Long-lived Transactions Made Less Harmful

Anonymous Author(s)
Submission Id: #376

ABSTRACT

Many systems use snapshot isolation, or something similar, as defaults, and multi-version concurrency control (MVCC) remains essential to offering such point-in-time consistency. Data anomalies still exist, however, that stymie serializable execution, but lower isolation with MVCC has made inroads into the database community. One major issue in MVCC is the timely removal of unnecessary versions of data items, especially in the presence of long-lived transactions (LLTs). We have observed that the latest versions of MySQL and PostgreSQL are still vulnerable to LLTs. Our analysis of existing proposals suggests that new solutions to this matter must provide rigorous rules for completely identifying unnecessary versions and elaborate designs for version cleaning lest it should be suspended by old versions required for LLTs. In this paper, we formalize such rules into our version pruning theorem and version classification, of which all form theoretical foundations for our new version management system, vDriver, that bases its record versioning on a new principle: *one version in-row and put trailer off-row.* We implemented a prototype of vDriver and integrated it with MySQL-8.0 and PostgreSQL-12.0. The experimental evaluation demonstrated that the engines with vDriver continue to perform the reclamation of dead versions in the face of LLTs, while retaining transaction throughput with reduced space consumption.

CCS CONCEPTS

• **Information systems → DBMS engine architectures**.

KEYWORDS

long-lived transaction, record versioning, version store

ACM Reference Format:
Anonymous Author(s). 2020. Long-lived Transactions Made Less Harmful. In *SIGMOD'20: 2020 International Conference on Management of Data (SIGMOD'20), Portland, OR.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The concept of record versioning dates back to the late 1970s right after major relational database systems (i.e., Ingres [23] and System R [3]) were rolled out, and concurrency control relying on versioned record items was designed to provide fast reads to users while concurrent writes on conflicting data items are being executed. The multi-version concurrency control, despite allowing data anomalies, has become the most popular method for arbitrating concurrent access to databases and by far the de-facto scheme to use in designing high-performance transaction engines. Contemporary database systems, therefore, set lower isolation levels, which use MVCC, as defaults to enhance performance on multicores.

Essential for MVCC to function normally is the strict condition requiring that the cleaning of old, unnecessary versions be exercised in a timely manner. Delaying version cleaning indeed causes version space to be bloated (rapidly in in-memory engines), and then other critical components of a database system are ill-affected by this overdue operation. In the worst case, the entire system would halt because database systems run out of storage space, and moreover customers still have to pay for the uncleaned version space in cloud databases, although systems suffer throughput fluctuations that are the clear signs of sub-quality performance metrics that customers had not envisaged when signing contracts.

1.1 Record Versioning in the Wild

When designing version storage, one should keep two types of information somewhere until they are not needed: (i) record version itself and (ii) locator pointing to a previous version. In this respect, the database community has made the decades-in-the-making efforts to reach two widely accepted designs for version store architecture: *in-row versioning* and *off-row versioning*, each one having its own advantages and disadvantages. In-row versioning stores old versions of a record in the same data page where the record item exists (though of course overflow pages occur more often if conditions are met). For version lookup, systems using in-row versioning may use a separate search structure outside data pages, typically leaves of an index structure. This design has a clear benefit of finding a version usually faster than off-row versioning if everything is under control, but the disadvantage is the hefty cost of index modifications (i.e., data page split) that must be paid inexorably when dead versions pile up quickly but remain unresolved.

In contrast, off-row versioning stores old versions in separate version space, and each version item contains locator information together with a record item. This design may take a long time to find a requested version since it may fetch one data block per each version, thus incurring multiple I/O activities while searching a proper version. However, the clear merit recompenses off-row versioning for the long lookup time by guaranteeing that index modifications would never occur even if there is an upsurge of new versions, since

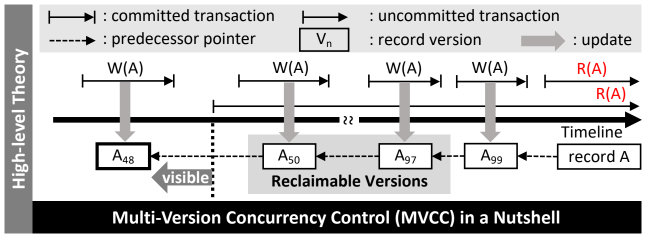


Figure 1: A long-lived transaction in MVCC.

versions and locator information are all stored in separate space so that main index structures remain unaffected.

One may consider a simple hybrid between two designs to cope with their shortcomings, but we are unaware of any prior attempts made or proven to be practical yet. In fact, we all discern the risk of naive trials that may bring all drawbacks, without settling the principal problem of version cleaning in presence of long-lived transactions (LLTs), and it is the main technical matter we are concerned with.

1.2 The Main Challenge

Important to notice after analyzing the root causes of the performance problems is the fact that all the problems can be resolved through the timely removal of unnecessary versions lacking in the two versioning designs. Version space can grow for two main reasons; (i) when new versions are generated more than a version cleaner is able to vacuum and (ii) when long-lived transactions obstruct the purging of reclaimable versions since some live versions required for LLTs must be kept in version space. The former can be overcome by adjusting the number of version cleaners, although it interferes with foreground transactions. However, unraveling the second matter is challenging since it is demanding to find an efficient mechanism to identify and reclaim dead versions without violating an essential representation invariant stating that *non-reclaimable record versions must be reachable at any moment as long as they are needed by live transactions.*

Figure 1 depicts how MVCC works under snapshot isolation while transactions having different lifetimes perform reads and writes. Writes on a record item create versions that are chained for reads to search a version committed before the concerned transactions. In this version chain, an old version A_{48} should remain visible to a long-lived transaction, while other versions — A_{50} and A_{97} — are no longer in use. Such uncleaned dead versions responsible for increased space usage are also suspected of being detrimental to transaction throughput. Therefore, the main goal of our work is to provide an efficient mechanism for version cleaning based on solid theoretical foundations, that addresses the pressing concern of the systems in presence of LLTs, while retaining transaction throughput with reduced space consumption.

1.3 Our Contributions

Since Jim Gray raised major concerns regarding LLTs [7], researchers and commercial database vendors have devised methods to ameliorate undesirable side-effects. Not surprisingly, our evaluation indicates that PostgreSQL-12.0 and MySQL-8.0 whose internal record versioning uses the most up-to-date techniques of in-row and off-row designs evolved in decades are still susceptible to LLTs. Our analysis, however, suggests that the strengths of two designs would still hold as transaction throughput increases, due in large part to the fact that short speedy transactions can benefit from fast in-row searching while the off-row scheme is favorable to keeping index structures intact regardless of heavy updates.

The present work is based on the premise that there are noteworthy gains by unifying two popular-but-seemingly dissimilar ideas. At the core of our approach is the following design strategy: record versions are managed by a new principle, called ***One Version In-Row And Put Trailer Off-Row*** (OVIRAPTOR), that keeps the first old version together with a record and moves older versions to off-row space. The OVIRAPTOR strategy ensures fast reads to most of the short transactions through accessing the first old version of a record in a data page (i.e., in-row) without the risk of frequent index modifications. As theoretical foundations for version cleaning, we first prove a *complete version pruning theorem* that gives necessary and sufficient conditions for identifying dead versions at any given time; then we lay down the *ground classification rules* under which versions with different lifetimes are classed separately so that cleaning dead versions would seldom be suspended by live versions required for LLTs. Essential for OVIRAPTOR to be feasible are concrete designs that provide fast access to versions through version buffering with lightweight in-memory index structures being augmented. Realizing our theoretical frameworks is enabled by a set of concurrent programming techniques.

By putting all these ideas together, we propose vDRIVER, an extension to existing engines to manage record versions. vDRIVER is composed of vSORTER and vCUTTER; vSORTER manages record versions by the OVIRAPTOR principle and maintains a version lookup structure, called *location lookaside buffer* (LLB), to provide a fast lookup to reads; and vCUTTER eagerly performs version cleaning by the version pruning theorem. We implemented a prototype of vDRIVER and integrated it with PostgreSQL-12.0 and MySQL-8.0. Experimental evaluation demonstrates that the engines with vDRIVER continue to perform version cleaning with LLTs, while retaining throughput and having low space overhead.

2 BACKGROUND AND MOTIVATION

Database systems should maintain data integrity by ensuring ACID. Among them is the isolation property intended

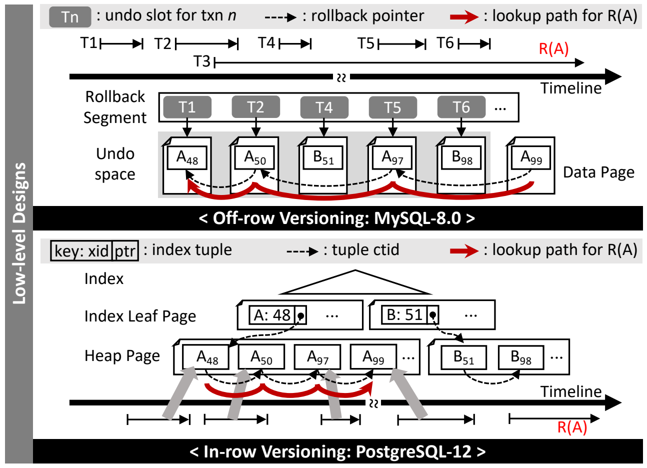


Figure 2: Effects of LLTs in two versioning designs.

