Long-lived Transactions Made Less Harmful

[Technical Report]

Jongbin Kim
Hanyang University

Hyunsoo Cho
Hanyang University

Kihwang Kim
Hanyang University

Jaeseon Yu
Hanyang University

Sooyong Kang
Hanyang University

Hyungsoo Jung
Hanyang University

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. 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 old versions required for LLTs should suspend garbage collection. 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: *Single In-row Remaining Off-row (SIRO) versioning*. 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

MVCC; record versioning; long-lived transactions

1 INTRODUCTION

The concept of object versioning dates back to the late 1970s right after major relational database systems (i.e., Ingres [28] and System R [5]) were released, and multi-version concurrency control emerged fully formed since then to provide fast reads to users while allowing concurrent writes on conflicting data. Data anomalies (e.g., *write skew* [6]) still exist that stymie serializable execution, but lower isolation with MVCC has made inroads into the database community; it has proven its efficiency in arbitrating concurrent access to databases and become the de-facto scheme to use in designing high-performance database systems. Transaction engines nowadays, therefore, set lower isolation levels, which use MVCC, as defaults to enhance performance on multicores.

Essential for MVCC to function efficiently is the strict condition requiring that databases must clean old, unnecessary versions promptly. 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 either halt due to a shortage of storage space or suffer severe performance degradation, and customers still have to pay for the uncleaned version space in cloud databases, although systems exhibit 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 the 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*. In-row versioning stores old versions of a record in the same data page where the record item exists. 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 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 for searching a proper version. However, the clear merit recompenses off-row versioning for the long lookup time by ensuring that index modifications would never occur

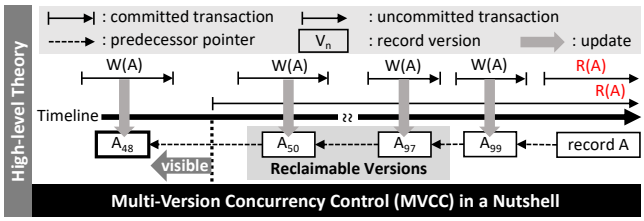


Figure 1: A long-lived transaction in MVCC.

even if there is an upsurge of new versions since versions and locator information are all stored in separate space, so that main index structures remain unaffected.

One may consider a pure hybrid between two designs, such as an LSM-tree, to cope with their shortcomings, but we all discern the risk of naive trials that may bring all drawbacks, without settling the principal problem of version cleaning in the presence of long-lived transactions (LLTs), and it is the primary technical matter we aim to address.

1.2 The Main Challenge

Essential to notice after analyzing the root causes of the performance problems is the fact that all the problems can disappear through the timely removal of unnecessary versions lacking in the two versioning designs. Version space grows when long-lived transactions obstruct the purging of reclaimable versions since some live versions required for LLTs must be kept in version space.* Unraveling this 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 live transactions need them.*

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 address the pressing concern of the systems in the presence of LLTs, while retaining transaction throughput with reduced space consumption.

*While version space can also grow when databases generate new versions more than a version cleaner can vacuum, it can be easily overcome by adjusting the number of version cleaners.

1.3 Our Contributions

Since Jim Gray raised significant concerns regarding LLTs [10], researchers and database vendors have devised methods to ameliorate undesirable side-effects. However, our evaluation indicates that the latest versions of PostgreSQL and MySQL, of which all have faithfully implemented in-row and off-row designs, respectively, are still susceptible to LLTs.

This work is based on the premise that there are noteworthy gains by unifying two popular-but-seemingly different ideas. At the core of our approach is a new versioning principle called ***S**ingle **I**n-row **R**emaining **O**ff-row versioning* (SIRO-VERSIONING), which keeps the first old version together with a record and moves other versions to off-row space. This 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 show a *complete version pruning theorem* that gives necessary and sufficient conditions for identifying dead versions at any given time. We then lay down the *ground classification rules* under which versions with different lifetimes are separately classed so that cleaning dead versions would seldom be suspended by live versions required for LLTs. Essential for SIRO-VERSIONING to be feasible are concrete designs that provide fast access to versions through version buffering with lightweight in-memory index structures augmented.

By putting all these ideas together, we propose VDRIVER, an extension to existing engines to manage record versions. 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. To the best of our knowledge, this is the first attempt to design and implement a practical hybrid of in-row and off-row versioning in open-source database engines.

2 MOTIVATION AND RELATED WORK

2.1 Motivation

Depending on how to organize versions and locators, the pros and cons of each versioning design are easily recognized. As shown on the bottom of Figure 2, PostgreSQL-12.0, which uses in-row versioning, stores versions in heap pages with an augmented search structure 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 repeatedly occur, depending on the pace. *Repeated page splits would severely impinge on a degree of concurrency.* As opposed to this, off-row versioning makes index structures intact, regardless of a massive influx of versions. MySQL-8.0 with the InnoDB storage engine uses undo space for this purpose.

