Counting Is All You Need for Instant Tuple Discovery: Enabling Real-Time HTAP in Standalone DBMSs

KYUNGMIN LIM, Seoul National University, Republic of Korea
MINSEOK YOON*, Hanyang University, Republic of Korea
KIHWAN KIM*, Hanyang University, Republic of Korea
ALAN FEKETE, The University of Sydney, Australia
HYUNGSOO JUNG†, Seoul National University, Republic of Korea

HTAP systems aim to unify operational and analytical workloads, yet real-time analytics remains constrained by the overhead of *extract-transform-load* (ETL) operations. Existing solutions often rely on dual-system architectures, incurring substantial resource costs and delays from data reformatting and relocation. We present TracerETL, a progressive ETL framework that enables real-time analytics in standalone DBMSs through instant tuple location discovery during transformation. At its core is Tracer, a counting-based tuple tracking mechanism that constructs a *tuple trace vector* using per-partition counters. This trace vector deterministically encodes each tuple's future relocation path with arrival order across transformation levels, enabling precise data access at any stage—without auxiliary indexes. We implement TracerETL in PostgreSQL and evaluate it against OLAP- and OLTP-optimized DBMSs. Experimental results show that PostgreSQL with TracerETL accelerates real-time HTAP queries by up to 127×, while efficiently handling progressive data conversion in a standalone DBMS.

CCS Concepts: • **Information systems → DBMS engine architectures**; **Online analytical processing engines**.

Additional Key Words and Phrases: Real-time HTAP, ETL, tuple location discovery

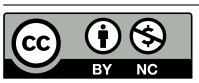
ACM Reference Format:
Kyungmin Lim, Minseok Yoon, Kihwan Kim, Alan Fekete, and Hyungsoo Jung. 2025. Counting Is All You Need for Instant Tuple Discovery: Enabling Real-Time HTAP in Standalone DBMSs. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 310 (December 2025), 28 pages. https://doi.org/10.1145/3769775

1 Introduction

The hybrid transactional and analytical processing (HTAP) paradigm has gained growing attention in the industry, offering to allow summaries and complex data exploration on the up-to-date state, in contrast to older data warehouse approaches which have separate OLAP systems and OLTP ones, and where the analytics are done on a system that *lags* the true situation. It is very challenging to find physical data arrangements that provide good performance for both complex queries that

*These authors contributed equally to this research.
†Corresponding author.

Authors' Contact Information: Kyungmin Lim, kyungmin.lim@snu.ac.kr, Seoul National University, Seoul, Republic of Korea; Minseok Yoon, minseok0yoon@hanyang.ac.kr, Hanyang University, Seoul, Republic of Korea; Kihwan Kim, kihwan@hanyang.ac.kr, Hanyang University, Seoul, Republic of Korea; Alan Fekete, alan.fekete@sydney.edu.au, The University of Sydney, Sydney, Australia; Hyungsoo Jung, hyungsoo.jung@snu.ac.kr, Seoul National University, Seoul, Republic of Korea.



ACM 2836-6573/2025/12-ART310
https://doi.org/10.1145/3769775

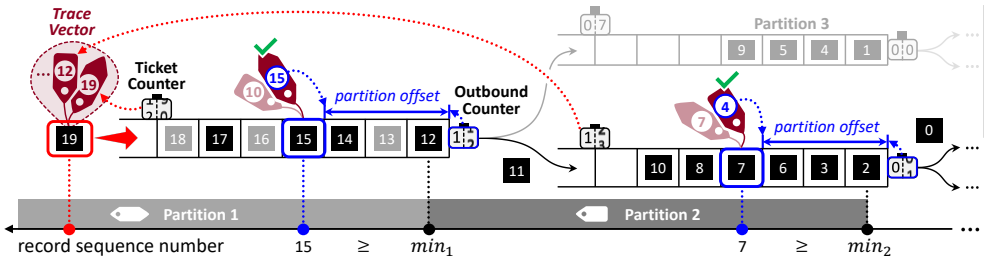


Fig. 1. Counting-based precise tuple location tracking.

typically process a few columns from a significant number of the rows, and for transactions that update the data, perhaps adding a few rows, or modifying them.

Commercial vendors have developed HTAP DBMSs [29, 35, 36, 40, 50, 62, 65, 69]. A common strategy streams fresh data from OLTP engines into *extract-transform-load* (ETL) pipelines that reshape row-oriented records into partitioned, columnar layouts optimized for OLAP. This design underlies cloud-native dual-format HTAP systems, such as AWS Zero-ETL [5, 6] and Google AlloyDB [24], which decouple transactional and analytical workloads via asynchronous transformation.

Recent work from both academia and industry [2, 59, 65] has explored *progressive partitioning* over log-structured storage engines (e.g., LSM-trees), where data is incrementally reorganized to improve I/O efficiency. While effective in distributed settings, these systems often rely on large-scale infrastructure to maintain low-latency transformation or suffer from reduced data freshness due to the *inherent delays of progressive reformatting and relocation*. In contrast, achieving real-time analytics in standalone HTAP DBMSs demands eliminating such delays. As data evolves from an OLTP-optimized row layout to an OLAP-friendly, partitioned columnar format, the system must ensure ***instant tuple accessibility***—guaranteeing that each record remains visible and precisely locatable throughout the transformation process.

To address this challenge, we draw inspiration from *airline ticketing systems*: when passengers check in at their departure city, they are issued all the tickets required for a multi-stop itinerary. As they pass through layover cities, the airline tracks their journey by verifying arrivals against the pre-assigned tickets. Similarly, we propose Tracer, a *counting-based method for precise tuple tracking* during progressive data transformation. Upon insertion, each tuple is assigned a compact *trace vector* composed of ticket numbers issued from *per-partition ticket counters* that deterministically encode its future path and arrival order through the transformation hierarchy. Each partition also maintains a *minimal sequence number* to verify **tuple presence** via comparison with the tuple sequence number and an *outbound counter* to compute the tuple's **partition offset** using the ticket number from the tuple trace vector. Together, this per-tuple trace vector and per-partition metadata enable the system to *precisely locate any record-attribute at any stage of progressive data transformation* (see Figure 1).

We present TracerETL, a physical layout and transformation framework built on Tracer that enables progressive data reformatting with efficient tuple access. Architecturally, TracerETL organizes data across a multi-level layout inspired by LSM-trees, where records are incrementally reshaped from row-oriented input into fine-grained, horizontally and vertically partitioned formats. A key innovation in TracerETL is the use of a compact *trace vector*, which deterministically computes each tuple's physical location via arithmetic over per-partition counters—eliminating the need for lineage tracking, compaction, or auxiliary indexes. This shift from index-reliant lookup to *index-free positional computation* introduces a new abstraction for progressive ETL and HTAP engines. Built on this mechanism, TracerETL enables *in-place updates* over evolving layouts and

offloads obsolete versions outside the primary data path, substantially reducing MVCC-induced write amplification and garbage collection overhead.

Contributions. Our primary contribution is TracerETL, a progressive ETL framework that enables instant tuple location discovery for real-time HTAP in standalone DBMSs.

- *Real-time ETL Framework*: Sections 3.2 and 3.3 detail the TracerETL layout, its progressive transformation algorithms, and the counting-based Tracer mechanism. At the core of Tracer is a compact trace vector built from per-partition counters, enabling instant data access and in-place updates during reformatting.
- *Efficient MVCC and Robust Recovery*: Section 3.4 describes how TracerETL decouples version management using an external version manager and ensures transactional consistency by adapting ARIES for crash recovery.
- *PostgreSQL Integration*: Section 4 presents a full implementation in PostgreSQL 15, extending B+tree indexing, query planning, and optimizer components to support partition-aware execution. Background workers manage continuous data transformation, and all components operate over disk-resident structures.
- *Evaluation*: Section 5 presents evaluation results, showing that TracerETL improves performance by up to 127× over state-of-the-art designs, with reduced latency and write amplification.

2 Background and Related Work

We derive the core design principles of TracerETL by building on key insights from prior work. This section reviews related research on HTAP database systems through the lens of ETL cost metrics, with a focus on data freshness, I/O overhead, and space amplification—factors that critically impact the viability of real-time analytics in standalone deployments.

HTAP databases with ETL pipelines. A broad spectrum of HTAP systems has been developed in recent years [23, 29, 32, 35, 36, 39, 40, 44, 50, 59, 61, 62]. Many commercial solutions [2, 29, 35, 36, 40, 46, 50, 62, 65, 69] rely on *resource-intensive dual-engine architectures*, where OLTP and OLAP workloads are decoupled via continuous ETL pipelines to achieve performance isolation. A key enabler in these systems is the abundance of computational and storage resources, which makes data partitioning tractable and efficient—especially for large-scale analytics over distributed clusters. In such environments, analytical queries over multi-terabyte datasets can return results within seconds [50, 51], driven by parallel execution on vertically and horizontally partitioned data spread across hundreds of nodes. However, this level of infrastructure is out of reach for resource-constrained standalone DBMSs, where such partitioning strategies remain largely infeasible.

Query acceleration via data partitioning. Physical and logical data partitioning—such as Pando [64]—can accelerate query processing via data skipping and predicate pruning. However, online partitioning incurs I/O amplification and repartitioning delays, especially over dynamic datasets. Prior work on LSM-based systems [16, 18–21, 52, 55, 57, 59, 60] has proved the benefits of log-structured storage engines, which amortize write overhead via batched, append-only updates. Nonetheless, they must trade off lookup latency for write efficiency. Likewise, logical partitioning approaches like Pando incur nontrivial repartitioning costs.

Despite these efforts, analytical queries over write-optimized LSM-trees remains inefficient due to deep search paths across multiple levels. GRF (Global Range Filter) [67], a notable effort to address this, applies a *counting* principle to track the number of flushed sorted runs per level and encodes this structural *shape* metadata into a global range filter. This MVCC-aware *shape encoding* enables early pruning by identifying sorted runs likely to contain the target key—regardless of version lineage—thereby improving range filter efficiency. However, once a candidate run is identified, GRF still depends on the SSTable's internal index for precise tuple resolution. Thus, GRF improves pruning efficiency but does not eliminate the need for index-based lookup, nor does it address space

amplification from MVCC. The underlying *out-of-place update* semantics of LSM-trees, coupled with compaction-driven garbage collection, continue to incur MVCC-induced space amplification in existing LSM-based systems. As such, achieving instant tuple location discovery remains an open challenge—and a critical enabler—for MVCC-efficient LSM-tree-based systems in HTAP settings.

Version management in MVCC and log-structured systems. Progressive partitioning strategies based on LSM-trees, which at a given time, store the data of a given table across several physical structures with different partitioning choices; a logical tuple will be moved from time to time between these structures. In many systems, this is further complicated by keeping multiple versions of a logical tuple, for use in MVCC. Although prior work has addressed version space overhead in traditional RDBMSs [4, 23, 31, 32, 37, 56], they do not readily apply to log-structured architectures. The core limitation stems from deferred compaction in LSM-trees, which delays the reclamation of obsolete versions and leads to space amplification.

To overcome this, TracerETL adopts a hybrid update strategy: *in-place updates* allow obsolete versions to be pruned immediately, reducing version bloat, while *out-of-place inserts* preserve the write efficiency of progressive log-structured transformation. Achieving both goals is non-trivial, but TracerETL resolves this tension by leveraging its instant tuple tracking capability—eliminating the need for full scans or multi-level traversal during updates. This decoupled design enables efficient version management without compromising analytical latency or transactional throughput, making it well-suited for standalone HTAP environments.

3 The TracerETL Architecture

3.1 Design Overview of TracerETL

This section presents an overview of TracerETL. TracerETL is grounded in two core design principles: ❶ progressive, log-structured data transformation, and ❷ tuple location discovery. These principles enable the unification of in-place updates and out-of-place inserts, achieving write efficiency and space compactness within a single, integrated storage layout. Table 1 summarizes the key notations used throughout the paper.

Table 1. Notations used throughout TracerETL.