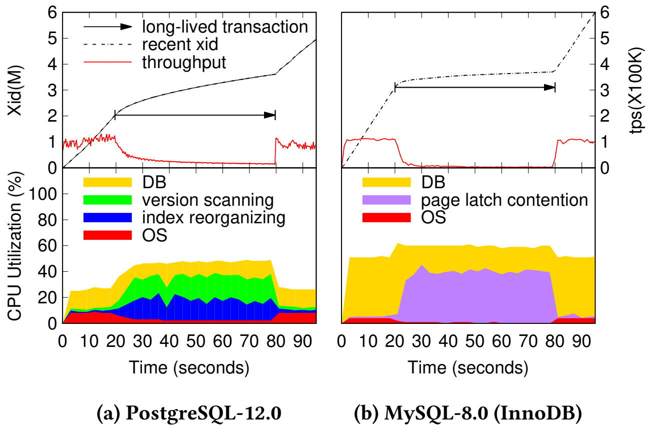


Figure 3: Effects of a long-lived transaction.

to provide users with the illusion of being a sole user in systems. Serializability, the highest isolation level, has been regarded as what all systems must pursue but thought to be somewhat hard to implement efficiently for general purposes. Practical alternatives have been proposed to relax strict data integrity by permitting some data anomalies in exchange for better performance. In this respect, snapshot isolation (SI), or something similar, has received widespread support from database vendors, so that modern database systems often set such isolation levels as defaults. At the heart of these is multi-version concurrency control that allows reads to access older versions of a record while writes act on the most recent version, thus avoiding read-write conflicts on physical data items and evading lock-induced blocking completely. In MVCC, record versions are conceptually chained for reads to search a proper version so that the version store inevitably keeps two types of information for each version item: record version and address locator to the version itself. Hence, version management systems form the bedrock of multi-version concurrency control in any database system.

For the past decades, researchers and database vendors have striven hard to reach two well-established designs: in-row versioning and off-row versioning. Depending on how versions and locators are organized, the pros and cons of each design are easily recognized. To sum up briefly, in-row versioning exploits spatial locality to expedite version searching by putting old versions with a record in the same data page as long as a page capacity permits. As shown on the bottom of Figure 2, PostgreSQL-12.0 stores versions in heap pages with an augmented search structure being embedded in a leaf page pointing to the first version in each heap page. However, a heap page can be overflowed with old versions so that page splits may occur frequently depending on the pace. *This will severely impinge on a degree of concurrency.* As opposed to this, off-row versioning makes index structures intact, regardless of versions. This is done by storing versions and locator information in a separate space. MySQL-8.0 with the InnoDB storage uses undo space for this purpose. As illustrated in the top of Figure 2, a record version and a roll pointer pointing to the predecessor version are all put together in a version item. A potential shortcoming is a long lookup time in searching for a requested version.

The two designs share the main cause of their drawbacks in common; reclaimable versions pile up rapidly if long-lived transactions exist in a system. Our evaluation revealed that the effect of an increase in the version chain length on transaction throughput is devastating to database engines, regardless of record versioning schemes. Any systems exposed to the risk of having such uncleaned dead versions would suffer storage space shortage as well as performance problems.

Motivation. The present work is motivated by the undesirable performance phenomena observed in two full-fledged database systems, when experimental evaluation is conducted with short transactions running OLTP workloads and long-lived transactions reading data items. Figure 3 shows the preliminary results. Common to both engines is the behavior that as a version chain gets longer, transaction throughput is sharply collapsing. Code analysis discovered that *latch duration on a page increases as the time taken for a transaction to traverse the long version chain grows*. This increased latch duration exacerbates contention on the concerned pages, thus leading to performance collapse. Answers for the question on who holds a latch long time depend on how version lookup is implemented in each engine. In PostgreSQL, the code for version lookup searches a version from the oldest, so that short transactions take much longer time than a long transaction. In contrast, MySQL searches a version from the newest so that a long transaction is badly impacted. In conclusion, whoever holds the latch longer will be the culprit liable for severe contention.

As shown in Figure 3(a), PostgreSQL suffers throughput collapse as a long transaction remains alive. In-depth looking through profiling revealed that PostgreSQL is internally fighting two enemies since a major portion of CPU time went to two activities: version searching and index modifications. Among two, frequent index modifications arise uniquely in database systems using in-row versioning, and they seem to be more complicated since even a split page is quickly flooded with a burst of newly generated versions. Moreover, page splits not only hamper the progress of foreground transactions, but also produce redo logs for capturing changes of databases. In summary, in-row versioning still exhibits bad signs in all aspects of performance metrics with LLTs.

A similar, bad story occurs in MySQL as well, with different signs of performance problems. As an LLT starts, MySQL also sharply degrades its throughput as a long-lived transaction performs massive reads that take longer time to search a version in undo space and I/O time to fetch undo pages aggravates latch contention more. The long latch duration caused by a long transaction surely prevents update transactions from accessing data pages, thus leading to throughput collapse. This is shown in Figure 3(b). Fortunately, no signs of index modifications have been discovered owing to its off-row versioning strategy. This advantage is favored to databases when updates are made on bigger size rows.

In-memory databases treat the issue as garbage collection problems since all versions reside in memory. However, prior proposals developed for in-memory engines cannot be applied to disk-based systems. Motivated by our preliminary results, the primary challenge can be posed as follows: *unnecessary versions should be readily classified and purging should be done without risking the representation invariant and incurring other latent performance matters.*

3 VERSION DRIVER ARCHITECTURE

This section describes the architecture of vDriver by first providing theoretical foundations essential for understanding our designs. The primary design rationale for the proposed architecture and the core design principles used in addressing major technical challenges will follow next.

3.1 Theoretical Foundations

This section reviews some salient concepts found in textbooks [5, 8, 28] and introduces new theoretical frameworks used throughout the present work. A database is a set of records $\mathbb{R}$ and their old versions $\mathbb{V}$, *all committed.* An i^{th} (old) version of a record $r \in \mathbb{R}$ is denoted as $v^{r,i}$ indexed in reverse chronological order, and, without loss of generality, r is equivalent to $v^{r,0}$. Hence, $\mathbb{V}$ is defined as $\mathbb{V} = \{v^{r,i} | \forall r \in \mathbb{R}, i > 0\}$. The visibility of a version is a measure of a time period that a version is visible to live transactions who started during that period, and the visibility of $v^{r,i}$ is therefore specified by start and end timestamps, denoted as $v_s^{r,i}$ and $v_e^{r,i}$, respectively. Let t_b^k and t_c^k be the begin and commit timestamps of a transaction T_k, respectively. By the definition, $v_s^{r,i}$ is indeed t_c^k of a transaction T_k who creates $v^{r,i}$ and makes it visible from t_c^k onwards, whereas $v_e^{r,i}$ is the start timestamp of a newer version $v^{r,i-1}$ since $v_s^{r,i-1}$ formally ends the visibility of $v^{r,i}$: So, $\forall v^{r,i} \in \mathbb{V} : v_e^{r,i} = v_s^{r,i-1}$ and $v_e^{r,0}$=∞. Hekaton engine [6] denotes this range as *valid time*. Following the definition of SI [4, 28], a *snapshot read* can be defined as follows:

Definition 3.1 (Snapshot read). A record version $v^{r,i} (i \geq 0)$ in $\mathbb{R} \cup \mathbb{V}$ is a *snapshot read* of r to T_k, denoted as s_r^k, if and only if $v^{r,i}$ is the latest version committed before t_b^k, i.e., $v_s^{r,i} < t_b^k < v_e^{r,i}$, except what T_k updates.

A snapshot of a database for a transaction T_k is logically decided at the time T_k starts, and can be defined as follows:

Definition 3.2 (Snapshot of a database). A snapshot $\mathbb{S}_k$ of a transaction T_k is a set of versions being the *snapshot reads* of records to T_k: $\mathbb{S}_k = \{v^{r,i} | \forall r \in \mathbb{R} : v^{r,i} = s_r^k\}$.

Then a version is said to be *dead* if it is not included in any valid snapshot of $\mathbb{T}$, a set of live transactions. The formal definition of a dead version can be written as follows:

Definition 3.3 (Dead version). A record version $v^{r,i} \in \mathbb{V}$ is said to be *dead* if $\forall T_k \in \mathbb{T} : v^{r,i} \notin \mathbb{S}_k$ (or equivalently expressed as $\nexists T_k \in \mathbb{T} : v_s^{r,i} < t_b^k < v_e^{r,i}$), or $\mathbb{T}=\emptyset$.

A dead version, once confirmed, can not revoke its status, and all dead versions are reclaimable. A group of non-overlapping time ranges can be provided as a set of *dead zones* such that versions born during one of the time ranges are identified as dead versions subject to Definition 3.3. A time range z=$[z_s, z_e]$ establishes a *dead zone*, if there is no live transaction started in between two: $\nexists T_k \in \mathbb{T} : z_s < t_b^k < z_e$. A complete set of dead zones can be defined as follows:

Definition 3.4 (Complete dead zones). A set of dead zones, $\mathbb{Z}_\mathbb{T}$=$\{z^1(=[-\infty,t_b^1]), \cdots, z^m(=[t_b^{m-1},t_b^m]), z^{m+1}(=[t_b^m, C^T])\}$, is *complete dead zones* for a nonempty set $\mathbb{T}$, where m=$|\mathbb{T}|$, $t_b^i (1 \leq i \leq m)$ is sorted in ascending order and C^T is the current time. If $\mathbb{T}$=$\emptyset$, then $\mathbb{Z}_\mathbb{T}$=$\{z^1(=[-\infty, C^T])\}$.

Following the definitions, a correctness condition for identifying dead versions with $\mathbb{Z}_\mathbb{T}$ constructed from $\mathbb{T}$, can be formalized as a *version pruning theorem* as below, and the version pruning theorem proved here subsumes the purging condition used in many database systems.