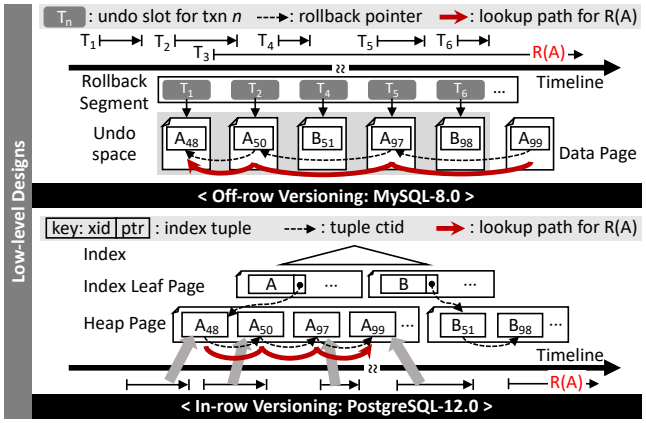
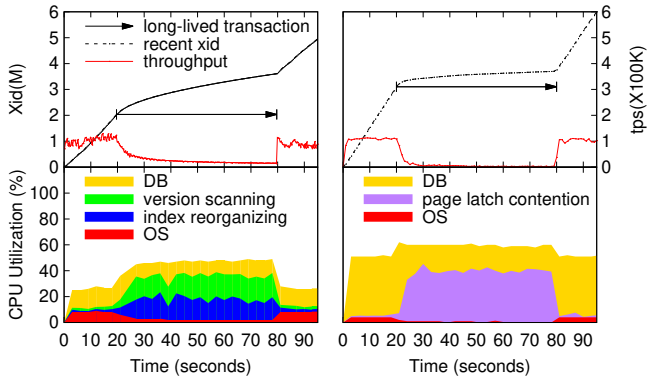


Figure 2: Effects of LLTs in two versioning designs.



(a) PostgreSQL-12.0 **(b) MySQL-8.0 (InnoDB)**

Figure 3: Effects of a long-lived transaction.

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 present work is motivated by the undesirable performance phenomena observed in two full-fledged database systems when we conduct the experimental evaluation with short transactions running OLTP workloads as well as LLTs that read data items. Figure 3 shows the preliminary results. Common to both engines is the behavior that as a version chain gets long, transaction throughput is sharply collapsing. Code analysis discovered that *latch duration on a page increases as a version chain grows in MySQL, whereas traversing an elongated version chain from the oldest version takes time in PostgreSQL*. The increased latch duration exacerbates contention on pages, and the long traversal time is fatal to short transactions. Which symptom arises depends on how databases implement version lookup; searching a version from the oldest, like PostgreSQL, would severely affect short transactions, in contrast, searching a version from the newest while holding a page latch, like MySQL, would worsen latch contention. In conclusion, an extended version chain is a prime culprit liable for performance problems.

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 overflows quickly 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 a 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-lived 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 arisen owing to its off-row versioning strategy. This advantage is favored to databases when updates occur on bigger size rows.

2.2 Related Work

Version cleaning in MVCC databases bears a strong resemblance to conventional garbage collection in that they uphold the same underlying principle; versions are formally classified as garbage if live transactions would never request them onward. Embodying the ideal policy, however, is demanding so that in practice the oldest live transaction is often chosen as a reasonable proxy for establishing the death line; this never violates the invariant—visible version must be reachable—but it is susceptible to LLTs due to the long suspension of version cleaning caused by the long-lived old transactions, leading versions to pile up rapidly.

In-memory databases treat version cleaning as garbage collection problems since all versions reside in memory, and there have been many proposals [7–9, 18, 20, 21, 23] addressing the issue by effectively collecting obsolete versions with different granularity levels. The main goal of prior work is to devise finer-grained approaches closer to the ideal solution that aims to purge all the versions identified as garbage at any

Table 1: Notations.

Notations	Description
$\mathbb{R}$	a set of records
$\mathbb{V}$	a set of committed old versions of all records
$v^{r,i}$	i^{th} (old) version of a record $r \in \mathbb{R}$ ($r = v^{r,0}$)
$v_s^{r,i}$, $v_e^{r,i}$	visibility start and end timestamps of $v^{r,i}$
t_b^k, t_c^k	begin and commit timestamps of a transaction T_k
s_r^k	a snapshot read of a record r to a transaction T_k
$\mathbb{S}_k$	a snapshot of a transaction T_k
$\mathbb{T}$	a set of live transactions
$\mathbb{Z}_\mathbb{T}$	a complete set of dead zones constructed from $\mathbb{T}$
$\mathbb{T}_k^{rv}$	the read-view of T_k at the time of t_b^k
C^T	the current time

moment. SAP HANA [20] proposed an interval garbage collector that periodically scans entire version chains for identifying and reclaiming obsolete versions in the background. Inheriting a similar interval-based method with enhanced scalability, Steam [7], deployed in Hyper [23], delegates garbage collection to the foreground transactions accessing the chain using a piggybacking approach, which resembles the hybrid approach in PostgreSQL.

Despite the notable improvements made to in-memory MVCC databases, efficient version cleaning has been a challenging matter to disk-based MVCC databases. The main hurdle is the I/O cost in that scanning version chains eagerly for reclaiming garbage versions inevitably incurs expensive disk I/O in disk-based systems, whereas such scanning is relatively a free lunch to in-memory counterparts. Moreover, delegating the version reclamation may severely hurt the transaction latency in disk-based systems; although a dedicate worker is used, it must compete with foreground transactions on shared resources, such as buffer pool and latches, thus obstructing transaction processing. Due to this difficulty, the concerned systems—MySQL and PostgreSQL—still use the age-old, coarse-grained criterion: using the oldest active transaction as a garbage-collection boundary.

3 VERSION DRIVER ARCHITECTURE

This section describes the primary design rationale for the proposed architecture and the core design principles used in addressing significant technical challenges.