Semantics	Notations	Description
Partition layout	p_i	A partition file in *PAX-format* [3]
	Z^R	A *record zone* within p_i
	Z^C	A *column zone* within Z^R
Tuple trace vector	K_r	The *partition key* of record r
	S_r	The *global sequence number* for r
	TC_{r,p_i}	Record r's *ticket number* at p_i
Partition counters	TC_{p_i}	The *ticket counter* at p_i
	OC_{p_i}	The *outbound counter* at p_i
	S^{min}_{r,p_i}	The *minimal* S_r, $r \in p_i$

Log-structured, gradual data transformation. Data partitioning is essential for both parallel query execution [2, 9, 11, 41, 42, 65] and accessing smaller datasets satisfying given predicates [64]. However, real-time partitioning in standalone databases is challenging as it can worsen I/O amplification. We adopt log-structured data partitioning, where low-dimensional partitions are progressively refined into high-dimensional ones as new data is ingested. Our approach is inspired by progressive partitioning [2, 59, 65, 69]. By gradually increasing partition dimensionality during reformatting, we can construct high-dimensional layouts without incurring the I/O overhead typically associated with eager re-partitioning.

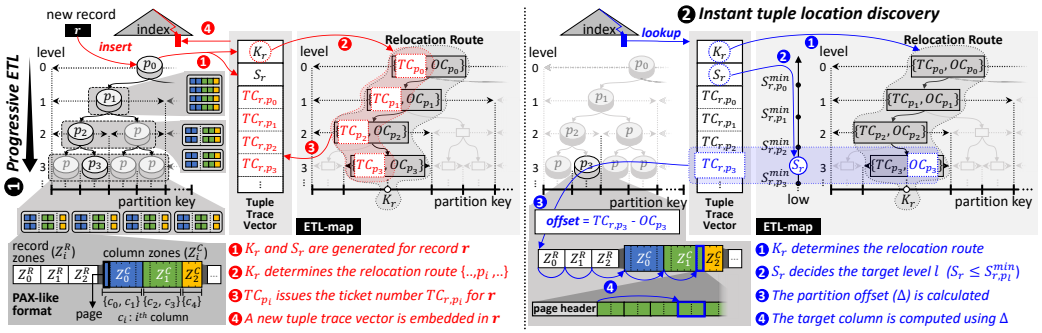


Fig. 2. Trace vector construction and use for instant lookup across transformations in TracerETL.

Tuple location discovery during transformation. Previous work [22, 33, 58, 70] has demonstrated the effectiveness of indirection mapping for efficient data access during transformation. For instance, TreeLine [70] leverages a *pure in-memory* `btree_map` [12] to maintain tuple-to-location mappings, offering fast access at the expense of increased memory consumption and volatility. In contrast, Tracer introduces a tracking mechanism based on *succinct, disk-resident metadata* designed for progressively transformed storage layouts. Each partition p_i —similar to a layover city in an itinerary—maintains three counters: (i) a *ticket counter* (TC_{p_i}), which issues monotonically increasing sequence numbers to arriving tuples, (ii) an *outbound counter* (OC_{p_i}), which records the number of tuples that have moved to *conceptually lower-level* partitions (i.e., *higher-numbered* ones), and (iii) a *minimum sequence number* (S^{min}_{r,p_i}) for efficient presence checks during lookup.

Upon tuple insertion, Tracer assigns the tuple r a sequence of ticket numbers from the ticket counters TC_{p_i} of all partitions along its deterministic relocation route. The ticket number from the root partition, TC_{r,p_0}, serves as the unique tuple sequence number S_r, while the full set {...,TC_{r,p_i},...} forms the core of the ***tuple trace vector*** (see ❶-❹ in Figure 2). Each partition p_i can efficiently check tuple presence by comparing S_r against S^{min}_{r,p_i}. To compute the in-partition offset for r, Tracer subtracts the current outbound counter OC_{p_i} from TC_{r,p_i}, allowing precise physical localization (see ❶-❹ in Figure 2). This resembles an airline ticketing system, where all boarding passes are issued at check-in, and each layover verifies the passenger's progression using those tickets.

Unifying in-place updates and out-of-place inserts. Although progressive transformation [2, 59, 65] reduces I/O overhead, supporting multi-versioning in HTAP workloads remains a fundamental challenge. TracerETL addresses this by enabling *in-place updates* while offloading obsolete versions to an external version manager responsible for garbage collection. This design decouples version management from the storage layout, significantly reducing MVCC overhead. Hence, TracerETL achieves two key objectives, *I/O efficiency* and *instant data access*, without embedding version control logic directly into the core engine.

3.2 Progressive Data Transformation

We adopt partitioning strategies that are well-established in the database community. For horizontal partitioning, we use the *reference partitioning technique* [48], which derives partition keys from foreign-key relationships, enabling logical grouping of related tuples. For vertical partitioning (i.e., columnar reformatting), row-oriented datasets are progressively transformed into column groups based on query access patterns. Hence, frequently co-accessed columns are physically co-located, improving query performance.

3.2.1 ***Data layouts for log-structured transformation.*** TracerETL consists of multi-level partitions, resembling LSM trees, where each partition is represented by a paginated file. Regarding

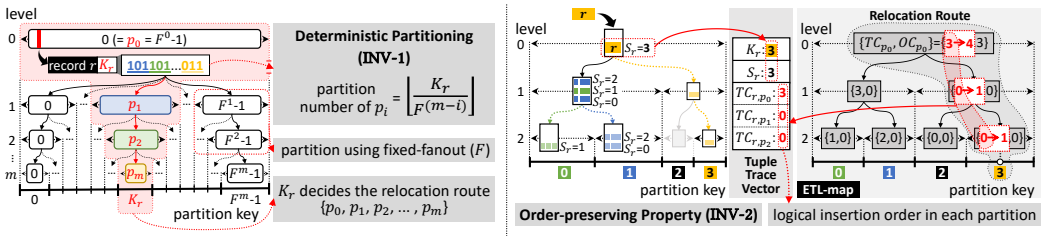
columnar reformatting, we logically divide the file space into column groups, thereby combining the physical partitioning of records with the logical partitioning of columns. As illustrated in Figure 2, lower-level partitions exhibit finer-grained logical vertical partitions represented by column groups. During data transformation, we mitigate random I/O when distributing column data across different pages within the partition file by following the log-structured writing policy. To this end, we utilize zoned data layouts, where a file comprises multiple zoned spaces to store groups of records (record groups) referred to as a record zone (Z^R). Each Z^R internally consists of multiple zoned spaces for column groups, known as a column zone (Z^C), indicating that Z^R encloses a set of column zones, with each Z^C_i storing all the combined columns conforming to a column group format (i.e., cg_i) contiguously. We then append a batch of column zones belonging to a record group into the last Z^R of the partition file, resulting in sequential writes (see ❶ **progressive ETL** in Figure 2).

3.2.2 ***Horizontal data partitioning.*** The reference partitioning scheme divides row datasets into multi-dimensional partitions using columns from *dimension tables*. This requires denormalizing foreign-key relationships to extract column sets—each representing a join path that propagates predicates from dimension to *fact* tables. Rather than storing raw values and incurring space overhead, we introduce *logical denormalization*: each column set is compactly encoded as a *hidden attribute*, which is stored in every table along the join path (e.g., `orders`, `order_line`). Fact tables may inherit multiple hidden attributes, some redundant but derived from distinct paths. These hidden attributes collectively define the multi-dimensional partition space. Upon insertion, new records populate their hidden attributes by consulting the system catalog and copying the encoded values from parent tables. Note that our design assumes static dimension keys; under *slowly changing dimensions* (SCDs) [34], key evolution may invalidate partition mappings and require trace vector reconstruction—incurring nontrivial maintenance overhead. That said, incremental reconstruction of hidden attributes using techniques similar to *virtual columns* [43, 49, 63] can address *Type 2* SCDs (newly added keys), with minimal costs.

3.2.3 ***Vertical data partitioning (columnar reformatting).*** Vertical partitioning aims to colocate semantically related attributes to improve OLAP performance by enabling selective, sequential access to only the required columns—avoiding unnecessary data retrieval. To guide column grouping in our implementation, we apply the following five heuristic rules based on an analysis of the TPC-CH schema: **R1**) group columns frequently modified by updates, **R2**) isolate columns that compose the primary key, **R3**) cluster attributes that serve as foreign keys, **R4**) group low-utility fields such as hidden or auxiliary attributes, and **R5**) place unused columns—irrelevant to both OLTP and OLAP queries—into a separate group. Although these rules are effective, they are not exhaustive. We leave column compression and learned layout optimization as future work, as they are beyond our primary focus.

After identifying the target column groups based on the given schema and queries, we either divide rows into the target column groups at once or gradually generate finer-grained ones. To support columnar reformatting, we leveraged publicly available columnar grouping formats [14] developed by Citus Data as an extension for PostgreSQL, initially presented in their research work [17]. Citus manages metadata for columnar grouping formats in relation rather than in the system catalog. Similarly, TracerETL also maintains per-table ETL metadata, ***ETL-map***, for storing metadata on data formats. These *ETL-map*s keep columnar grouping formats for each level, which are then utilized during scan operations.

3.2.4 ***Log-structured repartitioning rules.*** To enable fast data access, TracerETL adheres to two key invariants. The first, **INV-1**, enforces *deterministic data redistribution* based on a fixed set of attributes in the partition key. It guarantees that each tuple follows a predetermined relocation route



(a) Deterministic partitioning for a predictive route

(b) Order-preserving repartitioning for building a trace vector

Fig. 3. Key invariants in TracerETL: (i) deterministic partitioning and (ii) order-preserving property.

throughout the transformation process, allowing us to identify, in advance, all target partitions the tuple will travel (see Figure 3a).

The second invariant, **INV-2**, ensures *order preservation* during repartitioning; i.e., tuples are redistributed to lower-level partitions in an order-preserving manner. We maintain **INV-2** by sequentially repartitioning each Z^R within each partition. With all partitions tracking the minimum sequence number of their records, **INV-2** allows for precisely identifying a tuple's target partition at any point during transformation (§3.3.2).

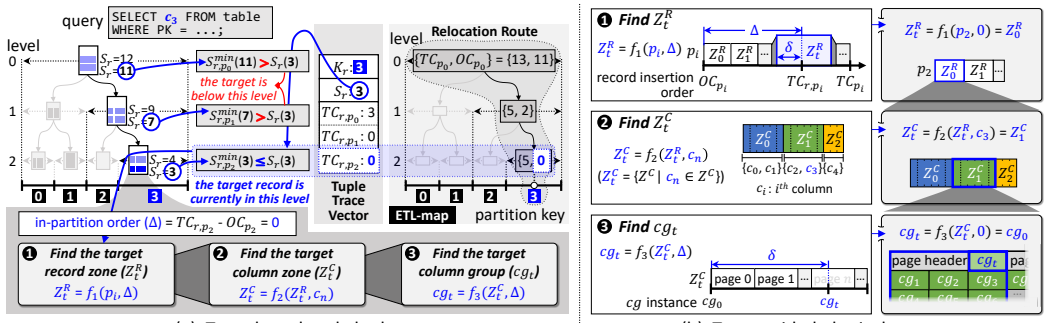
TracerETL's progressive partitioning preserves these invariants. Specifically, record items in upper-level partitions are redistributed deterministically by **INV-1** and appended in the unit of Z^R to lower-level partitions, maintaining **INV-2** (see Figure 3b). Each record r is physically assigned to a lower-level partition based on a horizontal partition key K_r (i.e., hidden attributes). The leveled partition tree follows a fixed fanout strategy with fanout factor F, such that a record is mapped to its level-i partition using $p_i = \lfloor \frac{K_r}{F^{(m-i)}} \rfloor$, for $i = 0 \cdots m$. This formula determines the target partition at each level based on the incremental extraction of bits from K_r. This fixed-fanout policy ensures that an upper-level partition deterministically maps to a known set of lower-level partitions. Figure 3a illustrates our deterministic, progressive partition strategy.

