# Diva: Making MVCC Systems HTAP-Friendly

Jongbin Kim*
Hanyang University

Jaeseon Yu*
Hanyang University

Jaechan Ahn
Hanyang University

Sooyong Kang
Hanyang University

Hyungsoo Jung
Hanyang University

## ABSTRACT

Multiversion concurrency control (MVCC) and design principles thereof are ingrained in modern database management systems, thus promoting remarkable progress in managing online transaction processing (OLTP) workloads for decades. However, MVCC systems would battle two vital concerns when facing hybrid transactional/analytical processing (HTAP). The first concern is to ensure rapid version searching for analytic queries with less I/O, and the second concern is to reclaim garbage data versions promptly for easing the strain on storage footprint. These are often tightly coupled since many MVCC systems rely on unified version storage that poses a space-time tradeoff in HTAP, giving rise to disappointing performance metrics that may negatively stereotype OLTP-friendly MVCC systems. This paper refutes the stereotype resulting from coupled design concerns and addresses the core problem by proposing Diva (**D**ecoupling **I**ndex from **V**ersion d**A**ta) that physically separates version index from version data; for decoupled concerns, we devise independent management policies: *provisional* version indexing and *time interval*-based version garbage collection. The separation of coupled concerns would render legacy disk-based MVCC systems more HTAP-friendly. We applied Diva to two full-fledged database systems—PostgreSQL and MySQL—and demonstrated that the systems with Diva escaped the space-time tradeoff under hybrid transactional/analytical workloads.

## CCS CONCEPTS

• **Information systems → DBMS architectures**; **Database transaction processing**; **Online analytical processing engine**.

## KEYWORDS

MVCC; HTAP; version searching; version cleaning

**ACM Reference Format:**
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 (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3514221.3526135

*Equal contribution †Contact author: Hyungsoo Jung


*SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA*

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526135

## 1 INTRODUCTION

MVCC has deeply permeated transaction processing systems and imbued modern database management systems (DBMSs) with all the virtues of transaction properties. It manages multiversioned data for point-in-time consistency to rescue conflicting operations that may otherwise get rejected/delayed in non-MVCC systems; however, it comes at the cost of storing multiple versions of data in databases. Over the past decades, much research in this area focused on system designs that enable DBMSs to offer better efficiency while ensuring correctness, especially in OLTP workloads. However, just because MVCC gains widespread acceptance in practice does not necessarily justify all the design decisions made in development since what looks seriously concerned in hindsight was latent at the time. Hence, we reexamine well-established DBMS designs in disk-based MVCC systems and renew legacy ones by unraveling design complexity from a new perspective to cope with HTAP workloads.

As databases evolved, many of the then-new ideas came up at different times and sometimes became entangled in design conflicts with prior decisions. When MVCC met database recovery decades back, a design conflict emerged; i.e., MVCC needs a history of data versions to offer point-in-time consistency, whereas recovery needs the most recently committed data version. Databases embraced this conflict by combining what MVCC and recovery required in unified storage space, thereby engendering three coupled concerns: recovery, version searching, and version cleaning. How? Unified management gives a data version *triple* roles: (i) *recovery target* if it is the latest commit, (ii) *search link* for version lookup, and (iii) *old version*. This design pervades database vendors in practice; many systems—Oracle, SQL Server, MySQL, and PostgreSQL—use the same, or something similar, in their storage architecture.

Unified storage architecture has strengths in that it reduces data redundancy and eliminates the burden of managing consistency. However, it is challenging to satisfy all three concerns for MVCC and recovery when facing HTAP workloads. For instance, the tightly coupled concerns of MVCC—rapid version searching and prompt cleaning of stale data versions—would become a thorny issue in a milieu where update transactions and complex analytic queries arrive together. Prior studies [41, 43, 48, 91] revealed that MVCC DBMSs, regardless of engine types, can exhibit at least one of dreadful performance metrics—low update rate, long query latency, and space expansion—if disrupted. Therefore, decoupling the three intertwined issues is vital to many DBMSs since they would suffer worse performance turmoil under complex hybrid loads.

Although the issue is critical, we have not witnessed prior research challenging the central premise of unified storage architecture, perhaps due to an uphill battle with I/O efficiency. In particular, prompt cleaning of data versions causes frequent storage access and repairs of broken search links, harming version search. Moreover,

version cleaning takes longer to identify stale data if storage contains many recovery targets. Due to these daunting costs, none of the disk-based DBMSs eagerly perform version cleaning, although suffering a significant slowdown in analytic queries. Instead, recent years have seen efforts [10, 39, 44, 49, 52, 68, 74, 79, 92] from industry and academia in that they build ETL pipelines, or its variants, to convert one data format emitted from OLTP workloads to OLAP-friendly columnar formats. Although such systems often maintain multiple dedicated instances for OLTP/OLAP loads and strive to reduce ETL latency between the instances, they offer the superior capability for performance isolation with HTAP.

We approach this problem from a different perspective in that we retain underlying legacy engines and aid them to be more HTAP-friendly by decoupling entangled designs. To this end, we propose Diva (**D**ecoupling **I**ndex from **V**ersion d**A**ta) that addresses the matter by applying separation of concerns [33] to MVCC databases; primarily, we pursue the clean separation of version searching and version cleaning. The intuition behind our rationale is that separating the coupled concerns would grant design freedom to database designers in alleviating their dilemmas. Of three coupled concerns, database recovery only minds the most recently committed data, so *co-locating it with uncommitted data in the main index* is a feasible solution for separating recovery from MVCC. We use one of the prior arts [17, 28, 41] conforming to our separation principle, thus isolating database recovery from MVCC.

Next, the core issues of MVCC await safe separation. When recalling their design goals, we perceive that *version searching requires a proper version index for time efficiency*, whereas *version cleaning needs an efficient method for identifying and removing stale data for space efficiency*. Since their concerns are conflict-free, it leads to a simple remedy that *constructs a version index physically outside of version data* with *independent management*. The essence of realizing such clean separation lies in concrete techniques responsible for indexing and cleaning data versions. To this end, we propose two design principles, i.e., *provisional version indexing* (§4) and *time interval-based version garbage collection* (§5). Below is our intuition behind concrete principles.

**Versions are ephemeral data streams**. Data versions are *continuous* and *visible for a sliding time window*, fitting the working set model, and they would *never survive failure* but be *durable* while alive, evading the duty of database logging and recovery. These characteristics foster our rationale for a concrete technique—*provisional* version index—that resembles the UNIX `inode` structure, and it *stays* in storage once DBMSs start and *vanishes* upon shutdown (i.e., *quasi-durable*). We confine version lookup to provisional indexes without referring to data versions in storage while navigating, thus attaining time efficiency and separation.

**The art of matchmaking**. *Data versions end their life once out of transactions' sight*; hence, the presence of a transaction whose snapshot falls into versions' visibility ranges decides data lifetime. Vacuuming should fetch versions from storage, check staleness by tracking live references, and purge them once confirmed. We improve conventional approaches using the duality between data and queries [24]; we set up preset *time intervals* and *wait for* versions and transactions included within the interval to be collected. This *matchmaking* eschews costly reference tracing and enables prompt *trimming* of stale data in bulk once *out of sight*.

All these ideas form a new approach to building MVCC databases facing the growing demand of HTAP workloads. We validate our design principles by applying Diva to recent versions of PostgreSQL and MySQL. Experimental evaluation demonstrates that the systems with Diva continue to process HTAP workloads while *escaping* the space-time tradeoff. This work is the first attempt to decouple salient design concerns successfully and make any disk-based MVCC databases more HTAP-freindly.

In summary, we make the following contributions:

- We show that the coupled designs in legacy disk-based MVCC systems are responsible for low update rate, long query latency, and space expansion under HTAP workloads.
- We show how to decouple the three design concerns in OLTP-friendly MVCC DBMSs and propose Diva for separating version searching and version cleaning while isolating recovery from MVCC using one of the prior arts.
- Diva introduces two methods for clean separation: provisional version indexing and interval-based version garbage collection. These techniques improve all performance metrics while keeping memory and storage footprints, I/O, and runtime overhead lower than vanilla systems.
- Evaluation with MySQL and PostgreSQL shows that Diva can aid any MVCC systems to be more HTAP-friendly by addressing deep-seated resource coupling problems.

## 2 BACKGROUND AND RELATED WORK

### 2.1 Background

Modern DBMSs rely on MVCC and ARIES [60] that have paraded its widespread adoption in practice. We start this section with a brief introduction of their core concepts (see [19, 36, 89]).

**Multiversion concurrency control**. The rationale for steering concurrent access through multiversion data is to prevent readers from being blocked on data already accessed by writers (or vice versa) through the concept of point-in-time consistency. Systems relying on MVCC usually assign a snapshot to a transaction and provide an illusion of a snapshot database comprising all the then-current data items at the time of the snapshot. It requires that the systems maintain a set of data versions visible to live transactions somewhere until the transactions commit. Although essential in MVCC, there is *no golden rule for managing temporary data versions*; DBMSs differ widely in version management.

**Transactional recovery**. ARIES restores a DBMS to be consistent using the most recently committed data. DBMSs store committed data in two different places. First, systems (e.g., PostgreSQL [82]) keep old versions in the main index for instant recovery, despite a high risk of frequent index reorganization under an upsurge of data versions. Second, other systems (e.g., MySQL) write versions to separate table space, where the primary index remains intact regardless of the volume of versions. When old committed versions are no longer necessary for recovery, DBMSs should clean garbage data to reduce space waste. DBMSs use different vacuuming strategies depending on how they organize committed data items.

**Coupled concerns**. From the MVCC perspective, transaction undo logs become natural candidates for serving old versions of data. In this respect, garbage collection can clean both undo logs and stale versions all at once if they are unified. The effect of killing two

birds with one stone was compelling to database designers so that version storage plays dual roles in transaction engines, coupling the design concerns of MVCC and recovery. The latent downside lies in delayed purging that may bloat version space, and it reveals itself when systems manage a growing pile of data versions under HTAP workloads. Besides that, MVCC poses a dilemma to database designers: purging stale versions should work without interfering with version lookup.

**Debunking the myth of unified management**. Version management is onerous to disk-based DBMSs under circumstances beyond their control. Prior attempts have never challenged unified architecture; thus, the poor metrics would devalue the presumed benefits of unified storage management. The past hesitancy for further elaboration is due to the exorbitant I/O cost charged for modifying *individual*, *temporary* versions. The over-priced cost is germane to two entangled activities. First, version cleaning must fix broken search links for version lookup. Second, version lookup needs a search structure vulnerable to version removal. These tightly coupled concerns may inflate version space and cause a slowdown in version search under HTAP loads, thereby leading to performance drop. Our analysis here conforms with what prior studies [41, 43, 48, 91] have reported. So, the plight necessitates separating unified resources to end the impasse of recovery, version search, and cleaning, the latter two of which are our primary focus.

## 2.2 Related Work

In an effort to analyze state-of-the-art systems regarding the concerned design factors, we use a shorthand notation, <·|·>, to express how they managed design concerns, where <·| denotes the coupling status between recovery and MVCC, whereas |·> indicates the status between version search and cleaning in MVCC. Two binary state (i.e., **c**: coupled and **s**: separated) variables produce four combinations: <**c**|**c**>, <**c**|**s**>, <**s**|**c**>, and <**s**|**s**>. Figure 1 shows the summary excerpted from what we have reviewed the systems.

