LiveBin: A Localized and Version-Aware Binned Scan Index

ZIKANG LIU, Fudan University, China
LINWEI LI, Tencent, China
FEI YE, Fudan University, China
ZHENYING HE*, Fudan University, China
YINAN JING, Fudan University, China
WEN HE, Tencent, China
HUICHAO DUAN, Tencent, China
LIXIONG ZHENG, Tencent, China
X. SEAN WANG*, Fudan University, China

Recent advancements in Binned Scan Index have demonstrated significant potential for accelerating scan operators in main memory analytical databases. This capability stems from the Draft-Refine paradigm – a two-phase approach that first generates approximate draft results, then refines them using auxiliary metadata stored in the index. Despite achieving state-of-the-art scan performance, current designs still suffer from two major inefficiencies: (1) Refinement-phase overhead: While modern techniques improve draft accuracy to reduce refinement iterations, each refinement operation incurs substantial random memory access costs, and (2) Dynamic data inefficiency: When considering MVCC, scan operators need to determine the visible version of records for current transaction. The overhead of visibility checking is substantially greater than the cost of predicate evaluation. Consequently, leveraging a Binned Scan Index for scans yields only marginal improvements in total execution time.

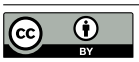
We propose LiveBin, a localized and version-aware Binned Scan Index that addresses these challenges through two key techniques: (1) Localization partitions a complete index into logically independent sub-indexes. While prior work strives to reduce refinement frequency through more complex draft generation, LiveBin employs Localization to decrease per-refinement random memory access overhead. This effectively reduces refinement-phase overhead while maintaining efficient draft generation. (2) Version-aware indexing embeds a hierarchical versioning structure, which extends the Draft-Refine paradigm to visibility checking. This integration consequently minimizes visibility validation and version chain traversal overhead. Experimental results demonstrate that LiveBin achieves 2.2–2.6× faster scan performance than state-of-the-art methods under identical memory budgets. We further integrated LiveBin into DuckDB, and the enhanced system achieved end-to-end speedups of 2.4–5.0× compared to the original DuckDB on TPC-H Q6 and SSB Q1 queries.

CCS Concepts: • **Information systems** → **Data scans**.

Additional Key Words and Phrases: scan index, analytical database, MVCC

*Corresponding authors

Authors' Contact Information: Zikang Liu, Fudan University, Shanghai, China, 24210240037@m.fudan.edu.cn; Linwei Li, Tencent, Shanghai, China, lwli15@fudan.edu.cn; Fei Ye, Fudan University, Shanghai, China, 21110240013@m.fudan.edu.cn; Zhenying He, Fudan University, Shanghai, China, zhenying@fudan.edu.cn; Yinan Jing, Fudan University, Shanghai, China, jingyn@fudan.edu.cn; Wen He, Tencent, Shanghai, China, arwinhe@tencent.com; Huichao Duan, Tencent, Shanghai, China, hcduan@tencent.com; Lixiong Zheng, Tencent, Shanghai, China, paterzheng@tencent.com; X. Sean Wang, Fudan University, Shanghai, China, xywangCS@fudan.edu.cn.



ACM 2836-6573/2026/2-ART50
https://doi.org/10.1145/3786664

ACM Reference Format:
Zikang Liu, Linwei Li, Fei Ye, Zhenying He, Yinan Jing, Wen He, Huichao Duan, Lixiong Zheng, and X. Sean Wang. 2026. LiveBin: A Localized and Version-Aware Binned Scan Index. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 50 (February 2026), 26 pages. https://doi.org/10.1145/3786664

1 Introduction

Scan operations represent a core computational primitive in main memory analytical database systems. A scan is defined as the process of evaluating a filter predicate over a designated column, materializing a bitmask or row identifier (rowID) list to flag qualifying records. Scan efficiency critically determines the end-to-end performance of query execution[7, 16, 19]. Indexing techniques[10, 12, 29] – specifically the construction of auxiliary metadata – serve as the primary mechanism to accelerate scan operations.

The rise of modern columnar database systems [4, 33, 39] has introduced new challenges for indexing. In columnar layouts, queries access only the required attributes, and advanced optimizations such as vectorized execution [31, 41] and data compression [15] make sequential scans more efficient. As a result, sequential scans can often outperform traditional indexes in many scenarios [20]. Achieving efficient index scans in this context therefore requires a careful balance between two types of memory access overheads: sequential traversal and random probing. While sequential traversal benefits from strong spatial locality but may process many irrelevant elements, random probing quickly narrows down candidate tuples but suffers from high per-access latency.

Binned Scan Index[11, 26] represents the state-of-the-art indexing technique, demonstrating superior performance by significantly reducing the volume of data accessed. It adopts the Draft-Refine paradigm: first generating an approximate draft result, then refining the draft to obtain the precise query result.

BinDex[26], the first proposed Binned Scan Index, consists of two components: (1) column values partitioned into multiple ranges (bins), each storing a direct filter bit vector to indicate qualifying rowIDs, and (2) a position array – a rowID sequence sorted by element values. Its query workflow first locates the target predicate's bin to copy the corresponding filter bit vector as the draft result, then searches the position array to identify rowIDs between the draft's value boundary and predicate, refining their corresponding bits in the draft to obtain the final result. Notably, finer-grained binning reduces random probing in refinement but requires storing more filter bit vectors, leading to excessive space overhead.

Cabin[11] retains the position array but replaces BinDex's filter bit vectors with compressed Filter Sketches, enabling more bins under identical space budgets. During queries, Cabin decompresses Filter Sketches to generate drafts (increasing sequential traversal overhead) but produces more accurate drafts that significantly reduce random probing iterations during refinement.

While Binned Scan Indexes achieve minimal data access through pre-materialized drafts, their current structural design suffers from two major inefficiencies that constrain their applicability in modern analytical databases.

The first inefficiency originates from costly random memory accesses during the refinement phase. As data scales, random probing costs grow disproportionately due to degraded spatial locality from scattered memory accesses, which exacerbate cache misses. Moreover, refinement exhibits inherent limitations in multi-threaded scalability: when distributing rowID refinements across threads, contention arises from blockwise bit storage (e.g., every 64 bits are stored as a `uint64_t`). Concurrent modifications to shared bit blocks cap parallelism. Furthermore, under tight memory budgets, a Binned Scan Index can only store a limited number of bit vectors, necessitating a large number of costly refinement operations. In such scenarios, the scan performance of the Binned Scan Index further deteriorates.

The second inefficiency manifests as scan performance degradation when processing dynamic data. The flat, clustered structural design impedes adjustments to the index. Structural updates (e.g., insertions/deletions in contiguous bit vectors and position arrays) incur high overheads. In modern MVCC systems[3, 8, 21], the requirement to maintain and process multiple versions of records during queries further degrades Binned Scan Index scan efficiency. The space overhead of a single Binned Scan Index is already substantial; maintaining multiple versions would introduce prohibitive storage costs, rendering this approach impractical. Moreover, after accumulated update and deletion operations, external version chain traversal and visibility checking during scanning significantly diminish the acceleration advantage of Binned Scan Indexes.

In this paper, we propose LiveBin, a localized and version-aware Binned Scan Index which addresses the aforementioned limitations through two key techniques.

The first is Localization – partitioning a complete index into logically independent sub-indexes. Prior research has focused on reducing the frequency of random memory probes during index scanning, whereas Localization adopts a novel perspective by decreasing the latency of individual random accesses. This cache-friendly approach significantly reduces refinement-phase costs by confining random memory accesses to localized sections. Furthermore, sub-index independence eliminates thread contention caused by monolithic flat structures during refinement. By isolating bit-block modifications to individual sections, LiveBin achieves better multi-thread scalability. Additionally, we propose a partial storage strategy to address the high memory footprint of Binned Scan Indexes. This approach enables storing a larger number of bit vectors under a comparable memory budget, allowing LiveBin to achieve a superior time-space trade-off.

The second key technique of LiveBin is version-aware indexing, where each sub-index maintains a lightweight versioning structure. We propose a hierarchical index adjustment strategy to achieve an improved balance between scan efficiency and index adjustment overhead. The first level organizes modification operations chronologically in an entry list. During multi-version processing, scan results are adjusted solely based on committed modifications identified in this list, eliminating high-overhead external version chain traversal and visibility checking. The second level, Valid Draft, records fully invalid rowIDs by a bit vector. When invalid elements accumulate, relying solely on the entry list triggers excessive random memory accesses. To mitigate this, fully invalid elements are synchronized into the Valid Draft, and final scan results are computed via bitwise AND operations between index scan results and the Valid Draft. This converts bulk random memory accesses into a single efficient bitwise AND operation. The final level, localized index adjustment, batches version information synchronization into sub-indexes when committed modifications reach a predetermined threshold or during system idleness. This process acquires more accurate index metadata, accelerating subsequent scan operations. Furthermore, confining each index adjustment exclusively to individual sub-indexes makes the adjustment overhead acceptable and ensures other sub-indexes remain available for concurrent query transactions.

The main contributions of this paper are fourfold.

- Through theoretical analysis and preliminary experiments, we identify two critical inefficiencies restricting Binned Scan Index adoption in in-memory analytical databases.
- We introduce LiveBin, which incorporates Localization technology to reduce random memory probing overhead in Binned Scan Index, further accelerating scan performance for in-memory analytical databases.
- LiveBin maintains lightweight hierarchical version information, effectively mitigating efficiency degradation when processing dynamic data under MVCC.
- We conduct extensive experimental evaluations of LiveBin under various workloads. Results show LiveBin achieves 2.2–2.6× speedup over state-of-the-art indexes under the same

space budget. We integrate LiveBin into DuckDB, a high-performance in-memory analytical database, demonstrating 2.4–5.0× scan performance improvements.

The rest of the paper is organized as follows. Section 2 analyzes the performance of the state-of-the-art approaches. Section 3 presents the LiveBin design and Section 4 describes the detailed algorithm of LiveBin. After that, Section 5 performs the experimental evaluation. Finally, Section 6 concludes the paper.

2 Background and Related Work

This section establishes the technical foundation and motivation for LiveBin. Section 2.1 discusses scan operations and the current research status. Section 2.2 discusses existing Binned Scan Index implementations. Section 2.3 details Multi-Version Concurrency Control (MVCC) mechanisms, specifically analyzing challenges Binned Scan Index encounters under MVCC.

2.1 Scan Operation and Scan Index

The scan operation takes two inputs: (1) a list of N values where each value is associated with a rowID indicating its ordinal position, and (2) a filtering condition comprising a predicate with a comparison operator (<, ≤, >, ≥, =, !=, or BETWEEN). The output can take two forms: a list of qualifying rowIDs, or an N-bit result bit vector in which the i-th bit is set to 1 if the i-th row satisfies the filter predicate and to 0 otherwise. The bit-vector representation has emerged as the predominant choice in modern scan indexes[13, 15, 28], owing to its superior operational efficiency. It enables dense, cache-friendly data layouts, facilitates SIMD-accelerated bitwise operations, and naturally supports efficient composition of multi-column predicates—advantages that are especially pronounced in analytical workloads involving complex filtering across multiple attributes.