Theorem 3.5 (Complete Version Pruning Theorem). *Given $\mathbb{Z}_\mathbb{T}$ for $\mathbb{T}$ at any moment, a version $v^{r,i} \in \mathbb{V}$ can be pruned if and only if $\exists z^k \in \mathbb{Z}_\mathbb{T} : (z_s^k < v_s^{r,i}) \wedge (v_e^{r,i} < z_e^k)$ (or equivalently expressed as $\nexists z^k \in \mathbb{Z}_\mathbb{T} : v_s^{r,i} < z_s^k < v_e^{r,i}$).*

Proof. **(Prunability: $\leftarrow$)** Given a version $v^{r,i}$, assume that there exists a dead zone $z^k \in \mathbb{Z}_{\mathbb{T}}$ that satisfies $(z_s^k < v_s^{r,i}) \wedge (v_e^{r,i} < z_e^k)$. If $\mathbb{T} \neq \emptyset$, for an arbitrary transaction $T_j \in \mathbb{T}$, it started either before z_s^k or after z_e^k since, by definition of a dead zone, there can be no transaction who started in between z_s^k and z_e^k. If it started before z_s^k then $t_b^j < z_s^k < v_s^{r,i}$. If it started after z_e^k then $v_e^{r,i} < z_e^k < t_b^j$. In either case, by Definition 3.1, $v^{r,i}$ is not s_r^j to T_j. Hence, $\forall T_j \in \mathbb{T} : v^{r,i} \notin \mathbb{S}_j$, and by Definition 3.3, $v^{r,i}$ is a dead version and can be pruned correctly. If $\mathbb{T}=\emptyset$, then $v^{r,i}$ can be pruned trivially.

(Completeness: $\rightarrow$) Consider an arbitrary dead version $v^{r,i} \in \mathbb{V}$. By Definition 3.3, $\nexists T_j \in \mathbb{T} : v_s^{r,i} < t_b^j < v_e^{r,i}$. Let T_l and T_e be two transactions, if exist in $\mathbb{T}$, who started latest before $v_s^{r,i}$ and earliest after $v_e^{r,i}$, respectively. Then, $(t_b^l < v_s^{r,i}) \wedge (v_e^{r,i} < t_b^e) \wedge \nexists T_j \in \mathbb{T}$: $(t_b^l < t_b^j < v_s^{r,i}) \vee (v_e^{r,i} < t_b^j < t_b^e)$. Now we have four cases depending on their existence.

(Case 1: $T_l \in \mathbb{T} \wedge T_e \in \mathbb{T}$**)** In this case, T_l and T_e are successive in begin timestamp order in $\mathbb{T}$, and by Definition 3.4, there is z^k=$[t_b^l, t_b^e] \in \mathbb{Z}_{\mathbb{T}}$, which satisfies $(t_b^l < v_s^{r,i} < v_e^{r,i} < t_b^e)$.

(Case 2: $T_l \in \mathbb{T} \wedge T_e \notin \mathbb{T}$**)** In this case, T_l is the last transaction in begin timestamp order in $\mathbb{T}$, and there is z^k=$[t_b^l, C^T] \in \mathbb{Z}_{\mathbb{T}}$, which satisfies $(t_b^l < v_s^{r,i} < v_e^{r,i} < C^T)$.

(Case 3: $T_l \notin \mathbb{T} \wedge T_e \in \mathbb{T}$**)** In this case, T_e is the first transaction in begin timestamp order in $\mathbb{T}$, and there is a dead zone z^k=$[-\infty, t_b^e] \in \mathbb{Z}_{\mathbb{T}}$, which satisfies $(-\infty < v_s^{r,i} < v_e^{r,i} < t_b^e)$.

(Case 4: $T_l \notin \mathbb{T} \wedge T_e \notin \mathbb{T}$**)** In this case, $\mathbb{T}=\emptyset$ and there is z^1=$[-\infty, C^T] \in \mathbb{Z}_{\mathbb{T}}$, which naturally satisfies $(-\infty < v_s^{r,i} < v_e^{r,i} < C^T)$. □

Theorem 3.5 connotes an important, overlooked rule stating that *if systems have no live transactions, then the entire* $\mathbb{V}$ *can be obliterated immediately*; thus Theorem 3.5 is essential to our version pruning. Crucial to enforcing Theorem 3.5 is visibility information that is specified by commit timestamps of update transactions. The reality, however, is different in that many database engines embed t_b of an updater instead of t_c in a record version. If a version has two t_bs corresponding to $v_s^{r,i}$ and $v_e^{r,i}$ in our taxonomy, respectively, then computing the visibility of a version is demanding.

To address the problem, such systems assign a set of begin timestamps of live transactions, called a *read-view* and also denoted as $\mathbb{T}_k^{rv}$, to T_k at the time of t_b^k. With T_k having $\mathbb{T}_k^{rv}$, then $v^{r,i}$ is said to be 'committed' at the time of t_b^k if the following condition holds: $(v_s^{r,i} < t_b^k) \wedge (v_s^{r,i} \notin \mathbb{T}_k^{rv})$. Extending this reasoning, the condition for $v^{r,i}$ to be a snapshot read to T_k, as defined in Definition 3.1, is equivalently rewritten as follows: $((v_s^{r,i} < t_b^k) \wedge (v_s^{r,i} \notin \mathbb{T}_k^{rv})) \wedge ((v_e^{r,i} \in \mathbb{T}_k^{rv}) \vee (v_e^{r,i} > \max_{\forall t_b^m \in \mathbb{T}_k^{rv}} t_b^m))$. The first part indicates that a transaction who updated $v^{r,i}$ already committed at t_b^k, while the second part implies that $v^{r,i-1}$ is either uncommitted (if it began) or yet to be created at t_b^k; thus, $v^{r,i}$ becomes a snapshot read to T_k. The rules described above are almost identical to what MySQL and PostgreSQL use now. The conditions stated in other definitions are equivalently rewritten likewise. The important condition in Theorem 3.5 for checking the inclusion property of the visibility of a version to $z^k \in \mathbb{Z}_{\mathbb{T}}$ can be written as follows: $\nexists z^{k+1}$=$[t_b^k, t_b^{k+1}] \in \mathbb{Z}_{\mathbb{T}}$:$((v_s^{r,i} < t_b^k) \wedge (v_s^{r,i} \notin \mathbb{T}_k^{rv})) \wedge ((v_e^{r,i} \in \mathbb{T}_k^{rv}) \vee (v_e^{r,i} > \max_{\forall t_b^m \in \mathbb{T}_k^{rv}} t_b^m))$, where $\mathbb{T}_k^{rv}$ is defined for T_k. If $\mathbb{T}=\emptyset$ ($\equiv \mathbb{Z}_{\mathbb{T}}=\{[-\infty, C^T]\}$), then $\mathbb{V}$ can be $\emptyset$.

As transactions begin and commit, dead zones are created and merge into a larger one. Combining two consecutive dead zones when a transaction whose t_b adjoins the left and the right dead zones commits, is a necessary condition for the current dead zones to be $\mathbb{Z}_{\mathbb{T}}$. Thus, designing succinct data structures for representing such dynamically changing $\mathbb{Z}_{\mathbb{T}}$ and efficient access methods not to incur contention, is one of the important technical matters to be addressed carefully in pursuit of removing dead versions efficiently.

3.2 Design Overview

Main design rationale. The foremost ignored basic fact in record versioning for MVCC is, record versions become reclaimable once none of the users can *read* them (Definition 3.3). By Theorem 3.5, the restart of a DBMS (i.e., $\mathbb{T} = \emptyset$) certainly renders all versions $\mathbb{V}$ reclaimable, thus emptying version space. However, a database system may crash any time so that it may have uncommitted records, denoted as $v^{r,\perp\rightarrow 0}$; then all the committed versions of the record are conceptually in a transient state and they can be denoted as $v^{r,i\rightarrow i+1}$ $(i \geq 0)$. The essential prerequisite for Theorem 3.5 is that a database must be in a consistent state requiring that a record and versions be committed. To ensure that, undo recovery should replace any $v^{r,\perp\rightarrow 0}$ with $v^{r,0\rightarrow 1}$; the same also works for handling aborted transactions. The rollback can, therefore, be expedited if $v^{r,0\rightarrow 1}$ is placed nearby $v^{r,\perp\rightarrow 0}$.

Considering the given designs in all aspects, we believe that in-row versioning outperforms the off-row method with respect to processing undo operations, while off-row versioning, as demonstrated in Figure 3, outdoes its counterpart in making index structures strongly resilient to a burst of incoming record versions. This whole argument provides us with the compelling design rationale for the essential versioning principle of vDriver: *one version in-row and put trailer off-row*. In fact, keeping more than one version in data pages increases space overhead and the chance of having page splits substantially, without commensurate merit. Challenges posed ahead of us are design details to fill the gap between ideal formulae and measurable performance metrics. The design principles described below are what we propose for addressing such matters.

Version buffering for fast reads. Due to our design rationale, vDriver basically resembles off-row versioning in

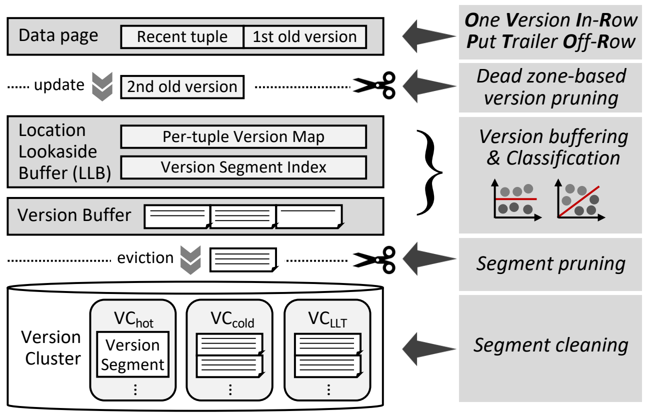


Figure 4: vDriver with core design principles.

that the vast majority of versions are stored in off-row space while leaving one version ($v^{r,0\to1}$ or $v^{r,1}$) in data pages. Key to space- and time-efficient version management is a set of our design principles applied to different layers of vDriver. Following the Oviraptor versioning principle, versions reside either in pages or in version space. The main difference between vDriver and legacy one in realizing off-row versioning lies in the version buffer layer of vDriver designed for fast version lookup, and this is the main workhorse to overcome the long lookup time required for traversing a version chain in legacy off-row versioning.