**Traditional disk-based databases**. Disk-based DBMSs can be classified into three types from the undo logging perspective. First, MySQL [13] and Oracle [15] write undo logs to a separate area—rollback and undo segments [11, 12]—and use them to provide users with a consistent view. Second, PostgreSQL stores old data versions in the main tablespace for efficient version search and hardens the state information of transactions (i.e., `pg_xact` [72]) for correct, instant recovery. Third, Azure SQL database [17] adopts a hybrid approach that decides the location dynamically, and it uses *unified* persistent version stores (PVS) for MVCC and writes additional information (i.e., *aborted transaction map*) for recovery. Despite the long history, design concerns on recovery and MVCC still get entangled—likewise, version search and reclamation act on unified version chains retaining design coupling. Hence, the selective removal of stale versions for compact version space will cause enormous I/O costs in mending broken links that have compelled traditional disk-based MVCC systems to remain in <**c**|**c**>.

**Modern key-value stores**. Despite falling beyond our main scope, key-value stores are an active subject in many research communities that are paying growing attention to the upheaval of HTAP [76]. Of modern key-value stores, we review two representative systems: RocksDB [35] and WiredTiger [63]. RocksDB, backend

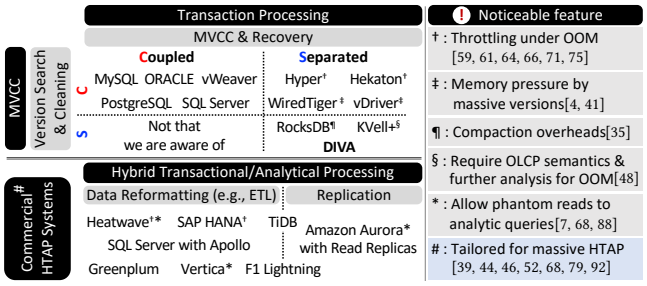



**Figure 1: A summary of literature review.**

storage for MyRocks [54, 80] and MongoRocks [51], writes only *committed* data to log-structured merge trees [67], and key-value pairs in storage work as old data versions, *not* undo logs. Version search uses a combination of *bloom filters* and *summary indexes*, and its global compaction can perform selective version removal. Therefore, we view RocksDB as *decoupled*, i.e., <**s**|**s**>. WiredTiger, the storage engine for MongoDB [62], separates its recovery concern from version management by adopting the *no-undo* policy. However, redo-only recovery is more susceptible to *memory pressure*, possibly caused by a growing volume of memory-resident data (e.g., *versioned* data) [77, 90], leading operating systems into thrashing. Since version search and cleaning access *memory-resident* skip lists, WiredTiger is tagged as <**s**|**c**>.

**Improvements to the space-time tradeoff**. There have been recent proposals to address the space-time tradeoff. vDriver [41] mitigates the ballooning of version space due to long-lived transactions by separating recovery from MVCC. It focused on tight space management of version storage but less concentrated on version search; although it enhances linear version lookup using a *memory-resident* index, version search and cleaning access the same structure so that we tag it as <**s**|**c**>. vWeaver [42] improved scan operations in a large version set under HTAP workloads by *frugal skip lists* and weaving techniques readily applicable to any MVCC systems (no architectural changes, i.e., <**c**|**c**>). As manifested in the paper, the fine-grained version reclamation may ruin their efficient search structures, left as an open issue. KVell+ [48] introduced the OLCP query concept that propagates old versions directly to the *concerned* active transactions, which separates all the design concerns nicely (i.e., <**s**|**s**>). However, the proposal demands a paradigm shift from conventional query semantics to its OLCP concept, and assessing the system behavior with limited resources (e.g., low in memory) requires further elucidation.

**Memory-optimized database engines**. Since *in-memory* engines only store committed data in storage, they avoid undo logging and devote themselves to version management—all the operations act on in-memory data structures without concerning I/O costs. Although SAP HANA [47], Hekaton [32], and Steam [22]—an extension to Hyper [40]—maintain entangled designs for managing versions, they succeeded in dealing with version management mainly due to the absence of the strict I/O concern. In-memory database systems, therefore, stay in <**s**|**c**> with *memory-optimized* designs.

**Commercial HTAP solutions**. If we view data reformatting as the core of ETL, many commercial products fall into the ETL domain[1]: Heatwave [68], F1 Lightning [92], SAP HANA [79], SQL

[1]Some claim to distance themselves from ETL, although reformatting data.

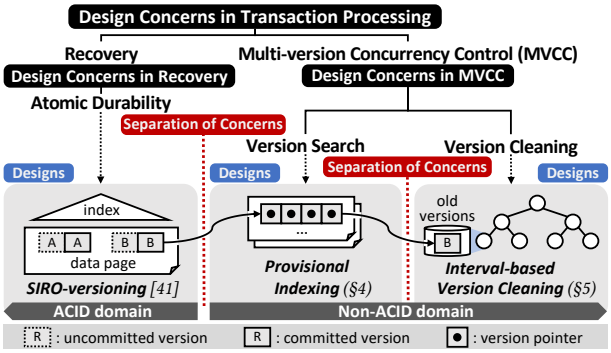



**Figure 2: Separation of the coupled concerns in databases.**

Server with Apollo [46], Vertica [45], Greenplum [52], and TiDB [39]. Most of them strive to reduce ETL latency for reformatting data. Some [46, 47, 68] adopt memory-optimized designs to gain performance despite more restrictions. Other systems [68, 88] permit phantom reads due to the lower isolation level (i.e., READ COMMITTED (RC)). Of HTAP solutions, three systems—Aurora, F1 Lightning, and TiDB—replicate data to earn high availability. Amazon Aurora with read replicas supports HTAP but exposes performance limitations (e.g., purge lag). Their guideline [7] recommends that replicas use the RC isolation for analytic queries to avoid performance problems. In this regard, the systems [7, 68, 88] with the RC isolation seem to trade MVCC to reduce latent space overhead, despite phantom reads. Overall, commercial HTAP systems are generally well optimized for large-scale HTAP, although some still possess design coupling. We take a different stance in that we focus on OLTP-friendly DBMSs and aid them to be more HTAP-friendly; Diva may assist some of the commercial products above.

**Summary**. Our analysis elicits detailed information regarding how significant concerns become coupled, although designers made their decisions with careful foresight at that time.

## 3 DESIGN OVERVIEW OF DIVA

This section provides insight into how Diva addresses the coupled concerns in MVCC systems. Figure 2 illustrates the hierarchy of design concerns in DBMSs with our separation principle imposed on adequate places. In essence, we restrict each concern to its isolated resource, and concrete design principles stated below would apply to any disk-based MVCC systems without loss of generality.

**Separation of recovery and MVCC**. Transactional recovery can be isolated from MVCC by separating the most recently committed data from its version store; then, recovery would solely manage redo logs and never access version storage. Stale undo logs beyond the recovery concern can safely relocate to version storage, acting as data versions in the MVCC landscape. Among the prior arts [17, 28, 41] is SIRO-versioning [41] that co-locates a record and its first old version in the main index and stores the rest versions in separate version space. Their versioning rule conforms to the separation principle and befits our approach; hence, we parlay it into our design in the rest of this paper.

**Separation of search and reclamation**. After isolating recovery from MVCC, we decouple two concerns in MVCC by physically separating a version index from referred data versions and adopting independent management policies; then, a version index and data versions can never cross-reference. Cleaning expired indexes or versions is asynchronous since we always inspect their staleness first, ensuring that none of the live transactions access stale data or index, which safeguards asynchronous cleaning policies.

**Design rationale for version indexing**. When delving deeper into multi-version concurrency control, we discovered two meaningful characteristics: (1) data versions are *ephemeral* data streams, and (2) the lifetime of data is determined *not* solely by its visibility range *but* by live transactions whose snapshot overlaps with the range. The two characteristics provide an appropriate rationale for our designs. The first characteristic shapes a *provisional* version index that resembles an `inode` in UNIX file systems. Since the pure `inode` design is not perfectly apposite for our purpose, we further refine our `inode`-like index by reducing internal fragmentation through multi-level index arrays (§4.3.1) and reclaiming stale index space by *online* space compaction (§4.3.2). More importantly, a provisional index avoids logging and recovery.

**Design rationale for version cleaning**. From the two characteristics, we gleaned valuable insight into *eliminating* expensive reference tracing in conventional version cleaning. The insight is, if we view reference tracking as a *query* acting on version streams and continuous transactions to judge staleness of *individual data*, then we can transform it into an *equivalent form* using the well-known concept of the *duality of data and queries* in continuous stream processing [23, 24, 50]. We never execute reference tracking queries in a new paradigm, but data versions and transactions now perform simple actions equivalent to evaluating such queries. **We transform the reference tracking query by laying out preset *time intervals*.** Then, **incoming versions and transactions bind themselves into an interval to which their visibility range and snapshot belong**. Once all the transactions, bound to an interval, have *committed*, the interval has a *null* reference and permits us to *wipe out* all the versions in it. This design *obviates* the need for costly reference tracking and *trims* stale versions in bulk; of course, the length of an interval affects tightness, so it is adjustable.

## 4 PROVISIONAL VERSION INDEXING

### 4.1 A Homage to the UNIX Philosophy

**Honoring tradition**. Since its emergence [73], UNIX-style file systems have had a solid foundation in managing file data in storage, and the `inode` structure has proven great practicality. At the heart of the `inode` design is its separated materialization from the data blocks, which conforms to our separation principle. Our version index resembles an `inode` in many ways; (1) a version index can span from direct to single indirect pointer blocks to locate a variable-sized set of visible data versions, and (2) it manages a shared pool of pre-allocated index nodes similar to those in file systems.

**Paving new paths**. The differences between our design and `inode` lie in two parts. First, we use *multi-granularity* index arrays to reduce *internal fragmentation* that may otherwise occur in fixed-sized arrays. The proper justification comes from the following observation; (short-lived) OLTP transactions and (long-lived) OLAP queries require *narrower* and *wider* windows of versions, respectively, to be reachable through the index. Second, *version space can be emptied upon restart* since no one needs versions in the previous generation, so crash consistency is unnecessary.

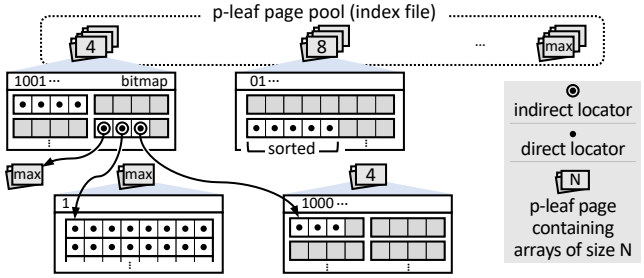



**Figure 3: The overall architecture of p-leaf index pages.**

## 4.2 Architecture

The base architecture of a provisional index inherits the layout of `inode` blocks in file systems. We initialize version index space on an index file with fixed-sized logical pages (pagination). An index page internally contains a pool of index arrays equal in size, and the array capacity may differ per index page (i.e., multi-granularity). The allocation state of each index array is managed by in-page array allocation bitmap. Like an `inode`, index arrays can work solely as either direct or indirect version locators; each entry in an index array comprises a triple of version id, lifetime information, and version locator. We call this collection of version index pages *provisional leaf* (or *p-leaf* in short) pages since they temporarily expand the covering space of a primary index (e.g., B+tree), reaching version storage. Figure 3 illustrates the version index layout featuring p-leaf pages with multigranular index arrays.