Indexing serves as the primary acceleration mechanism for scan processes by maintaining supplemental metadata. The evolution of columnar databases[4, 33, 39] has introduced new challenges for traditional index designs, as sequential scans often outperform traditional indexes in such systems[5, 20]. This advantage primarily stems from columnar storage allowing scans to avoid accessing irrelevant attributes.

Recent years have witnessed innovative approaches for accelerating sequential scans in columnar environments through various optimizations. (1) Data Skipping[29, 32, 35, 36] leverages precomputed statistics to skip non-relevant data zones, though effectiveness diminishes with unordered distributions; (2) Bit-level storage layouts[13, 28] decompose data into bitwise segments for hardware-parallel early pruning via most-significant-bit comparisons; (3) Lossy compression (e.g., Column Sketches[15]) scans approximated columns to reduce data volume but incurs verification overheads for ambiguous codes; (4) SIMD vectorization[22, 31, 40, 41] exploits batched processing of aligned data blocks using mask-based parallel operations, achieving significant speedups for arithmetic/logical workflows at the cost of memory alignment constraints.

Designing efficient indexes for columnar databases demands more refined structural and algorithmic designs, particularly requiring a balance between two memory access overheads: sequential traversal and random probing. Sequential traversal operates on continuously stored metadata to rapidly determine element eligibility, while incurring access volumes proportional to total elements. For example, Bitmap indexes[2, 9, 10, 38] are ideally suited for low-cardinality columns. Random probing employs targeted metadata access to efficiently eliminate non-qualifying elements, but each random memory access incurs significant latency. Tree-based indexes (e.g., B+ Tree[12], ART[24]) exemplify this paradigm, demonstrating efficiency primarily in low-selectivity scenarios (<1%).

2.2 Binned Scan Index

Binned Scan Index represents a hybrid technique combining sequential traversal and random probing, currently recognized as the state-of-the-art index for scan operations. Binning partitions the value domain into multiple ranges (bins), each maintaining a bit vector that directly or indirectly records common features of values within its range. Built upon the Draft-Refine paradigm, Binned Scan Index first identifies the predicate-matching range and utilizes its corresponding bit vector as initial query result draft, then refines the draft by validating individual elements against the predicate through value comparisons. We subsequently examine two contemporary Binned Scan Index implementations - BinDex[26] and Cabin[11] - analyzing their performance characteristics while identifying persistent bottlenecks in current methodologies.

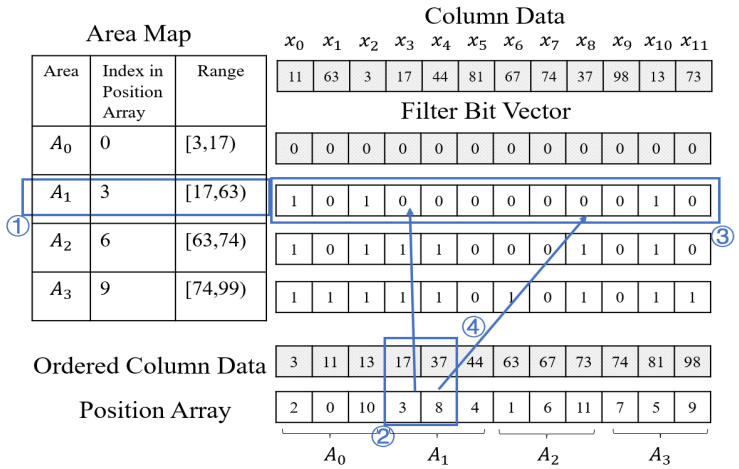


Fig. 1. Structure of BinDex and a scan example of BinDex (x < 40) (The shading indicates that this portion need not be stored within the index.)

BinDex. As illustrated in Figure 1, the BinDex structure consists of three components: Position Array, Area Map, and Filter Bit Vectors. Position Array stores rowIDs sorted by the original column data order; Area Map partitions the Position Array into discrete areas while recording each area's starting position; Filter Bit Vectors (one per area) encode rowIDs of elements smaller than each area's minimum value.

The BinDex scan procedure(using x < 40 as an example, blue-highlighted in the diagram) involves four phases: (1) Area Identification: Determine the containing area for the predicate value (40 resides in A_1); (2) Position Array Probe: Within A_1's designated Position Array segment, verify elements actually satisfying x < 40 (values 17, 37 corresponding to rowIDs 3,8); (3) Draft Generation: Copy A_1's Filter Bit Vector (pre-materializing rowIDs with values < 17) as the initial result draft; (4) Draft Refinement: Finalize the result by explicitly setting bits 3 and 8 in the draft vector.

BinDex does not store ordered column data. During the Position Array Probe phase, binary search is performed on the Position Array, and the corresponding column data (unordered) is accessed via the rowIDs to identify entries smaller than the predicate. Although this phase involves random access to column data, the number of such accesses is substantially lower than in the Draft Refinement phase. Thus, BinDex omits storing ordered column data to balance memory footprint and performance. Moreover, no separate Filter Bit Vectors are stored for other operators. For $\leq$, the condition $x \leq c$ can be equivalently processed as $x < (c + 1)$. For $>$ and $\geq$, BinDex applies a

fast bitwise NOT on the less-than Filter Bit Vector, which corresponds to the predicate, to generate the initial draft. This draft then undergoes a refinement process similar to that used for less-than queries. For column data, the index does not store the raw data internally, though the original values need to be accessible via rowIDs during scan processing.

Cabin. Figure 2 illustrates the structure of Cabin. Cabin replaces BinDex's per-area Filter Bit Vectors with Filter Sketches (in the red box). Specifically, Cabin assigns a w-bit code to each area, enabling division of the Position Array into $2^w - 2$ areas (reserving the minimum and maximum codes for metadata representation). Filter Sketches comprise w bit vectors that vertically store each element's encoded representation. In the figure, the parameter w is set to 3, the value x_0 resides in area A_1 and is encoded as 110 (the code of A_1). While retaining the Position Array, Cabin's Area Map also stores a corresponding code for each area (in the red box).

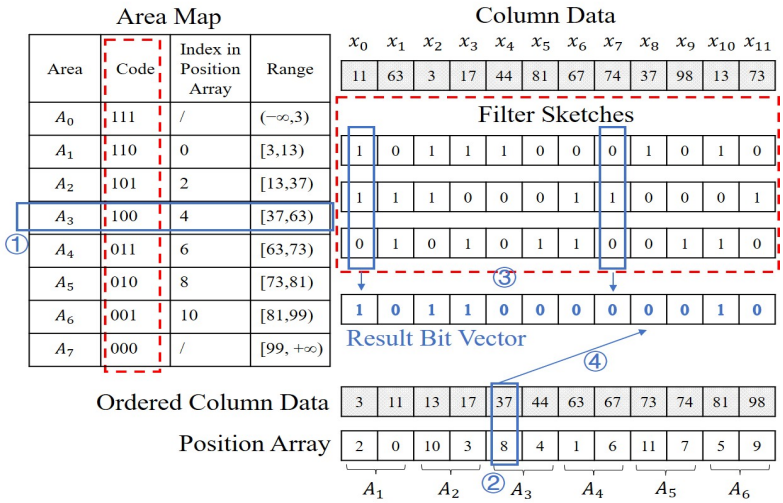


Fig. 2. Structure of Cabin and a scan example of Cabin (x < 40) (Red boxes indicate the main difference from BinDex)

The scan process of Cabin also consists of four phases, consistent with BinDex (using x < 40 as an example, highlighted in blue in the figure): (1) Area Identification: The value 40 resides A_3. (2) Position Array Probe: Within A_3, the element less than 40 is x_8. (3) Draft Generation: The only difference from BinDex lies in this phase. Cabin processes the encoded representation of each element in the Filter Sketches to generate the draft bit vector. In the figure, all elements belonging to A_0, A_1, and A_2—whose corresponding codes are 111, 110, and 101—are set to 1 in the draft; the remaining elements are set to 0. For example, x_0 with code 110 is set to 1, while x_7 with code 010 is set to 0. (4) Draft Refinement: The result is finalized by explicitly setting bit 8 in the draft ($x_8 < 40$).

Performance Analysis. Under identical space budgets (allocating w extra bit vectors), BinDex divides the domain into $w + 1$ ranges compared to Cabin's $2^w - 2$ ranges. During Draft Generation, BinDex directly copies the Filter Bit Vector whereas Cabin must sequentially traverse w bit vectors of Filter Sketches to construct the draft. During the Draft Refinement , Cabin's finer-grained partitioning of the value domain reduces the number of elements per range, thereby decreasing the number of bits requiring refinement.

To investigate performance bottlenecks in contemporary Binned Scan Index, we compared phase-level execution time of BinDex and Cabin for scan operations on 1 billion int32-type data

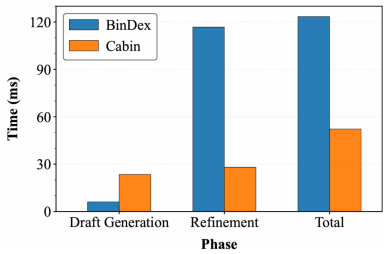


Fig. 3. Execution time comparison at different phases

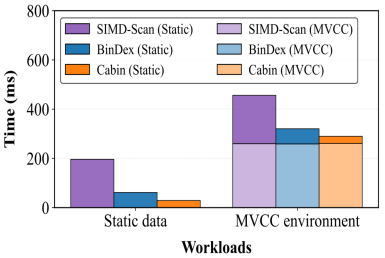


Fig. 4. Performance comparison on different workloads

records. Both indexes were configured with 16 additional bit vectors (Cabin using 4-bit codes), with results shown in Figure 3. Consistent with prior analysis, Cabin incurs higher Draft Generation latency, but achieves significant Refinement time reduction.

Nevertheless, the refinement phase continues to be the dominant performance bottleneck in Binned Scan Indexes. It consumes 94% of total execution time in BinDex and 56% in Cabin. Although Cabin mitigates this bottleneck to some extent, it does so at the cost of increased overhead in the Draft Generation phase. In this work, we aim to significantly reduce per-refinement latency while leaving the performance of all other execution stages unaffected.