Unlike horizontal partitioning, in-partition columnar reformatting reorganizes coarser-grained column groups into finer-grained ones—either incrementally during each repartitioning stage (when many groups exist) or all at once otherwise. As a result, lower-level partitions store tuples that satisfy more selective dimension attributes, organized into finer column groups. This design, however, introduces a trade-off. Middle-level partitions, which retain broader attribute ranges due to lower dimensionality, may include more generalized data than needed for query predicates. This reduces read efficiency in exchange for lower write amplification. Thus, queries may scan more tuples from these partitions than necessary. For efficiency, dedicated workers trigger repartitioning.

3.3 Counting-based Tuple Tracking

TracerETL introduces a counting-based mechanism for instant tuple discovery during progressive data transformation. At its core is the *trace vector*, a compact structure assigned to each tuple at insertion time. This section details how trace vectors and per-partition counters enable fast data access without auxiliary indexes.

3.3.1 ***Trace vector construction.*** To support instant tuple tracking during progressive transformation, TracerETL assigns each tuple a trace vector upon insertion. This trace vector consists of a compact sequence of ticket numbers $\{TC_{r,p_i}\}$, each issued by the corresponding partition p_i along the tuple's deterministic relocation path. The ticket from the root partition (TC_{r,p_0}) serves as the global insert sequence number S_r, which uniquely identifies the tuple. Each partition p_i maintains two additional metadata fields: (i) the outbound counter OC_{p_i}, which tracks the number of tuples sent to lower-level partitions, and (ii) the minimal sequence number $S^{min}_{p_i}$, which represents the



(a) Trace-based tuple lookup (b) Trace-guided physical access

Fig. 4. Tuple location discovery mechanism in TRACER.

smallest S_r among all tuples currently stored in p_i. These counters are updated as tuples move through the transformation process. Figure 3b illustrates this trace vector construction process.

3.3.2 ***Trace-based tuple lookup.*** As data undergoes progressive repartitioning, tuples move downward through the partition hierarchy in insertion order. As a result, the per-partition minimum sequence numbers S_{r,p_i}^{min} naturally follow a *descending sequence* along the relocation path. To locate a tuple r, the system identifies the first partition p_i such that $S_{r,p_i}^{min} \leq S_r$, eliminating the need to scan lower levels. For instance, in Figure 4a, a tuple with $S_r = 3$ is located in p_2, the first partition satisfying this condition. Once p_i is identified, the tuple's *logical in-partition offset* Δ is computed as $\Delta = TC_{r,p_i} - OC_{p_i}$, which determines its precise position under the order-preserving transformation invariant (**INV-2**). In Figure 4a, Δ evaluates to zero since $TC_{r,p_2} - OC_{p_2} = 0$.

3.3.3 ***Trace-guided physical access.*** After identifying p_i and computing Δ within p_i, the physical offset of the target column group instance is determined using standard calculations employed in PAX-like layouts such as Parquet [8] and Citus [17]. We first locate the target record zone Z_t^R containing the tuple by dividing Δ by the fixed record count per zone:

$$Z_t^R = \left\lfloor \frac{\Delta}{\text{records per zone}} \right\rfloor$$

We then compute the tuple's relative position δ within Z_t^R:

$$\delta = \Delta - \sum_{k=0}^{t-1} (\text{record count in } Z_k^R)$$

As shown in Figure 4b-❶, Z_0^R is the target zone for $\Delta = 0$, and δ is also zero, indicating that the tuple is the first logical position in Z_0^R.

Next, we locate the target column zone Z_t^C within Z_t^R using our *ETL-map*, which specifies the column grouping layout. The page offset of Z_t^C is computed by summing the page counts of all preceding column zones in Z_t^R. The target column group instance cg_t for tuple r is then identified as the δ^{th} instance in Z_t^C. Since all column zones within a record zone share the same tuple cardinality, this offset calculation is both precise and efficient. As shown in Figure 4b-❷❸, column c_3 belongs to the second column group Z_1^C, making Z_1^C the target, and with $\delta = 0$, cg_0 is the resolved instance. This layout-aware mapping completes the trace vector-guided access pipeline, enabling TRACERETL to locate any column group instance with minimal metadata and arithmetic.

3.4 In-Place Updates and Out-of-Place Inserts

B+trees, as a standard indexing structure, integrate naturally into TRACERETL. They support direct tuple retrieval, while physical reformatting occurs progressively via background repartitioning.

This section presents our approach for unifying in-place updates with out-of-place inserts and outlines the recovery protocol adaptations required to maintain correctness with TracerETL.

3.4.1 ***Out-of-place inserts with a tuple trace vector.*** The insert operation begins by navigating the B+tree index to find target leaf nodes to insert record items. When we add new record items to the B+tree index, we only insert RIDs to leaf nodes by preserving the primary key order to identify the appropriate leaf nodes for inserting record items. Next, we transition to the leveled partition tree by appending records to an in-memory partition buffer, *mempartition*. After generating a trace vector, we embed it within the corresponding RID. Once the mempartition buffer reaches capacity, it is flushed to storage. A background process continuously repartitions upper-level partitions into finer-grained partitions.

During progressive partitioning, preserving the original insert order is essential for predictable tuple placement. Failing to do so makes forecasting a tuple's future location within partitions infeasible.To support efficient lookup, the partition worker updates key per-partition metadata—$S^{min}r, p_i, OCp_i$—in the *ETL-map* for all affected partitions. Upon completion of repartitioning, the *ETL-map* is further updated to reflect new minimum sequence numbers for both upper- and lower-level partitions, thereby narrowing the search space during tuple resolution. Since the *ETL-map* encapsulates metadata that is also embedded in relation headers, we log all *ETL-map* changes to ensure durability. This enables accurate reconstruction of the *ETL-map* following a system restart.

3.4.2 ***Update-in-place with multi-versioning.*** When updates arrive, they traverse our B+tree index to locate the target RIDs in the leaf entries, each containing a tuple trace vector. Leveraging this trace vector along with the *ETL-map*, we rapidly identify the target partition and compute the tuple's precise in-partition offset, thus enabling instant tuple location discovery. Once the current tuple location is determined, our in-place update mechanism relocates the existing data version to a dedicated version manager and replaces it at the same location with new data. To mitigate version space amplification inherent in HTAP workloads, TracerETL employs specialized version managers integrated into PostgreSQL, inspired by prior research [31, 32].

However, access conflicts can arise between the background repartitioning process, which relocates data, and update transactions, which modify it. To resolve these conflicts with minimal disruption, we employ a conflict resolution technique that reduces locking on partition files. While more sophisticated concurrency control mechanisms could enhance parallelism, we avoid them to maintain a simpler recovery protocol and ensure robust and predictable system behavior.

3.4.3 ***Concurrent updates and repartitioning.*** To resolve access conflicts, we adopt a strategy allowing updates to proceed on the target (old) partition even during active repartitioning. Specifically, we use an approach similar to the delta-chain mechanism used in the Bw-tree [38, 68] and let update transactions temporarily store their modifications as *update deltas*. Figure 5 illustrates how repartitioning coordinates with concurrent updates. When an update transaction encounters an ongoing repartitioning operation, it appends update deltas to a temporary delta chain (termed "*stashing update deltas*"). These delta chains persist until the partition worker completes flushing the in-memory partition files. Conversely, if the repartitioning operation detects active updates, it briefly postpones its progress until those updates finalize. This approach mitigates contention between concurrent updates and repartitioning.

Once the in-memory redistribution process completes, the background worker proceeds to either write new in-memory partition files or append new Z^R pages to the end of existing lower-level partitions. To ensure consistency, the worker then briefly acquires a lock on the relevant partitions to apply all update deltas to the in-memory pages. This step guarantees that no updates are missed during the reorganization phase. Because replaying deltas in memory is significantly faster than the other two activities, update transactions briefly spin-wait (referred to as "*spin-waiting*"). A

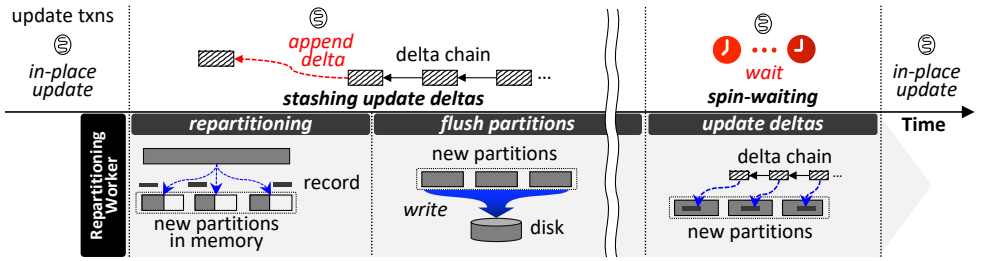


Fig. 5. Concurrent updates and repartitioning.

potential concern arises about whether newly updated partitions in storage might exclude the *update deltas*. However, recovery logs will restore such partitions to the correct state in the event of a crash. This mechanism shares similarities with the approach employed by MySQL for online DDL [7], which states "*recording the changes made by concurrent DML operations and applying those changes at the end.*" Nonetheless, it is crucial to note that all in-place updates must still generate redo/undo logs while creating update deltas to ensure correct recovery.

3.4.4 ***Adapting ARIES for TracerETL.*** TracerETL builds on ARIES [45] for crash recovery but requires key adaptations due to its hybrid update model—supporting both in-place updates and out-of-place inserts during progressive data transformation. A naive application of ARIES risks correctness violations, as the semantics of recovery differ across update types. To address this, we apply redo/undo logging for in-place updates and redo-only logging for out-of-place repartitioning. This separation enables consistent recovery semantics across both modes without introducing additional transactional complexity. As described in §3.4.3, repartitioning occurs concurrently with transactional updates. However, to simplify recovery, we enforce an exclusive *no-steal* policy during the final delta application stage, ensuring that the system is always in one of two valid states: *before* or *after* delta application. This design allows us to treat repartitioning as a logical atomic operation that can be rolled forward or undone based on recovery metadata.

To support this, TracerETL logs two distinct events during repartitioning: (i) a *pre-repartitioning marker* to signal the start of transformation and identify a stable base state and (ii) a *post-commit marker* indicating successful flush and delta application. Upon crash recovery, if the post-commit marker is absent, the system performs redo recovery using logs from the original partition only and truncates newly written lower-level files. If both logs are present, the transformation is considered committed, and both new files and delta applications are retained. Additionally, since update deltas may originate from uncommitted transactions, we require undo recovery to roll back only those deltas during ARIES's analysis and undo phases. This rollback is confined to lower-level partitions and does not affect the stable upper-level layout, preserving correctness under partial transformation. In summary, our adaptation to ARIES allows TracerETL to safely interleave progressive data reorganization with high-throughput transactional updates while preserving recoverability guarantees under failure.

4 Partition-Aware Query Processing

Efficient query processing over TracerETL's vertically and horizontally partitioned layout hinges on generating plans that access only the relevant partitions. To support this, TracerETL integrates partition-aware optimization and execution mechanisms. Inspired by Pando [64], which uses predicate rewriting for data skipping, we introduce general-purpose *partition-aware query rewriting and execution* techniques that retrofit pruning capabilities into engines lacking native support. Specifically, we inject predicates over *hidden attributes* into the logical plan, enabling effective

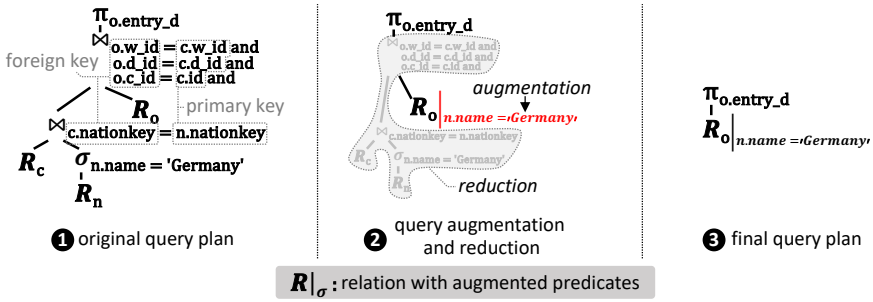


Fig. 6. Partition-aware query rewriting example.

partition pruning even during ongoing transformation. This rewriting reduces I/O by producing query plans that are natively TracerETL-aware.