3.1 Theoretical Foundations

We first establish the theoretical frameworks required for understanding the present work, and Table 1 shows the least set of notations. A database is a set of records ($\mathbb{R}$) and their old versions ($\mathbb{V}$), *all committed*. 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 ($v_s^{r,i}$) and end ($v_e^{r,i}$)

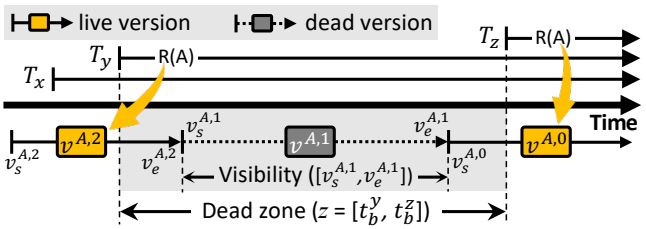


Figure 4: Visibility and dead zone.

timestamps. Assuming that a transaction T_k creates a record version $v^{r,i}$, the version is visible from the transaction's commit time onwards (i.e., $v_s^{r,i} = t_c^k$), whereas $v_e^{r,i}$ is the start timestamp of a newer version $v^{r,i-1}$ (i.e., $v_e^{r,i} = v_s^{r,i-1}$) since $v_s^{r,i-1}$ formally ends the visibility of $v^{r,i}$. Note that a subtle case is the end timestamp of a record r ($= v^{r,0}$) whose visibility interval is half-open (i.e., $v_e^{r,0} = \infty$). Hekaton [9] denotes this range as *valid time*. Following the definition of SI [6, 33], 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 does not belong to 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. We formally define a *dead zone*, a time range $z=[z_s, z_e]$ whose start (z_s) and end (z_e) times are set to the begin timestamps of two consecutive transactions. Then, any version whose visibility falls within a dead zone is literally a dead version (see Figure 4). We, therefore, identify and reclaim every existing dead version by accurately defining all dead zones. To this end, we define a *complete* set of dead zones 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 definition, a correctness condition for identifying dead versions with $\mathbb{Z}_{\mathbb{T}}$ can be formalized as a *version pruning theorem* as below, and the version pruning theorem 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: ←)** 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: →) 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 connote a critical, overlooked rule stating that *if systems have no live transactions, then one can empty the entire $\mathbb{V}$ 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 the begin timestamp of an updater instead of a commit timestamp in a record version. If a version embeds the begin timestamps of two transactions, one who created the version and the other one who created the next version, corresponding to $v_s^{r,i}$ and $v_e^{r,i}$, 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 when it begins (i.e., at t_b^k). Given a transaction T_k with its read-view $\mathbb{T}_k^{rv}$, we can say that a transaction $T_{r,i}$, who created a version $v^{r,i}$, has already committed at the time when T_k began if the following condition holds: the begin timestamp of $T_{r,i}$ is earlier than that of T_k but is not in T_k's read-view — formally, $(v_s^{r,i} < t_b^k) \wedge (v_s^{r,i} \notin \mathbb{T}_k^{rv})$. Then, the conditions for a version $v^{r,i}$ to be a snapshot read to T_k are: 1) $T_{r,i}$ has already committed and 2) the next version $v^{r,i-1}$ of $v^{r,i}$ is either uncommitted if it began (i.e., the begin timestamp of $T_{r,i-1}$ is in T_k's read-view) or yet to be created at the time when T_k began (i.e., the begin timestamp of $T_{r,i-1}$ is later than the largest timestamp in T_k's read-view) — formally, $((v_s^{r,i} < t_b^k) \wedge (v_s^{r,i} \notin \mathbb{T}_k^{rv})) \wedge ((v_e^{r,i} \in \mathbb{T}_k^{rv}) \vee (v_e^{r,i} > \max_{\forall t_b^m \in \mathbb{T}_k^{rv}} t_b^m))$. The rules described above are almost identical to what MySQL and PostgreSQL use now. Since begin timestamps of $T_{r,i}$ and $T_{r,i-1}$, which correspond to $v_s^{r,i}$ and $v_e^{r,i}$ in our taxonomy, are embedded in $v^{r,i}$, we can accurately identify dead versions by rewriting Theorem 3.5 as follows: *Given $\mathbb{Z}_{\mathbb{T}}$ for $\mathbb{T}$ at any moment, a version can be pruned if and only if it is not a snapshot read to all live transactions subject to the above conditions* — formally, $\nexists z^{k+1} = [t_b^k, t_b^{k+1}] \in \mathbb{Z}_{\mathbb{T}} : ((v_s^{r,i} < t_b^k) \wedge (v_s^{r,i} \notin \mathbb{T}_k^{rv})) \wedge ((v_e^{r,i} \in \mathbb{T}_k^{rv}) \vee (v_e^{r,i} > \max_{\forall t_b^m \in \mathbb{T}_k^{rv}} t_b^m))$, 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. When a transaction whose begin timestamp adjoins two consecutive dead zones commits, combining them is a necessary condition for the current dead zones to be complete. 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 primary objective of record versioning is to deliver prompt service to user queries arriving with different visibility conditions while maintaining the essential set of versions. Considering the given designs in all aspects, we argue that in-row versioning outclasses the off-row method in terms of fast recovery, whereas off-row versioning is superior to its counterpart in making index structures unaffected by a burst of incoming record versions. This whole argument provides us with the compelling design rationale for the essential versioning principle of vDriver: *single version in-row and remaining versions off-row.* Keeping only a single version alongside a record is to retain the strength of the fast recovery, and moving others to separate version storage is to be robust to massive incoming versions.

We inspect versions and collect garbage ones when versions physically change its status from in-row to off-row, thus gaining efficiency by minimizing unnecessary I/O. Challenges left unaddressed are design details to fill the gap between ideal concepts and practical implementations.

Architectural overview. The crux of our proposal for addressing the challenges is vDriver[†], of which the central role is version management responsible for three tasks: storing versions by the SIRO-versioning rule, pruning obsolete ones by our pruning theorem, serving reads as fast as it can. For the first task, vDriver hardens every version either in database pages or in off-row version storage with different lifetimes; databases ensure the durable atomicity of in-row versions for correct recovery, while off-row versions should never survive database reboots and crashes since none of the newly created transactions request them after the restart. For the second one, vDriver examines the eligibility of versions and prunes them if possible, all done whenever versions relocate from in-row pages to off-row space. Lastly, vDriver manages a buffer layer to overcome the speed gap between memory and version storage, and its role is the same as the database buffer.