2.3 Multi-Version Concurrency Control

Modern database systems widely adopt Multi-Version Concurrency Control (MVCC) [3, 8, 21, 42], which maintains multiple versions of each record to enable concurrent access. In particular, many OLAP-oriented systems employ optimistic concurrency control (OCC) [30, 43], allowing transactions to read or modify records without holding locks during execution. Conflicts—such as concurrent writes or overlapping read-write operations on the same record—are detected at validation time, and one of the conflicting transactions is aborted. This design is well suited for analytical workloads, as read-only queries never conflict, and concurrent updates to the same record are generally infrequent.

Under MVCC, the scan workflow operates as follows: Each element version's validity interval is defined by T_{start} and T_{end}. Upon initiation, a transaction receives a timestamp T_{id}. During scanning, the system must identify the version satisfying $T_{start} \leq T_{id} < T_{end}$ for each element as the visible version. This visible version is then evaluated against predicates to produce the scan result.

Tree-based indexes suffer less from MVCC complexities due to their easily adjustable structures. Taking B+ Tree[12] as an example: For multiple versions of a single record, each version inserts a corresponding node into the B+ Tree. Each leaf node is associated with a T_{start} and T_{end} pair defining its validity interval. During index traversal, a visibility check is performed on each encountered leaf node to determine whether its validity interval satisfies the temporal constraints of the scanning transaction. This structure naturally aligns with versioned data access, thereby mitigating the MVCC-induced performance degradation observed in scan-oriented indexes.

Challenges of Binned Scan Indexes under MVCC. The use of Binned Scan Indexes to accelerate scan operations under MVCC introduces new challenges: (1) how to maintain multiple versions of a record within the index structure, and (2) how to preserve high scan efficiency when dealing with multiple versions of data.

Flat indexes face significant challenges in efficiently storing multiple versions of records due to their flat clustered storage structure, where physical positions are tightly coupled with logical

ordering. For Binned Scan Indexes, the storage overhead of a single version is already substantial, making the simultaneous storage of multiple versions prohibitive.

The second challenge lies in how temporal validity verification for multi-version data degrades index scan efficiency. We conducted preliminary experiments comparing Binned Scan Index (BinDex[26], Cabin[11]) and a SIMD-accelerated naive scan (denoted as SIMD-Scan[41]) for scan operations on 1 billion int32 records. BinDex and Cabin were configured with 32 additional bit vectors. The experimental results are shown in Figure 4. The darker color represents the time spent by each method on processing raw data, while the lighter color indicates the time dedicated to version validation. Both SIMD-Scan and Binned Scan Index exhibit nearly identical increases in scan time under MVCC compared to static data, due to the shared version validation overhead required by both methods. While Cabin achieves 6.8× speedup over SIMD-Scan on static data, this acceleration ratio drops to 1.6× in MVCC scenarios. Under MVCC, the overhead from visibility validation fundamentally constrains Binned Scan Indexes' acceleration ratio, emerging as the critical bottleneck. The cost of visibility checking outweighs that of predicate evaluation. As a result, using a Binned Scan Index for scanning yields only marginal improvements. Therefore, to enhance the practical value of Binned Scan Indexes in modern MVCC systems, it is necessary to maintain additional metadata within indexes to reduce the overhead of visibility validation.

Design Principles. Based on the theoretical analysis and preliminary experiments outlined above, maintaining high-efficiency scans for Binned Scan Indexes under MVCC requires embedding supplemental version metadata within the index structure while adhering to three core principles: (1) Lightweight: The version metadata maintained in indexes should not incur significant storage overhead. (2) Rapidly processable: Metadata for each version must be compact and quickly processed. (3) Structurally flexible: Version metadata must facilitate easy maintenance to avoid compromising overall system throughput.

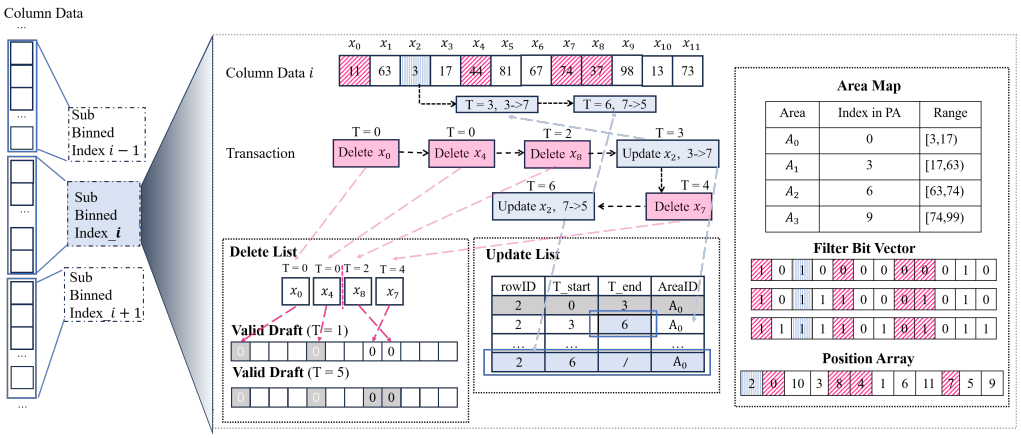


Fig. 5. The data structure of LiveBin (Pink indicates deletion operations, while blue represents update operations. Arrows illustrate the impact of operations on data structures. Slashed shading denotes data affected by modification operations.)

3 LiveBin Structure Design

This section presents the architectural design of LiveBin, whose comprehensive structure is depicted in Figure 5. Section 3.1 introduces the Localization mechanism for spatial data organization. Sections

3.2-3.3 subsequently detail the Delete List and Valid Draft components, which collaboratively optimize deletion processing. Section 3.4 describes the Update List, a dedicated structure designed to accelerate update operation handling.

3.1 Localization

Unlike conventional secondary indexes, LiveBin employs a horizontal partitioning strategy that decomposes a monolithic index into multiple sub-indexes, each autonomously managing contiguously stored data segments. For instance, given 1 billion data entries, if we set the segment size managed by each sub-index to 1 million entries, then 1000 sub-indexes will be constructed. The first sub-index manages the first 1 million entries, and so on.

Figure 5 illustrates the Localization mechanism of LiveBin, where each sub-index maintains an Area Map, Position Array, Filter Bit Vector (functionally identical to those described in Section 2.2), and MVCC structures (to be introduced in subsequent sections) for its designated data segment. During scanning, each sub-index generates a local result bit vector representing its managed segment's scan outcome. The parent index then sequentially concatenates these per-sub-index result bit vectors to yield the final scan result.

Localization delivers three key benefits for LiveBin: (1) enhanced scan efficiency, (2) improved multi-thread scalability, and (3) better structural flexibility.

Scan Efficiency. Section 2.2 identifies the bottleneck in the refinement phase of existing Binned Scan Indexes. Prior studies primarily attempt to reduce the number of random memory probes by producing more accurate drafts. Localization, in contrast, takes a complementary approach—it reduces the cost per probe to accelerate the refinement phase. Specifically, Localization confines each sub-index's random probing to its own local result bit vector, which typically fits entirely within the CPU cache. As a result, memory accesses exhibit strong spatial locality, avoiding frequent cache misses. In comparison, conventional Binned Scan Indexes probe a large global bit vector, causing scattered random accesses across memory and poor cache utilization. By localizing memory access patterns, Localization effectively enhances spatial locality and substantially lowers the per-probing overhead during refinement.

Multi-thread Scalability. Localization further enhances multi-thread scalability. Whereas refinement in conventional monolithic Binned Scan Indexes partitions rowID sets across threads, the block-wise storage of bits (typically using `uint64_t` types) introduces contention when multiple threads access the same blocks, reducing concurrency. In contrast, LiveBin's entirely independent sub-indexes enable concurrency at sub-index granularity: Each sub-index scan task is assigned to a single thread. This approach eliminates inter-thread data contention of conventional monolithic Binned Scan Indexes, consequently enabling superior scalability.

Structural Flexibility. Localization confers greater structural flexibility upon LiveBin. For incremental data, prior work adopts the Main+Delta strategy: newly arrived data is not immediately inserted into the index, but triggers full index reconstruction upon reaching a certain threshold. This approach suffers from two critical issues: (1) Unindexed new data degrades overall scan efficiency during queries when new data accumulates; (2) Full index reconstruction incurs substantial overhead. Through Localization, incremental data reaching a sub-index size threshold enables dedicated sub-index construction for that data segment, which facilitates immediate indexing of new data. LiveBin uses SIMD-Scan for newly inserted data below the sub-index construction threshold. This does not become a scan bottleneck due to the small volume of the unindexed data. Moreover, these sub-indexes require no merging with existing ones, enabling rapid construction. For data modifications, Localization enables LiveBin to reconstruct only the affected sub-indexes, as opposed to rebuilding the entire index. The overhead of reconstructing individual sub-indexes remains acceptable, as will be elaborated in Section 4.2.

Spatial Overhead. The spatial overhead introduced by Localization proves negligible. Since both Filter Bit Vector and Position Array scale linearly with raw data size, their space overhead remains completely unchanged regardless of whether a monolithic index or sub-index structure is employed. The sole addition stems from Area Map, where LiveBin maintains a dedicated Area Map per sub-index to record minimum values per area. This incremental overhead proves insignificant—for instance, with 1 million data entries, a single bit vector occupies 122KB while an Area Map only requires less than 1KB.

Tradeoff of Segment Size. We now discuss the impact of segmentation granularity. Consider a dataset of N elements partitioned into K segments (each maintaining a sub-index), with M Areas per sub-index. We analyze the effect of segmentation through the four phases of the Binned Scan Index algorithm (Section 2.2). For Area Identification, the overhead is negligible due to the small number of Areas. For Position Array Probe, the number of random accesses per sub-index is $log(\frac{N}{KM})$, resulting in a total of $Klog(\frac{N}{KM})$ random accesses. As K increases, this overhead grows. For Draft Generation, copying N bits as the draft is required, independent of K. For Draft Refinement, the number of random accesses per sub-index is $\frac{N}{2KM}$, leading to a total of $\frac{N}{2M}$ accesses, which is independent of K. However, owing to improved spatial locality of a larger K, the cost per random access decreases.

In summary, while Localization enhances spatial locality, it is not a guaranteed win. Finer-grained segmentation improves locality but increases overhead in the Position Array Probe phase. Our subsequent experiments confirm that an improper segment size can degrade performance compared to a non-partitioned approach. More importantly, not all Binned Indexes benefit from Localization. This is observed in Cabin, where Localization fails to address another of its bottleneck—the high cost of decoding Filter Sketches.