**Multigranular index arrays**. File systems (e.g., `ext4`) support a single file with a size up to 16 TiB, but p-leaf pages never face such a large amount of visible versions for a single record, so that index arrays range in capacity from $2^2$ to $2^8$ to avail oneself of the characteristics of transaction workloads; an indirect locator array can cover $(2^8)^2$ visible versions that serve the same number of live transactions. And arrays in a page have different types of locators (either direct or indirect), as in the first index page in Figure 3, for page utilization; of course, different-sized direct and indirect index arrays can form a hierarchy. Although there is no limit in increasing the indirection levels, a single indirection suffices to handle up to $2^{16}$ live transactions. Hence, **in practice, finding a version locator in Diva incurs at most two I/O operations** (cf. unbounded I/O in the legacy systems).

**Quasi-durable index space**. As with temporary tables [58, 69, 70] allowed in legacy databases, our version index need not survive failures, thus averting the heavy load of database logging and recovery. In addition, we never need other types of metadata (e.g., superblocks and data block bitmaps) to ensure crash consistency for *temporary* version space, nor do we enforce update ordering rules on our version index and separated data versions due to the clean separation between them. This simplicity only requires on-disk pages to be properly linked for correctness even on low-end servers (e.g., AWS EC2 t2.nano [9]), where page replacement occurs more often. As succinct memory-resident metadata just for bookkeeping the head of free index pages, we use *concurrent stacks*.

## 4.3 Managing Version Index Space

This section describes methods for managing our provisional version index, including concurrent access to the index.

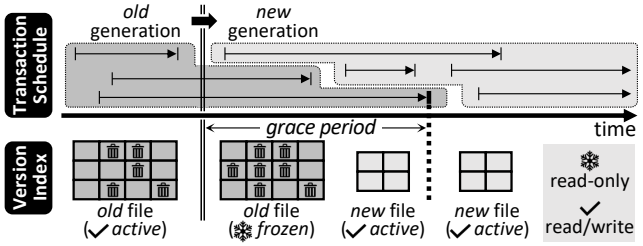



**Figure 4: Macro-compaction of the version index space.**

*4.3.1 Resource allocation.* According to the SIRO rule [41], the primary database index co-locates a record and its latest old version together. When a transaction updates a record, the latest record version needs to relocate to version storage, which may require a new p-leaf index array. If the record in a primary index already has a valid locator to its p-leaf array, it is enough to append the version pointer to the array. Otherwise, if it is the first time to push a version into the version store, we allocate a free array from the smallest capacity (=$2^2$) p-leaf page[2]. When an assigned array overflows, we double the capacity by assigning a new array twice in size, copy version pointers in the old array to the new one, and switch the record's array locator to the new one. Then we return the old (i.e., stale) array to its p-leaf page for recycling space.

When an overflow occurs in the maximum-sized array, we start indirection to extend the array capacity further. To this end, we assign an empty array (for indirect locators) with the smallest capacity to the record and make the first indirect locator in the array point to the full (maximum-sized) array. A new array with the smallest capacity is then allocated and pointed by the second indirect locator to store new version pointers. As data updates produce their versions, allocating free index arrays to accommodate all the incoming versions would quickly balloon the index space and incur I/O. We avoid excessive space expansion by using space compaction that reclaims stale index arrays *promptly* and *efficiently*.

*4.3.2 Space compaction.* Storage space keeps expanding if systems are remiss in not *cleaning* stale resources *promptly*, but fine-grained space reclamation is a *daunting* task in legacy disk-based databases. However, the separate materialization of a version index from data versions enables us to devise efficient cleaning strategies for the index and data versions. Note that there is no consistency issue in cleaning stale index arrays and garbage data versions, provided that whichever gets reclaimed occurs only after none of the live transactions access them. With this guarantee, two space compaction methods are devised solely for a version index: *macro-* and *micro-compaction*. Macro-compaction focuses on the entire index space when most space is filled with stale locators, whereas micro-compaction is for tightening dead space inside an index array.

**Macro-compaction**. The HTAP workloads make index space full of stale arrays once analytics complete. Macro-compaction, *online* resource reclamation, addresses such a waste of index space, and it uses a *grace period* to establish a strict *happen-before* relation between the old version index and upcoming transactions. The derived *invariant* is that new transactions would *never request* any from the old index space. This invariant permits us to *empty* the entire old space *on the spot*. Attaining the invariant is *extremely*

[2]We allocate more pages with $2^2$ capacity, if all of them are in use.

difficult if version storage collocates everything — undo logs, data versions, and version pointers — in the same space. The separation principle, on the other hand, easily ensures the invariant by separating a version index from version data.

We show the process of macro-compaction using the concept of graceful generation transition. As illustrated in Figure 4, it proceeds in three phases: (1. *begin transition*) *creates a new index file*, (2. *grace period*) waits for *quiescent states* requiring that transactions began before the quiescent period all commit, and (3. *end transition*) executes *deferred destruction* of old space by deleting the old index file. Upon creating a new index file in the first phase, the system enters a grace period during which the old index file becomes read-only (i.e., *frozen*) while the new index file allows read/write access to enable index creation and search for upcoming versions. When none of the live transactions can access the old index file (i.e., *invariant*), the grace period ends, and the system completes the generation transition by deleting the old file.

Although a quiescent phase is susceptible to long-lived transactions, the extended period does *never hurt* performance since it would only expand the index space for maintaining two index files for a moment. We activate macro-compaction only when the index file is full of garbage pages (refer to §6), and it would benefit systems by *instant reclamation* of version index space *on-the-fly*.

**Micro-compaction**. Foreground transactions, while accessing the index array, tighten valid entries inside the array by reclaiming stale ones while holding a reader-writer lock to be safe under concurrent access (§4.3.3). If such micro-compaction cannot secure enough free entries, this is the moment to expand the index array by relocating to a larger one. In contrast, if the array utilization is too low, a smaller array replaces the ample one by copying pointers from the old one and returns it to the pool.

*4.3.3 Managing concurrency.* Assuming that transaction concurrency control manages resource contention on the same data items in primary index pages, we focus on contention on our newly introduced p-leaf index pages; update transactions push version locators to p-leaf pages while analytic queries access them to obtain target version locators. Such concurrent access to our index structure would raise traditional resource contention issues between readers and writers on three occasions: (1) modifying the primary index page to set a locator to our p-leaf index, (2) allocating a p-leaf page, and (3) allocating an empty index array. Note that it is self-evident that there is no issue between readers.

**Contention between writers**. When update transactions are pushing version locators to p-leaf pages concurrently, malicious race conditions between writers may arise in two cases: (1) when they acquire a p-leaf page for a new index array, and (2) when they access the same in-page bitmap to search a free index array and mark it as *in-use*. Both races would occur when writers store their data versions for the first time or resize their arrays. We resolve the contention by using a concurrent stack for managing p-leaf pages with the same array capacity; a writer, once popped a p-leaf page from the stack, gains exclusive write access, which also precludes the second data race beforehand.

**Contention between a reader and a writer**. Malicious races between a reader and a writer occur when a reader traverses a p-leaf array while (1) a writer performs micro-compaction on the same array, or (2) a writer modifies a p-leaf array locator embedded in the primary index page after it resizes an index array. For such cases to happen, both should be allowed to access the same p-leaf array. Rather than inventing complicated mechanisms, we use a reader-writer lock per record to resolve the two cases. This strategy would not lead to performance regression since transaction concurrency control already steers concurrent access similarly. Even if we can avoid *locking* in the second case using more elegant RCU techniques [55, 56], we forbear from using it since it looks premature optimization at the moment.

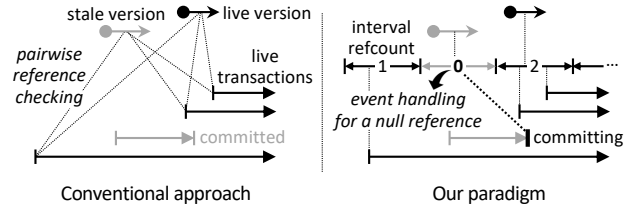



**Figure 5: Conventional approach versus our paradigm.**

## 5 VERSION GARBAGE COLLECTION

Garbage collection, in general, proceeds in two phases. The judgment of target data's staleness should occur first, and then cleaning stale data comes next. The judgment phase must ensure the absence of live references to stale data, and the vacuuming phase should purge identified garbage data without incurring heavy I/O. In version cleaning, one distinct tradeoff—between space (i.e., prompt cleaning) and I/O efficiency—is influential on disk-based DBMSs. However, we realize that just sacrificing timeliness to a reasonable margin can significantly raise I/O efficiency using batch cleaning of garbage data. Designing batch deletion poses pertinent questions: how do we *identify* reclaimable versions in a batch?, and how do we *organize* data versions in storage in a way that is apt for batch cleaning with less I/O? We address the former question using *interval-based* version-transaction matchmaking and resolve the latter by stratified version segmentation in storage.

### 5.1 The Power of a Matchmaker in Vacuuming

The duality between data and queries (§3) inspired us to transform the conventional approach into an efficient form—*interval-based matchmaking*—to identify and reclaim garbage versions in a batch by trading promptness for I/O efficiency. Figure 5 illustrates the difference between the conventional approach and our paradigm. **Matchmaking versions and transactions using an interval aids in deciding the fate of all versions in the interval at once**. We make a time inclusion relation play a matchmaker in a new paradigm. To this end, we invent a *hierarchical interval tree* for matchmaking, and versions in the same interval are in the same file segment. Later, the last outgoing transaction from the interval would activate the trimming of all the versions (i.e., deleting the file segment) without repairing broken search links due to our separated p-leaf index. Since the time interval plays dual roles—epoch and reference counting target, matchmaking fundamentally shares the same principle with garbage collection techniques based on intervals or epochs in main-memory DBMSs [32, 40, 47, 85] with the difference lying in concrete designs—hierarchical interval

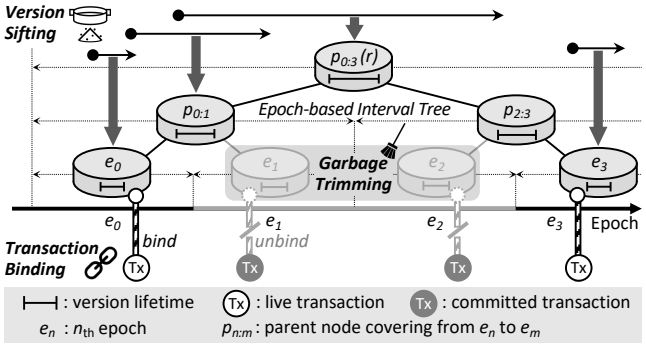



**Figure 6: Interval tree-based version garbage collection.**

trees—escaping the space-I/O tradeoff. So, **our approach enables fine-grained version cleaning in disk-based MVCC DBMSs.**

## 5.2 Matchmaker: An Epoch-based Interval Tree

To construct our matchmaker, a binary time-interval tree, we logically divide a timeline into constant intervals (i.e., *epochs*). Successive epochs spawn new leaf intervals from the *right-most* side on a timeline, and the parent's interval *subsumes* those of its child nodes (i.e., *parent-child interval invariant*). Each *leaf* interval in the tree has a reference counter to *bookkeep* the number of live transactions that began during the interval; the null reference to the interval ensures that all the versions whose visibility ranges are included in the interval are *reclaimable*. A leaf interval and its ancestors should exist in the presence of any of such transactions; i.e., an interval with a null reference (or no child) can be deleted from the tree. From above, we derive a *pruning rule*: incoming versions can be safely pruned in the *absence* of the matching interval since it logically implies a null reference to the interval.