Version buffering for fast reads. Due to our design rationale, vDriver resembles off-row versioning in 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 SIRO-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 consists 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. Our version buffer layer 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* [11, 12]; we use many of well-proven concurrent data structures and programming techniques.

[†]Code is available: https://github.com/hyu-scslab/vDriver.

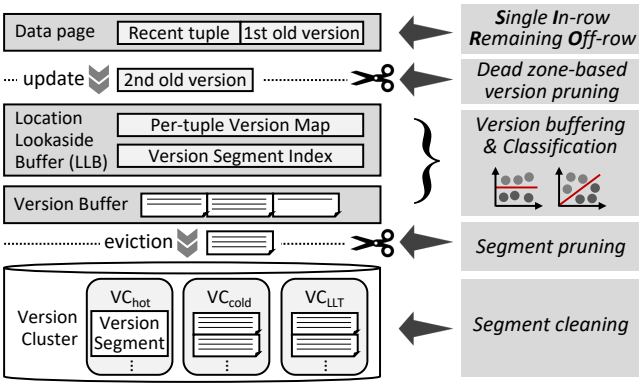


Figure 5: vDriver with core design principles.

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, we classify record versions into different categories by pre-defined rules, and a concept of version classification is devised and used in conjunction with Theorem 3.5. The central 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 hardened in version store, or it must *clean* 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 versions relocate from in-row pages to off-row version space in vBuffer, and we denote this as dead zone-based *version pruning*. Live versions survived the first pruning reside in proper in-memory version segments through version classification. The next pruning chance then comes when vDriver flushes the version segments 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 exercises *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.

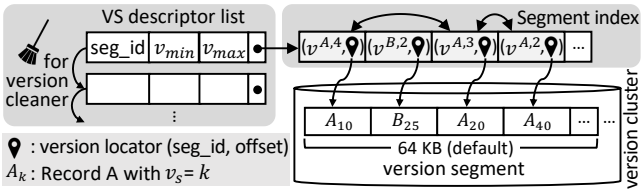
Summarizing what we have described here, Figure 5 shows the layered architecture of vDriver and the core design principles used in pursuing the primary virtue of vDriver; *live versions serve read requests while they remain in memory for fast responses, and we prune dead versions as early as we can to minimize space overhead.* Starting from vSorter, we present the critical components of vDriver.

3.3 vSorter

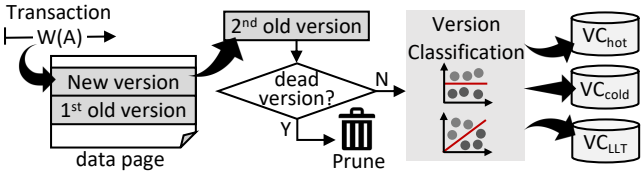
vSorter is a sub-component of vDriver responsible for managing the proper placement of record versions obeying the SIRO-versioning rule, and it manages vBuffer and performs version classification and pruning. Implicit here is that since segments in vBuffer and their segment indexes in LLB are managed 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 SIRO-versioning. We describe the dead zone-based version pruning and classification last.

Data structures. Version space is composed of three version clusters, each of which stores versions grouped by our 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. One can view the link structure as a 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 we can reclaim the current segment, where v_{min}=min$_{\forall v^{r,i} \in VS}$ $v_s^{r,i}$ and v_{max}=max$_{\forall v^{r,i} \in VS}$ $v_e^{r,i}$. Figure 6(a) illustrates the overall design of vSorter.

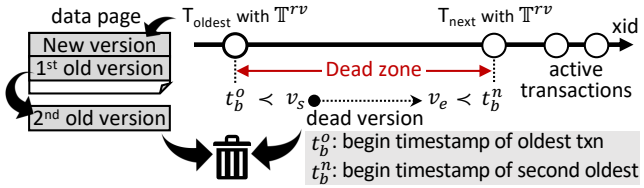
SIRO-versioning. The first step of SIRO-versioning (Figure 6(b)) commences when a transaction updates a record r (i.e., $v^{r,0}$). If $v^{r,0}$ already holds $v^{r,1}$ beside itself, then the current record becomes a new 1^{st} version $v^{r,0\rightarrow1}$, and $v^{r,\perp\rightarrow0}$ occupies the placeholder for $v^{r,1\rightarrow2}$ by pushing it to off-row space. We use a toggle bit to indicate which one is the current record, instead of physically swapping two. Before $v^{r,1\rightarrow2}$ moves to off-row version space, we execute *version pruning* to prune $v^{r,1\rightarrow2}$ by Theorem 3.5. If $v^{r,1\rightarrow2}$ survives this version pruning, then it becomes $v^{r,2}$ and is stored in one of the version clusters through version classification. For version



(a) Overall architecture of vSorter



(b) SIRO-versioning



(c) Dead zone-based version pruning

Figure 6: vSorter design.

classification, vSorter considers three factors: start and end timestamps of a version, and a clear sign of indicating that a version belongs to any snapshots of LLTs. In the present work, we classify versions as three classes: (i) hot versions (VC_{hot}), (ii) cold versions (VC_{cold}), and (iii) versions belonging to the 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.

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 augmented by a *read-view*. As depicted in Figure 6(c), vSorter checks whether any of the given dead zones bounds ($v_s^{r,i}$, $v_e^{r,i}$) of a version $v^{r,i}$. 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 we execute the pruning to identify and erase a dead version whenever record versions relocate.

Our version pruning has a trade-off between pruning accuracy and performance; the more precise 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 an array (PostgreSQL), one can construct $\mathbb{Z}_{\mathbb{T}}$ using the existing shared data structures. Besides, 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, $\mathbb{T}^{rv}$ is 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; instead, we update it 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 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; §5 will provide further discussions on this claim with evaluation results.