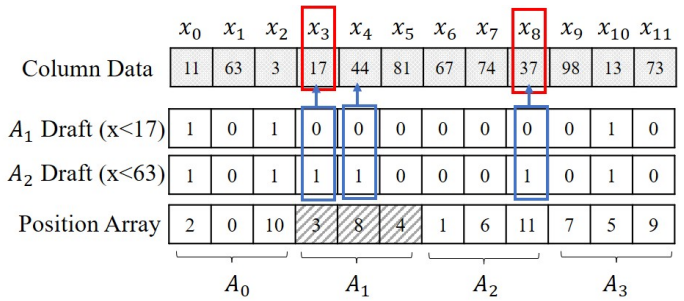


Fig. 6. Scan algorithm for partially stored Position array (x<40, 40 resides in A_1, Position Array of A_1 is not stored)

Memory Footprint Issue. High memory footprint is a significant challenge for Binned Scan Indexes. The memory overhead primarily comes from two components: Filter Bit Vectors and Position Array. We now discuss how LiveBin achieves efficient scan under limited memory budgets by optimizing both aspects. For Filter Bit Vectors, since Localization significantly reduces the cost of random accesses, LiveBin can achieve better scan efficiency with fewer Filter Bit Vectors compared to BinDex and Cabin.

For Position Array, rowIDs in each sub-index can be stored using a smaller bit width. For example, with 1 billion entries, a conventional index requires `uint32_t` to store rowIDs. In LiveBin, if we set the segment size to 50000, `uint16_t` can be used for storage instead. Moreover, when the memory space budget is particularly tight, for each sub-index, we can choose not to store the Position Array for a portion of the Areas. When the predicate falls within these Areas, we can refine the draft

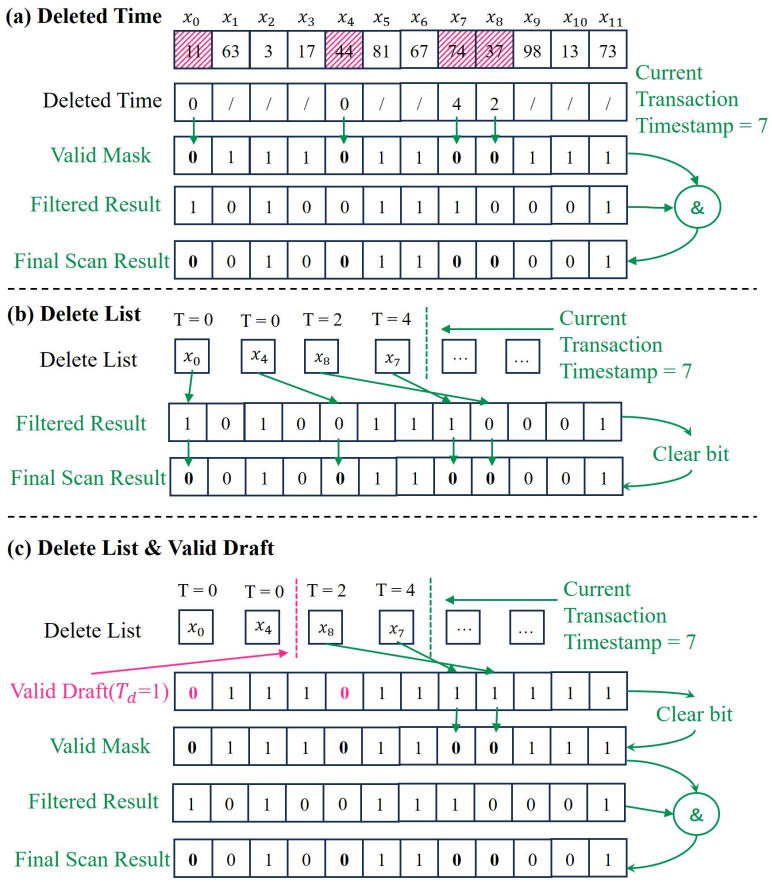


Fig. 7. Visibility validation with (a) Deleted Time, (b) Delete List, (c) Delete List and Valid Draft

using an additional algorithm. As shown in Figure 6, the A_1's Position Array is not stored. We can perform a bitwise XOR between the A_1 draft (x < 17) and the A_2 draft (x < 63) to obtain the elements in the range [17, 63) (which are x_3, x_4, x_8 originally in A_1's Position Array). Then we access these elements from the raw data and compare them with the predicate, and refine the draft based on the comparison results. (x_3 and x_8 satisfy the condition $x < 40$.)

3.2 Delete List

Under the MVCC mechanism, element deletion is implemented through deletion timestamp marking rather than immediate physical removal. As shown in Figure 7(a), elements x_7 and x_8 are marked with deletion timestamps T=4 and T=2 respectively. Visibility validation requires comparing the current transaction's timestamp against each element's inserted time and deleted time - an element becomes visible only when the transaction timestamp falls within [inserted time, deleted time). This process generates a Valid Mask (a bit vector where the i-th bit indicates the visibility of the i-th element). The final scan result is derived through a bitwise AND operation between the predicate-derived result bit vector (Filtered Result in the figure) and the Valid Mask.

In naive scan implementations, predicate evaluation requires sequential data comparisons while Valid Mask generation necessitates timestamp comparisons against all inserted time/deleted time pairs. Critically, Valid Mask computation often exceeds predicate evaluation time. To eliminate the prohibitive cost of full inserted time/deleted time traversal, we introduce the Delete List.

As shown in Figure 7(b), the Delete List maintains a sequence of <rowID, timestamp> tuples recording deleted elements' row identifiers and deletion timestamps, ordered chronologically by deletion time. With this structure, the visibility validation workflow becomes: (1) Locate all deletions committed before the current transaction's start timestamp within the Delete List (x_0, x_4, x_8, x_7 in the figure). (2) Modify the predicate-generated result bit vector by clearing bits corresponding to identified deleted rowIDs.

Maintenance overhead for the Delete List is minimal. When committing a deletion, we simply append a <rowID, timestamp> tuple to the list's tail in addition to updating the element's deleted time with the commit timestamp. This transforms metadata access patterns from per-element traversal to selective bitwise operations. The optimization significantly reduces visibility processing latency, particularly effective when few deletions have occurred.

3.3 Time-anchored Valid Draft

As committed deletions accumulate, the growing quantity of tuples in the Delete List necessitates substantial bit modifications to the filtered result bit vector. Due to the random memory access pattern, the associated overhead escalates rapidly. We address this by extending the Draft-Refine paradigm to Valid Mask generation through a time-anchored Valid Draft - a bit vector where bit i=0 indicates element i was deleted before timestamp T_d. As illustrated in Figure 7(c), the Valid Draft with $T_d = 1$ records elements deleted prior to T_d (specifically x_0 and x_4) by clearing their corresponding bits in the Valid Draft.

With Valid Draft, the visibility validation process becomes (shown in Figure 7(c)): (1) Delta Identification: Retrieve rowIDs in Delete List with timestamps between the Valid Draft's anchor time T_d and the current transaction's start time (x_8 and x_7 in the figure). (2) Draft Refinement: Generate Valid Mask by clearing bits corresponding to identified deletions in the Valid Draft. (3) The final query result is obtained through a bitwise AND operation between this Valid Mask and the predicate-generated result bit vector.

Valid Draft maintenance remains efficient. We could identify all Delete List tuples with timestamps prior to the minimum active transaction timestamp (a global variable). Then, the corresponding bits in the Valid Draft are cleared for these rowIDs and the author time of the Valid Draft is set to the minimum active transaction timestamp. Finally, the obsolete tuples can be safely bulk-removed from the head of the Delete List array as no active transactions require their visibility. For example, in the Valid Draft module shown in Figure 5, we synchronize both x_8 and x_7 into the Valid Draft and update its timestamp from 1 to 5.

The adjustment of the Valid Draft can be triggered under two distinct conditions: it can occur immediately upon the commit of each individual deletion operation, or lazily when the Delete List accumulates to a predetermined size threshold. This design deliberately transforms frequent fine-grained random bit modifications into batched bitwise-AND operations. As a result, it strikes an optimized balance between the access patterns of random probing and sequential traversal, while maintaining low runtime maintenance overhead.

3.4 Dual-filtering Update List

The integration of Delete List and Valid Draft elegantly mitigates the efficiency degradation caused by MVCC deletions during scan operations. In certain databases like PolarDB-IMCI[39], an update to columnar data is implemented as a delete operation followed by an insert operation. Here,

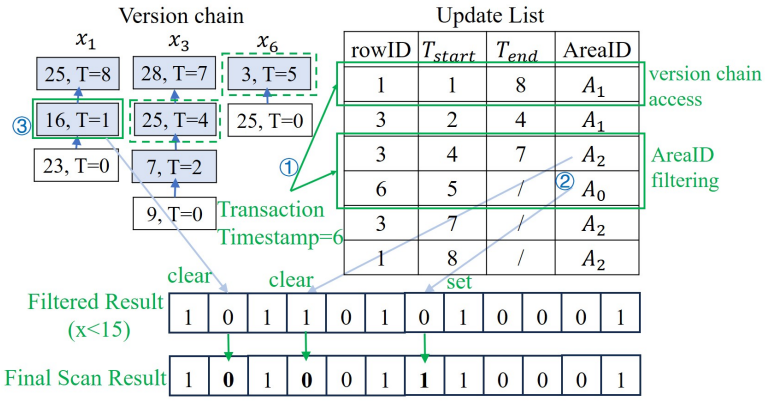


Fig. 8. Update processing of LiveBin (Suppose the elements with AreaID of 1 need to be further confirmed)

our proposed Delete List and Valid Draft mechanisms already fully accommodate such update operations. However, in databases such as DuckDB[33], updates are handled by inserting new versions into version chains. This necessitates maintaining supplemental metadata within indexes to efficiently process update operations under MVCC.

We now address efficient processing of MVCC updates. A baseline approach would traverse each element's version chain to identify visible versions, evaluate filter conditions, and modify corresponding bits in the result bit vectors of index scans. However, this necessitates full traversal of all version chains, resulting in prohibitive inefficiency under heavy update workloads. We introduce the Update List, which strategically reduces version chain access costs through timestamp filtering and AreaID metadata exploitation. The Update List employs quadruple entries formatted as <rowID, T_{start}, T_{end}, AreaID>, where each entry captures the row identifier of modified data, the update's commit timestamp T_{start}, its validity interval [T_{start}, T_{end}), and the AreaID where the updated value is located. The AreaID here corresponds to the same concept as the AreaID in the Area Map shown in Figure 5. Furthermore, it follows the same value range partitioning as defined in the Area Map.

The Update List inherently supports dual filtering: First, temporal filtering ensures that for any query timestamp, at most one quadruple per rowID satisfies visibility constraints, even with multiple updates to the same record. This filter efficiency is amplified by the spatial locality observed in update patterns, where most modifications target a limited subset of rowIDs. Second, AreaID filtering operates subsequent to temporal filtering, with access to the version chain data occurring exclusively when the result of predicate comparison cannot be determined based on AreaID.