Intuitively, our interval tree works as *per-interval reference tracking queries* for incoming versions and transactions. The matchmaking enables *consolidated reference counting* to an interval, thus allowing the *batch removal* of data versions. Although it seems to lose promptness, the matchmaking procedure itself costs nothing but waiting until the last outgoing transaction checks out. Figure 6 shows our interval tree and illustrates how version garbage collection works conceptually using the tree (see §5.4 for details). Our hierarchical interval tree diversifies version storage to reduce the potential risk of space expansion under long-running analytic queries with which many systems suffer from putting all of their eggs (i.e., data versions) into one basket (i.e., unified storage).

Given a p-leaf index and an interval tree, update transactions read and write both when storing their versions, while analytic queries read the p-leaf index to obtain version locators and update a leaf interval only for reference counting. We use two dedicated *background workers* for 1) macro-compaction of a provisional index and 2) the structure modification of our interval tree by inserting or deleting interval nodes together with their file segments.

## 5.3 Upkeep of a Rotation-free Interval Tree

Our interval tree *grows* as intervals holding one or more live transactions arrive and *shrinks* when transactions *drop* all the references to the interval. Unfortunately, textbook implementations of interval trees (e.g., Chapter 14 in [27]) cannot fit our purpose due to forceful tree rotations for balancing the tree, which may *forbid* concurrent access. We invent a *rotation-free* interval tree with *lock-free* insertion and deletion that *never rotate* the tree without *losing* efficiency. When inserting or deleting interval nodes, we modify our interval tree without violating the parent-child interval invariant. Insertion always occurs from the tree's right-most side, but deletion arises in arbitrary places although the tree's *left-most* side is a typical place of deletion. The focus is how the interval tree *grows* or *shrinks* as intervals come *in* and *out*.

*5.3.1 Lock-free insertion.* When a new interval arrives, we create a new parent node holding a new interval as a right child. We traverse from the right-most leaf interval to find an ancestor node whose *left* subtree ***covers a wider range*** than its *right* subtree, not *vice versa*. If detecting such a node, we *insert* the new parent between the node and its right subtree after *attaching* the right subtree to the new parent's left side. This sequence is to *balance* the tree, as exemplified in Figure 7-Ⓐ, Ⓒ, Ⓓ, and Ⓕ. On the contrary, if the existing tree is well balanced, the tree height grows by making the new parent a new root, and this is shown in Figure 7-Ⓑ & Ⓔ. As we keep inserting new nodes without deleting any, our interval tree would converge to a complete binary tree without rotating it since our insertion routine fully exploits the fact that new interval nodes arrive chronologically.

*5.3.2 Lock-free deletion.* For tree deletion, we maintain a separate *garbage queue* to keep deleted intervals for a short *grace* period to protect *ongoing* readers — i.e., update transactions that are traversing the tree to find matching intervals for their versions — from derailing. After quiescent states, we physically delete the garbage nodes and their file segments. Such deferred destruction of garbage

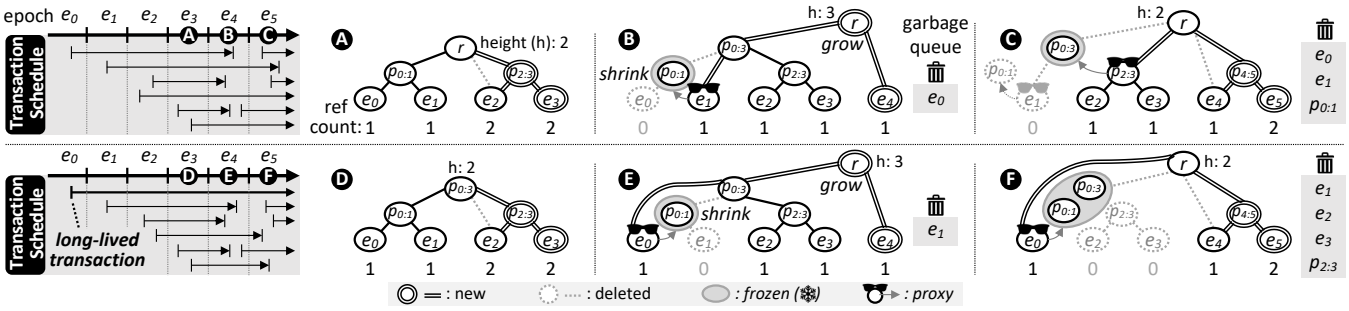



**Figure 7: Changes of the interval tree w/ (lower) and w/o (upper) a long-lived transaction.**

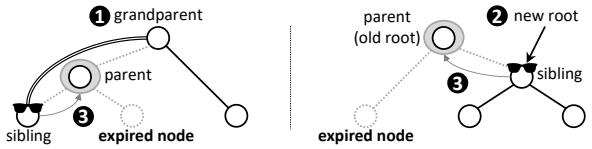



**Figure 8: Path compression w/ and w/o a grandparent node.**

nodes makes delete operations *lock-free*. A (sub)tree conceptually collapses into the void if both left and right subtrees are devoid of references from live transactions (see $p_{2:3}$ in Figure 7-❻). Our interval tree shrinks from the bottom while performing successive delete operations. This simple shrinking propagates upward until we find a parent node with a single child subtree.

When we spot a single parent-child path, we further shorten the tree traversal time through *path compression* that merges parent and child nodes by affixing the child subtree to its grandparent node, if it exists (Figure 8-❶). If the parent node is the root, the child node becomes the new root of the tree (Figure 8-❷). After path compression, the parent node becomes unreachable, although it still retains live versions in its file segment and exposes a valid interval for some live transactions to store their data versions into the file segment. To address the issue, we contrive a *proxy* mechanism that makes the live child node a proxy that 1) accepts incoming versions, matching the parent's interval, to its file segment on behalf of the parent (§5.4) and 2) temporarily holds the parent node for deletion (Figure 8-❸). A node can proxy multiple ancestors as a consequence of successive path compression (Figure 7-❻). We select a child node as a proxy for the parent node instead of vice versa to reduce latent data races in the latter case due to copying pointers and range values from the child node.

*5.3.3 Interval tree height analysis.* When transactions keep coming in and out, our interval tree grows from the right and shrinks from the left, thus maintaining a proper height. Here we provide a brief analysis of our interval-tree height. Figure 9 shows the worst-case shape of our interval tree (i.e., height unbalanced). Such scenarios occur when a new leaf node is inserted into the well-balanced interval tree (so a new root node is created), or when only one leaf node is alive in each right subtree after node deletions. It is certain that, in both cases, one or more (long-lived) transactions that began at the interval of the leftmost leaf node (i.e., the oldest transactions) are still alive. Then, we can formulate the maximum height of the interval tree in terms of the lifetime of the oldest live transaction, as shown in Theorem 5.1. It highlights that the height of our interval tree depends *only* on the lifespan of the oldest active transaction.

Theorem 5.1. *The height of the interval tree is not larger than* $\lceil \log_2 L_t \rceil$, *where* $L_t (\geq 1)$ *is the age of the oldest live transaction expressed as the number of spanning epochs.*

Proof. Since node deletions make additional rooms for accommodating new nodes, we assume the worst-case scenario that no more deletion would occur. As new nodes continue to arrive, our lock-free insertion makes the interval tree converge to a complete binary tree. To make a complete binary tree with height $H$, we need $2^H$ leaf nodes that require at least $2^H$ epochs. Hence, up to $2^H$ epochs (i.e., $L_t = 2^H$), the tree height is not larger than $H$. Therefore, given $L_t$, we can represent the height as $H \leq \lceil \log_2 L_t \rceil$. □

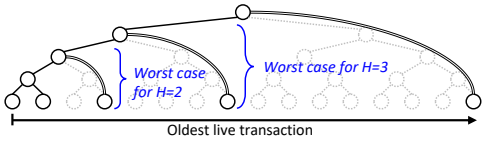



**Figure 9: The worst-balanced interval tree with height=4.**

## 5.4 Sift-Bind-Trim Version Cleaning

**Version sifting**. A data version with its visibility range — *start* and *end* timestamps — is *sifted* into an interval that *wraps* the range most *tightly*. If proper intervals are *absent*, we *prune* the version. However, confirming the absence of intervals should be careful since a node may proxy its ancestors' wider intervals. Given a version visibility range $R_v$ and the proxy's interval $I_p$, we face three cases: 1) $I_p$ includes $R_v$, 2) $R_v$ partially overlaps $I_p$, and 3) disjoint (*prunable*). In cases 1 and 2, the version is stored in the proxy's file segment since $R_v$ matches either the proxy's own or ancestors' intervals. Once reaching the destination, we store the version into its file segment. Hence, versions with similar lifetimes can *cluster* into the same interval through version sifting.

**Transaction binding**. A transaction, when commencing, binds itself to the current interval and increases its reference counter, which implicitly makes all the ancestors in the tree alive. Upon termination, it lowers the reference counter and delegates the job of deleting a node with *null* reference to the background worker that adds it to the garbage queue and waits for a quiescent period to ensure that none of the live transactions access it.

**Garbage trimming**. Since the sift-bind phases (i.e., matchmaking) eliminate time-consuming reference tracking to identify stale data, version cleaning just dequeues the garbage node from the garbage queue and trims its file segment to execute batch removal of all the garbage versions in it.

# 6 DESIGN PRINCIPLES IN ACTION

This section presents subtle issues in embodying Diva in disk-based DBMSs. We first discuss buffer designs and then describe topics pertinent to our provisional index and interval trees.

**Commonality**. First, we have contemplated buffer designs for our provisional index and version file segments and reached the design of a *separate, lightweight buffer* layer for each part, instead of integrating them into legacy database buffer management for *unified* buffer space, due to the potential risk of coupling all of the separated concerns again. Note that databases (e.g., MySQL) employing differential undo logging can retain their methodologies in our architecture but may lose some of our benefits since it should collect all the diffs required for building a version.

