logical NOT to the resulting bitvector of the predicate $X \leq v$. Similarly, $X < v$ is transformed to the predicate $X \leq (v-1)$. For two-sided queries of the form $v_1 \leq X \leq v_2$, evaluation proceeds similarly to the predicate $X \leq v_2$ (Equation 3) but begins aggregation from the first group fully contained within the range $[v_1, v_2]$. If v_1 crosses a group boundary, GE additionally sums the relevant point bitvectors from the preceding group.

Benefits. Figure 5 illustrates the GE bitmap index equivalent to the one shown in Figure 2. In addition to RE-like range and EE-like point query performance, our GE approach offers two benefits.

Space Efficiency. For high-cardinality attributes, the point bitvectors contain a small number of 1s, making them highly compressible. In this scenario, GE reduces the memory footprint by $\sim G\times$

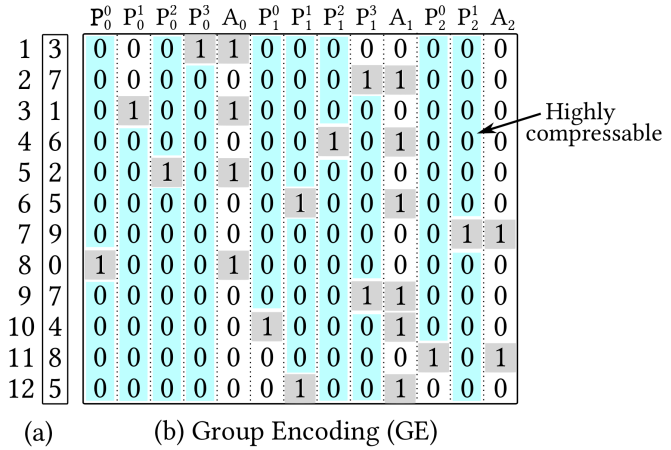


Fig. 5. The GE bitmap index equivalent to the one shown in Figure 2. By eliminating redundant encoding in RE and retaining only essential information, GE (1) substantially reduces the space overhead as its bitvectors become highly compressible, and (2) dramatically decreases the UDI overhead by minimizing the number of bit flips.

compared to its RE counterpart. For instance, when $G = 1,600$, GE reduces memory consumption by 1310×, from 1TB to ~800MB (see §5.3).

Update Friendliness. Each insert and delete operation in GE flips two bits in the underlying bit-matrix (one in the point bitvector and another in the cumulative bitvector of this group). In contrasts, RE requires flipping $\sim C/2$ bits. Similarly, each update operation in GE flips four bits, orders of magnitude fewer than the bits flipped in the RE update.

Alternative Query Evaluation. Given a predicate $X \leq v$, if the qualifying values cross group boundaries, we have two candidate evaluation methods. The first approach, illustrated in Equation 3, involves GE performing a logical OR on v mod G point bitvectors within the last group, then summing this result with the cumulative bitvectors. Alternatively, GE can aggregate the cumulative bitvector of the last group and perform logical XORs among the resulting bitvector and the remaining $G - (v \bmod G)$ point bitvectors in the group, effectively subtracting them. For attributes with high cardinality and large group sizes (e.g., thousands), when v is close to the last group's boundary, the latter approach significantly reduces the number of OR operations. We quantify these costs in §4.2.

Cost. Compared to the most query-efficient encoding, RE, the GE scheme introduces additional merge operations for RQs: (1) merging the cumulative bitvectors within the query range, and (2) merging the point bitvectors from the first and last groups when the query range spans group boundaries. We address these challenges by proposing new bitvector merging mechanisms (§4.2) and demonstrate that the associated overhead is predictable (§4.3).

4.2 Range Query (RQ) in RABIT

Executing RQs on GE requires ORing (1) a few cumulative bitvectors and (2) a set of point bitvectors when an RQ spans the boundaries of the first and last groups. To efficiently merge these bitvectors with varying bit densities, RABIT introduces a merging mechanism inspired by prior work [23, 35, 63]. We use the RQ predicate "$A \leq 8$" on the RABIT instance in Figure 5 to illustrate the mechanism.

① *Initializing Intermediate Bitvector (IB).* RABIT first creates a private copy of the cumulative bitvector of the first group fully covered by the query range as an intermediate bitvector (*IB*). In our example, the cumulative bitvector A_0 is selected. As *IB*'s bit density increases throughout the

Table 1. Notations in RABIT's performance model.

Data	N	Number of values
	C	Cardinality of the indexed attribute
	S_u	Size of uncompressed bitvector ($N/8$)
	r_{comp}	Compression ratio of point bitvector
	S_c	Size of compressed point bitvector (S_u/r_{comp})
	T_{dc}	Time to decompress point bitvector
HW	T_c	Time to access LLC (s)
	B_c	LLC bandwidth (GB/s)
	B_m	Memory bandwidth (GB/s)
	E_{or}	CPU speed for executing bitwise OR (GB/s)
RABIT	G	Group size of GE
Query	R	Query Range
	N_a	# cumulative bitvectors ORed in the query ($\lfloor R/G \rfloor$)
	N_p	# point bitvectors ORed in the query (R mod G)

query, *IB* is maintained in a decompressed form to enable efficient merging. Since *IB* is frequently used and segmented (§4.4), it is safe to assume it resides in the CPU's last-level cache.

② *ORing Cumulative Bitvectors.* RABIT then merges the cumulative bitvectors of subsequent groups fully covered by the query range into *IB*. In our example, only the cumulative bitvector A_1 is merged. Because these bitvectors have high bit density (e.g., > 2%) and low compressibility, RABIT maintains them in decompressed form to facilitate efficient merging with *IB*. This approach optimizes the merging process for modern SIMD operations by eliminating if-else branches and enhancing hardware prefetching.

③ *ORing Point Bitvectors.* When an RQ spans group boundaries, point bitvectors from the first and last groups are also merged. In our example, only P_2^0 is merged. These bitvectors contain very few 1s and are thus highly compressible. For example, each bitvector of RABIT on the *l_orderkey* attribute of the *LINEITEM* table in TPC-H contains ~4 1s and is ~16 bytes long. RABIT performs logical OR directly on these compressed point bitvectors, skipping most portions of *IB* during merging.

4.3 Group Size vs. Memory vs. Performance

For GE, using a smaller group size reduces the cost of merging point bitvectors but increases memory usage and the cost of merging cumulative bitvectors. To help users navigate these tradeoffs, we develop a memory consumption model and a performance model. Table 1 summarizes the parameters and notations used.

Memory Consumption Modeling. The size of a GE bitmap index consists of (1) point bitvectors, calculated as $C \cdot S_c$, with the same compression ratio as EE, and (2) decompressed cumulative bitvectors, calculated as $\lceil C/G \rceil \cdot S_u$. The total size of a GE bitmap index is $S(N, C, G) = C \cdot S_c + \lceil C/G \rceil \cdot S_u$. Our evaluation shows that for uniformly distributed datasets and using WAH compression with 32-bit words, the compression ratio of a bitvector can be approximated by $r_{\text{comp}} \approx C \cdot \alpha$. When the cardinality C of the indexed attribute exceeds 50, then $\alpha \approx 0.016$, which aligns with the findings reported by the WAH authors [65]. Thus, the additional memory overhead ratio compared to its EE counterpart can be expressed as:

$$Overhead_{GE}(C, G) = \frac{\lceil C/G \rceil \cdot S_c \cdot r_{\text{comp}}}{C \cdot S_c} \approx \frac{r_{\text{comp}}}{G} \approx \frac{C \cdot 0.016}{G} \tag{4}$$

When the group size G is set from 1/2 to 1/32 of C, GE consumes 3%–51% more memory than EE (the most compressed), which aligns with our evaluation (§5.1).

Performance Modeling. As discussed in §4.2, a typical query process is divided into three stages. In stage ①, data is sequentially read from the first cumulative bitvector in memory and written to *IB*, which resides in the cache. The latency of this stage is $T_C = S_u/B_m + S_u/B_c$. In stage ②, N_a cumulative bitvectors are read from memory, saturating the memory bandwidth. Each of them is merged with *IB* in the cache, followed by a write-back to *IB*. The latency of this stage is $T_A = N_a \cdot (S_u/B_m + S_u/E_{or} + S_u/B_c)$. In stage ③, N_p point bitvectors, which are highly compressed, are read from memory. They are decompressed and used to flip the corresponding bits in *IB*. The latency of this stage is $T_P = N_p \cdot (S_c/B_m + T_{\text{dc}} + T_c \cdot N/C)$.