For this purpose, the version buffer layer of vDriver is composed of vBuffer and *location lookaside buffer* (LLB); vBuffer manages a portion of the entire version segments that contain old record versions and LLB maintains per record chain of locators pointing to versions in version segments in both vBuffer and version store. The role of our version buffer is to serve reads with in-memory versions without being delayed by the modifications of vBuffer or LLB due to version insertions or deletions. For example, when versions are coming in and out, shared data structures change accordingly. Modifying such data structures while allowing concurrent transactions to access, therefore, makes heavy use of well-established *concurrent programming principles* [9, 10]; many of well-proven concurrent data structures and programming techniques are used here.

Version cleaning for space efficiency. As formalized in §3.1, rigorous theoretical frameworks are provided to any record versions to be checked for identifying dead versions. However, in practice identifying a reclaimable point precisely for each version is a daunting, wasteful task; performing such tasks against left-over versions held by long-lived transactions is a particularly acute matter to legacy systems, where versions are often stored in append-only store so that selective removal is challenging. To enable and expedite version cleaning in off-row version space, record versions are classified into different categories by pre-defined rules, and a concept of version classification is devised and used in conjunction with Theorem 3.5. The main role of version classification is to separate a group of versions possessing similar characteristics from others so that versions in the same category are likely to be removed altogether without minding other groups, and this is pivotal to curtailing version space, even if there are long-lived transactions. Hence, our version classification is for batch version cleaning that substantially relieves the burden of pruning individual versions.

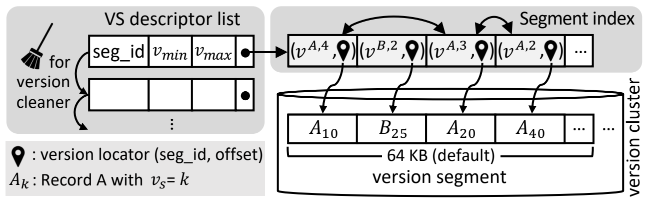
To reduce the space usage for dead versions, vDriver proactively *prunes* dead versions before they are hardened in version store or *cleans* hardened ones from stable storage, all done by applying Theorem 3.5. Hence pruning reduces memory footprint while cleaning curtails storage space. The first place pruning takes place is when a version is relocated from in-row pages to off-row version space in vBuffer, and this pruning is denoted as dead zone-based *version pruning*. Live versions survived the first pruning are stored in proper in-memory version segments through version classification. The next pruning chance then comes when the version segments are flushed to stable storage, and this is denoted as dead zone-based *segment pruning* and viewed as the batch pruning of dead segments in vBuffer. Despite our multilayered pruning, some versions are finally written to version space if they are snapshot reads to live transactions but never accessed before. vCutter will exercise *segment cleaning* as a radical last resort, by cutting dead segments in version space. Hence, segment pruning and cleaning are technically the same and differ as to the location it takes place.

Summarizing what we have described here, Figure 4 shows the layered architecture of vDriver and the core design principles used in pursuing the primary virtue of vDriver; *live versions are served to reads while they remain in memory for fast responses, and dead versions are pruned as early as we can in order to minimize space overhead.* Starting from vSorter, the major components of vDriver will be presented.

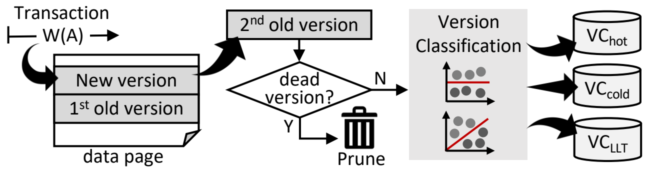
3.3 vSorter

vSorter is a sub-component of vDriver responsible for managing the proper placement of record versions obeying the Oviraptor rule, and it manages vBuffer and performs version classification and pruning. Implicit here is that since segments in vBuffer and their segment indexes in LLB are managed by vSorter, it undertakes our segment pruning. This section first describes data structures and storage layouts for version management, and it explains the overall sequence of Oviraptor versioning. The dead zone-based version pruning and classification will be described last.

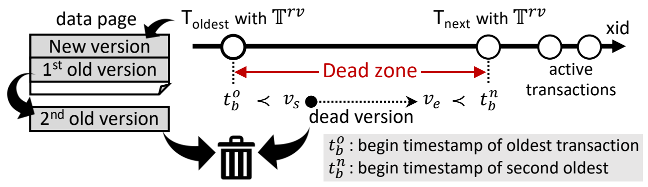
Data structures. Version space is composed of three version clusters, each of which stores versions grouped by our



(a) Overall architecture of vSorter



(b) Oviraptor versioning


data page
New version
1st old version
2nd old version
T_{oldest} with $\mathbb{T}^{rv}$
T_{next} with $\mathbb{T}^{rv}$
xid
Dead zone
active transactions
$t_b^o \prec v_s$
$v_e \prec t_b^n$
dead version
t_b^o : begin timestamp of oldest transaction
t_b^n : begin timestamp of second oldest

(c) Dead zone-based version pruning

Figure 5: vSorter design.

version classifier. Conceptually, a version cluster is a contiguous byte array, divided into fixed-sized segments the size of which is configurable. Version segments, once created, are buffered in memory and managed by our version buffering layer (i.e., vBuffer), and vBuffer builds an in-memory segment index for each segment. A segment index is simply an array of quadruples of (version, locator, llink, rlink), and links point to previous and next versions of a record. The link can be viewed as per record version chain, and LLB points the head and the tail of this chain. For version cleaning, vSorter maintains summary meta-data, called VS descriptor, for each uncleaned version segment VS. Each VS descriptor includes three fields — `seg_id`, v_{min} and v_{max} — used for deciding whether or not the current segment can be reclaimed, where v_{min}=$\min_{\forall v^{r,i} \in VS} v_s^{r,i}$ and v_{max}=$\max_{\forall v^{r,i} \in VS} v_e^{r,i}$. The overall design of vSorter is illustrated in Figure 5(a).

Oviraptor versioning. The first step of Oviraptor versioning commences when a transaction updates a record r (i.e., $v^{r,0}$). If $v^{r,0}$ already holds $v^{r,1}$ beside itself, then the current record will become a new 1st version $v^{r,0\rightarrow 1}$, and $v^{r,\perp\rightarrow 0}$ occupies the placeholder for $v^{r,1\rightarrow 2}$ by pushing it to off-row space. A toggle bit is used to indicate which one is the current record, instead of physically swapping two. Before $v^{r,1\rightarrow 2}$ is relocated to off-row version space, *version* pruning is executed to prune $v^{r,1\rightarrow 2}$ by Theorem 3.5. If $v^{r,1\rightarrow 2}$ survives version pruning, then it becomes $v^{r,2}$ and is stored in one of version clusters through version classification. For version classification, vSorter takes three factors into consideration: start and end timestamps of a version, and a clear sign of indicating that a version is included in any snapshot of LLTs. In the present work, versions are classified as three classes: (i) hot versions (VC_{hot}), (ii) cold versions (VC_{cold}) and (iii) versions that are included in snapshots of LLTs (VC_{LLT}). By classifying versions as such, the cleaning of a majority of versions once thought to be difficult due to LLTs now seems feasible as soon as they turn out to be unreachable. The overall flow of Oviraptor versioning is illustrated in Figure 5(b).

Dead zone-based version pruning. Pruning individual versions, or segments of versions, requires a complete condition, and the pruning criterion used to identify a dead version is Theorem 3.5 with the zone containment condition being augmented by a *read-view*. As depicted in Figure 5(c), vSorter checks whether $(v_s^{r,i}, v_e^{r,i})$ of a version $v^{r,i}$ is bounded by one of given dead zones. Once $v^{r,i}$ turns out to be a dead version, it is pruned immediately, thus reducing space consumption. Noticeable here is the way of enforcing Theorem 3.5 that the dead zone-based version pruning is opportunistic in the sense that it is exploited to identify and erase a dead version whenever a record version needs to be relocated.

Our version pruning has a trade-off between pruning accuracy and performance; the more accurate dead zones we aim to build, the longer time it may take to complete due to contention on shared structures. Due to this trade-off, we pursue a moderate, best-effort policy in designing algorithms. Since database systems internally manage a set of live transactions using a shared list (i.e., MySQL) or array (PostgreSQL), constructing $\mathbb{Z}_{\mathbb{T}}$ can be done using existing shared data structures. In addition to this, both engines put transaction id in a record version and decide the visibility of a version using the concept of read-views. In line with this design, $\mathbb{T}^{rv}$ needs to be constructed from a set of begin timestamps of live transactions. However, due to the accuracy-performance trade-off, updating $\mathbb{Z}_{\mathbb{T}}$ is not performed every time a transaction begins or commits, rather it is updated periodically. Although the lazy updates make $\mathbb{Z}_{\mathbb{T}}$ slightly outdated, it would not matter since a majority of dead versions will be identified by z^1 (i.e., [-∞, t_b^o] with $\mathbb{T}_o^{rv}$) or z^2, when a system runs short transactions. Otherwise, dead versions will fall into wide dead zones that can be created if there are some long-lived transactions[*]. Owing to this, though containing some dead versions, version space required for serving live transactions can be substantially reduced; §4 will provide further discussions on this claim with evaluation results.

[*]A wide dead zone is created when a series of short transactions, started right after an LLT, commit and accordingly the dead zone starting from t_b of the LLT is repeatedly merged with its adjacent dead zone.

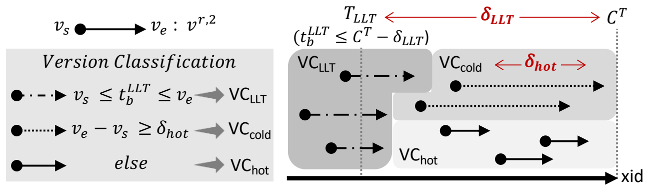


Figure 6: Version classification in action.