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 live versions cannot suspend the cleaning of dead versions; thus, batch cleaning of versions in the same class is feasible. The primary design rationale for having three classes is owing to 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 we can define more types of version classes 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 dead version is a daunting task to the current architecture. Even if the two engines may use garbage collection methods [9, 20, 23, 30] proposed for in-memory database engines, it entails a significant amount of I/O activities to make version chains correct concerning the representation invariant since purging individual dead versions in stable storage inevitably incurs disk I/Os.

[‡]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.

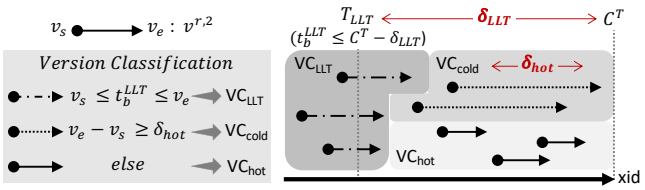


Figure 7: Version classification in action.

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 we define a long-lived transaction 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, we inspect a version (i.e., $v^{r,2}$) to see if 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 specific time window can be a good proxy for this purpose. Figure 7 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 due in large to workload characteristics, such as access patterns and transaction length. The time window during which misclassification occurs is called a *vulnerability window*, and it is the interval between the begin timestamp and the time a transaction becomes an LLT. The penalty for putting live versions in a wrong basket is the suspension of cleaning dead versions in the concerned class, thus leading to increased space consumption. But, our segment-based architecture can minimize the damage to version space through segment-based version cleaning, and in-depth analysis for this is discussed in §5.

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 the off-row version store by the SIRO-versioning 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

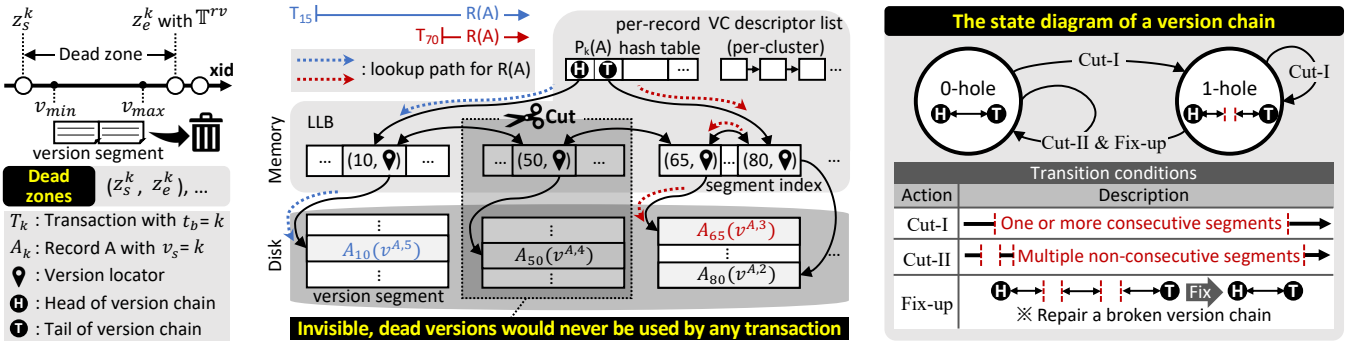


Figure 8: Version segment cleaning of vCutter.

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 8 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 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 8, a version in a dead segment contains locator links to previous and next versions that can be stored in other version segments. If the 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 8, 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 8. 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

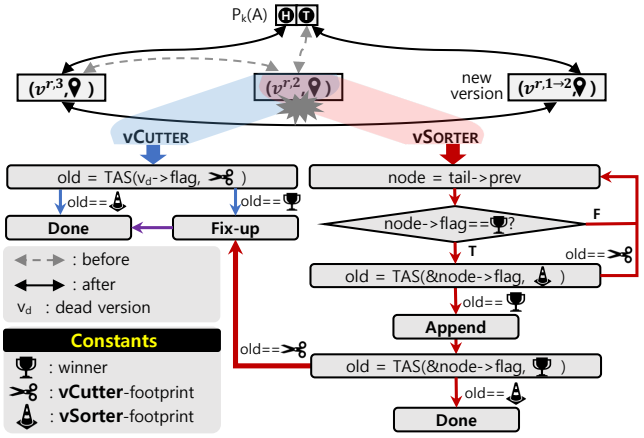


Figure 9: Collaborative version cleaning.

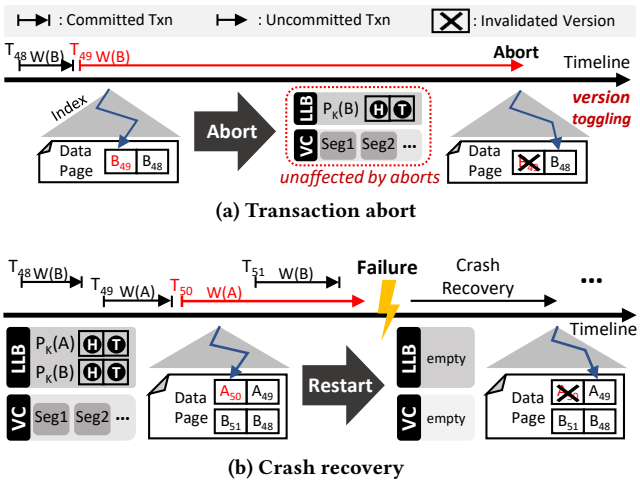


Figure 10: Undo recovery in vDriver.

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 9 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 $v^{r,1\to2}$ 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 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 9; 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 in vDriver

One of the claims proposed in PostgreSQL [29] 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 the first old version is stored in the same data page, thus making undo recovery simple. The same rationale holds for Azure SQL Server [4] and vDriver. By the SIRO-versioning principle, $v^{r,\perp\to0}$ and $v^{r,0\to1}$ are placed in the same data page and 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. Note that undo recovery affects vDriver differently, depending on the case. The completion of transaction abort toggles versions, with version segments and LLB unaffected by undo actions, but crash recovery renders all the versions in version segments obsolete because none of newly created transactions would request them, thus emptying everything in vDriver.