Our evaluation shows that the time to decompress a WAH bitvector ($C > 50$) scales approximately linearly with its compression ratio, which we denote as β. This observation aligns with prior findings [23]. The overall indexing time using RABIT is calculated as follows:

$$
\begin{aligned}
T_{GE}(N, C, G, R) &= \left(\frac{S_u}{B_m} + \frac{S_u}{B_c}\right) + N_a \cdot \left(\frac{S_u}{B_m} + \frac{S_u}{E_{or}} + \frac{S_u}{B_c}\right) + N_p \cdot \left(\frac{S_c}{B_m} + T_{\text{dc}} + T_c \cdot \frac{N}{C}\right) \\
&\approx N_a \cdot \left(\frac{S_u}{B_m} + \frac{S_u}{E_{or}}\right) + N_p \cdot \left(\frac{S_u}{B_m \cdot r_{comp}} + \frac{S_u \cdot \beta}{r_{comp}} + \frac{T_c \cdot N}{C}\right) \\
&\approx \frac{R}{G} \cdot \left(\frac{S_u}{B_m} + \frac{S_u}{E_{or}}\right) + (R \bmod G) \left(\frac{S_u \cdot (1 + B_m \cdot \beta)}{B_m \cdot C \cdot \alpha} + \frac{N \cdot T_c}{C}\right)
\end{aligned}
$$

The query latency equation takes four input parameters, while the remaining variables are determined by the system environment (both hardware and software). The equation reveals several important principles of using the GE encoding scheme. First, the group size G should be smaller than most query ranges; otherwise, GE degrades into EE when the query range is smaller than G. Second, a moderate group size G reduces the number of ORed point bitvectors to an average of $G/2$, which is orders of magnitude smaller than C. However, G can not be too small, which increases the number of cumulative bitvectors that must be ORed, introducing a tradeoff. Third, the equation reveals that the cost of merging each point bitvector scales inversely with the cardinality C. Therefore, for high-cardinality attributes, it is reasonable to increase the group size G accordingly. Taking these insights into account, and informed by our memory consumption model, we set the default group size $G = C/16$ unless otherwise specified throughout the paper.

4.4 RABIT Updates

RABIT is designed to efficiently support real-time UDIs on high-cardinality attributes, focusing on three primary goals: **(G1)** Ensuring that queries are non-blocking and can execute concurrently with update and flush operations. **(G2)** Minimizing the overhead of updates. **(G3)** Addressing the memory usage limitations inherent in using MV to achieve the first two goals.

Multi-Versioning. Similar to prior research [2, 58], RABIT employs out-of-place updates and multi-versioning with timestamps to enable parallel execution of queries and UDIs (note that we adopt CUBIT's terminology for consistency). Specifically, each UDI modifies a row in the conceptual bit-matrix (Figure 5). Instead of directly modifying bitvectors, which involves the expensive decode-flip-encode process, RABIT records the delta information in a per-RABIT-instance log, called *Delta Log*, to record committed UDIs. As discussed in Section 4.1, with GE, each UDI flips at most four bits. To compactly represent update information, RABIT adopts CUBIT's *Horizontal Update Deltas (HUDs)*, which stores each update as *<row_id, n_ones; p1, p2, ...>*, where n_ones indicates the number of bits set to 1 in the HUD, followed by their positions. The Delta Log consists of instances of UDI log entries, named (ULE), each of which contains the timestamp (*commit-ts*, in the bottom-right corner

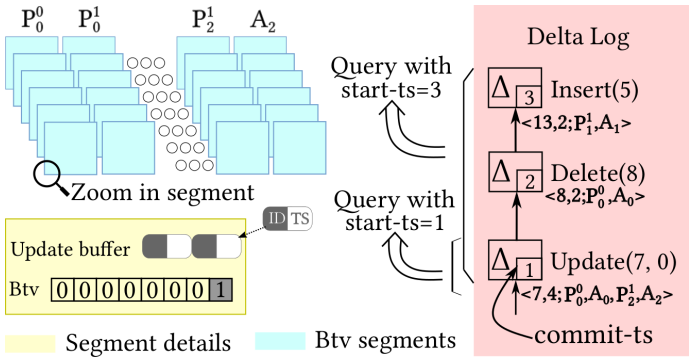


Fig. 6. RABIT incorporates out-of-place updates, bitvector segmentation, multi-versioning with timestamping, and a multi-layer update framework to enable wait-free RQs and real-time UDIs with minimal space overhead.

of each ULE). The corresponding operations and the generated HUDs are listed on the right-hand side of the ULEs. To support multi-versioning, RABIT maintains a *TIMESTAMP* counter, which is read at the start of each query and incremented by successful UDIs. UDIs commit by appending their update information in the form of ULE in chronological order, while queries traverse the Delta Log to retrieve the relevant HUD sets based on their starting timestamps.

As a concrete example, Figure 6 illustrates the basic architecture of RABIT that is initially equivalent to Figure 5b. An update operation changing the 7^{th} row to value 0 appends a ULE containing the HUD <7,4;P_0^0,A_0,P_2^1,A_2> to the Delta Log. Similarly, a delete operation removing the 8^{th} row, and an insertion of value 5 are performed by appending ULEs containing their HUD at the tail of the Delta Log. Queries with timestamp values of 1 and 3 retrieve different HUD sets, which, when incorporated into the bitvectors of RABIT, result in different snapshots of the bitmap index. The basic architecture helps RABIT to partially achieve the design goals **G1** and **G2**.

Bitvector Segmentation. In RABIT, background maintenance threads flush update information from the Delta Log into bitvectors (§4.5), reducing the need for queries to traverse a long log. To minimize flush overhead, RABIT employs segmentation that divides each bitvector into fixed-raw-size segments, with each segment independently compressed, as shown in Figure 6. Segmentation benefits flush operations by limiting the cost of generating new versions of segments, partially contributing to **G2**.

Concurrency Issue. Segmentation raises challenges in synchronizing queries with flush operations. The common approach relies on latches to serialize these operations. However, coarse-grained latching (e.g., per bitvector) results in performance degradation under high concurrency, undermining design goals **G1** and **G2**. On the other end, a fine-grained (e.g., per-segment) latching strategy risks query correctness. Consider a scenario where a flush operation latches the segments before updating, and a query operation acquires a segment's latch, reads the segment, and then releases the latch. Suppose a query $Q1$ from a thread $T1$ is reading the middle segment of a bitvector. If an update $U1$ from another thread $T2$ is flushed into a segment that $Q1$ has already read, and a subsequent update $U2$ from the same thread $T2$ is flushed into a segment that $Q1$ has not read yet, the result is that $T1$ observes the effects of $U2$ but not $U1$, even though $U1$ was successfully committed before $U2$. To address this issue, RABIT replaces latches with MVCC and a multi-layer update framework (§4.5). This design enables wait-free queries that complete in a finite number of steps, even while updates are in progress, ensuring that the RQ performance modeling in §4.3 remains valid even under heavy updates.

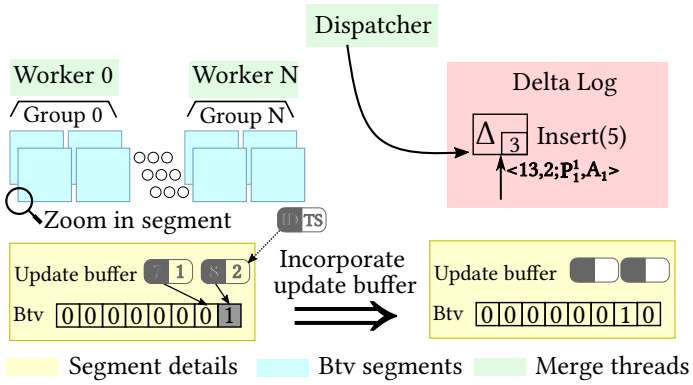


Fig. 7. A flush operation incorporates an update buffer into a bitvector by: (1) allocating a new bitvector segment, (2) copying the contents of the old version, (3) incorporating the update buffer into the new segment, (4) swapping the two segments, and (5) reclaiming the old segment using RCU.