4.1 Partition-Aware Query Rewriting

The partition-aware optimizer first augments the query plan with predicates over hidden attributes in *fact* tables (§4.1.1), enabling direct reasoning about partition boundaries. It then simplifies the plan by removing unnecessary `joins` and redundant accesses to dimension tables, producing a more efficient, partition-aware execution strategy (§4.1.2).

4.1.1 ***Augmenting query plan for hidden attributes.*** In TracerETL, we leverage *hidden* attributes embedded in fact tables to (i) enable partition-aware query processing and (ii) avoid costly `joins`. To support this, we augment the original query plan by injecting additional predicates over these hidden attributes. A *materialized view* is used to map domain values to concise encoded representations, allowing efficient predicate evaluation during query execution. This view is cached at runtime to interpret encoded values in the augmented predicates. Although some partitions may contain irrelevant records that do not satisfy the original predicates, the augmented plan enables accurate filtering by applying constraints on the hidden attributes.

Injecting new predicates over hidden attribute. We implemented plan augmentation within PostgreSQL to support partition-aware query rewriting. At DDL time, we extend the PostgreSQL engine to automatically append a variable-length byte-array *hidden attribute* to the schema when a table is created. During INSERT operations, we modify the foreign-key trigger to (i) validate referential integrity, (ii) retrieve the hidden attribute from the referenced parent tuple(s), and (iii) copy it into the new tuple's hidden attribute field. This logic incurs no I/O overhead, as the attribute fetch and copy occur entirely in memory. During query optimization, we augment the planner to analyze predicates over dimension attributes and inject equivalent filters over hidden attributes. PostgreSQL's planner already infers foreign-key relationships among tables in the `FROM` clause; we leverage this capability to identify join conditions suitable for predicate augmentation.

As shown in Figure 6-❷, the process proceeds in three steps: (i) *identify the augmentation target*, (e.g., the `orders` table). (ii) *track predicate propagation* from dimension tables and generate equivalent filters on hidden attributes; and (iii) *prune redundant joins* to dimension tables where possible. The rewritten plan thus includes the original predicates and additional filters over hidden attributes (e.g., `n.name = 'Germany'`), enabling effective partition pruning. To preserve optimizer behavior, we mark the selectivity of augmented predicates as 1.0 using `clause_selectivity_ext()`, ensuring they do not alter cost-based plan decisions. We leave cost-based optimization of these rewritten plans to future work.

4.1.2 ***Partition-aware query rewriting.*** Analytic queries frequently apply predicates to columns in dimension tables, which are propagated to fact tables through multi-way joins. In TracerETL, if

a predicate targets a *hidden attribute*—a dimension table column encoded into fact tables—we can rewrite the query to produce a more concise, semantically equivalent, and partition-optimized plan. *By doing so, we eliminate unnecessary joins and avoid redundant access to dimension tables.* Therefore, predicates are evaluated directly on hidden attributes within fact tables, reducing scan overhead and improving execution efficiency. The rewriting process begins by (i) *inspecting the schema* to identify relevant foreign key relationships. We then (ii) *analyze the predicates* to determine whether any target columns outside the known hidden attributes can be pushed down or pruned. Lastly, we (iii) *eliminate any superfluous columns* that are not required for query execution.

Eliminating unnecessary joins. We implemented the new rewriting rule to eliminate redundant tables, typically dimension tables, that serve no purpose beyond propagating partition keys to fact tables. Specifically, we determine whether a table contributes to the query beyond key propagation. If it plays any additional role (e.g., filtering, projection, or aggregation), it is retained. The optimizer can iteratively remove redundant tables from the `FROM` clause. The pruning process continues until no table satisfies all three criteria (**C1**-**C3**). Any table that meets all conditions is considered unnecessary and is safely removed from the query plan.

- **C1:** If a table's primary key participates solely in an equijoin with another table's foreign key, it can be removed by recursively analyzing transitive join relationships.
- **C2:** If all predicates on the table involve only columns used in the partition key, the table can be eliminated.
- **C3:** If the `SELECT` clause references only the partition key, and that key can be retrieved from other joined tables, the table can be safely discarded.

Our new rules eliminate several nodes in the augmented query plan shown in Figure 6-❷. These nodes access the `customer` and `nation` tables and perform `join` operations solely to propagate predicates from the dimension table (`nation`) to the fact table (`orders`). The resulting simplified plan, shown in Figure 6-❸, allows direct and sequential access to partitions of the `orders` table.

4.2 Partition-Aware Query Execution

This section describes our modifications to the PostgreSQL query executor to support partition-aware query evaluation, with a focus on *lazy partition enumeration* and data access using `io_uring`.

4.2.1 ***Lazy partition enumeration in PostgreSQL.*** The final stage of query optimization produces a physical plan that includes execution metadata, e.g., relation IDs and access methods. However, in TracerETL, enumerating all partition files during plan generation—especially for simple table scan queries—incurs unnecessary overhead. To address this, we extend PostgreSQL to adopt a *lazy partition enumeration* technique. During the initialization of the sequential scan node, we identify relevant partition IDs based on predicates applied to hidden attributes. These IDs are stored within the plan node and are resolved lazily at runtime by dynamically substituting the appropriate partition file names during scanning.

To support lazy enumeration, TracerETL leverages PostgreSQL's generalized inverted index (GIN) [27] to index partition keys of arbitrary length. We adopt GIN primarily due to its native support in PostgreSQL, despite its linear space complexity with respect to the partition key domain—e.g., indexing a 23-bit composite partition key incurs a 5 MiB footprint in our prototype. While more compact alternatives such as trie-based indexing may better exploit prefix sharing, GIN offers immediate compatibility and robust performance for our workload. The index maps encoded hidden attribute values to their corresponding partition identifiers, enabling fast retrieval of relevant partitions during query execution.

4.2.2 ***Private ring buffers for accessing partitions.*** Buffer contention in standalone HTAP DBMSs is a well-known concern, and TracerETL mitigates this by introducing *private ring buffers*

for OLAP workers. These private ring buffers are assigned automatically during query planning; when the planner selects a sequential scan node, such as a full table scan or a large-range scan with high I/O locality, the modified PostgreSQL engine allocates a dedicated private ring buffer for serving sequential scans over TracerETL partitions. Other access methods (e.g., index scans) continue to operate over the traditional shared buffer pool. However, this coexistence of private and shared buffers introduces the potential for data inconsistency, as the same page may reside in multiple buffer contexts. For correctness, OLAP workers must first check the shared buffer pool before loading a page into their private buffer. This coordination ensures that all reads observe the most up-to-date data. As a result, private ring buffers reduce memory contention during scans.

4.2.3 ***Fast data access using Linux*** `io_uring`. After the query plan is finalized, the optimizer hands it off to the executor, which sequentially scans partitions and identifies files that satisfy the augmented predicates. During this phase, TracerETL utilizes Linux `io_uring` [10] to efficiently fetch data blocks into private buffers. By leveraging `io_uring`'s asynchronous and low-latency I/O capabilities, TracerETL improves data access throughput. Further performance gains can be achieved by bypassing the kernel page cache—a technique adopted by systems like MongoDB and MySQL [30, 47]—to reduce data copying overhead and improve cache utilization.

5 Evaluation

This section presents the evaluation results of PostgreSQL-15 with and without TracerETL under HTAP workloads, and also under a pure OLAP subset, and with an OLTP subset plus one long-running query. To offload the significant burden of version management to a separate system, we leveraged Diva[32], one of the leading proposals in this area, so we incorporated TracerETL into its codebase. Therefore, we conducted our evaluation using three variants of PostgreSQL: (1) vanilla PostgreSQL, (2) PostgreSQL with Diva, and (3) PostgreSQL-Diva-TracerETL with full vertical and horizontal partitioning (TracerETL in short). To guide our evaluation, we seek to answer the following questions:

- **Q1 (§5.2):** *How effective is query rewriting for plan simplification and partition pruning on decision-support workloads?*
- **Q2 (§5.3):** *Can TracerETL maintain OLTP throughput while accelerating analytical queries under mixed workloads?*
- **Q3 (§5.4):** *Does TracerETL improve real-time HTAP over OLTP- and OLAP-optimized baselines in a standalone DBMS?*
- **Q4 (§5.4.2 and §5.4.4):** *How space- and I/O-efficient is progressive ETL with updatable layouts under diverse workloads?*
- **Q5 (§5.4.5):** *How does TracerETL compare to standalone log-structured HTAP systems like LASER [59]?*

5.1 Evaluation Setup

We use a 128-core machine featuring dual AMD EPYC 7003 processors, 256 MiB L3 Cache, 512 GiB DRAM, and two 2 TiB PCIe 4.0 NVMe SSDs, each delivering 7 GiB/s for sequential reads. To emulate larger-than-memory configurations, we use the Linux `cgroup` feature. We use a slightly-modified `TPC-CH` [15] framework, based on the tools [13] that Citus Data [17] developed for running CH-benCHmark with HammerDB-4.3 [1]. We created a dataset with 500 warehouses (WH). Colocating a record and its first old version, Diva, incurs space overhead [31, 32]; TracerETL has further space overhead for storing a trace vector for each record, so space needed is 60 GiB in vanilla PostgreSQL, 120 GiB in PostgreSQL-Diva, and 139 GiB in PostgreSQL-Diva-TracerETL. Table partitions, including *mempartitions*, are configured to have a capacity of 16 MiB; we assigned two dedicated workers to each relation.

5.1.1 ***Configurations for HTAP-capable engines.*** To stress the TracerETL pipeline and validate its ability to maintain high OLTP performance under real-time HTAP loads, we configure PostgreSQL with settings optimized for throughput. Specifically, we relax *strict durability* by enabling *asynchronous commit* [66] to increase write throughput and test TracerETL's ability for in-place updates during progressive transformation. We also enlarge WAL log files to reduce checkpoint frequency and disable PostgreSQL's parallel query execution [25], while enabling LLVM-based JIT query compilation [26]. The default isolation level is set to `REPEATABLE READ`. Unless otherwise specified, the configuration includes a 32 GiB buffer pool and 48 GiB DRAM, creating a larger-than-memory HTAP environment.

Real-Time LSM Trees (LASER). To evaluate performance under HTAP workloads, we compare against LASER [28, 59], a real-time LSM tree implementation built on RocksDB that supports columnar reformatting. Since RocksDB lacks SQL query support, we integrated LASER with PostgreSQL using the Foreign Data Wrapper (FDW) extension for RocksDB [53]. This setup enables PostgreSQL to execute query plans through RocksDB. Additionally, we extended the FDW implementation to support `REPEATABLE READ` isolation. However, running HammerDB-4.3 against PostgreSQL-LASER requires substantial modifications. Consequently, we created a single fact table and a single dimension table, replicating the setup from the original LASER work [59]. We then evaluated LASER's performance using simplified updates and analytical queries (i.e., point, range, and join queries) executed on these two tables.

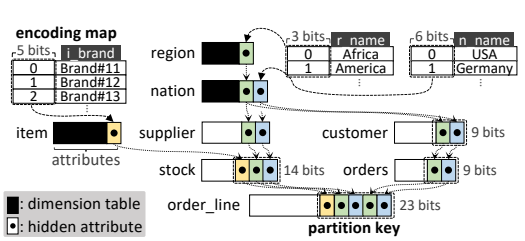
5.1.2 ***Configurations for OLAP-optimized DBMSs.*** For our evaluation, we utilized three OLAP-optimized database engines: ClickHouse[60], DuckDB[54], and Citus [17]. Each engine was configured with recommended settings to optimize performance for OLAP workloads. To ensure a fair comparison, the available memory was constrained to 48 GiB of DRAM using `cgroup`. With these engines, we focus solely on OLAP workloads, providing insights into performance in pure analytical query execution scenarios.