Figure 10 illustrates how undo operations are performed in two different events; (1) when a transaction aborts and (2) when a system crashes. Figure 10(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 10(b) describes the situation where crash 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 $v^{A,1\to0}$.

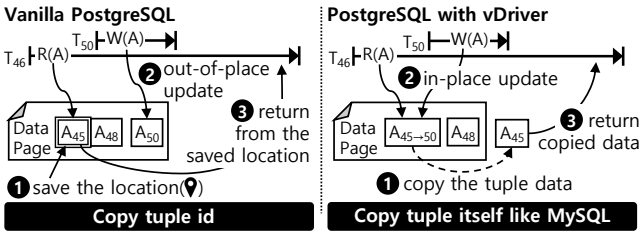


Figure 11: SIRO-versioning in PostgreSQL.

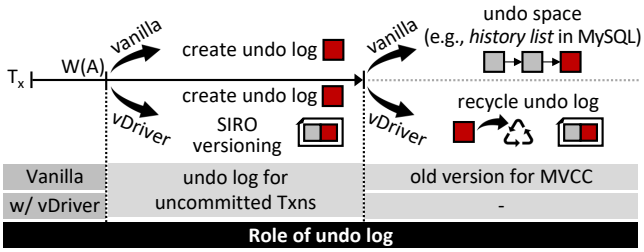


Figure 12: Temporal redundancy of undo logs.

4 PRACTICAL ASPECTS OF VDRIVER

We acknowledge that database designers are hesitant to espouse the notion of being lighthouse customers of new technologies, of which the practical effect is only validated through proof-of-concept systems. Nonetheless, we seek to make vDriver deployable for the full range of running systems by articulating the common abstraction required for embracing vDriver, as well as concrete implementation examples for developers to facilitate a better understanding. To this end, we first discuss the general common abstraction to cope with each nontrivial design concern and then provide specific implementations made to embody the concept. At the core of our effort here is the concern on how to smoothly integrate vDriver into well-established systems. Although we design vDriver to act as a standalone version manager readily pluggable to existing systems, adopting vDriver would require considerable engineering effort in two places: (i) changing the physical page layout to enforce SIRO-versioning and (ii) managing the redundancy of the existing undo space. Other critical components of databases remain intact. For addressing these concerns, we elaborate on general designs and concrete implementations below.

4.1 SIRO-versioning in Action

Abstraction. By the SIRO-versioning principle, vDriver should lay the first version together with a record, and this may entail substantial changes in the internal page layout of affected MVCC systems; in particular, all changes are made through in-place updates so that systems already using in-place updates—MySQL, SQL Server, and Oracle—have no difficulty in enforcing SIRO-versioning. However, systems employing in-row versioning with the out-of-place update policy, such as PostgreSQL, may sacrifice some benefits to apply SIRO-versioning on its data page. As shown in Figure 11, reads on a tuple in vanilla PostgreSQL return the tuple through copying the locator of the requested version, instead of copying the entire tuple, assuming that another update would never overwrite the tuple. As a general design for addressing this matter, any systems enforcing the out-of-place update policy should use the method of copying a requested tuple, which is widely used in other databases relying on the in-place update policy.

Implementation. The implementation of the abstract design first reserves one extra space for each record in a data page and uses it as a placeholder. Especially, for PostgreSQL relying on out-of-place updates, we change the in-row versioning page layout to the SIRO-versioning layout with in-place updates imposed. As noted above, any operations coming from the upper layer to request a version in PostgreSQL should return a copy of a tuple, instead of tuple id. Of course, we should hold a page latch when copying the tuple. In contrast, since MySQL uses in-place updates, implementing SIRO-versioning in MySQL only requires space reservation, which is much more straightforward than PostgreSQL. Besides, common to both engines is the modification to redirect version lookup to vBuffer when requested versions are absent in data pages.

4.2 Space Management for Undo Logs

Abstraction. Databases relying on the 'steal'/'no-force' policies must keep undo logs to roll back the effects of uncommitted transactions when needed. Undo logs, once the owner transaction commits, should be an integral part of the change history of a record, and this history is the authentic set of versions essential for MVCC. Undo logs, therefore, have dual roles as such but may have concerns in managing undo space when integrating with vDriver that also maintains separate version storage. The fundamental issue here is whether or not to retain customized undo logs for each engine. The answer to this question depends on how systems internally identify a loser transaction. Identifying losers is typically done by either directly checking transaction status in commit logs (e.g., '`pg_xact`' in PostgreSQL [27] and *aborted transaction map* in Azure SQL Database [4]) or indirectly scanning undo headers (e.g., '`rollback segment`' in MySQL [2] and Oracle [3]). The former type would have no difficulty in adopting vDriver, while the latter type must keep customized undo logs until transactions commit and then recycle logs since the undo log header should be used as an indicator for a loser until committed (see Figure 12.)