4.5 Flush Operation

Per-Segment Update Buffer. Each segment maintains a bitvector and an associated *update buffer*, an array of <ID, timestamp> pairs that record the bits to be flipped in the bitvector and the corresponding timestamps indicating when the flips should have taken place. Both the ID and timestamp are 64-bit integers, allowing each <ID, timestamp> pair to be read and written atomically on most modern CPUs. Figure 7 zooms in the first segment of the bitvector P_0^0, where the maintenance thread has written the first two related HUDs into its update buffer.

Benefits. Per-segment update buffers batch flushing updates from the Delta Log to bitvectors, and enable multi-versioning (§4.6). That way, queries can be executed independently alongside flush operations, thus achieving design goals **G1** and **G2**. Furthermore, update buffers allow the major body of RABIT (i.e., bitvectors in segments) to be single-versioned, eliminating the need for storing versioning information and enabling a higher compression ratio. This significantly reduces RABIT's space overhead, in particular when long-running RQs and frequent updates run in parallel, making RABIT orders of magnitude smaller than MV-based tree indexes, achieving design goal **G3**.

Flush using Embarrassing Parallelism. RABIT's flush procedure involves a single *flush scheduler* and multiple *flush worker* threads, as shown in Figure 7. The scheduler periodically traverses the Delta Log from its HEAD, collects HUDs that specify rows requiring bit flips in a row-major format, transposes them to column-major format, and identifies the bitvectors to update. It then partitions the bitvectors among worker threads, an *embarrassingly parallel* task [27] with no inter-thread communication. Each worker thread (1) updates the update buffers in the corresponding segments and (2) incorporates update buffers into the bitvector segments when needed. For example, Figure 7 shows the scheduler thread fetching the two oldest HUDs <7,4;P_0^0,A_0,P_2^1,A_2> and <8,2;P_0^0,A_0> from the Delta Log and distributing the bit-flipping tasks. The flush worker thread for bitvector P_0^0 then writes <7,1> and <8,2> to the update buffer of the first segment.

Benefits. This flush process simplifies the synchronization in RABIT in three key ways. (1) The Delta Log has a single consumer (the dispatcher), making it a multi-producer-single-consumer concurrent list, simplifying synchronization between UDI and flush operations. (2) Each update buffer is updated by only one flush worker thread at a time, simplifying it to a single-writer-multiple-reader structure. (3) A flush worker thread atomically writes bit-flip information with timestamps, while queries atomically read and validate them using their start timestamps. This eliminates the need for latches on update buffers, making RABIT's queries wait-free.

Incorporating Update Buffer. When the update buffer of a segment becomes full, the flush worker thread uses an epoch-based approach to determine whether it is appropriate to incorporate the updates into the single-versioned bitvector. Specifically, it assesses (1) the earliest *start-ts* among all active queries (denoted as TS_q) by querying the DBMS or an RCU [40] manager, and (2) the latest *commit-ts* of all entries in the update buffer (denoted as TS_c). The flush operation proceeds only when $TS_q > TS_c$, ensuring that all current and future queries only access the resulting bitvector of the flush. When this condition is met, the flush worker thread allocates a new segment by incorporating the update buffer into the old segment's bitvector. Figure 7 illustrates incorporating the update buffer into the first segment of bitvector P_0^0. The old segment remains accessible and is eventually vacuumed using an RCU mechanism once all queries referencing it have completed. The correctness of our flush mechanism is rooted in the embarrassingly parallel design, where each segment is managed by a single dedicated flush worker. This eliminates the need to handle write-write conflicts. Incorporating update buffers does not interfere with concurrent queries.

In our design, a long-running query may block flush workers from incorporating update buffers. To mitigate this, RABIT adopts a resize strategy: (1) allocate a larger buffer, (2) copy the contents from the old buffer, and (3) swap the two buffers and reclaim the old one via RCU. The resizing operation has an *O(n)* time complexity, where *n* is the number of elements in the buffer, which is typically small per segment. To prevent excessively large update buffers caused by very long-running queries, RABIT enables versioned bitvector segments in rare cases. Specifically, when the DBMS detects such a query, flush threads generate dedicated segments for it. For example, for a long-running query *Q1* with timestamp *T1*, the merge thread creates a segment version for *Q1* that excludes updates that happened after *T1*, then produces a newer version for subsequent queries. This adds an extra segment version for *Q1* for each flush during its execution, which is reclaimed once *Q1* completes. If the memory used to retain segment versions per query exceeds a configurable threshold (64 MB in our evaluation), the query is terminated to ensure system stability.

4.6 RABIT Queries

Basic Procedure. A query operation Q begins by retrieving a *start-ts* from the global *TIMESTAMP*, which essentially takes a snapshot of the bitmap index. For each bitvector V to be processed, Q reads its *start-delta* pointer, which identifies the starting ULE (denoted as S) to be collected by traversing the Delta Log. Then, Q collects the delta information in two steps: (1) it traverses each entry E in the update buffers and collects IDs with *E.commit-ts* < *S.commit-ts*, and (2) it traverses the Delta Log and collects HUDs with *commit-ts* ∈ [*S.commit-ts*, *Q.start-ts*]. If the delta information is not empty, RABIT makes private copies of the affected bitvector segments, applies the delta information by flipping the corresponding bits, and then evaluates the segments.

Optimizations. RABIT supports several query optimizations. (1) Queries can perform *deduplication* on delta information from both update buffers and the Delta Log by eliminating redundant bit flips on the same row ID, avoiding unnecessary bit flips. For example, in the RABIT instance shown in Figure 6, suppose a *delete(7)* operation is successfully executed, generating a new HUD <7, 2; P_0^0, A_0> at the tail of the Delta Log. Subsequent queries on P_0^0 no longer need to flip the 7^{th} bit in the bitvector. (2) The query workload over a bitvector can be parallelized by dividing it into subqueries over multiple groups of bitvector segments and distributing them across available CPU cores. In practice, this optimization can achieve nearly linear speedup. (3) For multi-column predicates, RABIT can perform logical operations at the granularity of segment groups. Consider a `WHERE` clause "*X and Y and Z*", where *X*, *Y*, and *Z* are RQ predicates on different attributes. Instead of first computing "*X and Y*" entirely, RABIT enables the evaluation of "*X and Y and Z*" segment

by segment, keeping intermediate bitvector segments in CPU caches and significantly reducing capacity cache misses [24].

Progress Guarantee. RABIT provides the following progress guarantees for queries.

- Both point and range queries are wait-free [26], meaning they complete in a finite number of steps, even while UDIs and flush operations are in progress.
- UDIs are lock-free [26], meaning that while UDIs may block each other, deadlock never occurs, because at least one UDI always makes progress (i.e., eventually successfully submits its ULE to the Delta Log) at a given period. This ensures the system as a whole continues to make progress. UDIs never compete with queries.
- Flush operations never block queries. However, very long-running queries may delay flush operations from incorporating update buffers into bitvector segments. RABIT employs resizing to minimize this possibility, and the transaction manager is responsible for canceling runaway queries to maintain system stability.

4.7 RABIT Optimizations

Exploiting Query Statistics can bring performance benefits. In particular, selecting an optimal GE group size according to frequent query ranges enables RABIT to execute RQs by reading only a few cumulative bitvectors. Similar techniques have been explored in prior research. For example, Rotem et al. [49] proposed an optimal binning strategy using a dynamic programming approach based on query statistics. In this paper, we assume queries are not known in advance and leave exploring query statistics to future work.