Version classification. The primary objective of version classification is to group versions exhibiting similar characteristics—update interval and being snapshot reads to LLTs—into the same category, so that cleaning dead versions are not suspended by live versions; thus batch cleaning of versions in the same class is feasible. The main design rationale for having three classes is based on two observations; (1) *records having long update intervals produce versions whose visibility is commensurate with the interval* and (2) *records being snapshot reads to long-lived transactions must survive as long as such LLTs are alive.* The former is to capture versions whose visibility is affected by access patterns, while the latter is to freeze snapshot reads for long-lived transactions. Hence, versions born around the same time are classed separately through version classification if they differ considerably as to their visibility. By doing this, live versions in one class would never impinge on the removal of dead versions in other classes, and this is of vital importance in addressing the primary challenge posed in §2. Although more types of version classes can be defined upon discovering new characteristics, vDRIVER uses three classes in the present work.

Since both MySQL and PostgreSQL store versions regardless of the characteristics, the selective removal of each individual dead version is a daunting task to the current architecture. Even if garbage collection methods [6, 17, 19, 25] proposed for in-memory database engines can be applied to the two engines, it entails a significant amount of I/O activities to make version chains correct with respect to the representation invariant since purging individual dead versions stored in stable storage inevitably incurs disk I/Os.

Version classification occurs when a version is relocated from a data page to vBUFFER after it survives dead zone-based *version* pruning. Among three classes, the first type VC_{LLT} is to store a set of versions that are still snapshot reads to at least one of long-lived transactions, and a long-lived transaction is defined as the one whose t_b is older than a certain threshold δ_{LLT}, which is a multiple of an average transaction length. As the first stage of classification, a version (exactly, $v^{r,2}$) is checked whether it is a snapshot read to any LLTs, or not.

After this stage, version classification performs the next test to classify versions into VC_{hot} and VC_{cold}; VC_{hot} stores a set of versions having short update intervals, and the threshold for identifying it is defined as δ_{hot} (i.e., $v_e - v_s < \delta_{hot}$), and then VC_{cold} manages the rest of versions whose visibility is relatively longer than versions in VC_{hot}. Though more research on finding δ_{hot} is needed, a multiple of an average update interval of versions within a certain time window can be a good proxy for this purpose. Figure 6 shows the core logic and illustrations of our version classification.

Despite our best effort in classifying versions accurately, some versions that later turn out to be VC_{LLT} were not classed as such at first. The classification error is mainly influenced by workload characteristics, such as access patterns and transaction length, etc. In fact, the time window during which misclassification is likely to occur is called a *vulnerability window*, and it is the interval between the begin timestamp and the time a transaction is identified as an LLT. The penalty for putting live versions in the wrong basket is the suspension of cleaning dead versions in the concerned class, thus leading to increased space consumption. But, our segment-based architecture is able to minimize the damage to version space through segment-based version cleaning, and in-depth analysis for this will be discussed in §4.

3.4 vCUTTER

vCUTTER is a sub-component of vDRIVER mainly responsible for cleaning reclaimable version segments. Unlike version pruning executed on a single version before being relocated to off-row version store by the OVIRAPTOR rule, the removal of version segments inevitably has a direct impact on the version lookup process since it may have to *cut* some locator links pivotal to reaching other record versions outside the cut-off segments. Therefore, cleaning versions involves *cut-and-fix* operations not to risk the representation invariant: *versions that are still snapshot reads to some live transactions must be reachable.* Moreover, the cut-and-fix routine of vCUTTER may conflict with the version insertion routine of vSORTER; hence alleviating the contention between the two routines demands a sophisticated technique. Further discussion on our *collaborative* approach to this matter will be presented later. Hence, the design focus of vCUTTER is on the efficiency of cut-and-fix operations to make it practical.

Version segment cleaning. For cleaning version segments, vCUTTER uses in-memory meta-data, VS descriptor. Among three fields, vCUTTER uses v_{min} and v_{max} to check whether the range $[v_{min}, v_{max}]$ defined for a version segment VS is included by any single dead zone of $\mathbb{Z}_{\mathbb{T}}$. From a macro perspective, version segment cleaning is indeed *dead zone-based segment pruning* in storage. The left side of Figure 7 illustrates the process of checking the range inclusion property of a given segment against $\mathbb{Z}_{\mathbb{T}}$. If a version segment turns out to be dead, then the segment is purged immediately. Noticeable here is our choice of segment timestamps (i.e., v_{min} and v_{max}) instead of individual version timestamps for

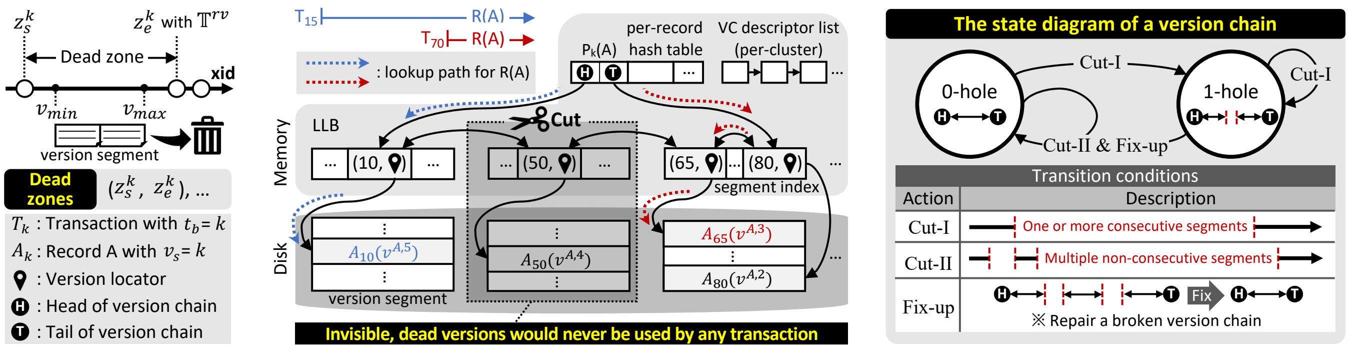


Figure 7: Version segment cleaning of vCutter.

cleaning. Although it expedites version cleaning by enabling segment-level batch cleaning, it hurts the *completeness* of version cleaning; dead versions may not be cleaned due to one or more live versions in the same segment. However, we prefer to opt for segment timestamps for two practical reasons: (1) version cleaning is not a time-critical task and (2) all dead versions will be removed eventually.

As shown in the center of Figure 7, a version in a dead segment contains locator links to previous and next versions that can be stored in other version segments. If links point to versions in the same dead segment, then nothing has to be done for the version. Otherwise, cutting these links break a version chain, thus creating some *holes* in the chain. Fixing these broken links may seem urgent at a glance, but careful looking reveals that versions are reachable although there is a hole in a chain. As shown in Figure 7, the removal of a version A_{50} ($=v^{A,4}$) due to segment cleaning creates a hole in a version chain for record A. However, reads on A by T_{15} and T_{70} can still correctly reach their snapshot reads — A_{10} ($=v^{A,5}$) and A_{65} ($=v^{A,3}$) — by traversing from the head and the tail of the chain, respectively. Note that the penalty for taking the wrong direction is negligible owing to the short-chain length. If the broken chain has one more hole at this moment, then there exists an isolated chain segment whose versions are never reachable by any means and become orphan versions. This is the time fixing the broken links becomes a pressing matter to vCutter since unreachable versions are the clear evidence of violating the representation invariant.

Formalizing a complete set of possible states with relevant actions, the state diagram of a version chain is depicted on the right side of Figure 7. A version chain has two states: 0-hole and 1-hole states. State transitions occur between two states with *triggering events* and *resulting actions*. An event that triggers a state transition is when version segments are cleaned. Depending on given dead zones, more than one version segments can be cut at any moment and this creates two types of triggering events. The first triggering event is denoted as 'Cut-I', and it is the case of having one or more *consecutive* dead segments, which can create at most one hole in any version chain. For example, Cut-I frequently occurs when z^1 keeps expanding and accordingly including subsequent segments while other zones have tight intervals, especially when systems run short transactions. Likewise, the second event is denoted as 'Cut-II' that is the case of having multiple nonconsecutive dead segments, which may create multiple holes in a chain. vCutter may face Cut-II if some LLTs create multiple wide dead zones that include nonconsecutive dead segments.

The concerned state is when a chain is already in 1-hole state and vCutter faces the Cut-II event that puts the reachability of some versions at risk. To avoid a structural crisis, a preemptive action, called 'Fixup', is taken to fix all the broken chains every time a dead version is encountered. The main operation of Fixup is to fill a hole by properly linking previous and next versions of a dead version. After Fixup is done, a broken chain will be in the 0-hole state. One technical matter has to be addressed clearly for this cut-and-fix procedure to be practically efficient; how to resolve latent contention between vCutter and vSorter.

Collaborative version cleaning. As vCutter performs segment cleaning, it has to fix broken links continuously. This may interfere with the insertion of newer versions into the same version chain by vSorter; in particular, Figure 8 illustrates the situation where vCutter logically deletes a version and fixes the pointers of neighbor versions while vSorter inserts a new version into this chain. The versions and fields changed by vCutter and vSorter are colored in blue and red, respectively. The contention, therefore, arises when vCutter is to delete $v^{r,2}$ while vSorter inserts a new $v^{r,1\rightarrow 2}$ into this chain. A naive solution is to use a latch to protect the whole version chain for mutual exclusion, but a system will suffer latch contention and fix-up operations will be severely delayed since transactions outnumber vCutter.