**Provisional version index in practice**. To activate online macro-compaction, we need a clear trigger and use the *space discrepancy* between a p-leaf index and an interval tree as an indicator. This design comes from realizing that our interval tree expands or contracts dynamically, whereas the index space would only grow so that a shrunken tree alludes to a waste of space in the p-leaf index if it remains swollen. The gap indicator can capture the right moments to invoke macro-compaction while systems are processing various long-term workloads. When integrating our p-leaf index to a database primary index, we utilize legacy record header fields as a placeholder for the entry locator to the p-leaf index. Although

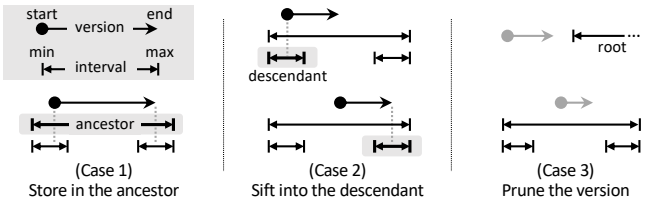



**Figure 10: Refined version sifting logic in practice.**

a record is delete-marked, modern MVCC DBMSs already enforce the retention of such meta-data fields for live transactions.

**Version garbage collection in practice**. Implementing version garbage collection is stand-alone work that would not require architectural changes in legacy systems. However, we find path compression helpful to refine the version sifting logic to make version sifting more efficient through one-time traversal from the root to leaf nodes. What we address here is the case where a version's visibility range may span over two descendant intervals. If the version overlaps with both intervals (Figure 10-(Case 1)), then it should stay in the current ancestor node, as expected. Otherwise, if the range overlaps with one of the intervals (Figure 10-(Case 2)), the refined logic is to sift the version into the overlapped descendant subtree recursively for finer-grained sifting that leads to more timely deletion. This refined sifting logic is meaningful when there are long-running analytic queries since staying in the upper-level node may *delay purging* that the legacy MVCC systems suffer.

## 7 EXPERIMENTAL EVALUATION

We validate our approach by materializing Diva into two full-fledged MVCC DBMSs: PostgreSQL-13.1 and MySQL-8.0. We borrowed the SIRO implementations from their codebase [3] and integrated the p-leaf index and the version cleaning framework with the referred systems. To further explore the impact of HTAP workloads on cloud-native MVCC databases, we use an Amazon Aurora MySQL [86, 87] instance without read replicas. For our interval tree, we use 100 *ms* as an epoch for evaluation.

**Computing platforms**. We use two types of servers for our evaluation. As a high-end platform, we use a 96-core machine with four Intel Xeon processors, 2 TiB of memory, and SSDs. As a low-end server, we use a low cost AWS Aurora MySQL t3.medium instance [8], featuring two virtual CPUs, 4 GiB of memory, and EBS only. All the databases operate robustly, regardless of servers and workloads, but the performance metrics differ widely. To limit CPU cores and memory, we use the `cgroup` feature of the Linux kernel.

**Workloads**. We use two types of workloads: `sysbench-1.0.20` [2] and `TPC-CH` [26]. For TPC-CH experiments, we adapted the tools [25] that Citus Data [29] developed for running CH-benCHmark with HammerDB [1]. In `sysbench` experiments, we add long-running join queries similar to TPC-CH to harness databases to explore how they behave. We emphasize three performance metrics: *version space overhead*, *latency* for OLAP queries, *throughput* for OLTP transactions. Our synthetic join queries are defined as follows:

JOIN Query $i$ ($Q_i$) ($1 \le i \le 4$):

```
SELECT SUM(sbtest-1.k + ... + sbtest-(i+1).k)
FROM   sbtest-1, ..., sbtest-(i+1)
WHERE  sbtest-1.id = sbtest-2.id AND ... AND
       sbtest-i.id = sbtest-(i+1).id
```

**Database configurations**. DBMS tuning impacts performance substantially and demands notable efforts in finding the right tuning formula. We need to tune DBMSs to suppress other noises when measuring various performance metrics of systems over a long period. We, therefore, follow conventional wisdom to preclude irrelevant contention or bottlenecks from arising in other components during our evaluation. To this end, we use large log files to suppress checkpoint activities and relax strict durability to permit more updates on tables by adopting "*asynchronous commit* [84]" for PostgreSQL and "*delayed log flushing* [65]" for MySQL. We disable query cache known to have scalability problems causing unwanted bottlenecks under HTAP. To lessen the adverse effects of background version cleaning in vanilla DBMSs, we disable `autovacuum` in PostgreSQL [37] and delay purging by setting a huge value for the maximum purge lag in MySQL [14]. The default isolation level for all databases is set to `REPEATABLE READ`. Other attempts to adjust more tuning knobs would be in vain since they could not hit other sweet spots, so we used the above formula throughout the evaluation, with varying buffer pool sizes.

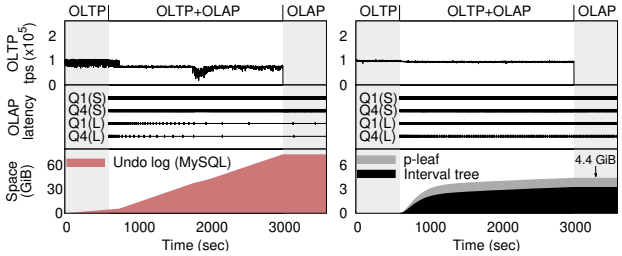
**Implementation caveat in PostgreSQL with Diva**. There is a well-known transaction ID wraparound problem in PostgreSQL with a remedy of periodic vacuuming [83]. When replacing the vacuum with Diva, we have not implemented the function of marking old tuples as frozen, carried while vacuuming in vanilla PostgreSQL. We notice this as a caveat since it bears an obvious regression issue if systems are up and running for a long time. One solution is to dispatch a similar task to our background process to mark old file segments as frozen. We leave this for future work.

### 7.1 Sysbench OLTP with Analytic Queries

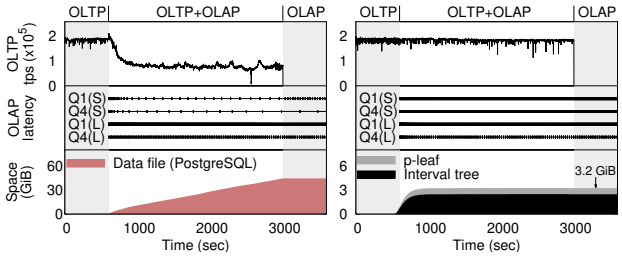
This experiment runs synthetic `join` queries, executing multi-table joins with `sysbench` OLTP workloads. We create 12 tables and adjust dataset size by varying the number of records for each table. We use 20 OLAP clients, half of which run queries within an open transaction to emulate long-running transactions (i.e., Q1(L), Q2(L), Q3(L), and Q4(L)), while the other half run each query within a single transaction (i.e., Q1(S), Q2(S), Q3(S), and Q4(S)). We only show latency results for Q1 (shortest) and Q4 (longest) for brevity. Unless stated otherwise, the experiment lasts for 1 hour consisting of three phases: OLTP only for 10 *min*, OLTP/OLAP for 40 *min*, and OLAP only for 10 *min*.

*7.1.1 Vanilla databases under HTAP workloads.* Since vanilla engines exhibit similar performance behavior under our workloads, we summarize the common phenomena upfront and avoid reiterating them hereafter. Initially, pure OLTP transactions show a decent update rate when running solely, and as analytic queries start running, data versions pile up since analytic queries hold their snapshot until the end of their execution and none of the vanilla engines perform fine-grained version cleaning. Then, expanding version space harms performance metrics: increasing space overhead, longer query latency, and degrading write throughput. Databases have diverse causes for throughput degradation.

*7.1.2 Evaluation on a high-end server.* In this experiment, we populate each table with 1 million records that result in a dataset of 2.7 GiB and set a 40 GiB buffer pool for vanilla databases. We set a

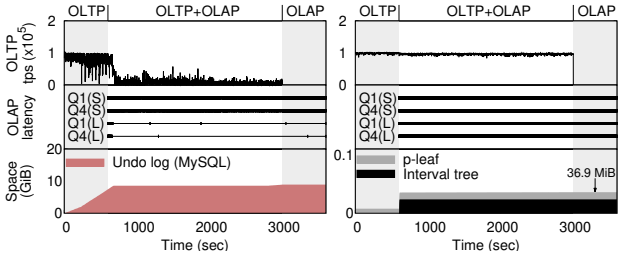



(a) Vanilla MySQL vs. MySQL with Diva

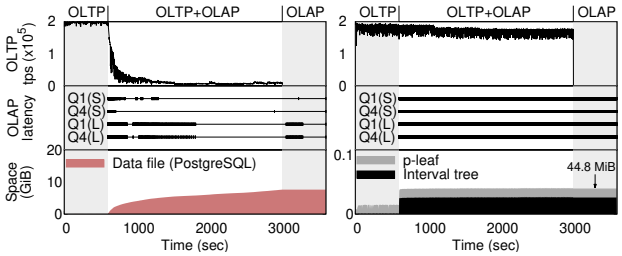



(b) Vanilla PostgreSQL vs. PostgreSQL with Diva

**Figure 11: Performance on a high-end server.**




(a) Vanilla MySQL vs. MySQL with Diva




(b) Vanilla PostgreSQL vs. PostgreSQL with Diva

**Figure 12: Performance with high CPU and low memory.**

20 GiB legacy buffer pool and 8 GiB buffer pages for our interval tree in modified systems. In vanilla databases, growing version space influences different places. In MySQL, long-lived transactions (i.e., Q1(L) and Q4(L) on the left side of Figure 11a) take time to complete `join` queries due to increased search time (refer to §7.3.1 for the latency breakdown of Q4(L)). Longer search time may block OLTP transactions on contending pages, degrading throughput. Meanwhile, in PostgreSQL, short query transactions (i.e., Q1(S) and Q4(S) on the left side of Figure 11b) run longer since searching for new versions traverses from the oldest versions in the so-called version chain. Because of this search characteristic, long-lived transactions (i.e., Q1(L) and Q4(L)) have short latency with old snapshots. **Indeed, the HTAP workloads pound more on query performance since even small extra I/O required for searching old versions impacts query latency badly.**

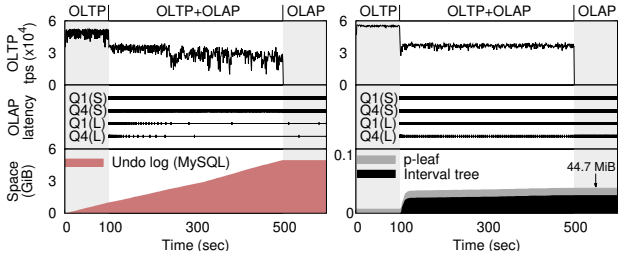
In contrast, the systems with Diva, as shown in the right side of Figure 11, sustain high OLTP throughput while processing analytic queries without showing a substantial latency increase. Interval tree-based version cleaning effectively reclaims stale data versions and keeps the least amount of live versions. Specifically, undo logs in vanilla MySQL occupy maximum 73.3 GiB of storage space while our modified engine consumes only 4.75 GiB (1.23 GiB for p-leaf and 3.52 GiB for interval tree), thus saving about 93.6% of storage space. **Essential to notice is the virtuous circle that since Diva assists the engines to find versions quickly, shorter query lifetimes help reduce version segments**. So, version search with bounded I/O is critical to MVCC systems under HTAP workloads.