With this structure, update processing proceeds as follows (shown in Figure 8): (1) Given a query transaction timestamp T, we first filter quadruples satisfying $T_{start} \leq T < T_{end}$ using SIMD-accelerated comparisons (green boxes in the figure). (2) For these qualified quadruples, we subsequently perform predicate evaluation based on stored AreaIDs and set corresponding result bits (e.g., quadruples with rowIDs 3 and 6). (3) Finally, for indeterminate cases by AreaID, we access version chains and set bits based on comparison outcomes (e.g., rowID 1 quadruple).

Update List maintenance remains lightweight: For new updates, the system simply appends configured quadruple entries to the list's tail while updating the previous version's T_{end} to the current transaction's commit timestamp, thereby maintaining continuous version life spans. For example, when processing the transaction "T=6, Update x_2, 7→5" in Figure 5, we append an entry at the end of the Update List and set the T_{end} of the previous entry of x_2 to 6. When handling

wide-bit data types (e.g., strings and large integers), we store AreaIDs in quadruples according to our lightweight version metadata principle. For narrow-bit data types, we directly store updated values instead of AreaIDs, eliminating version chain accesses entirely.

As the number of committed updates increases, the overhead of processing the Update List cannot be ignored. We therefore introduce a garbage collection mechanism to remove obsolete entries from the Update List. Specifically, entries whose T_{end} values are smaller than the minimum active transaction timestamp can be safely deleted, as they are no longer visible to any active transaction. This garbage collection is performed lazily, triggered only when the length of the Update List exceeds a predefined threshold.

Currently, the Update List is sorted by T_{start}, while an alternative scheme would sort it by rowID. We analyze the trade-offs between these two sorting methods along three aspects. First, the number of modified bits in the result remains unchanged regardless of the sorting method, since only one version of each updated rowID is visible to a transaction. A rowID-sorted list produces sequentially sorted rowIDs after dual filtering, enabling sequential access during result modification, whereas a T_{start}-sorted list leads to random access. However, due to each sub-index maintaining its own Update List, such random access is localized and thus incurs low overhead. Second, with T_{start} sorting, binary search can be performed based on the transaction timestamp to only process entries committed before the transaction,thereby avoiding a full scan of the list, whereas rowID sorting requires processing all entries. Third, when new updates are committed, T_{start} sorting allows efficient append-only operations, whereas rowID sorting may require costly insertion in the middle. Based on the above analysis, we favor the T_{start} sorting method for the Update List.

4 LiveBin Operations

In Section 3, we introduced LiveBin's Delete List, Valid Draft, and Update List structures to handle deletion and update operations under MVCC. We now integrate these components and detail the operations on LiveBin.

4.1 Scan Algorithm

We first present the complete scan execution workflow of LiveBin under MVCC. Algorithm 1 details the process for individual sub-indexes in LiveBin, where the final scan result requires sequential concatenation of all sub-indexes' result bit vectors.

Lines 1-5 execute the filtering process consistent with BinDex's implementation (detailed in Section 2.2). Following Area Identification and Position Array Probe (lines 1-2), we select the closest filter bit vector to the predicate as the draft while identifying its error margins relative to the final result during Draft Generation (lines 3-4). The draft's errors are subsequently corrected through Draft Refinement to obtain the filtered result vector RV (line 5).

Lines 7-8 handle deletion information to generate the Valid Mask (detailed in Section 3.3). Lines 9-15 process update information: The Update List's dual-filtering mechanism first identifies the set of rowIDs R_u requiring version chain access (lines 9-10). For each rowID in this set, the updated value is located in the version chain, and the corresponding bit in the result bit vector RV is set based on predicate evaluation of the retrieved value (lines 12-13). The final output combines the update-adjusted RV with the Valid Mask V_m through bitwise AND (lines 15-16).

For other operators ($\leq$, $>$, $\geq$, $=$, !=, BETWEEN), LiveBin employs the same processing logic as BinDex and Cabin. For example, we decompose the BETWEEN operator ($c_1 \leq x \leq c_2$) into $x \geq c_1$ and $x \leq c_2$ for processing. These operators differ only in the filtering phase (lines 1-5) without affecting subsequent deletion and update handling, hence we omit redundant algorithmic descriptions for these cases.

Algorithm 1: LiveBin's sub-index scan for predicate with "<" operator under MVCC

Input: a predicate "$x < c$" on column $X_{1...N}$, Filter Bit Vector $FV_{1...M}$, Area Map AM, Position Array PA, current transaction's timestamp T_0, Delete List DL , Valid Draft VD, Valid Draft's timestamp T_d, Update List UL, version chain VC

Output: result bit vector RV

1 A_c = binary_search_area(c, AM)
2 i_c = binary_search_pos(c, PA, X)
3 k_c, p_{start}, p_{end} = select_closest_draft(A_c, i_c, AM)
4 RV = draft_bit_vector_build(k_c, FV)
5 refine_draft(RV, p_{start}, p_{end}, PA)
6
7 j_s, j_e = binary_search_list(T_0, T_d, DL)
8 V_m = valid_mask_build(VD, j_s, j_e)
9 UL' = timestamp_filter_list(T_0, UL)
10 R_u = area_filter_list(c, UL', RV)
11 **for** r ***in*** Ru **do**
12 x_i = version_chain_access(r,VC)
13 compare_and_modify(x_i, c, RV)
14 **end**
15 RV = bitwise_and(RV, V_m)
16 **return** RV

Shortcut Optimization. LiveBin inherits the Shortcut Optimization from BinDex and Cabin. After Area Identification and Position Array Probe (lines 1-2), when the measured predicate selectivity falls below a predefined threshold, we initialize an all-zero bit vector during Draft Generation and selectively set qualifying tuples in Draft Refinement. This optimization proves particularly beneficial for BETWEEN operators ($c_1 \leq x \leq c_2$): With shortcut enabled, only one Draft Generation and one Draft Refinement iteration are required. Without it, two iterations per boundary ($x \geq c_1$ and $x \leq c_2$) plus a final bitwise AND operation become necessary. Leveraging LiveBin's Localization optimization, which substantially reduces Draft Refinement overhead, we can configure a higher activation threshold than BinDex/Cabin, thereby achieving broader applicability of the Shortcut optimization.

4.2 Hierarchical Index Adjustment

This section presents LiveBin's methodology for maintaining index efficacy through metadata timeliness enhancement. We implement a hierarchical adjustment strategy that optimizes scan acceleration while minimizing systemic efficiency degradation. This aligns with the index versioning principles discussed in Section 2.3.

The first level involves appending entries to the tails of the Delete List and Update List. When modification operations occur, we construct only lightweight entries documenting the changes and append them to the respective lists' tails. Since appending occurs exclusively at list tails, these operations maintain mutually non-blocking reads and writes, achieving superior concurrency.

The second level comprises lightweight adjustments to Delete List and Update List, which removes obsolete versioned entries. We synchronize obsolete deletion records from the Delete List to the Valid Draft and remove these outdated entries from the Delete List. Meanwhile, entries

corresponding to obsolete versions in the Update List are also deleted. This adjustment constrains the lists' length, thereby reducing the time spent scanning them. The adjustment at this level can adopt the lazy strategy and can be combined with the garbage collection mechanism[6, 23].

The third level involves moderate adjustments, specifically sub-index reconstruction. While the first two levels modify metadata related to data updates and deletions, they do not refresh filtering metadata (Filter Bit Vector and Position Array), leading to accumulated outdated information. Sub-index reconstruction is minimally interfering and blocking: (1) Independent sub-indexes allow reconstructing one sub-index without affecting scan operations on others; (2) Leveraging LiveBin's time-anchored design, ongoing scans continue using the legacy sub-index version until reconstruction completes, after which the new version is activated and the old version is recycled. (3) Given the spatial locality in data modifications—where most changes are concentrated within specific sub-indexes—only these impacted sub-indexes undergo reconstruction. Consequently, reconstruction overhead for sub-indexes is significantly lower than for monolithic indexes. Reconstructed sub-indexes refresh underlying metadata, accelerating subsequent scans. The adjustment at this level can be triggered either when modifications to a sub-index reach a predetermined threshold or during system idleness.

4.3 Complexity Analysis

This section establishes analytical models for LiveBin's scan performance to quantitatively evaluate the efficacy of LiveBin's optimization techniques. Since the processing flow is identical for each sub-index in LiveBin, the analyses presented here are described with respect to a single sub-index. All notations and their semantic definitions are summarized in Table 1.

Table 1. Notations in the complexity analysis

Term	Description
N	number of values
M	number of areas
r	rowID bits
t_{seqr}	time to sequentially read a byte
t_{seqw}	time to sequentially write a byte
t_{ranw}	time for a random write
D	length of Delete List
U	length of Update List
V	SIMD operation's bit vector length
t_{simd}	time of a SIMD logical operation

Scan performance of each sub-index in LiveBin comprises three temporal components: filter processing time, deletion handling time, and update resolution time. The total scan execution time is formally expressed as:

$$T_{total} = T_{filter} + T_{delete} + T_{update} \tag{1}$$

For the filter processing component, LiveBin's sub-index employs binary search with $O(logM)$ complexity during the Area Identification phase to locate the target area. During Position Array Probe, an $O(log\frac{N}{M})$ binary search identifies the discrepancy between the draft and final filtered result bit vector, where $\frac{N}{M}$ represents the number of elements per area. The Draft Generation phase requires $(\frac{Nt_{seqr}}{8} + \frac{Nt_{seqw}}{8})$ time to identify and copy the nearest precomputed filter bit vector as the draft. The spatial heuristic selects the preceding area's filter bit vector when the predicate

resides in the first half of the current area, and the subsequent area's filter bit vector otherwise, yielding an average refinement requirement of $\frac{N}{4M}$ bits. The Draft Refinement phase then performs sequential reads of $\frac{N}{4M}$ rowIDs from the Position Array and executes bit-flipping operations on the draft, consuming $(\frac{N}{4M} * \frac{rt_{seqr}}{8} + \frac{Nt_{ranw}}{4M})$ time.The total predicate processing time is:

$$T_{filter} = O(logN) + \frac{N}{8}(t_{seqr} + t_{seqw}) + \frac{N}{4M}(\frac{r}{8}t_{seqr} + t_{ranw}) \tag{2}$$

For the deletion handling component, LiveBin first identifies delta deletions occurring between the current transaction and the Valid Draft through binary search in the Delete List, requiring $O(2logD)$ time. Subsequently, generating the Valid Mask involves copying the Valid Draft with a time cost of $\frac{N}{8}(t_{seqr} + t_{seqw})$, followed by applying identified delta deletions by setting corresponding bits to 0 in the Valid Mask, which consumes $\frac{D}{4}(\frac{r}{8}t_{seqr} + t_{ranw})$ time for random-access modifications. $\frac{D}{4}$ is the average number of delta deletions. Finally, the filtered result bit vector is combined with the Valid Mask via bitwise AND operations, executed in $(\frac{N}{V}t_{simd})$ time through vectorized processing.The total deletion handling time is:

$$T_{delete} = O(2logD) + \frac{N}{8}(t_{seqr} + t_{seqw}) + \frac{D}{4}(\frac{r}{8}t_{seqr} + t_{ranw}) + \frac{N}{V}t_{simd} \tag{3}$$

For the update resolution component, LiveBin first filters update operations visible to the current transaction based on T_{start} and T_{end}, a comparison process with a latency of $O(logU) + \frac{U}{V}t_{simd}$. The filtered entries (averaging $\frac{U}{2}$) undergo predicate evaluation against either their AreaID or updated value, followed by bit setting in the result bit vector. This stage incurs a latency of $\frac{U}{2}(t_{seqr} + t_{ranw})$. The total update resolution time is:

$$T_{update} = O(logU) + \frac{U}{V}t_{simd} + \frac{U}{2}t_{seqr} + \frac{U}{2}t_{ranw} \tag{4}$$

Discussion. In the complexity model, the terms containing t_{ranw} dominate, corresponding to the random memory access pattern. LiveBin mitigates these costs through two optimizations: (1) Localization constrains random memory access ranges to reduce per t_{ranw} overhead. (2) A hierarchical adjustment strategy dynamically shortens Update List and Delete List lengths, thereby diminishing the t_{ranw} coefficient (U and D in Equation 3 and Equation 4).