Our approach to this matter is based on *collaborative processing* in that the execution of version cleaning is delegated to whichever obtains permission to access $v^{r,2}$ in a concerned

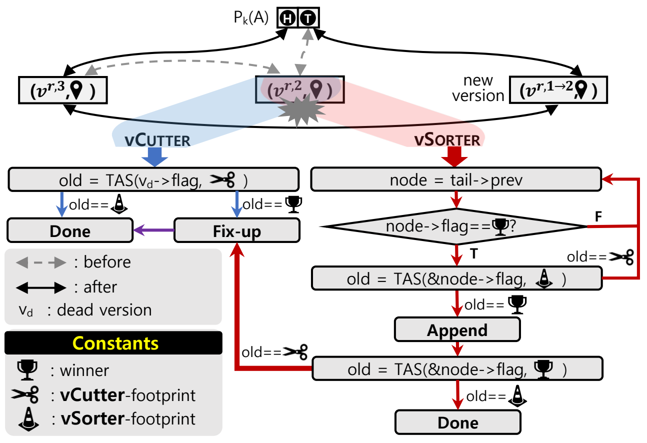


Figure 8: Collaborative version cleaning.

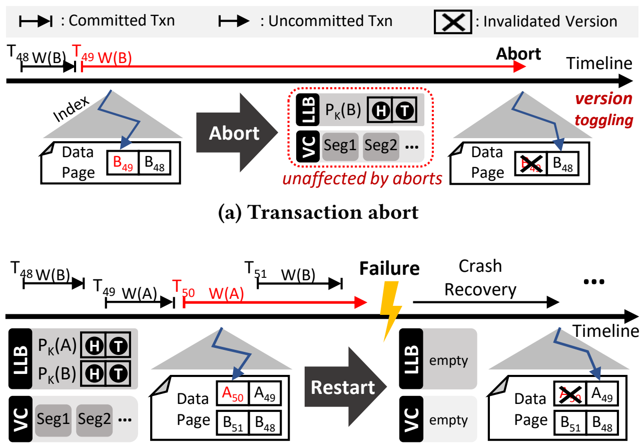


Figure 9: Undo recovery in vDRIVER.

chain. The winner is the one who succeeded in storing a special value into a flag first, and this is done by executing the atomic test-and-set (TAS) instruction. The default value set for a flag is predefined constant (i.e., `winner` constant value), and vCUTTER and vSORTER invoke TAS with their intention marks, `vCutter-footprint` and `vSorter-footprint`, to inform the winner of the footprint of the looser. An important invariant preserved here is, the dead version $v^{r,2}$ must be deleted by whoever wins this race. Otherwise, version segment cleaning would lead to malfunction. To this end, a protocol, called *collaborative version cleaning*, is devised and shown in Figure 8; if vSORTER wins, then it will do both tasks. Otherwise if vCUTTER wins, it returns immediately after it completes fix-up work, without inserting $v^{r,1\to2}$ to the end of the chain; fulfilling the suspended work will be done by vSORTER after it does spin-waiting. Although this seems to be a lopsided policy towards vCUTTER, it is acceptable since vCUTTER is more urgent than vSORTER and vCUTTER is battling with numerous foreground transactions.

3.5 Recovery

One of original claims proposed in PostgreSQL [24] is the ability to finish database recovery almost instantaneously. Since then instant recovery has been regarded as a major merit of in-row versioning because in-row versioning puts the first old version right besides a tuple hardened by an uncommitted transaction. Then the undo recovery is simply changing a few bits to toggle the pointer to the most recently committed record. The same design rationale holds for the recent changes made to the record versioning component of Azure SQL Server [2]. Likewise, the OVIRAPTOR principle permits $v^{r,\perp\to0}$ and $v^{r,0\to1}$ to be placed in the same data page, although a system crashes. Then, the undo recovery of vDRIVER just toggles a few bits to indicate that $v^{r,0\to1}$ reverts to $v^{r,0}$ since it is the most recently committed record.

Depending on circumstances, vDRIVER can dispose of version segments and buffered versions altogether, or none of them are affected by undo operations. Figure 9 illustrates how undo operations are performed in two different events; (1) when a transaction aborts and (2) when a system crashes. Figure 9(a) depicts the situation where a transaction T_{49} that updated $v^{B,0}$ (=B_{49}), is aborted, so that $v^{B,0}$ is logically deleted and $v^{B,1}$ (=B_{48}) becomes $v^{B,1\to0}$. Note that the completion of the abort leaves version segments and contents in LLB unaffected by undo operations. Figure 9(b) describes the situation where a database system crashed and the undo recovery performs rollback operations for a looser transaction T_{50} that updated $v^{A,0}$ (=A_{50}). Rolling back T_{50} is done by restoring $v^{A,1}$ (=A_{49}) to be $v^{A,1\to0}$. Important to observe here is the clear difference between two circumstances that everything stored in version clusters and LLB is completely obliterated in case of crash recovery by Theorem 3.5. Hence, the onset of restarting databases upon crashes just empties vDRIVER storage, thus being instant.

4 EXPERIMENTAL EVALUATION

This section presents our evaluation results. For our evaluation, two full-fledged database engines — MySQL-8.0 with InnoDB storage and PostgreSQL-12.0 — are chosen as the baseline engines, and properly modified to plug vDRIVER.

4.1 Implementations

vDRIVER is implemented as a proof-of-concept module and integrated into MySQL-8.0 (InnoDB storage engine) and PostgreSQL-12.0. Since MySQL is based on the multi-threaded architecture whereas PostgreSQL uses the multi-process model, vBUFFER and LLB in vDRIVER are tailored for each engine due to the difference in sharing their memory.

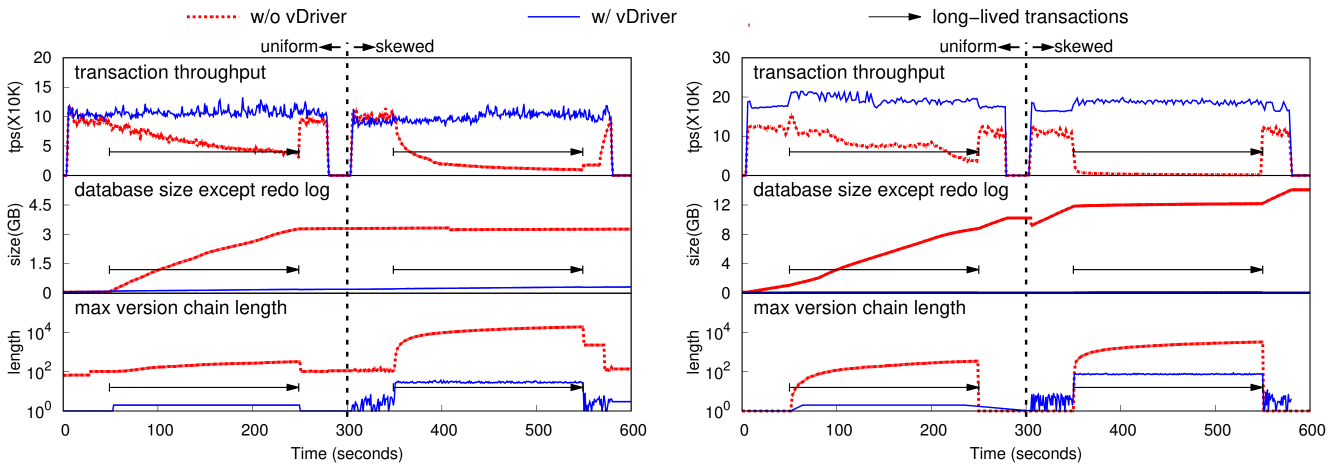


Figure 10: Throughput and version space overhead. (Left: PostgreSQL, Right: MySQL)

The visibility of a version. Since MySQL and PostgreSQL rely on a begin timestamp and a concept of a read-view for deciding the visibility of a version, their record format contains a `tid` field. For read-views, the InnoDB engine already maintains a set of read views for live transactions in `trx_sys->mvcc`. It is indeed $\mathbb{Z}_{\mathbb{T}}$ required for our version pruning, but vSorter does not access `trx_sys->mvcc` for performing $\mathbb{Z}_{\mathbb{T}}$-based version pruning, mainly due to latent contention on `trx_sys->mutex`, which protects many shared data structures relevant to transaction management. vSorter rather periodically creates a copy of it. In contrast, PostgreSQL bookkeeps an array of live processes in its shared memory, and there is no shared structure managing read-views for live processes. Therefore, proper changes are made to share the read-views of all live transactions; $\mathbb{Z}_{\mathbb{T}}$ is periodically constructed from PostgreSQL's live transactions. Our two pruning steps, of course, may use a slightly outdated version of $\mathbb{Z}_{\mathbb{T}}$ due to the periodic updates.

Implementing Oviraptor versioning. As a proof-of-concept implementation (not commercial-grade), the page layout of respective storage engines is properly modified to put one more record version and to redirect version lookup to LLB and vBuffer. Some of the reserved space in a record format, if exists, is used as toggle bits to indicate which one is $v^{r,0}$. Versions generated by record updates are the main target of vDriver. While reserving extra space in a data page and using that space as a placeholder, Oviraptor's in-place update policy conficts with the out-of-place update policy adopted by PostgreSQL. Since versions are seldom relocated once confirmed its location, tuple pointers can be returned to upper layers without holding a tuple (or buffer) lock. Our in-place update strategy may violate this invariant so that the original invariant is preserved by copying required tuples, although it may not be a robust solution. Since MySQL uses in-place updates, implementing Oviraptor versioning in MySQL is less demanding than PostgreSQL.

4.2 Evaluation Setup

Experimental evaluation is conducted on our 96-core machine with four Intel Xeon E7-8890 processors, 1 TiB memory, and two high-end NVMe SSDs. The machine runs Ubuntu 16.04.4 LTS distribution with a Linux kernel 4.4.0. We use the sysbench-like OLTP workload in the presence of long-lived transactions, by varying access patterns from uniform to skewed distributions. We then measure throughput, space consumption and other metrics of vanilla and vDriver-based engines. Unless stated otherwise, all (modified or vanilla) engines run under REPEATABLE READ isolation. We set vBuffer to 8 MiB for all experiments.