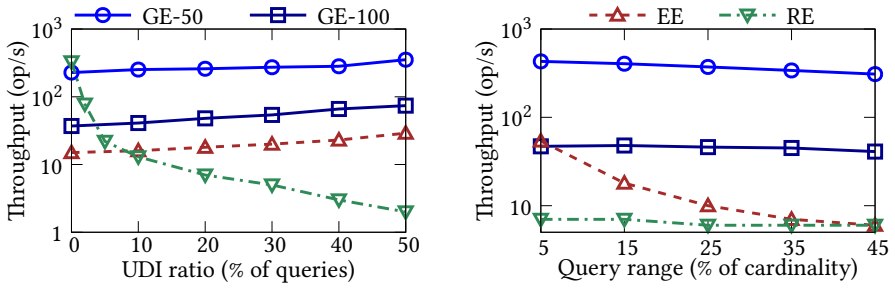
SIMD for Bitmap Indexing. The columnar bitvector structure of bitmap indexes is well-suited for SIMD instructions. RABIT leverages SIMD in three notable use cases. (1) When ORing decompressed cumulative bitvectors, RABIT uses `_mm512_or_epi32(A,B)`, where A and B are 16 packed 32-bit WAH literal words from two operand bitvectors, to compute 16 logical ORs in a single step. (2) AVX-512 masked operations enable selective computation based on a bitmask. When probing a column for aggregation in RABIT, the bitvector of matching values and the column data fit naturally into the mask and operand registers of these AVX-512 instructions. For example, in TPC-H Q1, RABIT aggregates the l_quantity column by using `ret=_maskz_compress_epi64(bitvector,column)` to extract column values matching the 1s in *bitvector* into *ret*, and then applies `reduce_add_epi64(ret)` to sum up 8 matching values in *ret* in one instruction. (3) Inspired by prior research [34], RABIT uses AVX-512 mask instructions to convert the resulting bitvector to an ID list in a batched manner.

5 Experimental Evaluation

We now show that RABIT enables efficient RQs and real-time updates with minimal memory overhead, making bitmap indexing viable for analytical queries.

Methodology. We first compare our GE to existing encoding schemes for RQs (§5.1) and analyze its sensitivity and tuning knobs (§5.2 and §5.3). Next, we integrate RABIT into a row-store DBMS and compare it against state-of-the-art tree indexes (§5.4). Finally, we integrate RABIT into a column-store DBMS and compare RABIT against its production-grade query engine (§5.5).

Testbed. All experiments are conducted on a server with two Intel Xeon 5317 CPUs (24 cores with hyper-threading, 3.0GHz, 18MB shared L3 cache). The system has 196GB DDR4 DRAM and a 1TB SSD. RABIT and the baselines are compiled using g++ 13.2 on Ubuntu 24.04 with -O3 optimization level. We use the open-source *liburcu* as the framework of safe memory reclamation and *numactl* to spawn threads to CPU cores in a round-robin fashion. For performance tests, we run each query 1,000 times and report the mean values. Standard deviations > 2% are denoted by error bars.



(a) 15% query range (b) 20% UDI ratio

Fig. 8. GE outperforms EE and RE for workloads with (a) UDIs and (b) RQs. The y-axis is in log scale.

Workloads. We use an in-house tool to generate synthetic workloads containing integer values by varying domain cardinality. We further evaluate using an industry-grade HTAP benchmark called *TPC-HC* [53] that runs TPC-C transactions on TPC-H workloads with scale factor (SF) = 10.

5.1 GE is Optimal for RQs

We evaluate RABIT with our GE encoding scheme against two baselines, CUBIT with EE and UpBit with RE (CUBIT does not support RE due to the overhead of synthetic ULEs generated by flush operations [58]), by varying two key factors: UDI ratio and query range. We use a dataset containing 100 million entries with a cardinality of 1000. GE is evaluated with three group sizes (50, 100, 200), indicated by the suffix. All bitvectors, regardless of encoding scheme, are compressed using the classic 32-bit WAH compression [66]. We spawn 16 threads, with each thread performing RQs and UDIs following a predefined ratio. We make the following observations.

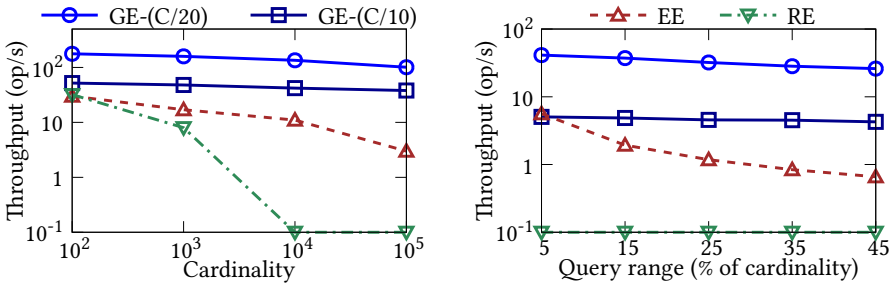
GE is Space Efficient. Table 2 shows the index sizes for different encoding schemes. EE is the most space-efficient, storing highly compressed bitvectors. RE contains many more 1s, making them less compressible, with RE being 16× larger than EE. In contrast, GE increases the size by only 8%–30% over EE.

Table 2. Bitmap index sizes (second row) and increments over the most space-efficient EE (third row). RE is 16× larger, while GE with group sizes of 200, 100, and 50 incurs overheads from 8% to 30%.

	EE	RE	GE-50	GE-100	GE-200
Size (MB)	779	12,498	1,015	908	844
Increment	–	+16×	+30%	+17%	+8%

GE is Fast with Updates. We fix the query range at 15% of the domain cardinality (150 unique values), vary the UDI ratio from 0% (read-only) to 50% (frequently updated), and measure the overall throughput. As shown in Figure 8a, when the workloads are read-only, RE provides the highest performance, as it requires only a single logical operation between two bitvectors. In contrast, EE is the slowest, since it must OR all bitvectors with the query range (150 bitvectors in this evaluation). GE outperforms EE by replacing intra-group ORs with reading a single cumulative bitvector.

The group size in GE significantly affects query performance. In this experiment, the query covers 150 consecutive values. GE-100 is the least efficient GE variant, as it must OR one cumulative bitvector along with 50 point bitvectors. Conversely, GE-50 achieves the best performance by ORing only three cumulative bitvectors, making it nearly as efficient as RE (Figure 8a).



(a) 15% query range, 20% UDI ratio (b) 1B entries, 1000 cardinality

Fig. 9. GE's throughput remains stable with varying (a) cardinality and (b) dataset size (log scale y-axis).

GE-50 and GE-100 represent the two performance extremes, with other group sizes falling in between. For example, GE-200 ORs 50 point bitvectors and then subtracts the result from the cumulative bitvector, being slightly worse than GE-100. For clarity, we omit these intermediate settings from Figure 8 but provide a detailed discussion on selecting the optimal group size in §5.3.

When UDIs are present, RE performance degrades sharply due to severe conflicts between UDIs and RQs, as shown in Figure 8a. EE, relying on CUBIT, remains unaffected by UDIs. The overall throughput with EE improves as the UDI ratio increases because UDIs incur lower workloads than RQs. GE experiences minimal impact from UDIs and consistently outperforms both baselines. With 20% UDIs, GE-100 is 2.7× (6.9×) faster, and GE-50 is 14× (37×) faster than EE (RE). With 50% UDIs, GE-50 is 118× faster than RE.

GE is Stable with Increasing Query Ranges. We fix the UDI ratio at 20% and vary the query range from 5% to 45% of the domain cardinality. As shown in Figure 8b, RE achieves the lowest throughput due to the conflicts between UDIs and RQs. When the query range is small (5%) and below the group size, GE-100 degrades into EE (see §4.3); both schemes OR 50 bitvectors, resulting in similar throughput. GE-50 is the fastest because it reads one cumulative bitvector. EE throughput drops as the query range grows due to more bitvectors ORed. In contrast, GE experiences negligible degradation. At a 25% query range, GE-50 is 18× (31×) faster than EE (RE). When the query range reaches 45%, GE-50 outperforms the baselines by over 52×.

5.2 Sensitivity Analysis

We now present a sensitivity analysis using the same experimental setup as in §5.1, varying two key parameters: domain cardinality and dataset size. Each evaluation allocates 64GB of memory for indexes. If a bitmap index exceeds this limit, a least-recently-used (LRU)-based swap mechanism evicts retired bitvectors to disk, which significantly degrades performance.

Impact of Domain Cardinality. We vary the domain cardinality from 100 to 10 million, while fixing the UDI ratio at 20% and the query range at 15% of the cardinality. For GE, the group size is set to $1/20^{\text{th}}$ and $1/10^{\text{th}}$ of the cardinality, indicated by the suffix in Figure 9a. We make the following observations. First, the query performance of RE degrades sharply and becomes prohibitively slow when cardinality exceeds 10^3 because of its increase in index size that causes frequent swapping of bitvectors in and out of main memory. Second, EE's performance also degrades with increasing cardinality, as more bitvectors must be ORed. Third, GE maintains stable performance, as its use of cumulative bitvectors introduces negligible overhead even as cardinality increases.