*7.1.3 Evaluation with high CPU and low memory.* In this experiment, we limit available memory to 4 GiB and allocate 2 GiB for legacy buffer pools in vanilla systems. We also reduce the dataset size by using 10K records for each table, leading to a dataset of 120 MiB. The main reason for using a tiny dataset is because analytic `join` queries with 1M records take an overly long execution time to finish once the buffer is full. For modified databases, we set 1 GiB for a legacy buffer pool but intentionally allocate much smaller buffer space for our p-leaf index and interval tree (i.e., 4 MiB for the p-leaf buffer and 16 MiB for our interval tree) to explore how modified engines react to memory pressure in our buffer space.

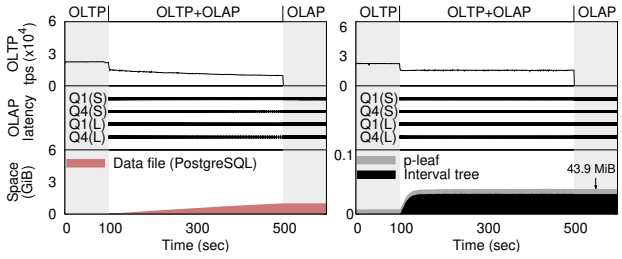
As shown on the left side of Figure 12, OLAP queries in vanilla engines run significantly longer due to the growing number of I/O operations once the buffer overflows with data versions, considering the tiny dataset. It causes OLTP throughput to plummet in all vanilla engines (e.g., from 200kTps to 5kTps in vanilla PostgreSQL) since severe contention in buffer pools and long search time aggravate the situation: the average OLTP latency is 1.628 *ms* with the peak longer than 1 sec. The situation in PostgreSQL is worse than MySQL because PostgreSQL accumulated more versions that hurt `join` queries: all single-transaction queries, i.e., Q1(S)-Q4(S), have enormous latency after the buffer is full. **So, the workloads plunge the vanilla engines into an impasse**.

In contrast, memory pressure has a limited impact on the modified engines since version lookup incurs two I/O operations, one for the p-leaf page and the other for the version segment. As shown on the right side of Figure 12, the latency for `join` queries in modified systems is bounded, and OLTP throughput and its latency (avg: 0.056 *ms* and max: 76.9 *ms*) are less affected by analytic queries. Although it is preliminary, the results show that **Diva aids the legacy engines to be less affected by memory pressure**.

*7.1.4 Evaluation with low CPU and memory.* This experiment uses 4 CPU cores with 4 GiB memory and the same configuration for vanilla databases with a small dataset. We run the experiments for a shorter period (i.e., 10 *min*). For engines with Diva, we allocate 1 GiB for legacy buffer pools and 1 GiB for our components. As shown in the left side of Figure 13, vanilla databases exhibit similar performance metrics to those in the previous section. Initially, a lower OLTP throughput produces a smaller volume of data versions that slowly degrade query latency. The low-end server impacts modified databases that degrade their throughput due to the low




(a) Vanilla MySQL vs. MySQL with DIVA




(b) Vanilla PostgreSQL vs. PostgreSQL with DIVA

**Figure 13: Performance on a low-end server.**

core count. However, the query latency and version space overhead, as shown in the right side of Figure 13, are not significantly affected by limited computing cores.

*7.1.5 Evaluation on cloud databases.* This experiment explores the performance of cloud databases [16, 31, 86] that rely on disaggregated architecture exposed to unpredictable delay jitters between front-end and back-end servers. We use the Aurora MySQL t3.medium instance and set up a t3.medium EC2 instance for our modified MySQL. The configuration is similar to §7.1.4, except that the modified MySQL stores database files in network-attached storage (i.e., Amazon EBS), with longer network delays. Figure 14 shows throughput and query latency since Aurora MySQL denies our access to its raw database log files. Although Aurora MySQL delivers low throughput (i.e., 5kTps) due to network latency between our client machine and the database, slowly expanding version space, once triggering buffer overflow, suddenly impacts query latency. This behavior is quite in contrast to the previous results in Figure 13 and implies informal SLA violations. Once it occurs, unreasonable cloud billing reports would betide and perplex customers who have run such workloads. Perhaps, the naive use of OLAP-optimized platforms [6, 21, 30, 38] cannot effortlessly resolve the problem. MySQL with DIVA on the EC2 instance shows similar results to what we obtained in Figure 13.

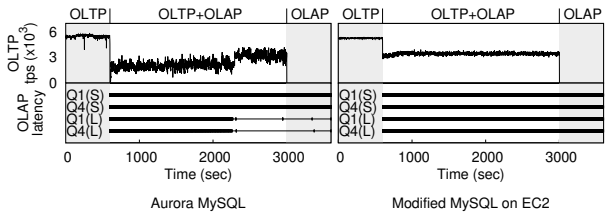
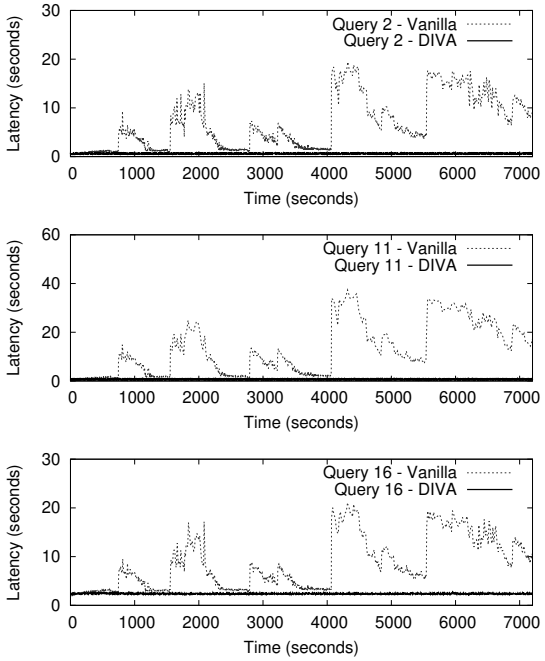
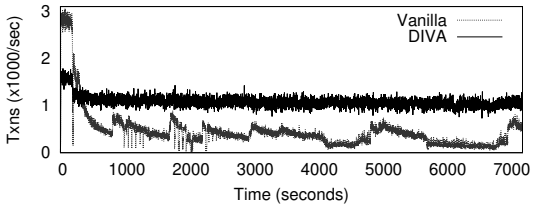



**Figure 14: Performance on cloud computing platforms.**




(a) Query latency for Q2, Q11, and Q16.



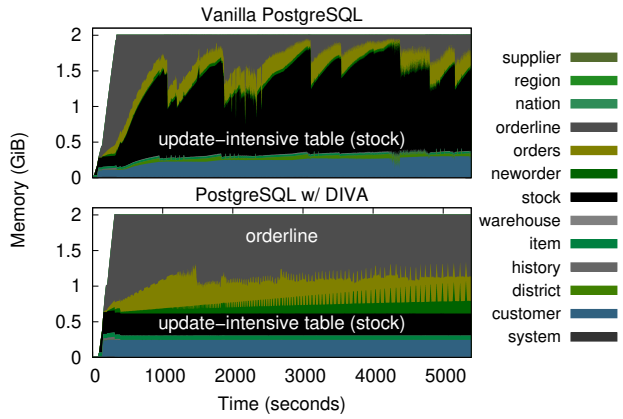

(b) TPC-C throughput (`NewOrder` transaction).

**Figure 15: Performance under TPC-CH benchmark.**

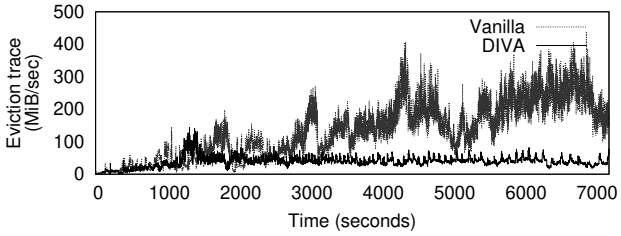
## 7.2 TPC-CH Workloads

In this section, we run TPC-CH workloads with PostgreSQL and use the benchmark developed by Citus Data [25]. The test configuration is similar to the low-memory configuration in §7.1.3, i.e., 4 GiB memory and 2 GiB for the legacy buffer pool. To see the behavior itself, we use a small dataset using two warehouses (WH). In this experiment, we do not intentionally create long-running transactions since some TPC-CH queries run an excessively long time. Note that a majority of updates in TPC-CH workloads are data inserts into tables that grow as time elapses, and the execution time of most queries increases accordingly. Queries affected more by an update-intensive table (i.e., the `stock` table) are Q2, Q11, and Q16, all of which run `join` operations on the `stock` table. We only show query latency for those queries in Figure 15a and throughput for `NewOrder` transactions in Figure 15b.

*7.2.1 Overall performance.* PostgreSQL with DIVA effectively suppresses their latency for the three queries due to the reduced I/O cost by our p-leaf index and interval tree. It also sustains throughput better than the vanilla engine. However, throughput and query latency in the vanilla engine show similar fluctuating patterns with different peak values. In-depth analysis through profiling reveals that `NewOrder` transactions acting on the stock table continue to




(a) Breakdown of buffer pool usage




(b) Eviction trace (i.e., evicted pages per second)

**Figure 16: In-depth analysis of buffer pool metrics.**

produce numerous data versions that cause three side-effects: 1) it leads to undo space expansion, 2) it results in memory pressure in a buffer pool, and 3) finally, it impacts other queries badly. Furthermore, the queries trying to load the same data page of the `stock` table from storage incurs undesirable page latch contention, which leads to such latency spike. `StockLevel` transactions in TPC-C also aggravate the contention since they also act on the `stock` table. Although more investigation is needed, our analysis may explain the fluctuating behavior in TPC-C throughput.

*7.2.2 In-depth analysis of database buffer cache.* To provide further insight into performance problems, we peek inside the database buffer cache, obtain the usage breakdown, and collect buffer eviction trace while running the same TPC-CH benchmark. As shown in Figure 16a, there is a remarkable difference between vanilla and modified systems. First, the buffer usage breakdown in the vanilla PostgreSQL shows that the `stock` table occupies considerable buffer space (e.g., the peak is 75%) as queries spend time searching required versions in the table for join operations. As analytic queries fetch more pages during version lookup, it evicts the other large table (i.e., the `orderline` table) from the buffer cache, which would damage `NewOrder` transactions. The tension between these two tables competing for the same buffer space is the main culprit for performance fluctuation. This behavior explains what we observed in Figure 15.