4.3 Evaluation Results

This section presents experimental results. Except for the first experimental result showing important performance metrics, MySQL and PostgreSQL will be used alternately in the following experiments due to the space limit. It is noteworthy that both show almost similar behaviors in all experiments, although their version searching order differs.

4.3.1 Throughput and Space Overhead. This section evaluates the performance of the systems over two important metrics: transaction throughput and space usage. The evaluation consists of two phases; the first phase uses a uniform access pattern while the second phase uses a highly-skewed one. In each phase, a group of long-lived transactions join around 50s in the first phase, and 350s in the second phase and perform a point query continuously against a table randomly chosen out of 48 tables, each of which contains 1000 records, the size of which is 256 bytes.

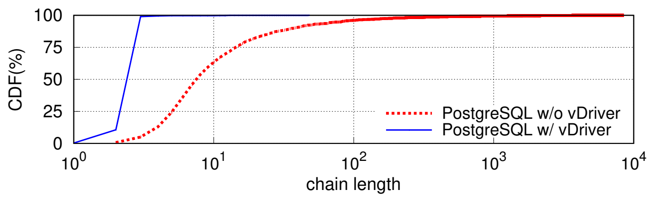


Figure 11: The CDF of version chain length.

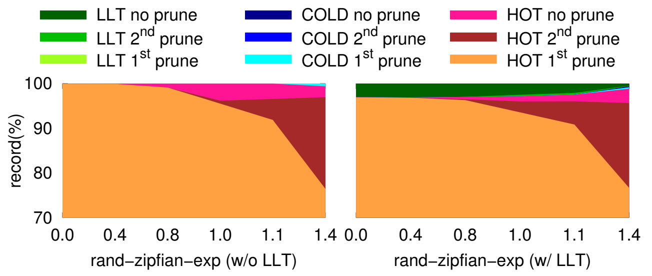


Figure 12: Pruning effects of vDRIVER on MySQL.

Figure 10 shows the results, and all the vanilla systems suffer performance degradation more sharply under skewed distributions, and their storage usage grows as time elapses. In PostgreSQL, update transactions are impacted badly due to long version chains, especially under the skewed workloads, while long latch duration caused by read transactions incur severe latch contention in MySQL. However, the vDRIVER-based engines retain transaction throughput and low space usage in the presence of LLTs, since vDRIVER successfully prunes and cleans dead versions and segments as early as they turn out to be dead. We observe two notable phenomena in MySQL; (1) MySQL with vDRIVER outperforms the vanilla system mainly because vDRIVER does not need undo logs for updates and so safely skips the undo generation routine that internally uses a few giant latches and (2) MySQL abruptly truncates undo space during the test due to the new undo space management policy [1] cutting one of '`undo_001`' and '`undo_002`' files alternately. Although we have seen prior proposals [6, 11–16, 18, 20–22, 26, 27] that have addressed performance issues caused by latch contention in database components, the two engines seem to possess unresolved matters that need attention from our community.

To further investigate the effect of our pruning, we trace the length of the longest valid version chain that allow transactions to traverse over the experimental period. The figure on the bottom (note we use a log scale in Y-axis) shows that the max chain length of the vanilla engines grows but rather steeply under highly-skewed workloads in presence of LLTs, while the max valid version chain in the engines with vDRIVER can be retained shorter than vanilla systems even with LLTs under uniform workloads. Even under highly-skewed workloads where a few hot records may lead to long version chains, vDRIVER can control the max chain length under 100 and retain the length until the end of LLTs.

In contrast, the max chain length of the vanilla engines reaches 10^4 and keeps growing as long as LLTs remain alive. Interesting to see in MySQL is its version lookup mechanism that although version space becomes large due to many possible reasons, MySQL's purge system internally limits the version traversal to the oldest read view of MySQL, i.e., `purge_sys->view`. This is why the length is 1 if there is no long-lived transaction. Figure 11 shows the CDF of version chain length under highly-skewed workloads, and most records retain short chain length in vDRIVER while the vanilla system shows a very wide spectrum. As verified here, the version chain length has an immense impact on the important performance metrics, and therefore the timely removal of dead versions is of importance to databases relying on multi-version concurrency control.

4.3.2 Pruning Effect. To see the effects of version and segment pruning, we show a breakdown of our pruning in the MySQL engine w/ or w/o LLTs, by varying the zipfian parameter (i.e., '`rand-zipfian-exp`' of sysbench). While vSORTER prunes dead versions and segments, we do extra work to obtain version class information just for this evaluation. In this experiment, four long-lived transactions join 10s after OLTP workloads start and end around 70s, and they reads randomly selected records continuously until they ends. Note that the breakdown of pruning effects with LLTs may differ as to how long we run the test. We first explain the common phenomena and discuss the effects of LLTs on our pruning.

Figure 12 shows the results, and 1^{st} `prune` and 2^{nd} `prune` indicate that versions are pruned in version and segment pruning, respectively. As shown in the figure, a majority of versions have been purged in one of two version pruning stages (i.e., '`HOT` 1^{st} `prune`' and '`HOT` 2^{nd} `prune`'), regardless of workload distributions and the presence of LLTs. Noticeable here is the impact of our version pruning that up to zipfian value 1.1, more than 92% of $v^{r,1\to2}$ versions can be pruned when being relocated from data pages to vBUFFER, regardless of the presence of LLTs. This has a significant implication that our version pruning with the OVIRAPTOR principle may play a crucial role in reducing version space by proactively pruning garbage versions in in-memory engines, as well as cloud databases, if concerned engines are properly modified to faithfully embody our designs. This area is a good place for future work. Interesting to see here is that frequently updated records tend to generate versions rapidly under highly-skewed workloads, and such versions are likely to legitimately pass the first pruning stage (i.e., version pruning). But, most of survived versions are pruned in the second pruning stage by the segment pruning, and only a small

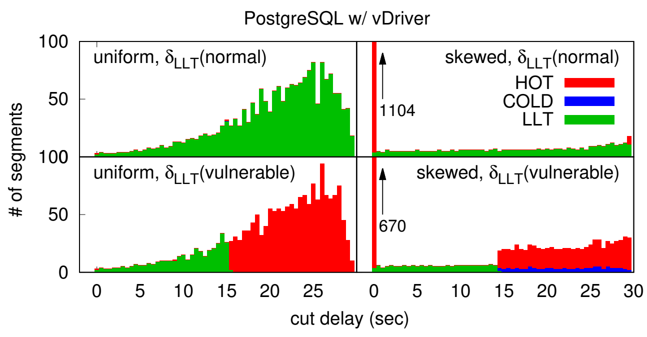


Figure 13: Effect of classification errors (stacked).

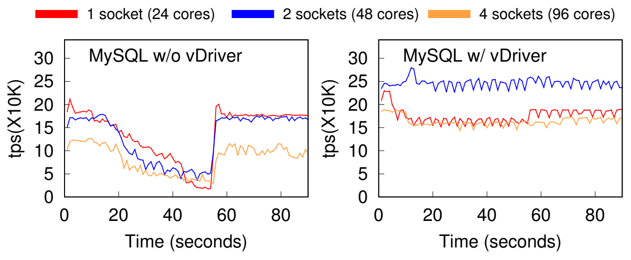


Figure 14: Throughput behavior on multicores.

portion of versions pass our two pruning stages and finally reach version space (i.e., 'HOT no prune'). Nonetheless, such durable segments will soon be cleaned by vCutter through the segment cleaning, thus having a short lifetime.

When long-lived transactions are present, the breakdown of pruning is different because versions in VC_{LLT} appear as non-reclaimable ones (i.e., 'LLT no prune'). As we use more skewed workloads, the portion of VC_{LLT} is diminishing while the region for non-reclaimable hot versions grows. It may seem counterintuitive at first glance, but the reason is because some versions generated by frequently updated records turn out to be VC_{LLT} but they were not classed as such at first due to the classification error. These versions are put in the wrong basket indeed, thus increasing the portion of 'HOT no prune'. Our explanation is supported by the observation that the portion of 'HOT no prune' grows while that of 'LLT no prune' shrinks, and the sum of the two is maintained as the same. We will discuss the effect of classification errors in detail below.

4.3.3 Effect of Classification Error. As we briefly discussed in §4.3.2, versions can be stored in wrong segments during the vulnerability window. To see the effects of classification errors in our version classifier, we conduct experiments on PostgreSQL with an LLT, by using two δ_{LLT} values — normal and very large ones — under uniform and highly-skewed workloads. To see the effect of classification errors clearly, each table is populated with 10,000 record items. To emulate highly-skewed workloads, we set 'rand-zipfian-exp' to 1.2 for this experiment; very few are updated enormously. We measure the *cut delay*, which represents the elapsed time for a given segment until being purged from stable store, for those segments who survived all pruning steps and have been hardened to the version store. If there is no classification error, then the timely removal of VC_{HOT} segments is what one should expect (i.e., very short cut delay).

Figure 13 shows the results using stacked graph. With normal δ_{LLT} and a uniform distribution, a majority of versions required for an LLT are correctly classified as VC_{LLT} but the cut delay for VC_{LLT} segments is spread over the lifetime of an LLT. This is due to the behavior of the uniform distribution that with low probability, some records are seldom updated and their versions are relocated from data pages to vBuffer later than others and accordingly, their segments are hardened later than other segments. In that case, their segments have short remaining time until LLTs commit, while other segments hardened earlier have longer time to be cut, that would lead cut delays to spread widely. Note that VC_{HOT} versions are mostly pruned by our version pruning ('1st prune') under uniform distribution as shown in Figure 12. Hence there is no VC_{HOT} segment in this case. Meanwhile, under a highly-skewed distribution, we can observe the effect of classification errors such that there are a few VC_{HOT} segments left uncleaned long time, as depicted at time 30s, because it contains VC_{LLT} versions that were not classed as such at first. This is due to the behavior of the skewed distribution that a small number of records are updated very frequently and their versions are classified as VC_{HOT} since transactions are yet to be identified as LLTs.