5 Experimental Analysis

In this section, we evaluate the performance of LiveBin under extensive workloads and configurations. The source code is available at https://github.com/fducslzk/livebin2025.

5.1 Experimental Setup

Experimental Configuration. All experiments are conducted on a machine equipped with a 2.6 GHz AMD EPYC 7K83 processor (16 cores, 32 MB L3 cache). The machine is equipped with a 1TB disk and 64GB DRAM. The operating system is 64-bit CentOS 8 with Linux kernel version 5.4.241-1. The programs are compiled using g++ 8.5 with optimization flag -O3 and SIMD flags (-mavx, -mavx2, -mbmi1, -mbmi2). All experiments are performed in main memory.

Solutions to Compare. We compare LiveBin against the following methods: Binned Scan Indexes (BinDex[26], Cabin[11]), tree-structured indexes (B+ Tree[12], BwTree[25]), SIMD accelerated sequential scanning (SIMD-Scan[41]), and the lossy compression index Column Sketches[15]. We obtain BinDex and Cabin source codes from their repositories. For others, we faithfully implement these techniques according to their papers, applying all feasible optimizations. All methods output a bit vector as the scan result.

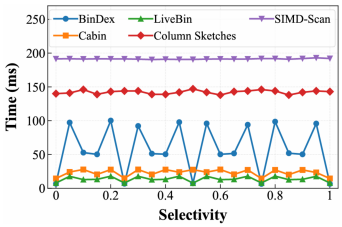


Fig. 9. Scan performance at different selectivities

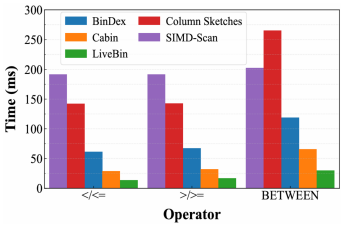


Fig. 10. Scan performance of different operators

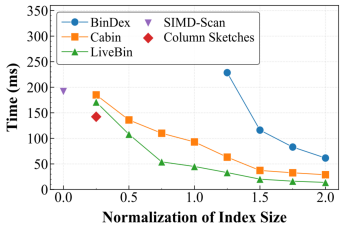


Fig. 11. Space-time analysis for scans

Workloads. We generate columns containing one billion values. For static workloads, we consider two distribution scenarios: uniform and skewed distributions. For uniform distribution, values are uniformly distributed over the full value range. For skewed distribution, values follow a Zipf distribution. We manipulate scan selectivity by setting predicate constant c.

For MVCC environment evaluations, we conduct two primary experiments: First, comparing scan time across methods under varying levels of data modifications. This experiment evaluates scan performance when handling multi-version information. Second, measuring throughput of various methods under hybrid workloads. This experiment assesses comprehensive processing capabilities for hybrid workloads.

5.2 Performance on Static Workloads

This subsection evaluates the scan performance on a static dataset of 1 billion 32-bit integers. All experiments except the selectivity-controlled study measure average scan latency with selectivity increasing from 0 to 100% in 1% increments. For each experiment, we conducted 10 independent trials and used the average outcome as the final result.

Performance of Varying Selectivities. Figure 9 presents the scan performance under the $\leq$ operator across varying selectivity levels. Binned Scan Indexes are configured at twice the original data size. We control the selectivity by varying the predicate constant. The number of refinements equals the count of elements between the predicate constant and the area boundary. As the predicate constant increases from the start of an Area to its end and enters the next Area, the number of refinements first grows and then drops, creating the observed periodic pattern in the scan time of Binned Scan Indexes. BinDex exhibits a more pronounced fluctuation because each refinement is non-localized and incurs higher overhead. Consistent with prior experimental findings, Binned Scan Indexes outperform Column Sketches and SIMD-Scan. The results show that LiveBin outperforms its counterparts across all selectivity levels. Furthermore, LiveBin demonstrates robustness, as evidenced by its smaller vertical amplitude. The maximum latency gap between its fastest and slowest queries is 10.1 ms, compared to 13.3 ms for Cabin and 94.2 ms for BinDex.

Scan with Different Operators. Figure 10 evaluates scan performance across different filter operators. For $>$ and $\geq$ operators, results are derived by applying bitwise negation to the outputs of $\leq$ and $<$ operations. The BETWEEN operator decomposes the predicate $c_1 \leq x \leq c_2$ into $x \geq c_1$ and $x \leq c_2$, thus requiring approximately twice the execution time of a single $\leq$ operation. LiveBin demonstrates performance advantages: it achieves 3.94× and 1.87× speedups compared to BinDex and Cabin under the $>$ operator, and 3.98× and 2.20× speedups for the BETWEEN operation.

Index Space-time Analysis. Figure 11 presents the space overhead and corresponding scan performance under the $\leq$ operator for five methods. The x-axis indicates the index size normalized to the column data size. BinDex can only be constructed when x > 1, as it does not support partial storage of Position Array. For x < 1, LiveBin adopts partial Position Array storage and sets the

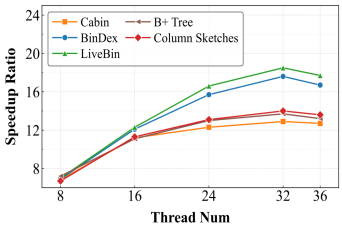


Fig. 12. Comparison of multi-thread scalability

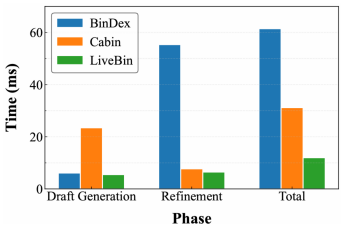


Fig. 13. Scan performance at different phases

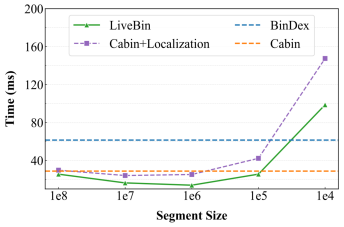


Fig. 14. Scan performance with different segment sizes

segment size to 50,000 to enable 16-bit Position Array. The results demonstrate that when the memory budget is very limited (x = 0.25), Column Sketches achieve the best performance. However, once x reaches 0.5, LiveBin becomes the top-performing method. For instance, at x = 1.25 (where each Binned Scan Index can store 8 additional bit vectors), LiveBin achieves 2.21× and 4.3× speedups over Cabin and Column Sketches, respectively. At x = 2 (with 32 additional bit vectors), the speedups are 2.1× and 13.6× over Cabin and Column Sketches. In summary, LiveBin reduces random access overhead through Localization while decreasing space consumption via a partial storage strategy. The combination of these techniques enables it to achieve a better time-space trade-off.

Multi-thread Scalability. We conducted multi-thread scalability experiments on a machine equipped with a 2.70 GHz Intel Xeon Platinum 8374C CPU (36 cores, 54MB L3 cache, 100GB DRAM). The amount of data in this experiment was 5 billion. Experimental results are presented in Figure 12. LiveBin demonstrates the best scalability, as Localization enables logically and spatially independent sub-indexes, allowing scan tasks per sub-index to be assigned to individual threads, thereby avoiding data contention. However, its speedup ratio decreases at 36 threads due to memory bandwidth saturation and the reduced workload per thread. In contrast, BinDex, Cabin, and B+ Tree exhibit limited scalability owing to bit-block contention when setting their result bit vectors. Although Column Sketches experiences no data contention in multi-threading, it processes the largest data volume per thread, making it more susceptible to memory bandwidth saturation.

Analysis of Different Scan Phases. To investigate the source of LiveBin's performance advantages, we profile the time consumption during Draft Generation and Refinement phases for all three Binned Scan Indexes under a 2.0× space budget (32 additional bit vectors). In this configuration, Cabin's optimal parameters are code width=5 and group=6, partitioning the Position Array into 180 areas, while BinDex and LiveBin partition it into 33 areas.

Experimental results are shown in Figure 13. In the Draft Generation phase, BinDex and LiveBin simply copy specific filter bit vectors as draft results, whereas Cabin must traverse 5 bit vectors and compute to obtain drafts. This fundamental difference makes Cabin incur the highest time cost in this phase. During Refinement, Cabin requires fewer refinement iterations due to its finer-grained Position Array partitioning. Although LiveBin employs coarser partitioning than Cabin, its Localization technique confines refinement operations to smaller memory ranges, thereby reducing per-iteration random memory access costs. This enables LiveBin to significantly reduce Refinement phase latency without introducing additional Draft Generation overhead. The combined benefits demonstrate that LiveBin effectively addresses Refinement phase bottlenecks while maintaining efficient draft generation.

Influence of Segment Size. The critical parameter in the Localization technique is the segment size. We further investigate LiveBin's scan performance under varying segment sizes, as shown in Figure 14. As the segment size decreases (to no less than 1e6), scan latency progressively improves. This occurs because smaller segments enhance the spatial locality of random memory accesses.