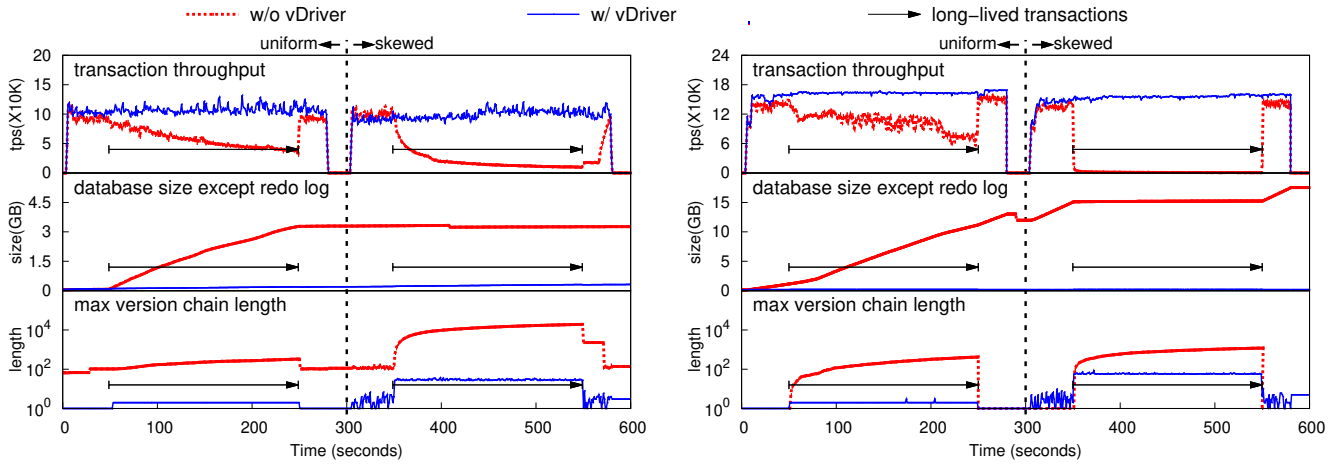


Figure 13: Throughput and version space overhead. (Left: PostgreSQL, Right: MySQL)

Implementation. The implementation of the abstract design only affects substantially to MySQL since it internally uses undo log headers in rollback segments to resurrect uncommitted transactions when it restarts. Following the general design, we keep MySQL's undo logs until a transaction commits and recycle them upon commit by relinking them to '`update_undo_cached`', without appending it to the global '`history_list`', which otherwise would have been done in stock MySQL for *off-row versioning*. Although the undo log and an old version coexist redundantly, it is temporal redundancy (i.e., until the transaction commits) and so has negligible overhead. Another issue is the deletion of ghost records in MySQL; stock MySQL removes ghosts when purging delete undo logs by using the tracking information. Since we eliminate the redundancy of MySQL's undo space, we change the role of MySQL purging in that it fulfills the job of deleting ghosts and modifying the affected B-tree index in the background by scanning the entire user table, like PostgreSQL. On the contrary, PostgreSQL uses a commit log (i.e., '`pg_xact`') to identify uncommitted transactions when judging the eligibility of versions, there is no particular issue worth mentioning for handling undo space.

4.3 Engine Specific Implementations

Since MySQL and PostgreSQL use a begin timestamp and a concept of a read-view for deciding the visibility of a version, their record format contains a transaction identifier. For read-views, the InnoDB storage 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`. 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 for managing read-views. Therefore, we introduce a global shared read-views of all the live transactions and periodically update $\mathbb{Z}_{\mathbb{T}}$ by constructing it from live transactions.

5 EXPERIMENTAL EVALUATION

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

5.1 Evaluation Setup

For evaluation, we conduct experiments 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.

5.2 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.

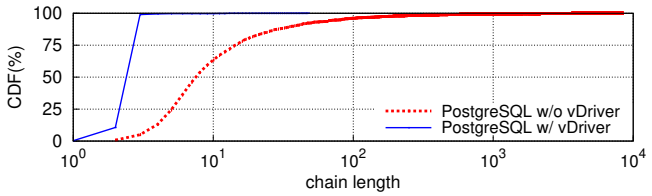


Figure 14: The CDF of version chain length.

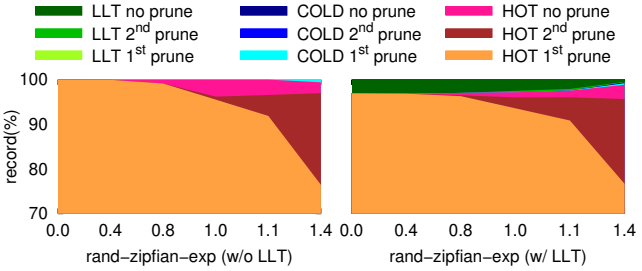


Figure 15: Pruning effects of vDriver on MySQL.

5.2.1 Throughput and Space Overhead. This section evaluates the performance of the systems over two critical 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 13 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 poorly impacted due to long version chains, especially under the skewed workloads, while long latch duration caused by read-only 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. First, MySQL with vDriver outperforms the vanilla system mainly because of the elimination of the redundancy of MySQL's undo space and the relevant work for managing it, which internally uses a few giant latches. Second, MySQL abruptly truncates undo space during the test due to the new undo space management [1] cutting one of '`undo_001`' and '`undo_002`' files alternately. Although we have seen prior proposals [9, 13–17, 19, 22, 24–26, 31, 32] that have addressed performance issues caused by latch contention in databases, both engines may still 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 allows 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 the 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 14 shows the CDF of version chain length under highly-skewed workloads in the presence of an LLT, and most records retain short chain length (up to the maximum of 35) in vDriver while the vanilla system shows a 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.

5.2.2 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). We run the experiment for 90s and measure performance metrics, and the breakdown pattern remains the same regardless of time. 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 read randomly selected records continuously until the end. 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 15 shows the results, and 1^{st} `prune` and 2^{nd} `prune` indicate that versions are pruned in version and segment pruning, respectively. A majority of versions have been

[§]In MySQL, we assign the sysbench worker threads to two sockets by using `taskset` to prevent unexpected performance fluctuations due to thread migration in NUMA architectures.