Impact of Data Size. Figure 9b shows the evaluation results with datasets containing 1B entries. As the dataset size increases, the relative behavior of different indexes remains consistent, and

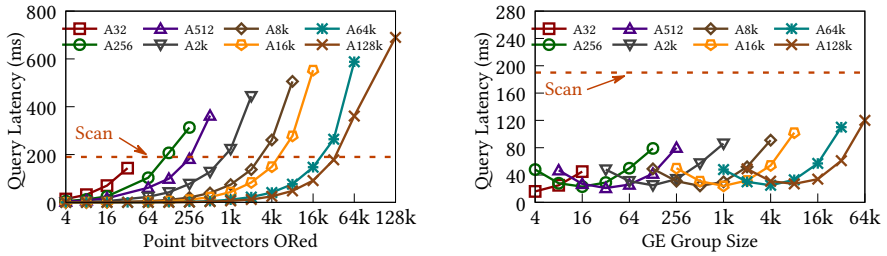


Fig. 10. (a) RQ performance using only point bitvectors degrades below table scan (one-sided predicates) as cardinality and query range grow, while (b) GE with a proper group size, remains an order of magnitude faster across all cases.

Figure 9b looks almost identical to Figure 8b. The exception is RE, which becomes prohibitively slow because the overall index size is 110GB, exceeding the memory reserved for indexes.

5.3 Tuning RABIT

We demonstrate that the static tuning of RABIT's parameters achieves consistent performance across diverse workloads.

Group Size of GE. As analyzed in §4.3, the group size G for GE is a key performance parameter that should be tuned according to constraints on storage space and query latency requirements.

Group Size vs. Query Performance. Figure 10a shows the execution time for queries involving OR operations across multiple point bitvectors (achieved by setting GE group size larger than the query range, resulting in degradation to EE), with attribute cardinalities varying from 32 to 128K (denoted by legend suffixes). For comparison, we also include the execution time for equivalent one-sided range predicates evaluated using a highly optimized columnar DBMS scan (§5.5). Our results indicate that for attributes with cardinalities exceeding 256, EE-based RQs may be more costly than a full table scan. For attributes with cardinality of 128k or greater, EE-based RQs become ~3× more expensive.

In contrast, selecting an appropriate group size G can significantly improve RQ performance. Figure 10b shows the worst-case query latency, which is defined as the time to OR half of the cumulative bitvectors ($C/2G$) along with half a group's point bitvectors ($G/2$). Our evaluation shows that an appropriate G reduces query latency for all workloads. For example, for the attribute with cardinality = 16K (A16K), setting the GE group size to 1K yields a worst-case query latency of 24 ms, which is 8× faster than the baseline scan. Furthermore, the modeling results in §4.3 indicate that (1) decreasing G reduces the number of point bitvectors ORed but increases the number of cumulative bitvectors ORed, and (2) the optimal G depends heavily on the specific hardware configuration. As shown in Figure 10b, on our evaluation platform, the best performance is consistently achieved when $G \approx C/16$ across all workloads, allowing for static tuning of GE. Our artifact provides scripts to quickly determine the optimal G for specific servers.

Group Size vs. Space Overhead. We also evaluate the space overhead of GE compared to the most space-efficient EE counterparts, using the same workloads and setting the group size G to around $C/16$. Our results show that, irrespective of cardinality, the space overhead of GE inversely correlates with G, as illustrated in Figure 11. For example, for the workload with cardinality 16K, increasing G from 512 to 2K reduces the memory overhead from 43% to 3%, and the best-performing configuration results in a space overhead of 25%, which aligns with the modeling results in §4.3.

Number of Maintenance Threads. It is critical for RABIT's performance to determine the optimal number of worker threads that flush updates into bitvectors and perform garbage collection. Our

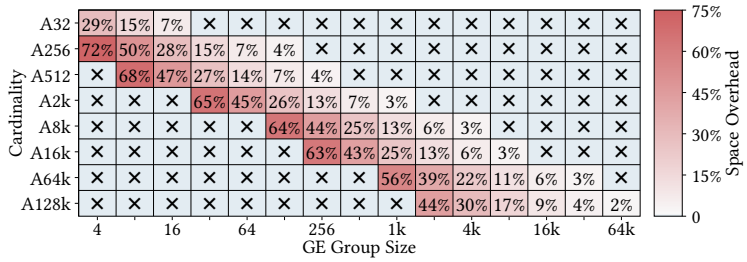


Fig. 11. GE's space overhead over the most space-efficient EE grows nearly linearly with the cardinality to group size ratio, aligning with the model in §4.3. With the group size yielding the best query performance, the overhead is ~26%.

evaluation shows that in typical use cases (UDI ratio < 90% and query range < 50%), one worker thread is sufficient. This is mainly because our GE encoding scheme limits the number of bitvectors that need to be updated to four, and one worker thread can handle flushing without affecting concurrent RQs. In contrast, CUBIT maintains a group of maintenance threads to handle flushing, the size of which increases linearly with concurrent queries and UDIs.

Flush Granularity. In each iteration, the *flush scheduler* traverses the Delta Log and attempts to process up to K ULE entries, where K is a predefined threshold and set to 32 by default. Because RABIT's flush process is lightweight, as it typically involves only writing the update buffers of the affected segments, the scheduler consistently keeps pace with incoming updates. In our evaluation, even under workloads with 50% updates (Figure 8a), the Delta Log contains only a small number of entries, and the scheduler traverses the whole log.

5.4 RABIT Benefits Row-Based DBMSs

We demonstrate that RABIT outperforms tree indexes by integrating it into an in-house row-based DBMS built on top of DBx1000, with several critical components adapted from PostgreSQL.

Prototype DBMS. Similar to PostgreSQL, our prototype DBMS organizes base data into fixed-size pages (8KB) and accesses tuples via tuple identifiers of the form *(Block, Offset)*. The system implements MVCC using timestamps and incorporates a transaction manager that assigns unique timestamps to concurrent operations. Additionally, we leverage *liburcu* to efficiently track transaction states with minimal overhead, as the library automatically manages transaction states upon thread registration and deregistration.

Algorithms. We implement the following baseline tree indexes based on their open-source implementations: (1) a classic B$^+$-Tree with a fixed fanout of 32; (2) a latch-free Bw-Tree [60] with a maximum inner node size of 256 bytes; (3) ART [32] with a maximum node fanout of 256; and (4) P-Tree [53], which implements multi-versioning by creating snapshots of the tree through path copying for each update. We optimize all algorithms within our testbed (e.g., by replacing linked lists with arrays to handle duplicate keys) and carefully tune parameters.

RABIT. We construct a RABIT instance for each attribute appearing in query predicates. For example, we build three RABIT indexes on attributes *l_shipdate*, *l_discount*, and *l_quantity*, with group sizes set, respectively, to 3 months, attribute cardinality (effectively reducing to EE encoding), and 5, accommodating the requirements of most queries, including TPC-H Q6.

Tree Indexes. Other single-versioned tree indexes employ multi-column indexing across these attributes, which is the best case for the baselines. Queries utilizing these indexes follow two stages: (1) retrieving tuple IDs via the index, and (2) fetching these tuples to compute the final query result. RABIT, Bw-Tree, and P-Tree provide latch-free queries, whereas other single-versioned trees are protected by optimistic lock-coupling [33]. P-Tree employs the universal constructor PAM

[54], which uses an immutable, balanced binary tree structure to facilitate a divide-and-conquer parallel strategy and implement an efficient work-stealing scheduler. Moreover, P-Tree inlines table entries, allowing queries to directly return results without additional base data retrieval (the second stage). This feature significantly improves query performance at the cost of increased memory consumption (see below).

Index-Only Scans. Single-versioned tree indexes lack transactional visibility, requiring the DBMS to access the base table to verify the visibility of indexing results. Mitigating this requires either auxiliary structures (e.g., visibility maps in PostgreSQL [55] and version synopses in HyPer [42]) or coarse-grained index latches, both of which introduce additional storage and update overhead [53]. We thus only support the *read committed* isolation level with these tree indexes, which are fast but susceptible to *phantom reads*. In contrast, P-Tree and our RABIT provide *snapshot isolation* and natively support index-only scans through their MV design.

Table 3. The sizes of different indexes and the size increment compared to RABIT with GE. RABIT achieves multi-versioning with one-hundredth of the size of multi-versioning trees.