ClickHouse. For evaluation, we configured ClickHouse to use the MergeTree engine, which is optimized for OLAP workloads. MergeTree organizes data similarly to an LSM-tree and relies on several caching mechanisms, including Mark Cache, Uncompressed Cache, and the operating system's page cache, to accelerate query execution. Data within MergeTree tables is sorted and stored based on a sort order defined during table creation, and we used the primary key of each table in the TPC-CH benchmark (also referred to as CH-benCHmark) as the sort order to align with the benchmark's query patterns. The dataset used for our ClickHouse evaluation was 26 GiB in size, compressed using ClickHouse's default LZ4 compression scheme. We applied default configurations for the Mark Cache (5 GiB) and disabled the Uncompressed Cache.

DuckDB. For our evaluation, we configured DuckDB to use Parquet [8] files for all tables. We began by generating table files in PostgreSQL and then converted them to the Parquet format. Using the gzip compression scheme, the total dataset size was 21 GiB. We also configured DuckDB to execute queries in a single-threaded mode, which simplifies resource usage but may limit performance compared to multi-threaded configurations.

Citus. Citus is a distributed extension for PostgreSQL that leverages the PAX layout [3] for efficient columnar storage and employs the `zstd` compression scheme to reduce storage footprint. To mitigate performance degradation caused by the absence of indexes in columnar storage, we restricted the transformation to two large tables: `orders` and `order_line`. The dataset size for Citus was 49 GiB, reflecting the impact of its columnar layout and compression optimizations.

5.1.3 ***TPC-CH benchmark setup.*** We modified the original TPC-CH schema to make our evaluation more faithful to the original TPC-H specifications. We added the "`i_brand`" column to the `item` table (see Figure 7a), which is included in the `part` table according to the TPC-H specifications



(a) Horizontal partitioning with *hidden attributes* (i.e., partition keys)

Augmentation Effect	TPC-CH Queries with Predicates on *hidden attributes*
High	Q1{n} Q2{n,b} Q3{n} Q5{n} Q6{n} Q7{n} Q8{n,b} Q9{n,b} Q11{n} Q14{b} Q15{b} Q16{b} Q18{n} Q19{b} Q20{n,b} Q22{n}
Medium	Q4{n} Q10{n} Q12{n} Q13{n} Q17{b} Q21{n}

(b) Partition keys and modified queries

Rule	orders	order_line
R1	o_carrier_id	ol_delivery_d
R2	o_id, o_w_id, o_d_id	ol_w_id, ol_d_id, ol_o_id, ol_number, ol_amount
R3	o_c_id	ol_supply_w_id, ol_i_id
R4	o_entry_d, o_ol_count, hidden_attributes	ol_quantity, hidden_attributes
R5	o_all_local	ol_dist_info

(c) Vertical partitioning of `orders` and `order_line` using column grouping

Fig. 7. TPC-CH benchmark setup for data partitioning.

but excluded in the TPC-CH schema. Our evaluation of TracerETL focuses on real-time data partitioning of large tables that may grow during execution.

Augmenting TPC-CH queries. To evaluate the benefits of horizontal partitioning, we modified all 22 OLAP queries by adding predicates on at least one hidden attribute used in the partitioning keys. In some cases, our reference partitioning strategy provides limited benefit due to the query structure. This also requires including `join` operations with other tables. We refer to the modified query as $Q\#_c$, with a subscript $_c$ indicating which column(s) have a predicate imposed: C might be b for `i_brand`, n for `n_name`, or r for `r_name`. Figure 7b shows all 22 modified TPC-H queries used across all PostgreSQL variants in our evaluation, annotated using the notation described above.

Target for horizontal partitioning. We have selected four tables—i.e., `orders`, `order_line`, `customer`, and `stock`—as partition targets since OLTP transactions keep modifying these. We designated three columns as *hidden attributes* for horizontal partitioning in TracerETL: "`r_name`" from the *region* table, "`n_name`" from the `nation` table, and "`i_brand`" from the `item` table. These columns have cardinality numbers of 5, 62, and 25, respectively, and require bit lengths of 3, 6, and 5 bits for encoding as hidden attributes. Fact tables may inherit multiple hidden attributes through different join paths, leading to varying attribute sets per table. For instance, `stock` has two three hidden attributes (14 bits), while `orders` and `customer` each have two hidden attributes (9 bits). But, `order_line` inherits five (23 bits total)[1]. All these tables use their hidden attributes as a partition key (see Figure 7a). Upon the arrival of new tuples, these hidden attributes are appended to the new records.

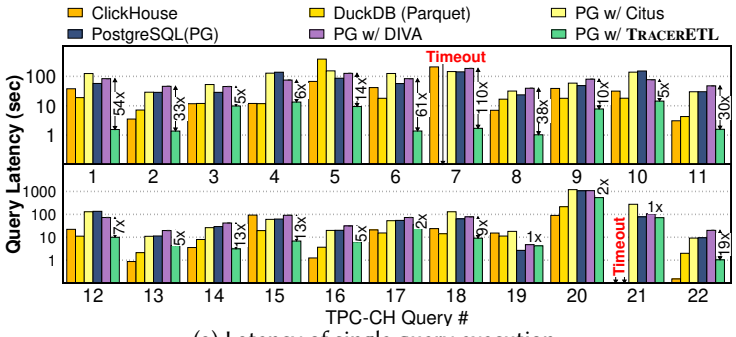
Target for vertical partitioning. In TracerETL, we target columnar grouping for two fact tables: `orders` and `order_line`. In contrast to other tables, which mainly profit from horizontal partitioning, these two tables can significantly benefit from columnar formatting. Following the five rules outlined in §3.2.3, we reformat the rows from these tables into five distinct columnar groups. The sizes of these column groups vary for each table based on query semantics, as depicted in Figure 7c.

5.2 OLAP Workloads

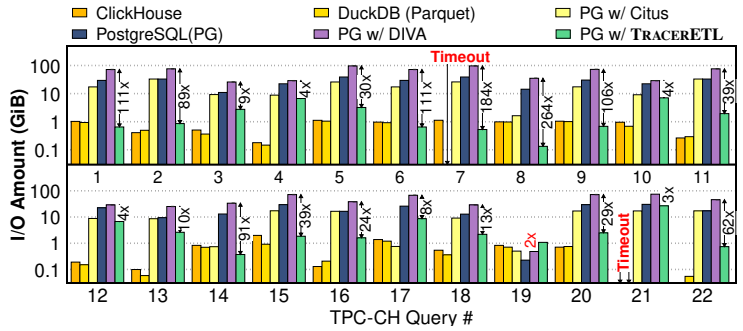
We evaluate TracerETL on static datasets using the augmented queries in Figure 7b. Figure 8 reports two key performance metrics: query latency and physical I/O amount. Each query is executed in a cold buffer state to assess the effectiveness of TracerETL's partitioning and query rewriting techniques in reducing I/O costs.

5.2.1 ***Performance metrics for OLAP engines.*** As illustrated in Figure 8a, OLAP engines such as ClickHouse and DuckDB exhibit lower query latencies compared to PostgreSQL-based engines

[1]The resulting *ETL-map* sizes for `order_line`, `orders`, `stock`, and `customer` tables (~ 139 GiB) are 1.8 MiB, 4 KiB, 37 KiB, and 5 KiB, respectively. These succinct *ETL-map* files are typically resident in memory throughout query execution.



(a) Latency of single query execution



(b) I/O amount of single query execution

Fig. 8. Performance metrics of analytic workloads.

like vanilla PostgreSQL and Diva, primarily due to their use of columnar and compressed layouts. However, despite their space efficiency, OLAP-optimized engines face decompression overhead, which impacts overall query performance. Additionally, certain queries (e.g., Q5, Q7, Q21) frequently require materialization due to memory pressure from their OLAP-specific execution logic, leading to higher latencies in DuckDB.

In contrast, Citus, which adopts PostgreSQL's execution logic while utilizing columnar storage, only converts two tables (`orders` and `order_line`) to a columnar format. As a result, queries scanning these tables (e.g., Q4, Q12) benefit from reduced I/O and lower latency compared to vanilla PostgreSQL. However, for other queries, Citus experiences significant I/O overhead and higher latencies compared to ClickHouse and DuckDB. This is largely due to its reliance on PostgreSQL execution logic, which is not optimized for columnar storage. In fact, the absence of indexes on the two columnar tables can make Citus perform worse than vanilla PostgreSQL in certain cases. The results highlight an important insight: **simple columnar reformatting with compression may deliver suboptimal performance for OLAP workloads** unless accompanied by static tables and OLAP-optimized execution logic.

5.2.2 ***Performance metrics for PostgreSQL variants.*** Vanilla PostgreSQL and PostgreSQL-Diva exhibit a similar trend, as these platforms do not exploit partitioning. An interesting observation is the inferior performance of Diva compared to the vanilla engine, mainly because of Diva's policy of inlining records and their first versions, leading to the retrieval of larger datasets. TracerETL inherits this policy from its codebase, which adversely affects performance. However, the latency results for TracerETL generally demonstrate the advantages of partitioned data and unveil intriguing patterns. Particularly, columnar grouping in TracerETL often reduces I/O volume

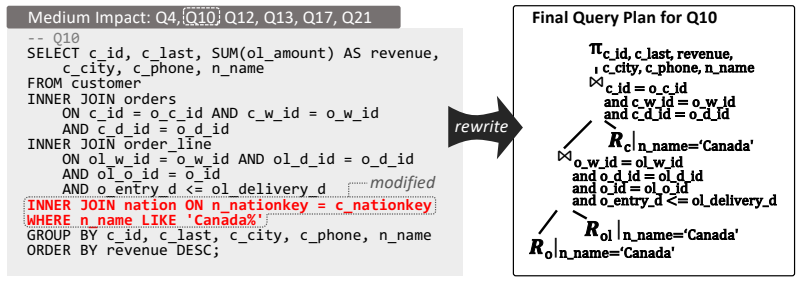


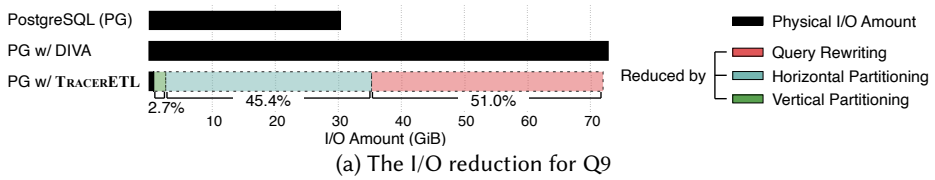
Fig. 9. The effect of query rewriting for augmented TPC-CH queries in the medium-impact group.

by minimizing the number of pages accessed during query execution (Figure 8b), although its effect on overall query latency (Figure 8a) may be less prominent.

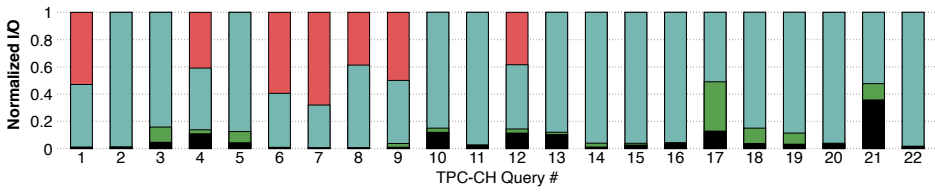
For a subset of queries (e.g, Q1, Q2, Q5–Q9), TracerETL achieves lower latency and reduced I/O compared to Diva, mainly due to query rewriting that eliminates unnecessary table accesses and joins. In contrast, queries such as Q10 still require join operations—particularly between `order_line` and `orders`—despite accessing smaller table partitions. Figure 9 illustrates the impact of TracerETL's rewriting on Q10, a representative of the medium-impact group. While high-impact queries benefit from streamlined plans with sequential scans over a narrow partition subset (as shown in Figure 6), medium-impact queries retain expensive joins in their final plans, resulting in longer execution times.