However, if we use very large δ_{LLT} (~15s), then the spectrum of the segment distribution resembles prior results with normal δ_{LLT} but differs as to whether or not VC_{HOT} segments are cleaned after the long vulnerability window. Cleaning of some VC_{HOT} segments containing VC_{LLT} versions are likely to be suspended until LLTs are all committed, irrespective of workload distributions. Under highly-skewed workloads, we observe two prominent phenomena; (1) VC_{LLT} segments are rarely flushed to version space since a majority of VC_{LLT} versions remain unmodified and reside in data pages as either $v^{r,0}$ or $v^{r,1}$; (2) some VC_{LLT} versions having long update intervals are classed as VC_{COLD} and remain uncleaned until the end of LLTs. The first phenomenon is indeed the obvious strength of the Oviraptor versioning but, nonetheless, the penalty for the classification errors is the suspension of segment cleaning due to misclassified VC_{LLT} versions in VC_{HOT} segments, thus increasing version space. Although our classifier is vulnerable to the classification errors as such, our segmented version store architecture can minimize the damage by limiting the bad impact on a small number of contaminated segments.

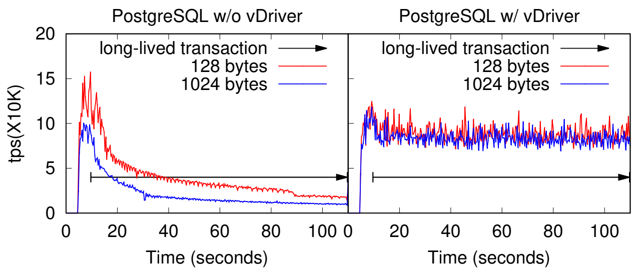


Figure 15: Effect of record size.

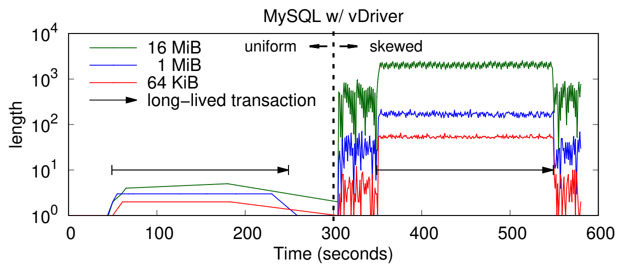


Figure 16: Effect of segment size.

4.3.4 *Throughput Behavior on Multicores.*

To see whether the negative performance effects caused by long-lived transactions are limited to high-end multicore servers, we conduct experiments that evaluate the throughput behavior of the systems with LLTs being present, under special distribution workloads, varying the number of cores. To limit available CPU cores, we use the `taskset` command. Since one CPU socket features 24 cores, we permit all `mysqld` server threads to run on any of 24, 48 and 96 cores, respectively. Since this evaluation is to see the behavior, not complete throughput trajectory, four long-lived transactions join around 10s and end 55s. As shown in Figure 14, the vanilla engine suffers the same performance collapse due to the longer version chain created by LLTs, while vDriver escapes performance matters but shows somewhat awkward throughput behavior in the sense that throughput on two sockets is higher than that on four sockets. Profiling revealed that it is because cache invalidation messages are generated much larger on four sockets than experiments on two sockets. From the results, we confirm that performance issues will arise irrespective of the count of cores.

4.3.5 *Effect of Record Size.*

As discussed in §2, record size can affect performance, especially when database systems use in-row versioning for implementing their MVCC. To further investigate the effect of record size on transaction throughput under skewed workloads (`rand-zipfian-exp` value = 1.1) in the presence of LLTs, we conduct experiments on PostgreSQL by setting record size to 128 and 1024 bytes. Figure 15 shows the throughput of the systems when an LLT begins at about 10s. Since PostgreSQL with vDriver only keeps one version in data pages, the throughput does not change noticeably although record size gets bigger. However, the vanilla system exhibits worse throughput as we increase the record size since page splits occur more frequently with the larger size than the smaller one.

4.3.6 *Effect of Segment Size.*

As analyzed in §4.3.1, we claim that the segment size can affect the maximum chain length among all version chains. A large segment has a benefit in terms of the segment management while a small segment effectively suppresses the max chain length. To see the effect of the segment size, we vary the segment size from 64 KiB to 16 MiB with the same configurations used in Figure 10. As shown in Figure 16, the max chain length is well controlled under uniform workloads, while there are significant differences under skewed workloads. Especially, when the segment size is 16 MiB, the max chain length is greater than 10^3 that will surely affect latch duration. Although vCutter checks candidate version segments continuously for segment cleaning, unfilled segments cannot be cleaned since it lacks information about v_{min} and v_{max}. If all versions for a frequently updated record are stored in the same segment that is yet full, then such versions would account for increasing the chain length, and the maximum valid chain becomes longer until the segment is cleaned.

5 CONCLUSION

For the past decades, we have witnessed the remarkable evolution of database technologies, in particular multi-version concurrency control, and there is agreed consensus in the database community on the basis of MVCC. In this regard, record versioning has been essential to ensuring correct, efficient concurrency control on versioned data. Despite the immense effort, contemporary database systems are still vulnerable to the age-old issue: long-lived transactions. As hardware vendors are provisioning more computing cores, the side-effects of long-lived transactions will intensify its impact on all aspects of performance metrics. In the present work, we have addressed this matter by first formalizing theoretical frameworks in our theorem and proposing version management architecture, called vDriver. For version management, we have proposed a new versioning principle, Oviraptor, and realized it through version classification and version pruning based on our theorem. Experimental evaluation demonstrated that vDriver generally resolves many important issues arising when long-lived transactions are present. Moreover, the proactive version pruning and the segment pruning/cleaning aligned with version classification in vDriver are expected to provide valuable insight to in-memory engines in designing version garbage collection where relieving memory pressure is of utmost importance.

REFERENCES

[1] Oracle Corporation and/or its affiliates. 2019. MySQL 8.0 Reference Manual: 15.6.3.4 Undo Tablespaces. https://dev.mysql.com/doc/refman/8.0/en/innodb-undo-tablespaces.html.

[2] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mitturand Venkataramanappa. 2018. Constant Time Recovery in Azure SQL Database. *PVLDB* 12, 12 (Oct. 2018), 2143–2154. https://doi.org/10.14778/3352063.3352131

[3] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.* 1, 2 (June 1976), 97–137. https://doi.org/10.1145/320455.320457

[4] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/223784.223785

[5] 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.

[6] 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 (SIGMOD '13)*. ACM, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[7] Jim Gray. 1988. Readings in Database Systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter The Transaction Concept: Virtues and Limitations, 140–150. http://dl.acm.org/citation.cfm?id=48751.48761

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

[9] Maurice Herlihy. 1993. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Trans. Program. Lang. Syst.* 15, 5 (November 1993), 745–770. https://doi.org/10.1145/161468.161469

[10] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[11] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *PVLDB* 3, 1-2 (Sept. 2010), 681–692. https://doi.org/10.14778/1920841.1920928

[12] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2011. Scalability of write-ahead logging on multicore and multisocket hardware. *The VLDB Journal* 21, 2 (2011), 239–263. https://doi.org/10.1007/s00778-011-0260-8

[13] Hyungsoo Jung, Hyuck Han, Alan D. Fekete, Gernot Heiser, and Heon Y. Yeom. 2013. A Scalable Lock Manager for Multicores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. 73–84.

[14] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. 2017. Scalable Database Logging for Multicores. *PVLDB* 11, 2 (Oct. 2017), 135–148. https://doi.org/10.14778/3149193.3149195

[15] Jongbin Kim, Hyeongwon Jang, Seohui Son, Hyuck Han, Sooyong Kang, and Hyungsoo Jung. 2019. Border-Collie: A Wait-free, Read-optimal Algorithm for Database Logging on Multicore Hardware. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, New York, NY, USA, 723–740. https://doi.org/10.1145/3299869.3300071

[16] Per-Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-performance Concurrency Control Mechanisms for Main-memory Databases. *PVLDB* 5, 4 (Dec. 2011), 298–309. https://doi.org/10.14778/2095686.2095689

[17] 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 (SIGMOD '16)*. ACM, New York, NY, USA, 1307–1318. https://doi.org/10.1145/2882903.2903734

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

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

[20] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. 2011. PLP: Page Latch-free Shared-everything OLTP. *PVLDB* 4, 10 (July 2011), 610–621. https://doi.org/10.14778/2021017.2021019

[21] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1583–1598. https://doi.org/10.1145/2882903.2882958

[22] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2015. VLL: A Lock Manager Redesign for Main Memory Database Systems. *The VLDB Journal* 24, 5 (Oct. 2015), 681–705. https://doi.org/10.1007/s00778-014-0377-7

[23] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. 1976. The Design and Implementation of INGRES. *ACM Trans. Database Syst.* 1, 3 (Sept. 1976), 189–222. https://doi.org/10.1145/320473.320476

[24] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD '86)*. ACM, New York, NY, USA, 340–355. https://doi.org/10.1145/16894.16888

[25] 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 (SOSP '13)*. ACM, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[26] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging Through Emerging Non-volatile Memory. *PVLDB* 7, 10 (June 2014), 865–876. https://doi.org/10.14778/2732951.2732960

[27] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *PVLDB* 10, 2 (Oct. 2016), 49–60. https://doi.org/10.14778/3015274.3015276

[28] Gerhard Weikum and Gottfried Vossen. 2002. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.