	RABIT	B^+-Tree	Bw-Tree	ART	P-Tree
Size (MB)	156	1,959	1,971	1,932	18,402
Increment	-	+12.6×	+12.6×	+12.4×	+118×

RABIT is Space Efficient. Table 3 presents the sizes of various indexes. Despite supporting multi-versioning, RABIT is the most space-efficient, as it stores single-versioned, highly compressed bitvectors. In contrast, single-versioned tree indexes, such as B^+-Tree, Bw-Tree, and ART, are generally 12× larger due to the overhead of maintaining pointers and metadata. P-Tree, which is multi-versioned, must track multiple versions of tree paths. Including inlined table entries, P-Tree is 118× larger than RABIT.

RABIT is Fast for RQs. Except for P-Tree, RABIT delivers the fastest RQ performance. Figure 12a presents the query latency of TPC-H Q6 (selectivity ≈ 2%), which selects 365 consecutive days out of 2,526 and 24 consecutive quantity values out of 50. The RABIT-powered DBMS performs logical AND/OR operations on ∼ 11 bitvectors (4 out of 28 cumulative bitvectors for *l_shipdate*, 3 out of 11 for *l_discount*, and four cumulative bitvectors along with a few point bitvectors for *l_quantity*). This approach is faster than tree indexing because (1) RABIT has a smaller memory footprint, leading to fewer cache misses compared to tree indexes, and (2) bitwise logical operations are more efficient on modern hardware than pointer chasing in tree indexing. Furthermore, the outcome of RABIT (i.e., the ID list) is clustered, enabling the RABIT-powered DBMS to access base data in a single, sequential pass. Overall, RQ performance of RABIT is 1.6×–2.2× faster than the baselines. P-Tree's inlined table entries enable faster RQ performance but incur significantly higher space consumption and update overhead.

We further stress-test indexes by enlarging the query range. Figure 12b presents the results of a variant of Q6 that selects four out of seven years (selectivity ≈ 8%). The query latency of tree indexes increases almost linearly with selectivity. In contrast, the increased overhead of RABIT mainly comes from accessing more base data rather than indexing, making it as efficient as P-Tree.

RABIT Enables Real-Time Updates. RABIT outperforms tree indexes in update performance, with each UDI taking only 0.1ms, 130×–520× faster than updating a tree index, as shown in Figure 12a. This is because RABIT appends the update information to its Delta Log. In contrast, tree indexes must traverse the tree to locate the target node, update it, and potentially perform structural adjustments. Furthermore, retries are necessary when RQs and UDIs conflict. As the query range increases, the UDI latency of tree indexes also grows due to increased conflicts, whereas RABIT

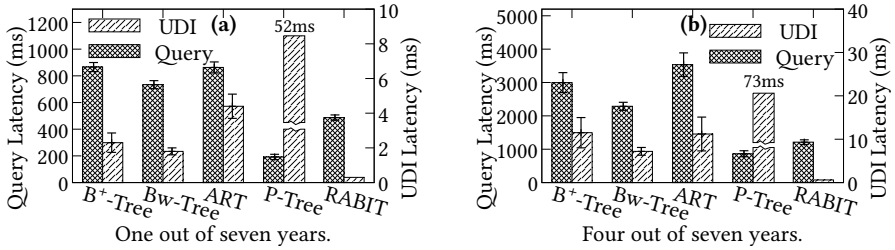


Fig. 12. RABIT outperforms all tree indexes across a wide spectrum of RQs, from (a) small to (b) large.

maintains stable update performance, as shown in Figure 12b. P-Tree incurs the highest update cost due to its immutable data structure: each update triggers a copy-on-write process along the path from the root to the leaf node. Its update performance further degrades with larger query ranges due to increased CPU cache pressure (according to our discussion with the P-Tree authors), as shown in Figure 12b.

5.5 RABIT Benefits Column-Store DBMSs

DuckDB$^+$. We further demonstrate the benefits of RABIT for columnar DBMSs by integrating it into DuckDB (version 1.0). In our evaluation, we observed that DuckDB's query optimizer misses some opportunities for better query plans for TPC-H. For example, Q10 performs a group-by on attributes *c_custkey*, *c_name*, and *c_acctbal*. Since *c_custkey* is the primary key and determines the other attributes in the same table, an optimized plan can defer loading the other attributes until after the joins, thereby reducing materialization overhead. However, DuckDB's optimizer has not yet incorporated this optimization. For a fair comparison, we addressed these known performance choke points in DuckDB [4, 16] to demonstrate that even if these bottlenecks are resolved, RABIT-powered DBMS still outperforms state-of-the-art column-store systems. We denote this optimized implementation as DuckDB$^+$ and use it as our baseline.

CUBIT without Binning. Like prior updatable bitmap indexes, CUBIT is primarily designed to support point queries (PQs) of the form "$A = x$" and its performance degrades when handling RQs, since it has to bitwise-OR multiple bitvectors. When query templates are known beforehand, CUBIT addresses this performance issue by employing a binning strategy that maps RQs to PQs on CUBIT instances tailored for each query. For example, in TPC-H Q6 with a predicate "$l\_quantity < QTY$", where *QTY* is randomly selected as 24 or 25, the corresponding CUBIT instance maintains only three bitvectors, respectively for values less than, equal to, and greater than 24, eliminating the need to merge bitvectors when processing Q6. However, such tailored binning introduces significant space and maintenance overhead, making CUBIT impractical for real-world queries with RQs. For example, in Q19, where *QTY* is randomly selected within other ranges, CUBIT must maintain another instance for the same attribute. Moreover, when *QTY* cannot be statically determined (e.g., in Q17), binning becomes impractical, forcing the system to merge bitvectors for queries. We refer to the CUBIT without the binning technique as *CUBIT-no-Binning* and use it as a baseline.

RABIT Indexing. For the first time, RABIT enables bitmap indexing on frequently updated, high-cardinality attributes. We create RABIT instances both for low-cardinality attributes such as *l_quantity* (cardinality=50) and *l_shipdate* (cardinality=2,526), and for the high-cardinality *l_orderkey* (cardinality=15M when SF=10) in the LINEITEM table with updates, introducing a novel form of secondary indexing, which we illustrate using Q1, Q5, and Q6.

Fast Scan with RABIT. Scan is widely used for RQs. For instance, scan operations account for 95% of the execution time of Q6. In addition to the optimized scan in DuckDB$^+$, we compare RABIT

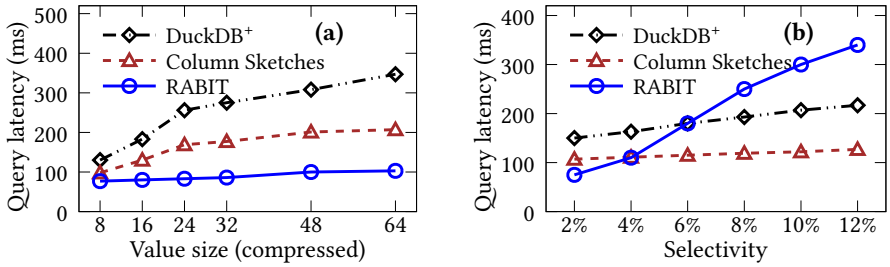


Fig. 13. For single-column RQ predicates, (a) RABIT-based queries outperform Column Sketches and highly-optimized scans, and (b) this trend continues for selectivity < 4%.

against sparse indexes [25, 50], which have a lower memory footprint and provide efficient RQs. To that end, we implement Column Sketches (1-byte sketch size) and the associated optimizations [25] in DuckDB$^+$, where sketches are evaluated using AVX-512 instructions before accessing each chunk of base data (2,048 values).

Single-Column Predicate. We first evaluate one-sided RQ predicates "$A < x$" on a uniformly distributed attribute in the LINEITEM table (SF = 10), varying the compressed storage size of each value (by adjusting the cardinality of attribute A), as it is a key factor affecting scan performance. The query selectivity is set to 2%, which is typical for analytical queries like Q6. As shown in Figure 13a, as value size increases, the query latency of DuckDB$^+$ increases sharply due to the larger amount of data scanned. The query latency for Column Sketches also increases due to the higher number of candidate checks resulting from the increased cardinality. RABIT's performance decreases slightly because of the increased TLB misses. For attributes with 64-bit values, RABIT is 2× and 3.4× faster than Column Sketches and DuckDB$^+$, respectively.