In contrast, the breakdown of buffer usage in PostgreSQL with Diva shows stable results; especially, the `stock` table occupies much less cache space (i.e., 15%), thus yielding more buffer pages to other tables (e.g., `orderline`). The eviction trace in Figure 16b backs up this argument in that the vanilla system evicted more pages than Diva. The eviction rate is growing in the vanilla (the peak rate is 439 MiB/s) as time elapses, whereas Diva stabilizes around 41

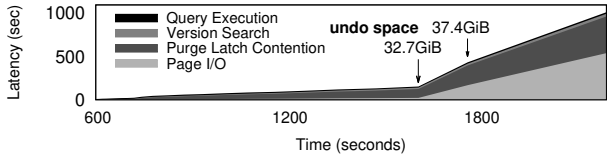



**Figure 17: The latency breakdown for Q4(L) in MySQL.**

MiB/s with the peak of 144 MiB/s. The dual impact of low memory footprint and eviction rate benefits PostgreSQL with Diva in all crucial performance metrics. Overall, **this investigation confirms that Diva aids the engine to reduce resource contention and achieve better performance metrics under HTAP workloads.**

## 7.3 Miscellaneous

*7.3.1 Query latency breakdown for Figure 11.* As shown on the left side of Figure 11, analytic queries are more susceptible to I/O caused by memory pressure. To understand system behaviors clearly, we peek inside the system and obtain latency breakdown for Figure 11. Figure 17 shows the latency breakdown for Q4(L) in vanilla MySQL between 600s and 2400s. As `sysbench` produces data versions, the latch contention overhead dominates if everything is in buffers. As undo space approaches the buffer capacity (i.e., 40 GiB), I/O overhead grows substantially. After the critical point, latch contention and page I/O are responsible for most of the ever-growing latency (cf. bounded I/O operations for each version search in Diva). **The breakdown confirms that the legacy MVCC engines would exhibit severe read amplification in version search once data versions overflow its buffer**.

*7.3.2 P-leaf vs. Memory-resident Index.* Despite a large footprint, an in-memory index covering the entire dataset is the core of in-memory MVCC systems for fast data retrieval. In contrast, we use a disk-resident index. To validate the adequacy of our design for disk-based DBMSs, we compare the memory footprint of Diva with that of vDriver, a prior art sharing the similar goal of managing data versions for disk-based systems and adopting a memory-resident index for data versions. We use PostgreSQL-based implementations of both approaches with `shared_buffer` set to 2 GiB. To see the change of their memory footprint, we run sysbench update-only workloads with two datasets (i.e., 5M and 10M records) while adding 20 long-lived queries with 3 *min* delay in between, resulting in a stepwise increase of memory footprint in vDriver (see Figure 18).

Figure 18 shows that vDriver internally requires more memory since its in-memory index grows proportional to the dataset (i.e., table size and the number of data versions) due to memory-resident data structures. So, it implies that the accretion of memory footprint would lead to memory pressure under larger-than-memory

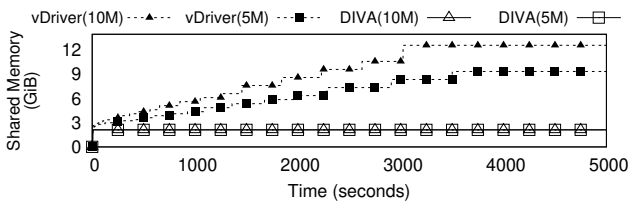



**Figure 18: Comparison of memory footprint with vDriver.**

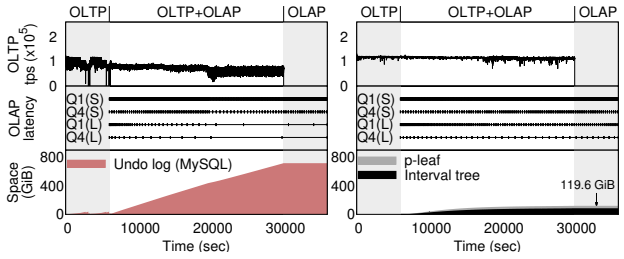



**Figure 19: Performance of MySQL with a large dataset.**

analytic workloads once spilled. In contrast, Diva is free from such restrictions with bounded I/O for version searching and can operate on low-end servers with less memory footprint, regardless of the dataset. The results adequately justify our design rationale. Since other metrics look similar, we omitted them due to space limitations.

*7.3.3 Evaluation with a large dataset.* In this experiment, we populate each table with 30 million records that result in a raw dataset of 80 GiB for vanilla MySQL and 160 GiB for MySQL with Diva due to the SIRO-versioning [41]. We set a 512 GiB buffer pool for the vanilla system. We intentionally create a huge log file to suppress checkpoint activities in vanilla MySQL and use a vast value for the maximum purge lag to delay purging undo pages [14]. Otherwise, vanilla MySQL would suffer performance collapse since background purging and checkpointing would severely interfere with foreground transactions. In MySQL with Diva, we set a 350 GiB buffer pool with 150 GiB dedicated for our p-leaf index and interval trees. We used the same synthetic hybrid workloads used earlier and ran the evaluation for 10 hours with the same proportion. The overall behavior is almost identical to Figure 11a except for the difference in time scale. As shown in Figure 19, DBMSs equipped with ample memory show similar performance metrics under larger-than-memory workloads.

*7.3.4 Evaluation under skewed workloads.* To explore the performance effect of skewed workloads, we measure the performance of PostgreSQL under skewed updates with the same configuration used in Figure 11 (high-end server setting). We ran the test for 20 minutes, with the first half running OLTP-only and the rest running OLTP/OLAP. Figure 20 shows that OLTP throughput collapses right after OLAP loads start in the vanilla engine with ample memory, unlike Figure 11. The reason is that OLTP transactions take enormous time in traversing long version chains from the oldest to the newest in vanilla PostgreSQL. The space overhead is mild due to skewed updates on a few hotkeys. We omit OLAP latency since the effect of the hotkeys on analytic queries is negligible. In contrast, PostgreSQL with Diva is less affected by skewed loads.

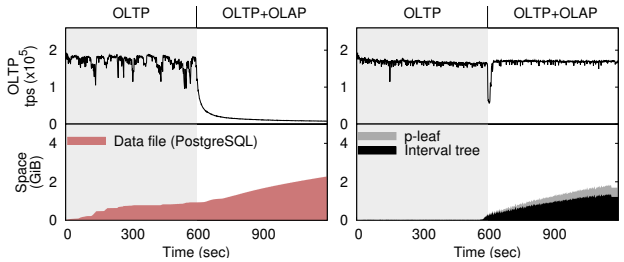



**Figure 20: Performance under skewed updates ($\theta$=1.0).**

## 8 DISCUSSION

The HTAP workloads set three clear objectives: high OLTP throughput, low OLAP latency, and compact storage space. Earning all three requires careful design guidelines to strike a balance between individual objectives. We draw some lessons worth further research.

- ***Database buffer: unified or partitioned?*** Buffer cache influences analytic queries considerably, and how to manage the cache for separated components needs further investigation to gain practicality. In particular, deciding between partitioned or unified cache for our designs is worth the effort.
- ***Designs for disaggregated cloud architecture.*** We showed that the disaggregated storage architecture for cloud-native DBMSs exposes several weaknesses under HTAP loads. The steep increase in query latency comes from network storage architecture and is worse than on-premise DBMSs. As we have seen recent efforts [34, 93] that addressed data layout issues for cloud analytics, exploring the cloud storage architecture is worth the effort to overcome inherent obstacles (e.g., network delay).
- ***Data version locality.*** Version locality in version storage has received less attention than primary indexes, despite its growing importance in HTAP environments. This paper affirms the strong influence of version locality on analytic queries, like how much column-store systems [5, 18, 20, 44, 53, 57, 78, 81, 94] impact data analytics. Considering the practical impact in this area, we believe that there is significant room for improvement.

## 9 CONCLUSION

This paper revisits well-proven designs in disk-based MVCC DBMSs and decoupled the entangled design concerns to address the space-time tradeoff we encounter when facing HTAP workloads. The undesirable performance metrics have rendered legacy disk-based MVCC systems ineffective for HTAP loads. To refute some negative stereotypes and address this critical issue, we propose Diva. The separation of concerns, the underlying principle of Diva, would empower database designers to fend off the undue burden on managing version search and cleaning under HTAP workloads. For the independent management of two separate concerns, we proposed two concrete designs—provisional indexing and interval-based garbage collection—that embody the clean separation. We validate our approach by applying Diva to two full-fledged MVCC DBMSs and conducting experimental evaluation on various computing platforms, including cloud environments. Since our work focused on devising correct, general-purpose designs, it remains open for further research to develop bespoke systems that may outshine ours in particular domains. We hope that database designers will find our design principles helpful in advancing new database architecture or revisiting already-closed, worth-looking issues.

## ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for helping us improve this paper. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2022R1A2C2008427).

## REFERENCES


[1] 2020. HammerDB Version 4.0. Available at https://github.com/TPC-Council/HammerDB/releases/tag/v4.0.

[2] 2020. sysbench-1.0.20. Available at https://github.com/akopytov/sysbench.

[3] 2020. vDriver github codebase. https://github.com/hyu-scslab/vDriver.

[4] 2020. vDriver: version chain codebase. https://github.com/hyu-scslab/vDriver_PostgreSQL/blob/72e68f1ed864b8e90dd8f2c660da4310fe3fa47c/src/backend/storage/vcluster/vchain.c.

[5] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[6] Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, Lazar Davidovic, Maja Damjanic, Slobodan Djunic, Nemanja Djurkic, Charles Feddersen, Cesar Galindo-Legaria, Alan Halverson, Milana Kovacevic, Nikola Kicovic, Goran Lukic, Djordje Maksimovic, Ana Manic, Nikola Markovic, Bosko Mihic, Ugljesa Milic, Marko Milojevic, Tapas Nayak, Milan Potocnik, Milos Radic, Bozidar Radivojevic, Srikumar Rangarajan, Milan Ruzic, Milan Simic, Marko Sosic, Igor Stanko, Maja Stikic, Sasa Stanojkov, Vukasin Stefanovic, Milos Sukovic, Aleksandar Tomic, Dragan Tomic, Steve Toscano, Djordje Trifunovic, Veljko Vasic, Tomer Verona, Aleksandar Vujic, Nikola Vujic, Marko Vukovic, and Marko Zivanovic. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3204–3216. https://doi.org/10.14778/3415478.3415545

[7] Amazon Web Services Inc. 2021. Amazon Aurora MySQL reference: Aurora MySQL isolation levels: READ COMMITTED isolation level for reader instances. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/AuroraMySQL.Reference.html#AuroraMySQL.Reference.IsolationLevels.

[8] Amazon Web Services Inc. 2021. Amazon Aurora Pricing: Database Instances. https://aws.amazon.com/rds/aurora/pricing/#Database_Instances.

[9] Amazon Web Services Inc. 2021. Amazon EC2 T2 Instances. https://aws.amazon.com/ec2/instance-types/t2/.

[10] Amazon Web Services, Inc. 2022. What Is AWS Glue? https://docs.aws.amazon.com/glue/latest/dg/what-is-glue.html.

[11] Oracle Corporation and/or its affiliates. 2019. MySQL 8.0 Reference Manual: 15.6.6 Undo Logs. https://dev.mysql.com/doc/refman/8.0/en/innodb-undo-logs.html.

[12] Oracle Corporation and/or its affiliates. 2019. Oracle 19 Database Administrator's Guide: 16 Managing Undo. https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-undo.html#GUID-2C865CF9-A8B5-4BF1-A451-E8C08D3611F0.

[13] Oracle Corporation and/or its affiliates. 2021. 15.3 InnoDB Multi-Versioning. https://dev.mysql.com/doc/refman/8.0/en/innodb-multi-versioning.html.