Noticeable is that Q19—a high-impact query—exhibits higher latency under TracerETL compared to Diva, primarily due to the lack of bitmap scan support. In log-structured environments, where tuples may be dispersed across multiple evolving partitions, it is challenging to memoize record positions for efficient bitmap-based access. As a result, TracerETL falls back to sequential scans, incurring the associated I/O and latency overhead. We also note that despite the I/O reduction, **TracerETL incurs higher I/O than OLAP engines for certain queries due to the space overhead of trace vectors and the absence of data compression**. Nonetheless, for some of these queries, TracerETL achieves much lower latency than OLAP engines by utilizing `io_uring`.

5.2.3 ***The breakdown of I/O reduction in TracerETL.*** To gain deeper insights into the extent of I/O reduction achieved by different strategies in TracerETL, we instrumented PostgreSQL to analyze the breakdown of I/O reduction for each augmented TPC-CH query. As illustrated in Figure 10a, we examine the I/O reduction attributed to three key effects: (i) horizontal partitioning



(a) The I/O reduction for Q9



(b) The I/O reduction for analytic workloads

Fig. 10. The I/O reduction in PostgreSQL-TracerETL.

(HP), (ii) vertical partitioning (VP), and (iii) query rewriting (QR), in comparison to the total I/O amount of PostgreSQL-Diva using Q9 as a reference. It is important to note that the reduction effects may vary depending on the query plans. **Notably, the impact of partition-aware query rewriting, accounting for 51.0%, is more significant than initially anticipated**. This outcome underscores the importance of adapting the query optimizer to comprehend various partitioning strategies fully, enabling proper augmentation and rewriting of query plans. Without such adaptation, achieving comparable performance benefits would prove challenging.

Another intriguing discovery is the lower-than-expected effect of columnar grouping, amounting to only 2.7%, in TracerETL. This result might be attributed to our choice of reference point, which is the total I/O amount. However, if we shift the reference point to the I/O amount after HP and QR, the reduction ratio increases to approximately 70%, indicating a significant impact of columnar formatting on I/O reduction. This consequence suggests the need for further investigation into how to integrate compression techniques into dynamically changing columnar datasets. Nonetheless, **our observation underscores the potential of continuous columnar grouping in standalone HTAP database systems**.

Next, we delve into the breakdown of I/O reduction for all augmented TPC-CH queries. Figure 10b presents the breakdown of I/O, normalized to the baseline without TracerETL. We observe that queries sharing structural properties tend to benefit similarly from our partitioning strategies and rewriting rules, independent of the benchmark suite. Overall, *horizontal partitioning* is the primary contributor to I/O reduction. In particular, queries such as Q2, Q11, Q15, Q16, Q20, and Q22—each of which joins a fact table with one or more dimension tables and includes low-cardinality predicates on dimension attributes—benefit substantially by avoiding scans over irrelevant partitions. Additionally, *query rewriting*-based *join elimination* provides substantial I/O savings for queries that join a fact table with multiple dimension tables but project only fact table attributes. This category includes Q1, Q4, Q6–Q9, and Q12.

Vertical partitioning (VP), based on columnar grouping, shows more modest I/O benefits overall. However, certain queries with narrow projections—namely Q3, Q5, Q17, Q18, Q19, and Q21—still derive meaningful I/O reduction by selectively accessing relevant column zones. Notably, this benefit is achieved without a compressed columnar store: VP in TracerETL reads columns dispersed across partition files, yet still lowers the I/O footprint effectively. These results, summarized in Figure 10, confirm that TracerETL can improve analytical query performance through optimized query planning and partition-aware execution. However, this is not the primary focus of our contribution, as similar improvements are possible with other static partitioning approaches. Our key claim lies in demonstrating that such benefits persist under mixed OLTP/OLAP conditions, which we evaluate in subsequent sections.

5.3 OLTP Workloads

To evaluate OLTP performance under mixed workloads, we configured HammerDB to run TPC-C transactions over the TPC-CH schema using 12 concurrent clients. The workload executed for 30 minutes, during which we introduced four long-lived transactions (LLTs) partway through the experiment. These LLTs repeatedly executed Q1, Q7, Q14, and Q22 until the end of the test. It is well recognized that vanilla PostgreSQL suffers substantial throughput degradation in the presence of even a single LLT [31]. In contrast, PostgreSQL-Diva maintains stable performance by mitigating version contention. PostgreSQL-TracerETL initially exhibits slightly lower throughput than both vanilla and Diva, due to increased contention in the legacy shared buffer pool. This contention arises from TracerETL's background partitioning, which disperses update hotspots and thereby increases memory pressure. In contrast, vanilla PostgreSQL experiences pronounced initial throughput drops caused by vacuuming activity.

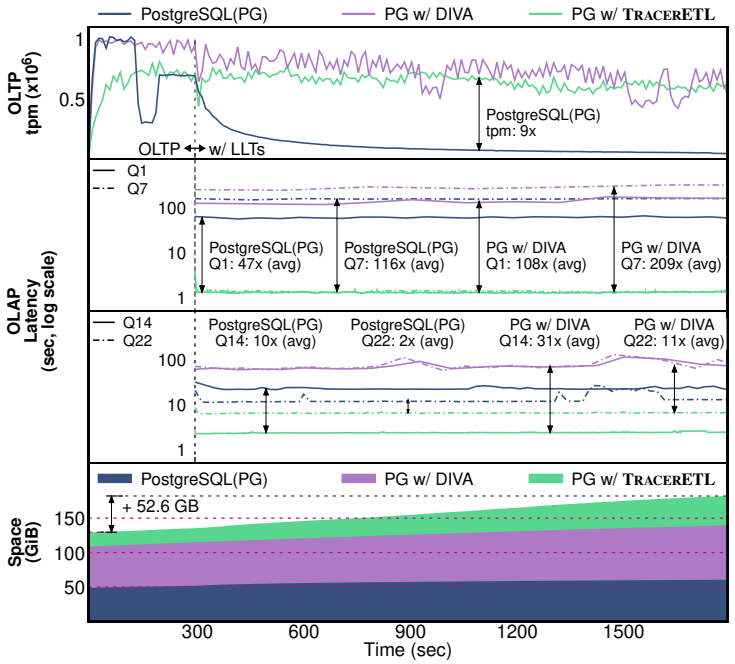


Fig. 11. OLTP performance with long-lived transactions.

As shown in Figure 11, PostgreSQL-TracerETL inherits the robustness of PostgreSQL-Diva, maintaining high OLTP throughput even in the presence of concurrent, long-running analytical queries with diverse execution plans. This resilience is rooted in TracerETL's architecture, which decouples version management from the partitioned physical layout. Notably, its support for *in-place updates* allows efficient versioning without the write amplification and commit latency penalties commonly observed in MVCC over LSM-based systems. Consequently, all engines—except vanilla PostgreSQL—sustain stable throughput, underscoring the effectiveness of version-isolated design for real-time HTAP workloads.

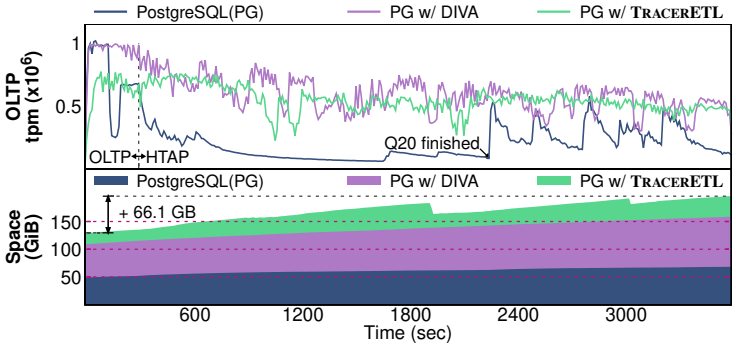
TracerETL's partition workers progressively repartition and relocate records into higher-dimensional partitions. This real-time transformation reduces the data footprint accessed by OLAP queries (Q1, Q7, Q14, and Q22), allowing them to operate over narrower partitions and thus achieve lower latency compared to Diva. An unexpected observation is that Diva has exhibited a higher query latency than the vanilla engine. Internal profiling reveals that it is mainly due to the larger datasets with increased random access to data versions. TracerETL overcomes the same issue by reducing accessed datasets through fine-grained partitioning.

5.4 HTAP Workloads

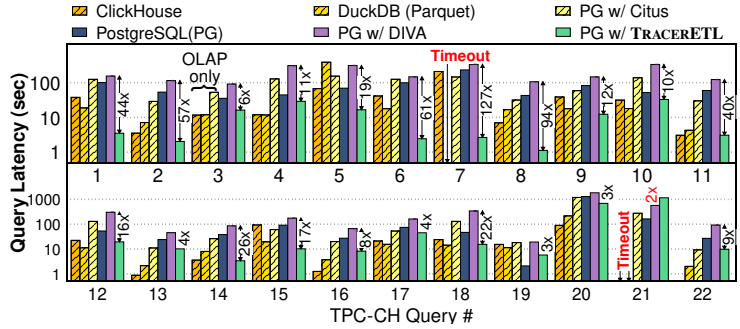
For HTAP loads, we used 12 workers running TPC-C workloads and 4 workers running modified TPC-CH queries. We ran experiments for 60 minutes and measured the average query latency.

5.4.1 ***Performance evaluation with TPC-CH queries.*** Figure 12 shows the OLTP throughput and query latency of the augmented TPC-CH queries summarized in Figure 7b.

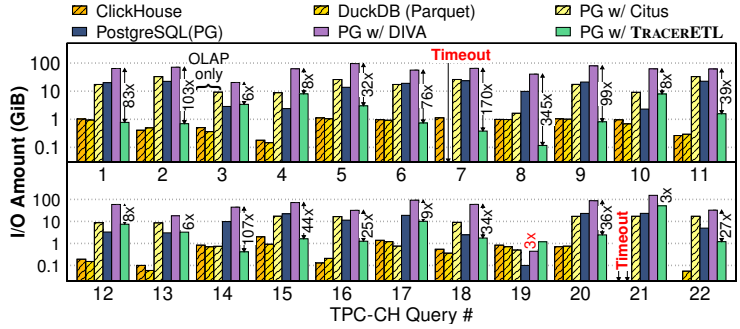
PostgreSQL variants without TracerETL. First, the vanilla engine and PostgreSQL-Diva experience severe drops or occasional fluctuations in OLTP throughput during periods nearly coinciding with the latency numbers of long-running queries (i.e., ~ 100*s* for vanilla PostgreSQL and ~ 300*s* for PostgreSQL-Diva). In the vanilla engine, such queries had the low watermark of



(a) OLTP throughput in HTAP



(b) Average latency of each query in HTAP



(c) Average I/O amount of each query in HTAP

Fig. 12. Performance metrics under HTAP loads.

vacuuming, which led to space inflation with garbage data. Interestingly, due to the low update rate in vanilla PostgreSQL, queries suffer less from performance interference with OLTP transactions.

PostgreSQL with TracerETL. PostgreSQL-TracerETL sustains high throughput while delivering strong query performance, primarily due to reduced data access. Queries in the high-impact group benefit significantly by accessing a small number of data pages without disrupting OLTP activity, resulting in substantially lower latency. Data partitioning and query rewriting in TracerETL plays a pivotal role in reducing I/O, achieving performance improvements of 83× (Q1), 103× (Q2), 76× (Q6), 170× (Q7), 345× (Q8), 99× (Q9), and 107× (Q14) over PostgreSQL-Diva. These improvements allow TracerETL to consistently outperform OLAP-optimized engines in mixed workloads. **TracerETL enables standalone PostgreSQL to support real-time analytics**

through progressive ETL and updatable data layouts. However, some medium-impact queries (e.g., Q4, Q17, Q21) experience increased latency due to contention in the legacy shared buffer pool.