We then vary query selectivity by adjusting x. The value size is set to 13 bits. As shown in Figure 13b, for moderate selectivity (e.g., 2%), RABIT outperforms the alternatives. As selectivity increases, RABIT, used as a secondary index, experiences a sharp performance decline due to the increased probes to base storage, aligning with earlier findings [28]. In Column Sketches, the increased overhead arises from the decreased efficiency in filtering out chunks. Unexpectedly, the scan performance of DuckDB$^+$ also deteriorates, primarily due to the overhead of maintaining the growing ID list as selectivity increases. Overall, RABIT outperforms Column Sketches and scans for up to 4% and 6% selectivity, respectively.

Multi-Column Predicate. Real-world queries often involve multi-column predicates. For instance, to execute Q6, DuckDB$^+$ performs full table scan on four columns: *l_extendedprice*, *l_shipdate*, *l_discount*, and *l_quantity*. Since Column Sketches do not natively support multi-column sketches on numerical attributes (Appendix C of [25]), we leverage the best approach in our evaluation. Specifically, we evaluate the sketches of *l_shipdate* and *l_quantity*, perform logical AND operations between the results, and then fetch the other two columns, all using AVX-512 instructions. In contrast, the RABIT-powered Scan operator performs logical OR/AND operations among RABIT instances on the attributes involved in the `WHERE` clauses. For Q6, this stage involves only 11 logical operations, mainly between GE cumulative bitvectors (see §5.4 for details).

Our evaluation shows that the Q6 query latency with RABIT-powered scan is 204ms (72ms for indexing), 2.4× faster than the optimized scan in DuckDB$^+$, because RABIT probes only two columns, effectively halving the data read from storage. Compared to Column Sketches, which incur significant overhead when composing and merging the results from multiple sketches, RABIT-powered scan eliminates these costly stages and achieves a 4.2× query speedup. As most TPC-H queries involve multi-column predicates, we omit Column Sketches from the remaining evaluations.

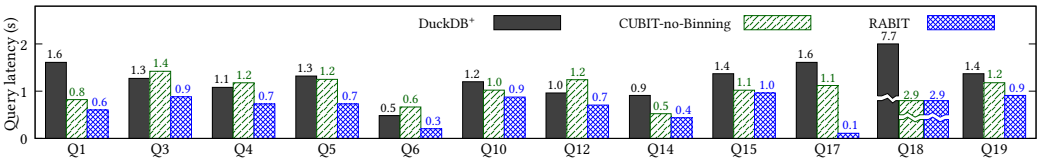


Fig. 14. RABIT enables bitmap indexing for the LINEITEM and ORDERS table with large-cardinality attributes. Our RABIT-powered query engine outperforms or matches alternatives across all TPC-H queries.

Fast Join with RABIT. Join is the most costly relational operator and often involves RQs on input tables. Take Q5 as an example, DuckDB$^+$(1) performs a full table scan on the LINEITEM table and an RQ on ORDERS, and (2) joins the two tables on the *orderkey* attribute using a hash table. Our evaluation shows that, when SF = 10, DuckDB$^+$ spends 340ms and 750ms for these two stages.

In contrast, we maintain RABIT instances on the *orderkey* attributes of both tables. The RABIT-powered join operator performs the RQ on the ORDERS table by ORing four cumulative bitvectors, eliminating the need to read the corresponding column. It then retrieves the matching bitvectors from the instance for the LINEITEM table and performs logical ORs between them. When SF = 10, this process involves ~357K logical ORs, which are efficiently handled by RABIT's merging mechanism. The efficiency stems from the fact that, for large-cardinality attributes, each bitvector contains a few 1s (~4 1s in this RABIT instance) and is highly compressible (~16 bytes/bitvector), allowing the merge mechanism to skip most portions of the bitvector (§4.2). Furthermore, when most bitvectors in a group are selected, we use the cumulative bitvector and subtract the non-matching ones (§4.1). The resulting bitvector is then used to scan the LINEITEM table and perform aggregation in one pass. Our evaluation shows that the RABIT-powered operator reduces overall query latency, making it 1.8× faster than DuckDB$^+$.

Fast Aggregation with RABIT. Aggregation occurs in all TPC-H queries and often involves RQ on input tables. Take Q1 as an example, DuckDB$^+$ (1) executes an RQ on the *l_shipdate* attribute of the LINEITEM table, and (2) groups the matching tuples using a perfect hashing by iterating over each entry, identifying its group-by category, and updating the corresponding statistic counters.

By maintaining RABIT instances on the group-by attributes, our RABIT-powered operator first calculates the positions of matching tuples of each group-by category and then processes each category by reading the matching entries in one pass. This design is more amenable to modern SIMD instructions for aggregation. Our experimental results (SF = 10) show that the RABIT-powered aggregation operator takes only 45ms for stages (1) and (2), and 315ms for stage (3). The overall query latency is 604ms, 2.7× faster than the implementation of DuckDB$^+$.

Applicability. Our evaluation shows that RABIT overall benefits analytical queries, except those that do not contain obvious performance choke points (e.g., Q2 and Q16). We demonstrate this through experiments on 12 out of 22 TPC-H queries, covering join-dominated (Q4, Q5, Q17, and Q19), aggregation-heavy (Q1 and Q18), and scan-intensive (Q6 and Q12) queries. Our RABIT-powered query engine achieves a speedup of 1.4×–14.8× compared to the native approaches in DuckDB$^+$. Except for Q18, where bitmap indexes are primarily used for point queries, RABIT is 1.3×–10.3× faster than CUBIT without binning, as shown in Figure 14.

6 Related Work

We classify recent research on indexing for RQs as follows.

Latch-Free Tree Indexes. A key challenge in indexing for RQs in HTAP DBMSs is efficient synchronization with concurrent updates. To increase parallelism, researchers have proposed various index structures, primarily tree-based, by adopting general parallel-programming constructors [13, 17, 26, 31] or transactional memory techniques [5, 59] to enable latch-free execution of RQs

and UDIs. However, when long-running RQs and updates coexist, these approaches face limitations, including cache-line bouncing due to frequent *compare-and-swap* retries [30] and scalability bottlenecks from centralized metadata management [15].

Multi-Versioning Tree Indexes. MV is a promising approach to enhance parallelism for RQs. Researchers have developed various MV tree structures by: (1) creating versioned tree nodes organized into versioned lists [61, 68], (2) maintaining versioned references to nodes created at different timestamps [30, 41], or (3) appending timestamps to keys to construct temporal indexes [48]. A recent study, P-Tree [53], utilizes the universal constructor PAM [54] on top of a balanced binary tree structure, facilitating efficient parallel execution through a divide-and-conquer strategy. The structure is immutable; each update creates a new path from the root to the leaf, allowing queries and updates to operate on separate snapshots. P-Tree inlines table tuples to achieve performance improvements, but this comes at the cost of higher memory consumption, potentially up to two orders of magnitude greater than RABIT (§5.4).

Sparse Indexes [25, 50] are much smaller than tree-based indexes, making them a promising alternative to full table scan [69]. Sparse indexes rely on lossy summaries and act as scan accelerators, whereas RABIT is lossless and supports query processing without accessing base storage. This enables RABIT to support index-only scans and multi-column predicates efficiently. Subsequent work [9, 38] reduces candidate check overhead in sparse indexes by encoding additional information into the structures. However, these structures are read-only. Another trend improves scan performance by bit-packing attribute values and leveraging SIMD instructions to perform comparisons in parallel [18, 39, 62]. In contrast, RABIT provides a more general indexing approach that supports efficient updates, multi-column predicates, and index-only scans.

Bitmap Indexes in DBMSs. Although some production DBMSs use bitmap indexes (e.g., Oracle [45] and Greenplum [21]), their adoption remains limited. For example, PostgreSQL introduced a native bitmap index in version 8.3.23, but later removed it due to its limited applicability: low-cardinality attributes under read-mostly workloads. Subsequently, PostgreSQL implemented a Bloom filter index [22] that enables rapid exclusion of non-matching tuples via bit-string signatures. However, it only supports equality queries. In contrast, RABIT overcomes the core limitations of traditional bitmap indexing, enabling broader adoption.