[14] Oracle Corporation and/or its affiliates. 2021. 15.8.9 Purge Configuration. https://dev.mysql.com/doc/refman/8.0/en/innodb-purge-configuration.html.

[15] Oracle Corporation and/or its affiliates. 2021. Oracle Release 21. Database Concepts: Part IV-11 Data Concurrency and Consistency. https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/data-concurrency-and-consistency.html#GUID-8DC0D1D1-C2B1-4237-9B77-27889B6467C1.

[16] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1743–1756. https://doi.org/10.1145/3299869.3314047

[17] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant Time Recovery in Azure SQL Database. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2143–2154. https://doi.org/10.14778/3352063.3352131

[18] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. *To Partition, or Not to Partition, That is the Join Question in a Real System.* Association for Computing Machinery, New York, NY, USA, 168–180. https://doi.org/10.1145/3448016.3452831

[19] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[20] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*. www.cidrdb.org, 225–237. http://dblp.uni-trier.de/db/conf/cidr/cidr2005.html#BonczZN05

[21] Nemanja Borić, Hinnerk Gildhoff, Menelaos Karavelas, Ippokratis Pandis, and Ioanna Tsalouchidou. 2020. Unified Spatial Analytics from Heterogeneous Sources with Amazon Redshift *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2781–2784. https://doi.org/10.1145/3318464.3384704

[22] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *Proc. VLDB Endow.* 13, 2 (Oct. 2019), 128–141. https://doi.org/10.14778/3364324.3364328

[23] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 668. https://doi.org/10.1145/872757.872857

[24] Sirish Chandrasekaran and Michael J. Franklin. 2002. Streaming Queries over Streaming Data. In *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong, China) *(VLDB '02)*. VLDB Endowment, 203–214.

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

[26] 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. https://doi.org/10.1145/1988842.1988850

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[28] Microsoft Corporation. 2021. tempdb database. https://docs.microsoft.com/en-us/sql/relational-databases/databases/tempdb-database?view=sql-server-ver15.

[29] 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

[30] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. 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, 215–226. https://doi.org/10.1145/2882903.2903741

[31] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to Be Fast, Available, and Frugal in the Cloud. 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, 1463–1478. https://doi.org/10.1145/3318464.3386129

[32] 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)*. ACM, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[33] Edsger W. Dijkstra. 1982. *On the Role of Scientific Thought.* Springer New York, New York, NY, 60–66. https://doi.org/10.1007/978-1-4612-5695-3_12

[34] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proceedings of the 2021 International Conference on Management of Data.* Association for Computing Machinery, New York, NY, USA, 418–431. https://doi.org/10.1145/3448016.3457270

[35] Facebook. 2021. RocksDB Universal Compaction. Available at https://github.com/facebook/rocksdb/wiki/Universal-Compaction.

[36] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[37] The PostgreSQL Global Development Group. 2021. PostgreSQL: Documentation: 25.1. Routine Vacuuming. https://www.postgresql.org/docs/current/routine-vacuuming.html.

[38] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795

[39] 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. https://doi.org/10.14778/3415478.3415535

[40] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE

Computer Society, USA, 195–206. https://doi.org/10.1109/ICDE.2011.5767867
[41] 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. https://doi.org/10.1145/3318464.3389714
[42] Jongbin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2021. Rethink the Scan in MVCC Databases. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 14 pages. https://doi.org/10.1145/3318464.3389714
[43] 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. https://doi.org/10.1145/2882903.2882905
[44] 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. https://doi.org/10.14778/2367502.2367518
[45] 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. https://doi.org/10.14778/2367502.2367518
[46] 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. https://doi.org/10.14778/2824032.2824071
[47] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 1307–1318. https://doi.org/10.1145/2882903.2903734
[48] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2020. Kvell+: Snapshot Isolation without Snapshots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 425–441. https://www.usenix.org/conference/osdi20/presentation/lepers
[49] 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. https://doi.org/10.14778/3436905.3436913
[50] Hyo-Sang Lim, Jae-Gil Lee, Min-Jae Lee, Kyu-Young Whang, and Il-Yeol Song. 2006. Continuous Query Processing in Data Streams Using Duality of Data and Queries. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/1142473.1142509
[51] Percona LLC. 2021. MongoRocks. https://www.percona.com/doc/percona-server-for-mongodb/3.4/mongorocks.html.
[52] 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
[53] Roger MacNicol and Blaine French. 2004. Sybase IQ Multiplex - Designed for Analytics. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 1227–1230.
[54] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3217–3230. https://doi.org/10.14778/3415478.3415546
[55] Paul E. McKenney, Joel Fernandes, Silas Boyd-Wickizer, and Jonathan Walpole. 2020. RCU Usage In the Linux Kernel: Eighteen Years Later. *SIGOPS Oper. Syst. Rev.* 54, 1 (Aug. 2020), 47–63. https://doi.org/10.1145/3421473.3421481
[56] Paul E. McKenney and Jonathan Walpole. 2008. Introducing Technology into the Linux Kernel: A Case Study. *SIGOPS Oper. Syst. Rev.* 42, 5 (July 2008), 4–17. https://doi.org/10.1145/1400097.1400099
[57] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886
[58] Microsoft Corporation. 2021. SQL Server Doc: CREATE TABLE (See "Temporary Tables"). https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql?view=sql-server-ver15#database-scoped-global-temporary-tables-azure-sql-database.
[59] Microsoft Inc. 2019. SQL Server In-Memory OLTP Internals for SQL Server 2016: Memory Requirement (page 74). https://download.microsoft.com/download/8/3/6/8360731A-A27C-4684-BC88-FC7B5849A133/SQL_Server_2016_In_Memory_OLTP_White_Paper.pdf.
[60] 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 (March 1992), 94–162. https://doi.org/10.1145/128765.128770
[61] MongoDB Inc. 2018. JIRA/Core Server/SERVER-38170: mongod stopped due to out of memory. https://jira.mongodb.org/browse/SERVER-38170.
[62] MongoDB Inc. 2021. MongoDB. https://www.mongodb.com/.
[63] MongoDB Inc. 2021. WiredTiger Storage Engine. https://docs.mongodb.org/manual/core/wiredtiger/.
[64] MongoDB User Group. 2016. MongoDB Wiredtiger 3.2.0 crashes with out of memory. https://groups.google.com/g/mongodb-user/c/0lJeT55qAo8?pli=1.
[65] MySQL Community. 2021. MySQL 8.0 Reference Manual: The InnoDB Storage Engine - InnoDB Startup Options and System Variables (i.e., `innodb_flush_log_at_trx_commit`). https://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html#sysvar_innodb_flush_log_at_trx_commit.
[66] Ned Otter Blog. 2018. In-Memory OLTP Resources, Part 4: OOM, the most feared acronym in all of In-Memory OLTP. http://nedotter.com/archive/2018/01/in-memory-oltp-resources-part-4-oom-the-most-feared-acronym-in-all-of-in-memory-oltp/.
[67] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048
[68] Oracle. 2021. Heatwave User Guide. Available at https://downloads.mysql.com/docs/heatwave-en.pdf.
[69] Oracle and/or its affiliates. 2021. MySQL 8.0 Reference Manual: 13.1.20.2 CREATE TEMPORARY TABLE Statement. https://dev.mysql.com/doc/refman/8.0/en/create-temporary-table.html.
[70] Oracle and/or its affiliates. 2021. Oracle Doc: Creating a Temporary Table. https://docs.oracle.com/cd/E18283_01/server.112/e17120/tables003.htm#i1006400.
[71] Redgate Hub. 2014. Exploring In-memory OLTP Engine (Hekaton) in SQL Server 2014 CTP1. https://www.red-gate.com/simple-talk/sql/database-administration/exploring-in-memory-oltp-engine-hekaton-in-sql-server-2014-ctp1/.
[72] PostgreSQL repository. 2021. README on The Transaction System. https://github.com/postgres/postgres/blob/master/src/backend/access/transam/README.
[73] Dennis M. Ritchie and Ken Thompson. 1974. The UNIX Time-Sharing System. *Commun. ACM* 17, 7 (July 1974), 365–375. https://doi.org/10.1145/361011.361061
[74] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The Art of Balance: A RateupDB™ Experience of Building a CPU/GPU Hybrid Database Product. *Proc. VLDB Endow.* 14, 12 (Aug. 2021), 2999–3013. https://doi.org/10.14778/3476311.3476378
[75] SAP Community. 2014. 6 Tips to avoid HANA Out of Memory (OOM) Errors. https://blogs.sap.com/2014/01/05/6-tips-to-avoid-hana-out-of-memory-oom-errors/.
[76] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F Ilyas. 2021. Real-Time LSM-Trees for HTAP Workloads. *arXiv preprint arXiv:2101.06801* (2021).
[77] William Schultz, Tess Avitabile, and Alyson Cabral. 2019. Tunable Consistency in MongoDB. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2071–2081. https://doi.org/10.14778/3352063.3352125
[78] Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Ariel Cary, Vivek Bharathan, and Chuck Bear. 2013. Materialization strategies in the Vertica analytic database: Lessons learned. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 1196–1207. https://doi.org/10.1109/ICDE.2013.6544909
[79] 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. https://doi.org/10.1145/2213836.2213946
[80] Facebook Open Source. 2021. MyRocks: A RocksDB storage engine with MySQL. http://myrocks.io/.
[81] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 553–564.
[82] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., USA) *(SIGMOD '86)*. Association for Computing Machinery, New York, NY, USA, 340–355. https://doi.org/10.1145/16894.16888
[83] The PostgreSQL Global Development Group. 2021. PostgreSQL: Documentation for PostgreSQL 12: 25.1.5. Preventing Transaction ID Wraparound Failures. https://www.postgresql.org/docs/current/routine-vacuuming.html#VACUUM-FOR-WRAPAROUND.
[84] The PostgreSQL Global Development Group. 2021. PostgreSQL: Documentation for PostgreSQL 12: Section 29.3. Asynchronous Commit. https://www.postgresql.org/docs/12/wal-async-commit.html.

[85] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[86] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. 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, 1041–1052. https://doi.org/10.1145/3035918.3056101

[87] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, and Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 789–796. https://doi.org/10.1145/3183713.3196937

[88] Vertica Inc. 2021. Version 11.0.x Documentation: Vertica Concepts: Common Vertica Concepts: Transactions. https://www.vertica.com/docs/11.0.x/HTML/Content/Authoring/ConceptsGuide/Other/Transactions.htm?tocpath=Vertica%20Concepts%7CCommon%20Vertica%20Concepts%7CTransactions%7C_____0.

[89] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[90] WiredTiger. 2021. WiredTiger Version 10.0.0: Eviction (See "Update restore eviction"). https://source.wiredtiger.com/develop/eviction.html.

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

[92] 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. https://doi.org/10.14778/3415478.3415553

[93] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-Tree: Learning Data Layouts for Big Data Analytics. 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, 193–208. https://doi.org/10.1145/3318464.3389770

[94] Marcin Zukowski and Peter Boncz. 2012. From X100 to Vectorwise: Opportunities, Challenges and Things Most Researchers Do Not Think About. 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, 861–862. https://doi.org/10.1145/2213836.2213967