Ironically, TracerETL may appear to hinder long-running analytical queries. Q20, a high-latency query (taking 10–30 minutes), illustrates this trade-off. In vanilla PostgreSQL, OLTP throughput sharply declines during Q20, limiting dataset growth and thereby reducing the I/O footprint for subsequent queries like Q21. In contrast, both Diva and TracerETL maintain high OLTP throughput throughout Q20, which leads to continued data growth, larger datasets, and increased I/O demand—increasing Q21's latency. Notably, Q21 under TracerETL incurs even higher latency than under Diva. This overhead stems from TracerETL's column grouping mechanism: to materialize the large intermediate result generated by Q21's anti-join, full records must be reconstructed from disjoint column groups, introducing extra processing delays that exceed those observed in Diva.

5.4.2 ***The I/O breakdown with TPC-CH queries.*** We evaluated the I/O behavior of sequential scans on the `order_line` table using augmented TPC-CH queries on PostgreSQL-TracerETL (Figure 13). This analysis quantifies the number of data pages fetched via `io_uring` and PostgreSQL's shared buffer pool. As illustrated, the majority of I/O is served directly from NVMe storage, while only a small fraction is served from the shared buffer, typically corresponding to upper-level partitions. Notably, **TracerETL completes each stage of progressive repartitioning for individual partition files in `order_line` in just 60** *ms*, primarily due to the OS page cache.

I/O breakdown in private ring buffers. Figure 13 presents the breakdown of I/O for private ring buffers (i.e., `io_uring`), indicating that queries perform direct read I/O from lower-level partitions of target tables. As anticipated, most queries utilize our private ring buffers to access the third-level partitions of `order_line`, as these partitions contain archived data items that are no longer modified and are finely partitioned based on specified partition keys and column grouping criteria.

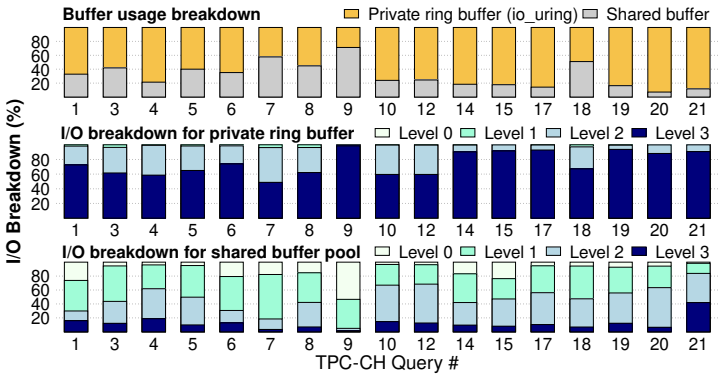


Fig. 13. The I/O breakdown of scanning `order_line`.

I/O Breakdown in the shared buffer. Figure 13 also shows the I/O breakdown for PostgreSQL's shared buffer pool. Only a small fraction of data is served from the shared buffer, from upper-level partitions (levels 0–2) of `order_line`. These partitions are more likely to reside in memory due to frequent OLTP updates. The remaining data is retrieved from deeper partition levels via `io_uring`, making analytic queries seldom interfere with fast OLTP transactions.

5.4.3 ***ETL-map recovery.*** We log all modifications to the *ETL-map* and ensure crash consistency via a replay-based recovery mechanism. As described in §3.4.4, redo logs generated during repartitioning adhere to our modified ARIES protocol. For instance, if a crash occurs before the appearance of a *post-commit marker*, the recovery process replays only those log records that precede the *pre-repartitioning marker*, discarding the rest—thereby avoiding partial or inconsistent state restoration. TracerETL flushes these logs asynchronously at *group commit* time and checkpoints

the *ETL-map* file in sync with PostgreSQL's checkpointing schedule. Since the *ETL-map* logs are lightweight, the total log volume under the TPC-CH benchmark remains below 20 MiB, enabling fast recovery (under 1 sec including file I/O) for the *ETL-map*.

5.4.4 ***TracerETL under skewed workloads.*** To evaluate the robustness of TracerETL under skewed workloads, we employ the sysbench framework with configurable Zipfian distributions. Our schema consists of one dimension table (512 rows) and one fact table (5 million rows), where the fact table includes a foreign key referencing the dimension table. The system is configured with 48 GiB of DRAM and a 4 GiB PostgreSQL buffer pool to reflect a resource-constrained, standalone HTAP setting. We introduce access skew by using the Zipfian coefficient (α=1.2) and run four **concurrent** workloads—*INSERT*, *UPDATE*, *FULL SCAN*, and *JOIN*—to emulate realistic HTAP pressure. This experiment addresses the key question: **How does access skew impact the performance and efficacy of TracerETL?**

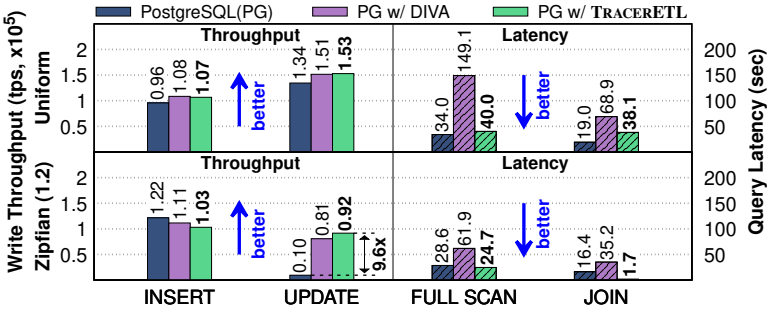


Fig. 14. Performance metrics under skewed workloads.

We evaluate TracerETL under skewed workloads, comparing it against two baseline PostgreSQL configurations—vanilla and Diva. The evaluation focuses on write throughput and query latency under varying degrees of data skew. As shown in Figure 14, TracerETL's insert throughput declines under skewed loads due to increased repartitioning pressure on hot partitions. This outcome is expected, as skewed access patterns concentrate insert activity, triggering more frequent data migration within localized regions of the partition tree. In contrast, TracerETL consistently outperforms both baselines in update throughput. In particular, vanilla PostgreSQL suffers from excessive version chain traversal, Diva partially mitigates this through dedicated version management. The performance advantage of TracerETL stems from its **in-place update mechanism**, which distributes write contention across multiple partition files. This design effectively alleviates hotspot bottlenecks induced by concurrent skewed INSERTs, as evidenced by the distribution of per-partition counter values in Figure 15.

Analytical query performance also benefits from TracerETL's partition-aware architecture. Under skew, full-scan operations are marginally more efficient than under uniform access due to reduced partition file access. For join queries, TracerETL maintains its performance advantage

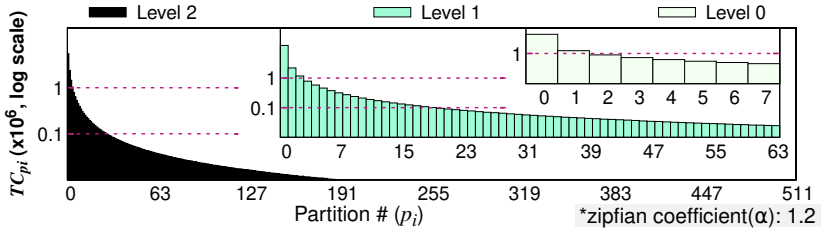


Fig. 15. Per-partition counters under skewed workloads.

over the baselines due to reduced data movement and effective partition pruning—enabled by query rewriting on hidden attributes. Although preliminary, these results indicate that **TracerETL's progressive partitioning with in-place updates reduces contention and sustains performance under skewed, mixed workloads in standalone HTAP settings**.

5.4.5 ***Performance comparison with LASER***. To isolate the impact of our counting-based tuple tracking and updatable ETL design, we compare TracerETL against LASER [59], which performs log-structured ETL but lacks instant tuple discovery and in-place updates. We adopt LASER's original configuration and benchmark three query types—point, range (scanning 100 records from a randomly chosen key), and join—under background insert-only (100 inserts/s) and update-only (1000 updates/s) loads. We include ClickHouse and vanilla PostgreSQL for reference. Dataset sizes are 12 GiB (ClickHouse), 22 GiB (PostgreSQL), 4.1 GiB (LASER), and 30 GiB (TracerETL); PostgreSQL's shared buffer pool is set to 8 GiB.

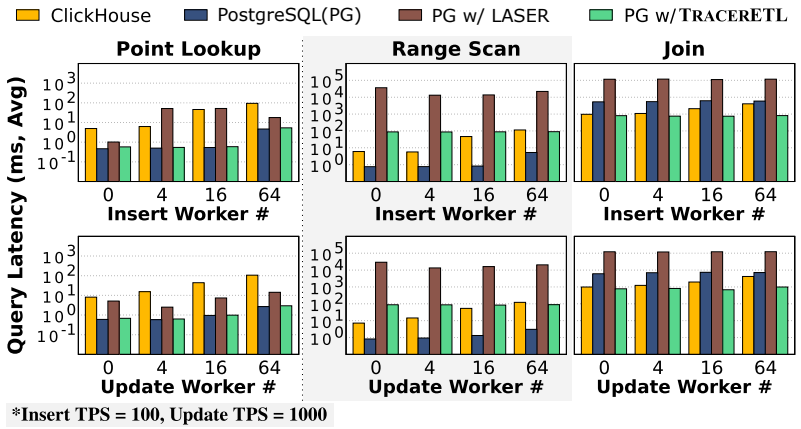


Fig. 16. Performance comparison with LASER [59].

As shown in Figure 16, query latencies increase with transactional load across all engines. ClickHouse excels at range and join queries, while LASER performs well on point queries but suffers from high latency due to the lack of a buffer pool. Vanilla PostgreSQL benefits from high cache hit rates and efficient range scans but lags in join performance. In contrast, **TracerETL achieves strong results in join processing by leveraging its updatable data layouts for real-time data access and reduced I/O overhead**. However, range queries on non-partitioned keys may degrade performance due to broader partition access.

6 Limitations and Discussion

While our evaluation demonstrates the feasibility of TracerETL, it also highlights key insights and limitations that inform future research on standalone HTAP systems.

- ***Deterministic partitioning***. TracerETL may face limitations in dynamic environments, such as evolving partition keys and schema drift. In such cases, maintaining the validity of trace vectors may require adjusting partition boundaries or reorganizing the entire partition layout, potentially incurring nontrivial overhead. Promising directions to address this challenge include adaptive fanout control and lightweight online repartitioning strategies. As this limitation is shared by many systems based on static partitioning, we believe that tackling this core issue presents an important avenue for future research.
- ***ETL-aware query optimizer***. To fully support real-time HTAP workloads, query optimizers must account for performance interference at runtime. Even with carefully partitioned layouts

and separate I/O paths for OLTP and OLAP, unmonitored I/O contention can degrade query latency. Leveraging real-time statistics for adaptive planning is a promising direction.

- ***Cloud-scale disaggregated storage***. Extending TracerETL to cloud-native ETL architectures backed by disaggregated storage systems is a compelling direction for future work. Such integration could enable scalable, low-latency data transformation that aligns with emerging trends in cloud database design.
- ***Applicability of counting-based tuple tracking***. The trace vector mechanism enables instant tuple discovery by combining per-tuple trace vectors with lightweight per-partition metadata. These design principles generalize beyond TracerETL, offering a scalable alternative to indexing or full-table scans in HTAP engines, especially those employing log-structured or progressively partitioned storage in cloud-native environments.

7 Conclusion

HTAP has become increasingly important, yet standalone DBMSs still struggle to deliver real-time performance due to interference between transactional and analytical workloads. To address this gap, we presented TracerETL, a progressive ETL framework that enables low-latency data transformation through counting-based tuple tracking. At the core of TracerETL is the trace vector, a lightweight structure generated from per-partition counters. This design enables instant tuple location discovery even during gradual data transformation. By unifying in-place updates and out-of-place inserts, TracerETL achieves space-efficient version management and write-optimized ingestion. Our implementation in PostgreSQL demonstrates that counting alone is sufficient to support fast, scalable, and low-overhead tuple discovery in disk-based HTAP systems. We believe that this insight—embedding deterministic, per-tuple trace vectors within log-structured partition trees—offers a broadly applicable design principle for standalone DBMSs, distributed LSM-based engines, and modern log-structured storage systems.