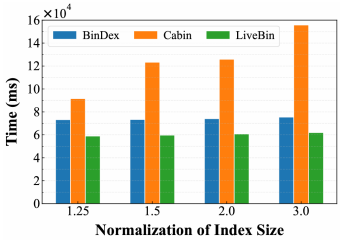


Fig. 15. Comparison of construction time

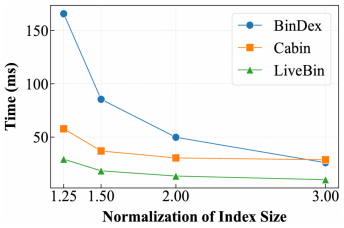


Fig. 16. Scan performance on datasets with skewed distribution(Zipf=1)

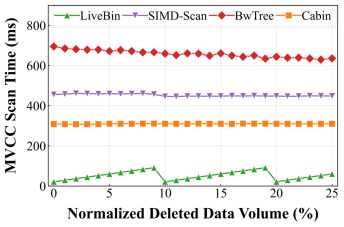


Fig. 17. Scan at different deleted data volume

Notably, when the segment size reaches 1e4, LiveBin's scan performance becomes inferior to that of BinDex. At this configuration, the increased number of sub-indexes introduces substantial overhead from Position Array probes (required per sub-index), which becomes the dominant cost factor. Moreover, we apply the Localization technique to Cabin (denoted as Cabin+Localization in the figure). Results show Localization only provides limited improvement for Cabin. This occurs because Localization primarily optimizes the refinement phase, while another of Cabin's bottlenecks lies in Draft Generation. The above results indicate that Localization requires specific parameter settings and is not a universally applicable technique for all Binned Scan Indexes.

Build Time. Figure 15 compares the construction time of the three indexes under varying space budgets. Both LiveBin and BinDex employ a space budget-agnostic construction algorithm. In contrast, Cabin's construction time grows with increased space allocation due to its use of more groups. The results demonstrate that LiveBin achieves superior scan performance while introducing no additional construction overhead.

Scan Performance on Other Datasets. We further evaluate the Binned Scan Indexes on skewed distributions (Zipf=1), with Figure 16 showing their scan performance under varying space budgets. The comparative results remain consistent with prior experiments on uniform distributions. Due to space constraints, we omit results for different data widths and additional distributions, as LiveBin's optimization strategies demonstrate general applicability across these scenarios.

5.3 Performance under MVCC

This subsection presents two kinds of experiments. The first evaluates scan efficiency under MVCC, aiming to test each method's capability in handling multi-versioned data when scanning. The second examines overall performance under hybrid workloads, designed to assess the integrated behavior of each method in dynamic workload scenarios.

Scan Performance at Different Modified Data Volumes. We first evaluate the MVCC scan performance of various methods under different amounts of modified data. Specifically, one thread (called UDI thread) continuously synchronizes data changes to the multi-version maintenance structures of each method. Meanwhile, another thread (called Scan thread) executes scan operations on the modified states and reports both the current modified data volume and the corresponding scan time (representing the x-axis and y-axis values in Figures 17, 18, and 23). The x-axis represents the ratio of modified data volume to the original data volume. The reported scan time includes both base scanning and version processing overhead. This experiment aims to assess each method's capability in handling multi-version information during scans.

Figure 17 illustrates the scan latencies under varying amounts of deleted data. SIMD-Scan and Cabin exhibit nearly constant latency because deletions only modify timestamps of corresponding

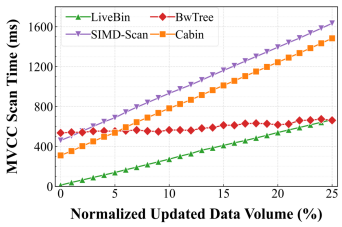


Fig. 18. Scan at different updated data volume

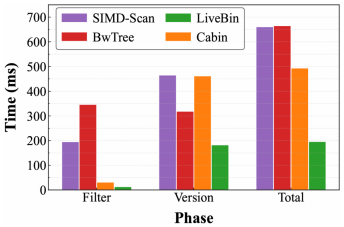


Fig. 19. Performance at different phases

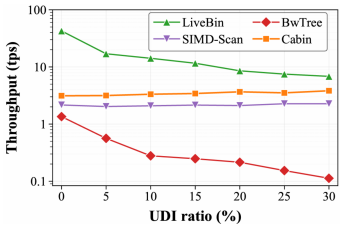


Fig. 20. Single-threaded throughput under different UDI ratios

elements without altering the processed data volume. BwTree's scan time decreases with more deleted data: Deletions modify node timestamps, reducing the number of leaf nodes satisfying temporal constraints and thereby minimizing random accesses during final table lookups. LiveBin employs Delete List to record deletions. The jagged pattern in the figure results from the UDI thread synchronizing deletion records from the Delete List to the Valid Draft at every 10% deleted data volume. During this process, while new deletions introduce proportional random access overhead, the Valid Draft mechanism converts most random accesses into sequential bitwise AND operations. This design confines deletion-induced overheads to bounded thresholds to preserve scan efficiency with average speedups of 11× and 7.4× over BwTree and SIMD-Scan.

Figure 18 presents the scan latency under varying amounts of updated data. BwTree exhibits minimal scan efficiency degradation, as its tree structure enables timely insertion of updated values into leaf nodes. In contrast, the flat and clustered storage schemes of SIMD-Scan and Cabin require separate storage of update metadata, leading to proportional increases in random access overhead. Although LiveBin also requires additional storage for update information, its Update List design enables efficient processing of such data. By leveraging AreaID filtering and temporal filtering, LiveBin rapidly isolates the required version information while minimizing random memory accesses. Overall, LiveBin achieves average speedups of 2.1× over BwTree and 3.1× over Cabin, which is attributed to its efficient predicate evaluation and update information processing.

Analysis of Different Phases of MVCC Scan. Figure 19 presents the average latency breakdown across the two scan phases under MVCC. The Filter phase identifies base data satisfying predicates, while the Version phase processes versioned information (timestamps and multi-version metadata). LiveBin and Cabin exhibit significantly faster performance in the Filter phase. Cabin achieves 6.1× speedup over SIMD-Scan during the Filter phase. However, due to unoptimized multi-version handling, Cabin's acceleration ratio drops from 6.1× to 1.3× under MVCC when compared to SIMD-Scan. By leveraging Valid Draft and Update List, LiveBin effectively reduces multi-version data processing overhead, achieving 3.3× speedup against SIMD-Scan. This demonstrates the necessity of designing a specialized MVCC support structure.

Query Throughput on UDI Hybrid Workloads. We next compare the throughput of the four methods under hybrid workloads. In this experiment, the hybrid workload consists of four transaction types: inserting 1% of the base elements, deleting 1% elements, updating 1% elements, and executing scan queries. Each thread performs certain numbers of transactions, with UDI transactions executed according to the specified UDI ratio and the remaining transactions dedicated to scan queries. This experiment aims to assess the overall performance of various methods when handling hybrid workloads.

We first evaluate the throughput under single-thread execution with varying UDI ratios, as shown in Figure 20. Overall, LiveBin achieves average throughput 6.9× and 4.4× higher than SIMD-Scan and Cabin respectively. This superiority derives from LiveBin's rapid MVCC-Scan execution and

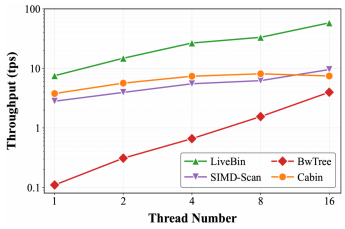


Fig. 21. Throughput at different numbers of threads

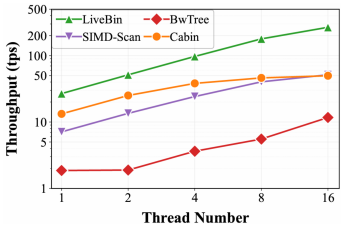


Fig. 22. Throughput on TPC-H Q6 workload

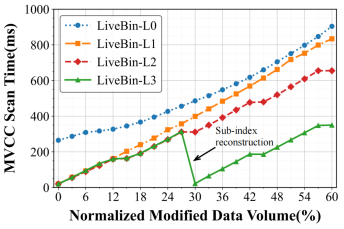


Fig. 23. Benefit of hierarchical adjustment in LiveBin

lightweight UDI operation processing. LiveBin exhibits throughput degradation with increasing UDI ratios due to two factors: (1) its data modification operations incur higher overhead than scan queries, and (2) the accumulation of multi-version metadata prolongs scan processing time.

We next fix the UDI ratio across all threads and compare the throughput scalability of the four methods as thread count increases. The experimental results are shown in Figure 21. Leveraging its well-engineered multi-version management architecture, LiveBin achieves orders-of-magnitude higher throughput than SIMD-Scan and BwTree in multi-threaded configurations. Two factors constrain the multi-thread scalability of these methods: (1) increased write contention among concurrent threads, and (2) the accumulation of version metadata from write operations, which degrades subsequent scan performance across all threads.

Query Throughput on TPC-H Workloads. We further evaluate the throughput of these methods on the TPC-H workload (scale factor = 20 with RF1 and RF2), which is a common analytical workload. RF1 (RF2) inserts (deletes) tuples equivalent to 0.1% of the original data volume in batch. Each thread executes the same workload, which comprises 98% Q6, 1% RF1, and 1% RF2. As shown in Figure 22, LiveBin achieves 3.6× and 4.6× higher throughput than Cabin and SIMD-Scan on average. These gains primarily stem from: (1) LiveBin's rapid predicate evaluation efficiently handling the high volume of Q6; (2) Its embedded multi-version management structure swiftly processing versioning overhead from data modifications.

Benefits of Hierarchical Adjustment. LiveBin rapidly processes multi-version information under MVCC through hierarchical structural adjustment. We next evaluate the MVCC scan performance under varying modified data volumes when different levels of LiveBin's index adjustment mechanisms are enabled. The experimental setup remains consistent with that described in Section 5.3 "Scan Performance at Different Modified Data Volumes". Figure 23 demonstrates scan performance across progressively enabled adjustment levels: LiveBin-L0 uses external version metadata; LiveBin-L1 incorporates Delete List and Update List; LiveBin-L2 adds Valid Draft and garbage collection; while LiveBin-L3 implements sub-index restructuring. Results show L1 significantly outperforms L0 at low volumes of modified data by avoiding exhaustive version chain traversal, whereas L2 surpasses L1 at mid-to-high volumes of modified data through effective list length control. The sudden drop in the L3 curve at the 30% data modified occurs because the UDI thread triggers a sub-index reconstruction once the ratio reaches the preset 30% threshold. This process refreshes index metadata to accelerate subsequent scans. Notably, the overhead of reconstructing a sub-index is acceptable. In our experimental evaluation, reconstructing a single sub-index takes approximately 61 ms, whereas a full scan operation requires about 34 ms.