7 Conclusion

Bitmap indexes are effective for range queries but have traditionally been restricted to read-only, low-cardinality attributes due to their inherent limitations. We present RABIT, a bitmap indexing design that combines a novel encoding scheme, an efficient bitvector merge mechanism, and a multi-layer update framework. RABIT enables fast range queries and real-time updates with minimum memory overhead, on attributes of any cardinality in both read-only and frequently updated tables. This advancement facilitates the broad adoption of bitmap indexes in DBMSs and opens up new strategies to accelerate their key operations. Our evaluation on both row-store and column-store DBMSs shows that RABIT significantly accelerates analytical queries.

Acknowledgments

We thank the anonymous reviewers as well as Wei Xi for their constructive feedback. Manos Athanassoulis is partially supported by the National Science Foundation under Grants No. IIS-2144547 and CCF-2403012, a Facebook Faculty Research Award, and a Meta Gift. Junchang Wang and Fu Xiao are partially supported by the National Natural Science Foundation of China under Grant No. 62427809, and the Natural Science Foundation of Jiangsu Province under Grant No. BK20243053.

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] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 461–466. http://dx.doi.org/10.5441/002/edbt.2016.42

[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] 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

[5] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. *ACM SIGPLAN Notices* 45, 5 (2010), 257–268. http://dl.acm.org/citation.cfm?id=1837853.1693488

[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. http://dl.acm.org/citation.cfm?id=1270403.1271866

[7] Chee-Yong Chan and Yannis E. Ioannidis. 1998. Bitmap index design and evaluation. *ACM SIGMOD Record* 27, 2 (1998), 355–366. http://dl.acm.org/citation.cfm?id=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. http://dl.acm.org/citation.cfm?id=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. http://dx.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. http://doi.acm.org/10.1145/1988842.1988850

[12] Fabian Corrales, David Chiu, and Jason Sawin. 2011. Variable Length Compression for Bitmap Indices. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*. 381–395. http://dx.doi.org/10.1007/978-3-642-23091-2_32

[13] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2020. A wait-free universal construction for large objects. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 102–116. https://doi.org/10.1145/3332466.3374523

[14] 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. http://dl.acm.org/citation.cfm?id=1739041.1739071

[15] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. 2011. Why STM can be more than a research toy. *Commun. ACM* 54, 4 (2011), 70–77. https://doi.org/10.1145/1924421.1924440

[16] 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. http://www.vldb.org/pvldb/vol13/p1206-dreseler.pdf

[17] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 275–286. https://doi.org/10.1145/3323165.3323197

[18] 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. http://doi.acm.org/10.1145/2723372.2747642

[19] 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. http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/I01.pdf

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

[21] Greenplum. 2010. Online reference. *http://www.greenplum.com/* (2010).

[22] PostgreSQL Global Development Group. 2016. Bloom Filter Index Access Method. https://www.postgresql.org/docs/current/bloom.html. [Online; accessed 1-Feb-2025].

[23] Gheorghi Guzun and Guadalupe Canahuate. 2016. Hybrid query optimization for hard-to-compress bit-vectors. *The VLDB Journal* 25, 3 (2016), 339–354. https://doi.org/10.1007/s00778-015-0419-9

[24] John L. Hennessy and David A. Patterson. 2012. *Computer Architecture - A Quantitative Approach, 5th Edition.* Morgan Kaufmann.

[25] 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. http://doi.acm.org/10.1145/3183713.3196911

[26] 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

[27] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The Art of Multiprocessor Programming (Second Edition)*. Morgan Kaufmann. https://doi.org/10.1016/C2011-0-06993-4

[28] 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. http://dl.acm.org/citation.cfm?doid=3035918.3064049

[29] Jong-Bin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2021. Rethink the Scan in MVCC Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 938–950. https://doi.org/10.1145/3448016.3452783

[30] 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

[31] Alex Kogan and Erez Petrank. 2012. A methodology for creating fast wait-free data structures. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 141–150. http://doi.acm.org/10.1145/2145816.2145835

[32] 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. http://dl.acm.org/citation.cfm?id=2511193

[33] 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. http://doi.acm.org/10.1145/2933349.2933352

[34] 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].

[35] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François 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

[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. http://dl.acm.org/citation.cfm?id=2510649.2511251

[37] Guoliang Li and Chao Zhang. 2022. HTAP Databases: What is New and What is Next. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2483–2488. https://doi.org/10.1145/3514221.3522565

[38] 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

[39] 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. http://doi.acm.org/10.1145/2463676.2465322

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

[41] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. 2022. Bundling linked data structures for linearizable range queries. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 368–384. https://doi.org/10.1145/3503221.3508412

[42] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 677–689. http://doi.acm.org/10.1145/2723372.2749436

[43] 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=645575.658338

[44] 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

[45] Oracle. 2013. Oracle11g Database Data Warehousing Guide. https://docs.oracle.com/cd/E11882_01/server.112/e25554/indexes.htm. [Online; accessed 1-May-2024].
[46] Oracle. 2024. Oracle23ai Data Warehousing Guide. https://docs.oracle.com/en/database/oracle/oracle-database/23/dwhsg/index.html. [Online; accessed 1-October-2024].
[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] Christian Riegger, Tobias Vinçon, Robert Gottstein, and Ilia Petrov. 2020. MV-PBT: Multi-Version Indexing for Large Datasets and HTAP Workloads. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 217–228. https://doi.org/10.5441/002/edbt.2020.20
[49] Doron Rotem, Kurt Stockinger, and Kesheng Wu. 2005. Optimizing candidate check costs for bitmap indices. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*. 648–655. https://doi.org/10.1145/1099554.1099718
[50] 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
[51] Avi Silberschatz, Henry F Korth, and S Sudarshan. 2020. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company. https://www.db-book.com/
[52] Haoze Song, Wenchao Zhou, Heming Cui, Xiang Peng, and Feifei Li. 2024. A survey on hybrid transactional and analytical processing. *The VLDB Journal* 33, 5 (2024), 1485–1515. https://doi.org/10.1007/s00778-024-00858-9
[53] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multi-Versioned Indexes. *Proceedings of the VLDB Endowment* 13, 2 (2019), 211–225. http://www.vldb.org/pvldb/vol13/p211-sun.pdf
[54] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. 2018. PAM: parallel augmented maps. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 290–304. https://doi.org/10.1145/3178487.3178509
[55] PostgreSQL Team. 2024. Chapter 11.9: Index-Only Scans and Covering Indexes. https://www.postgresql.org/docs/17/indexes-index-only-scans.html. [Online; accessed 1-May-2024].
[56] Sybase IQ Team. 2009. Chapter 15.1: Overview of Indexes. https://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.infocenter.dc00170.1510/html/iqapgv1/CIHECEAI.htm. [Online; accessed 1-October-2024].
[57] TPC. 2021. TPC-H benchmark. *http://www.tpc.org/tpch/* (2021).
[58] Junchang Wang and Manos Athanassoulis. 2024. CUBIT: Concurrent Updatable Bitmap Indexing. *Proceedings of the VLDB Endowment* 18, 2 (2024), 399–412. https://www.vldb.org/pvldb/vol18/p399-athanassoulis.pdf
[59] Xin Wang, Weihua Zhang, Zhaoguo Wang, Ziyun Wei, Haibo Chen, and Wenyun Zhao. 2017. Eunomia: Scaling Concurrent Search Trees under Contention Using HTM. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 385–399. https://doi.org/10.1145/3018743.3018752
[60] 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
[61] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 31–46. https://doi.org/10.1145/3437801.3441602
[62] 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
[63] Harry K T Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. 1985. Bit Transposed Files. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 448–457. http://www.vldb.org/conf/1985/P448.PDF
[64] 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. http://iopscience.iop.org/1742-6596/180/1/012053
[65] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2002. Compressing Bitmap Indexes for Faster Search Operations. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. 99–108. https://doi.org/10.1109/SSDM.2002.1029710
[66] 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. http://doi.acm.org/10.1145/1132863.1132864

[67] Kesheng Wu, Arie Shoshani, and Kurt Stockinger. 2010. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Transactions on Database Systems (TODS)* 35, 1 (2010), 1–52. http://dl.acm.org/citation.cfm?id=1670243.1670245

[68] 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. http://www.vldb.org/pvldb/vol10/p781-Wu.pdf

[69] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proceedings of the VLDB Endowment* 17, 2 (2023), 148–161. https://www.vldb.org/pvldb/vol17/p148-zeng.pdf

[70] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1771–1775. http://doi.acm.org/10.1145/3035918.3054784

Received April 2025; revised July 2025; accepted August 2025