Acknowledgments

We appreciate the reviewers' insightful feedback. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2022R1A2C2008427). Additionally, SK hynix Inc. provided partial support through a research project.

References

[1] 2020. HammerDB Version 4.3. Available at https://github.com/TPC-Council/HammerDB/releases/tag/v4.3.

[2] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Zongchang (Jim) Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Zhi (Adam) Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Yalan (Maya) Meng, Prashant Mishra, Jay Patel, Rajesh S. R., Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Ye (Justin) Tang, Junichi Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Gensheng Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divyakant Agrawal, Jeff Naughton, Sujata Kosalge, and Hakan Hacıgümüş. 2021. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2986–2997. doi:10.14778/3476311.3476377

[3] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. 2002. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal* 11, 3 (Nov. 2002), 198–215. doi:10.1007/s00778-002-0074-9

[4] Adnan Alhomssi and Viktor Leis. 2023. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 6 (apr 2023), 1426–1438. doi:10.14778/3583140.3583157

[5] Amazon Web Services, Inc. 2023. Amazon Aurora zero-ETL Integration with Amazon Redshift. https://aws.amazon.com/rds/aurora/zero-etl/. Accessed: 2025-04-10.

[6] Amazon Web Services, Inc. 2023. AWS zero-ETL. https://aws.amazon.com/what-is/zero-etl/. Accessed: 2025-04-10.

[7] Oracle Corporation and/or its affiliates. 2023. MySQL 8.0 Reference Manual: InnoDB storage engine: InnoDB and Online DDL: Online DDL Performance and Concurrency. https://dev.mysql.com/doc/refman/8.0/en/innodb-online-ddl-performance.html.

[8] Apache Parquet. 2024. Apache Parquet File Format. Retrieved July 28, 2024 from https://parquet.apache.org/docs/file-format/ https://parquet.apache.org/docs/file-format/.

[9] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-Invented. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2205–2217. doi:10.1145/3514221.3526045

[10] Jens Axboe. 2023. Efficient IO with io_uring. Retrieved October 15, 2023 from https://kernel.dk/io_uring.pdf

[11] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 168–180. doi:10.1145/3448016.3452831

[12] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. Available at https://panthema.net/tlx.

[13] Citus Data. 2020. Tools for running CH-benCHmark with HammerDB. https://github.com/citusdata/ch-benchmark.

[14] Citus Data, Inc. 2024. Citus GitHub project: Distributed PostgreSQL as an extension. Retrieved February 28, 2024 from https://github.com/citusdata/citus https://github.com/citusdata/citus.

[15] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The Mixed Workload CH-BenCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems* (Athens, Greece) *(DBTest '11)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. doi:10.1145/1988842.1988850

[16] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proc. ACM Manag. Data* 1, 1, Article 46 (may 2023), 27 pages. doi:10.1145/3588726

[17] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *Proceedings of the 2021 International Conference on Management of Data* (Xi'an, Shaanxi, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2490–2502. https://doi.org/10.1145/3448016.3457551

[18] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 79–94. doi:10.1145/3035918.3064054

[19] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 505–520. doi:10.1145/3183713.3196927

[20] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush the Wacky Continuum. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 449–466. doi:10.1145/3299869.3319903

[21] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: Granulating LSM-Tree Compactions Correctly. *Proc. VLDB Endow.* 15, 11 (jul 2022), 3071–3084. doi:10.14778/3551793.3551853

[22] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. doi:10.1145/2463676.2463710

[23] Nuno Faria, José Pereira, Ana Nunes Alonso, Ricardo Vilaça, Yunus Koning, and Niels Nes. 2023. TiQuE: Improving the Transactional Performance of Analytical Systems for True Hybrid Workloads. *Proc. VLDB Endow.* 16, 9 (jul 2023), 2274–2288. doi:10.14778/3598581.3598598

[24] Google Inc. 2023. Google AlloyDB for PostgreSQL. https://cloud.google.com/products/alloydb. Accessed: 2024-04-10.

[25] The PostgreSQL Global Development Group. 2023. Documentation PostgreSQL 15: Chapter 15. Parallel Query. Retrieved February 28, 2023 from https://www.postgresql.org/docs/15/parallel-query.html

[26] The PostgreSQL Global Development Group. 2023. Documentation PostgreSQL 15: Chapter 32. Just-in-Time Compilation (JIT). Retrieved February 28, 2023 from https://www.postgresql.org/docs/15/jit.html

[27] The PostgreSQL Global Development Group. 2023. Documentation PostgreSQL 15: Chapter 70. GIN (Generalized Inverted Index) Indexes. Retrieved February 28, 2023 from https://www.postgresql.org/docs/15/gin.html

[28] Hemant Saxena. 2024. Github repository for LASER (ICDE'23). Retrieved December 1, 2024 from https://github.com/hemant271990/laser-lsm-htap https://github.com/hemant271990/laser-lsm-htap.

[29] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3072–3084. doi:10.14778/3415478.3415535

[30] MongoDB Inc. 2023. MongoDB's Issue Tracker: WiredTiger [WT-6833] - Implement asynchronous IO using io_uring API. Retrieved October 15, 2023 from https://jira.mongodb.org/browse/WT-6833

[31] Jongbin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-Lived Transactions Made Less Harmful. 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, 495–510. doi:10.1145/3318464.3389714

[32] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. 2022. Diva: Making MVCC Systems HTAP-Friendly. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 49–64. doi:10.1145/3514221.3526135

[33] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. 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, 1675–1687. doi:10.1145/2882903.2882905

[34] Ralph Kimball and Margy Ross. 2002. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling* (2nd ed.). John Wiley & Sons, Inc., USA.

[35] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1790–1801. doi:10.14778/2367502.2367518

[36] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-Time Analytical Processing with SQL Server. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1740–1751. doi:10.14778/2824032.2824071

[37] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2020. KVell+: Snapshot Isolation without Snapshots. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 24, 17 pages.

[38] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-Tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 302–313. doi:10.1109/ICDE.2013.6544834

[39] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, and Andrew Pavlo. 2020. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *Proc. VLDB Endow.* 14, 4 (dec 2020), 534–546. doi:10.14778/3436905.3436913

[40] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. *Greenplum: A Hybrid Database for Transactional and Analytical Workloads*. Association for Computing Machinery, New York, NY, USA, 2530–2542. https://doi.org/10.1145/3448016.3457562

[41] S. Manegold, P. Boncz, and M. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge Data Engineering* 14, 04 (jul 2002), 709–730. doi:10.1109/TKDE.2002.1019210

[42] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2000. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 339–350.

[43] Microsoft Corporation. 2025. SQL Server Computed Columns. https://learn.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms191250(v=sql.105)?redirectedfrom=MSDN. Accessed: 2025-07-26.

[44] Elena Milkai, Yannis Chronis, Kevin P. Gaffney, Zhihan Guo, Jignesh M. Patel, and Xiangyao Yu. 2022. How Good is My HTAP System?. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1810–1824. doi:10.1145/3514221.3526148

[45] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (mar 1992), 94–162. doi:10.1145/128765.128770

[46] Amazon Web Services Inc. or its affiliates. 2023. Amazon Aurora zero-ETL integration with Amazon Redshift. https://aws.amazon.com/about-aws/whats-new/2023/06/amazon-aurora-mysql-zero-etl-integration-redshift-public-preview/.

[47] Oracle. 2023. MySQL 8.0 Reference Manual: 15.8.6 Using Asynchronous I/O on Linux. Retrieved October 15, 2023 from https://dev.mysql.com/doc/refman/8.0/en/innodb-linux-native-aio.html

[48] Oracle Corporation. 2009. Oracle Database 11g Release 2: Reference Partitioning. https://docs.oracle.com/cd/E11882_01/server.112/e25523/part_admin001.htm. Accessed: 2024-04-10.

[49] Oracle Corporation. 2025. Oracle Virtual Columns. https://www.oracletutorial.com/oracle-basics/oracle-virtual-column. Accessed: 2025-07-26.

[50] Oracle Corporation and/or its affiliates. 2022. Heatwave User Guide. Available at https://downloads.mysql.com/docs/heatwave-en.pdf.

[51] Oracle Corporation and/or its affiliates. 2022. MySQL HeatWave: One MySQL Database for OLTP, OLAP, and Machine Learning. Available at https://www.oracle.com/a/ocom/docs/mysql-database-service-with-heatwave.pdf.

[52] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (jun 1996), 351–385. doi:10.1007/s002360050048

[53] Postgres Professional. 2024. PostgreSQL extension implements a Foreign Data Wrapper (FDW) for RocksDB. Retrieved December 1, 2024 from https://github.com/postgrespro/lsm https://github.com/postgrespro/lsm.

[54] 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

[55] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 497–514. doi:10.1145/3132747.3132765

[56] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2023. One-Shot Garbage Collection for In-Memory OLTP through Temporality-Aware Version Storage. *Proc. ACM Manag. Data* 1, 1, Article 19 (may 2023), 25 pages. doi:10.1145/3588699

[57] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine for Semi-Sorted Data. *Proc. VLDB Endow.* 10, 13 (sep 2017), 2037–2048. doi:10.14778/3151106.3151108

[58] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. 2013. Making Updates Disk-I/O Friendly Using SSDs. *Proc. VLDB Endow.* 6, 11 (aug 2013), 997–1008. doi:10.14778/2536222.2536226

[59] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F. Ilyas. 2023. Real-Time LSM-Trees for HTAP Workloads. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 1208–1220. doi:10.1109/ICDE55515.2023.00097

[60] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17, 12 (Nov. 2024), 3731–3744. doi:10.14778/3685800.3685802

[61] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. Accelerating Analytical Processing in MVCC Using Fine-Granular High-Frequency Virtual Snapshotting. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 245–258. doi:10.1145/3183713.3196904

[62] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 731–742. doi:10.1145/2213836.2213946

[63] SQLite. 2025. SQLite Generated Columns. https://www.sqlite.org/gencol.html. Accessed: 2025-07-26.

[64] Sivaprasad Sudhir, Wenbo Tao, Nikolay Laptev, Cyrille Habis, Michael Cafarella, and Samuel Madden. 2023. Pando: Enhanced Data Skipping with Logical Data Partitioning. *Proc. VLDB Endow.* 16, 9 (jul 2023), 2316–2329. doi:10.14778/3598581.3598601

[65] Junichi Tatemura, Tao Zou, Jagan Sankaranarayanan, Yanlai Huang, Jim Chen, Yupu Zhang, Kevin Lai, Hao Zhang, Gokul Nath Babu Manoharan, Goetz Graefe, Divyakant Agrawal, Brad Adelberg, Shilpa Kolhar, and Indrajit Roy. 2023. Progressive Partitioning for Parallelized Query Execution in Google's Napa. *Proc. VLDB Endow.* 16, 12 (jul 2023), 3475–3487. doi:10.14778/3611540.3611541

[66] The PostgreSQL Global Development Group. 2023. PostgreSQL: Documentation for PostgreSQL 15: Chapter 30.4. Asynchronous Commit. https://www.postgresql.org/docs/15/wal-async-commit.html.

[67] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. GRF: A Global Range Filter for LSM-Trees with Shape Encoding. *Proc. ACM Manag. Data* 2, 3, Article 141 (May 2024), 27 pages. doi:10.1145/3654944

[68] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 473–488. doi:10.1145/3183713.3196895

[69] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, Yuan Gao, Qilin Dong, Junxiong Zhou, Jeremy Wood, Goetz Graefe, Jeff Naughton, and John Cieslewicz. 2020. F1 Lightning: HTAP as a Service. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3313–3325. doi:10.14778/3415478.3415553

[70] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-in-Place Key-Value Store for Modern Storage. *Proc. VLDB Endow.* 16, 1 (sep 2022), 99–112. doi:10.14778/3561261.3561270

Received April 2025; revised July 2025; accepted August 2025