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 $v^{r,1\rightarrow 2}$ versions can be pruned when being relocated from data pages to vBuffer, regardless of the presence of LLTs. The result has a significant implication that our version pruning with the SIRO-versioning 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 the survived versions are pruned in the second pruning stage by the segment pruning, and only a small 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 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 that 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`'. The observation that the portion of '`HOT no prune`' grows while that of '`LLT no prune`' shrinks, supports our claim, and the sum of the two remains the same. We will discuss the effect of classification errors in detail below.

5.2.3 Effect of Classification Error. As we briefly discussed in §5.2.2, vDriver can store versions 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—legitimate and huge ones—under uniform and highly-skewed workloads. We populate each table with 10,000 records. To emulate skewed workloads, we set '`rand-zipfian-exp`' to 1.2, and we measure the *cut delay*, which represents the elapsed time for a given segment until being purged from stable storage. 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 16 shows the results using a stacked graph. With normal δ_{LLT} and uniform distribution, a majority of versions

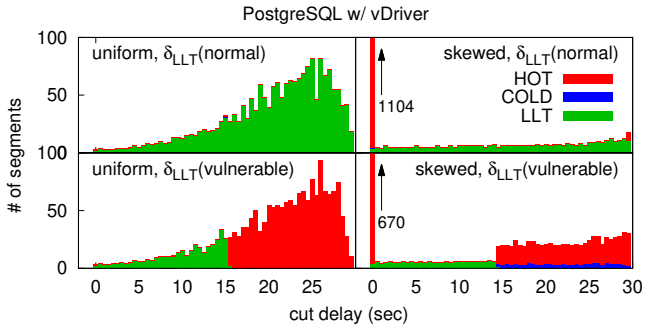


Figure 16: Effect of classification errors (stacked).

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 a longer time to be cut, which would lead cut delays to spread widely. Note that VC_{HOT} versions are mostly pruned by our version pruning ('1^{st} `prune`') under uniform distribution as shown in Figure 15. 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 mainly due to the characteristic of the skewed distribution; a small number of records are modified 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 apparent strength of the SIRO-versioning but the penalty for the classification errors is the suspension of segment cleaning due to misclassified VC_{LLT} versions in VC_{HOT} segments,

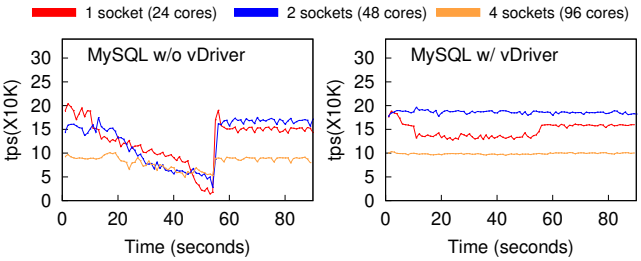


Figure 17: Throughput behavior on multicores.

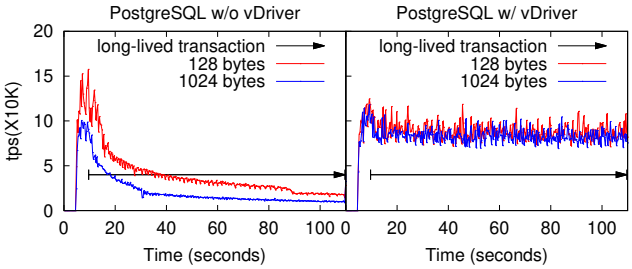


Figure 18: Effect of record size.

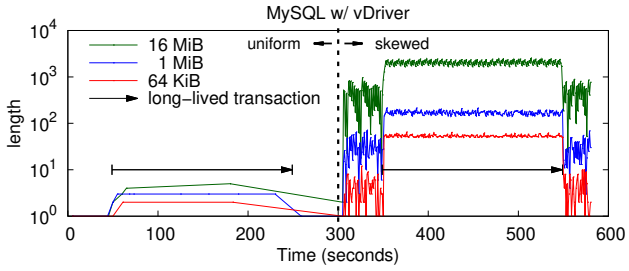


Figure 19: Effect of segment size.

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 harmful impact on a small number of contaminated segments.

5.2.4 Throughput Behavior on Multicores. To see whether the adverse performance effects caused by LLTs are limited to high-end multicore servers, we conduct experiments that evaluate the throughput behavior of the systems with LLTs being present, under specialized distribution, 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 the entire throughput trajectory, four long-lived transactions join around 10s and end 55s. As shown in Figure 17, the vanilla engine suffers the same performance collapse due to the longer version chain created by LLTs, while vDriver escapes performance matters.

5.2.5 Effect of Record Size. As discussed in §2, record size can affect performance, mainly 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 (i.e., `rand-zipfian-exp` = 1.1) in the presence of LLTs, we conduct experiments on PostgreSQL by setting record size to 128 and 1024 bytes. Figure 18 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 becomes enormous. 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.

5.2.6 Effect of Segment Size. As analyzed in §5.2.1, 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 13. As shown in Figure 19, 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 surely affects 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.

6 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, SIRO-versioning, 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] 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.

[3] 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.

[4] 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

[5] 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

[6] 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

[7] 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

[8] Kalen Delaney. 2016. SQL Server In-Memory OLTP Internals for SQL Server 2016. https://download.microsoft.com/download/8/3/6/8360731A-A27C-4684-BC88-FC7B5849A133/SQL_Server_2016_In_Memory_OLTP_White_Paper.pdf

[9] 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

[10] 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

[11] 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

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

[13] 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

[14] 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

[15] 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.

[16] 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

[17] 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

[18] 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 (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1675–1687. https://doi.org/10.1145/2882903.2882905

[19] 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

[20] 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

[21] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015)*. https://www.microsoft.com/en-us/research/publication/high-performance-transactions-in-deuteronomy/

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] PostgreSQL repository. 2019. README on The Transaction System. https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/backend/access/transam/README;hb=8548ddc61b5858b6466e69f66a6b1a7ea9daef06#l334.

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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.