5.4 System Integration

In this section, we integrate LiveBin, Cabin, BinDex, and Column Sketches into DuckDB—a high-performance OLAP database. Their practical applicability in OLAP systems is then evaluated

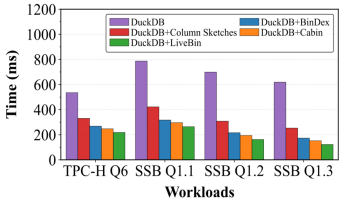


Fig. 24. Performance on static data in DuckDB

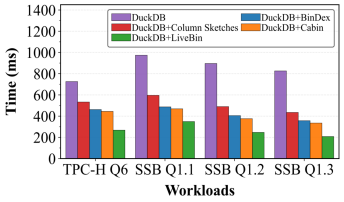


Fig. 25. Performance under MVCC in DuckDB

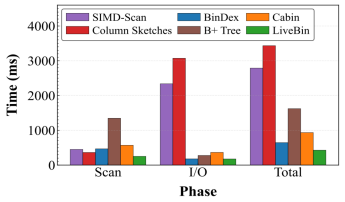


Fig. 26. Scan performance on SSD (cold data)

using the TPC-H and SSB benchmarks. We configure the scale factor at 20 and execute Q6 for TPC-H and Q1.1, Q1.2, Q1.3 for SSB. These four queries are selected because their scan operations constitute the dominant portion of overall execution. Our integration with DuckDB is designed to be minimally invasive. Specifically, we introduce a new scan operator that utilizes LiveBin for predicate evaluation. A detailed technical report describing the integration process has been made available in our repository.

Performance on Static Data. Figure 24 displays their performance on static data. For TPC-H Q6, LiveBin-enhanced DuckDB completes the query in 218 ms, achieving a 2.45× acceleration over native DuckDB. Although DuckDB demonstrates leading OLAP performance, its scan efficiency is significantly boosted by LiveBin. We observed only a 1.13× speedup for LiveBin+DuckDB over Cabin+DuckDB on TPC-H Q6, since other operations like projection and aggregation constitute a notable portion of the execution time. When considering only the scan procedure, LiveBin+DuckDB took 53ms compared to 83ms for Cabin+DuckDB, yielding a 1.57× speedup. These results confirm that integrating LiveBin further improves DuckDB's scan performance, demonstrating LiveBin's practical applicability in high-performance OLAP databases.

Performance under MVCC. Figure 25 demonstrates the query performance under MVCC with 10% modified data volume. The need to compare transaction timestamps with version timestamps to select appropriate versions introduces additional overhead under MVCC. Across the four workloads tested, native DuckDB incurs an average additional latency of 196 ms due to MVCC requirements, whereas LiveBin requires only 77 ms. For TPC-H Q6, LiveBin+DuckDB achieved a 2.71× speedup over native DuckDB under MVCC, while Cabin+DuckDB achieved only a 1.63× speedup over native DuckDB. These results confirm that LiveBin's version-aware indexing efficiently handles multi-version information, mitigating scan performance degradation under MVCC.

5.5 Applicability on SSDs

The increasing performance and affordability of SSDs enhance their value for analytical workloads. We further evaluate LiveBin's scan performance on SSDs to explore its practical utility. Following prior experimental setups[1, 18, 27, 34], we adopt `O_DIRECT` for page-aligned data reads and implement an LRU buffer pool manager with a 4 KB page size. The tests were run on a 1 TB SSD WDS100T2G0A-00JH30 (SATA 3.2 interface, 545 MB/s sequential read). Based on existing results[14, 17, 27, 37] that B+ Tree remains competitive for scan performance on SSDs, we include B+ Tree as an additional baseline. The dataset contains one billion int32 integers. We use 10% selectivity, allocate twice the raw data size for Binned Scan Indexes, and optimize all methods for efficient page access.

Figure 26 presents cold data results with a buffer pool sized at 10% of the raw data. LiveBin and BinDex achieve the lowest I/O time by loading only the predicate-relevant area's Position Array and Filter Bit Vector. This is further reflected in the number of pages read—a key factor determining

I/O time: LiveBin and BinDex read 61K pages, Cabin 126K, B+ Tree 100K, SIMD-Scan 976K, and Column Sketches 1220K. The pure scan time of different methods is affected to varying degrees by their I/O access patterns. For example, Column Sketches achieves a lower scan time than BinDex and Cabin because it performs fully sequential reads, which minimize system-level interference.

In summary, LiveBin demonstrates leading performance on SSDs, primarily attributed to two factors: (1) Loading Batch on-demand – it loads and processes sub-indexes in batches on-demand, significantly reducing memory pressure; and (2) Reduced I/O volume – it only needs to load the Position Array and Filter Bit Vector of the area relevant to the predicate.

6 Conclusion

In this paper, we introduce LiveBin, a novel Binned Scan Index that significantly improves scan performance by addressing key limitations of existing designs. LiveBin employs Localization to minimize random memory access overhead during the refinement phase and adopts a partial-storage strategy to reduce its memory footprint. Furthermore, by embedding lightweight versioning information directly within the index structure, LiveBin effectively maintains high scan efficiency under MVCC, a common challenge for prior binned indexes. Experimental evaluation demonstrates that LiveBin offers a robust and efficient indexing solution for in-memory analytical databases.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions. This work was supported by NSFC (Grant No. U24A20232 and 62272106).

References

[1] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-tree: approximate tree indexing. *Proc. VLDB Endow.* 7, 14 (Oct. 2014), 1881–1892. doi:10.14778/2733085.2733094

[2] Manos Athanassoulis, Zheng Yan, and Stratos Idreos. 2016. Upbit: Scalable in-memory updatable bitmap indexing. In *Proceedings of the 2016 International Conference on Management of Data*. 1319–1332.

[3] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading.

[4] Satish Tanaji Bhosale, Tejaswini Patil, and Pooja Patil. 2015. Sqlite: Light database system. *Int. J. Comput. Sci. Mob. Comput* 44, 4 (2015), 882–885.

[5] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2018. Smooth scan: robust access path selection without cardinality estimation. *The VLDB Journal* 27 (2018), 521–545.

[6] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable garbage collection for in-memory MVCC systems. *Proceedings of the VLDB Endowment* 13, 2 (2019), 128–141.

[7] David Broneske, Sebastian Breß, and Gunter Saake. 2013. Database scan variants on modern CPUs: A performance study. In *International Workshop on In-Memory Data Management and Analytics*. Springer, 97–111.

[8] Michael J Cahill, Uwe Röhm, and Alan D Fekete. 2009. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–42.

[9] Guadalupe Canahuate, Michael Gibas, and Hakan Ferhatosmanoglu. 2007. Update conscious bitmap indices. In *19th International Conference on Scientific and Statistical Database Management (SSDBM 2007)*. IEEE, 15–15.

[10] Chee-Yong Chan and Yannis E. Ioannidis. 1998. Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA) *(SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 355–366. doi:10.1145/276304.276336

[11] Yiyuan Chen and Shimin Chen. 2024. Cabin: A Compressed Adaptive Binned Scan Index. *Proc. ACM Manag. Data* 2, 1, Article 57 (March 2024), 26 pages. doi:10.1145/3639312

[12] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[13] 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 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 31–46. doi:10.1145/2723372.2747642

[14] A. Fevgas, L. Akritidis, P. Bozanis, et al. 2020. Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. *The VLDB Journal* 29, 2 (2020), 273–311. doi:10.1007/s00778-019-00559-8

[15] Brian Hentschel, Michael S. Kester, and Stratos Idreos. 2018. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 857–872. doi:10.1145/3183713.3196911

[16] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database Cracking.. In *CIDR*, Vol. 7. 68–78.

[17] P. Jin, C. Yang, C. S. Jensen, et al. 2016. Read/write-optimized tree indexing for solid-state drives. *The VLDB Journal* 25, 6 (2016), 695–717. doi:10.1007/s00778-015-0406-1

[18] Peiquan Jin, Chengcheng Yang, Xiaoliang Wang, Lihua Yue, and Dezhi Zhang. 2020. SAL-Hashing: A Self-Adaptive Linear Hashing Index for SSDs. *IEEE Transactions on Knowledge and Data Engineering* 32, 3 (2020), 519–532. doi:10.1109/TKDE.2018.2884714

[19] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. 2008. Row-wise parallel predicate evaluation. *Proceedings of the VLDB Endowment* 1, 1 (2008), 622–634.

[20] 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 2017 ACM international conference on management of data*. 715–730.

[21] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[22] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*. 311–326.

[23] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid garbage collection for multi-version concurrency control in SAP HANA. In *Proceedings of the 2016 international conference on management of data*. 1307–1318.

[24] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

[25] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 302–313.

[26] 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 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 909–923. doi:10.1145/3318464.3380563

[27] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. 2010. Tree indexing on solid state drives. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 1195–1206. doi:10.14778/1920841.1920990

[28] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 289–300. doi:10.1145/2463676.2465322

[29] Guido Moerkotte. 1998. Small materialized aggregates: A light weight index structure for data warehousing. *None* (1998).

[30] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 677–689. doi:10.1145/2723372.2749436

[31] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1493–1508. doi:10.1145/2723372.2747645

[32] Wilson Qin and Stratos Idreos. 2016. Adaptive Data Skipping in Main-Memory Systems. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 2255–2256. doi:10.1145/2882903.2914836

[33] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. doi:10.1145/3299869.3320212

[34] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2011. B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. arXiv:1201.0227 [cs.DB] https://arxiv.org/abs/1201.0227

[35] Lefteris Sidirourgos and Martin Kersten. 2013. Column imprints: a secondary index structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 893–904. doi:10.1145/2463676.2465306

[36] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1115–1126. doi:10.1145/2588555.2610515

[37] Chundong Wang, Gunavaran Brihadiswarn, Xingbin Jiang, and Sudipta Chattopadhyay. 2022. Circ-Tree: A B+-Tree Variant With Circular Design for Persistent Memory. *IEEE Trans. Comput.* 71, 2 (2022), 296–308. doi:10.1109/TC.2020.3047972

[38] Junchang Wang and Manos Athanassoulis. 2024. CUBIT: Concurrent Updatable Bitmap Indexing. *Proc. VLDB Endow.* 18, 2 (Oct. 2024), 399–412. doi:10.14778/3705829.3705854

[39] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, Yujie Wang, Baokai Chen, Chang Cai, Yubin Ruan, Xiaoyi Weng, Shibin Chen, Liang Yin, Chengzhong Yang, Xin Cai, Hongyan Xing, Nanlong Yu, Xiaofei Chen, Dapeng Huang, and Jianling Sun. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proc. ACM Manag. Data* 1, 2, Article 199 (June 2023), 25 pages. doi:10.1145/3589785

[40] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing Database Column Scans with Complex Predicates.. In *ADMS@ VLDB*. 1–12.

[41] 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. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 385–394. doi:10.14778/1687627.1687671

[42] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.* 10, 7 (March 2017), 781–792. doi:10.14778/3067421.3067427

[43] Per Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. arXiv:1201.0228 [cs.DB] https://arxiv.org/abs/1201.0228

Received July 2025; revised October 2025; accepted November 2025