Highly Scalable SQL Azure MVCC Version Cleaner

Jithendra Gandikota
Microsoft
Austin, USA
jigandik@microsoft.com

Prashanth Purnananda
Microsoft
Redmond, USA
praspu@microsoft.com

Chaitanya Sreenivas Ravella
Microsoft
Redmond, USA
ravellas@microsoft.com

Adrian-Leonard Radu
Microsoft
Redmond, USA
aradu@microsoft.com

Hanuma Kodavalla
Microsoft
Redmond, USA
hanumak@microsoft.com

Max Garton
Microsoft
Kansas City, USA
maxgarton@microsoft.com

Allison Ohara
Microsoft
Corona, USA
allisonohara@microsoft.com

Wayne Chen
Microsoft
Redmond, USA
waync@microsoft.com

Raghavendra TK
Microsoft
Redmond, USA
raghavendra.tk@gmail.com

ABSTRACT

SQL Azure utilizes Multi-Version Concurrency Control (MVCC) to support several features such as Read Committed Snapshot Isolation (RCSI), Snapshot Isolation (SI), Constant Time Recovery, and Metadata Versioning. MVCC maintains multiple versions of data rows to support these features. A background version cleaner lazily reclaims older versions that are not required by any of these features. The prior version cleaner algorithms suffered from several drawbacks which prevented it from cleaning older versions quickly leading to space bloat. Under conflicting workloads, the cleaner failed to acquire locks, delaying cleanup. Cleanup also slowed down significantly while handling very large databases and multi-database instances. When a consumer of some versions lagged, it held up many versions in the system from being cleaned up. This paper outlines the innovative solutions that SQL Azure implements to address these drawbacks, such as transaction-level version tracking, statement level cleanup, dynamic multi-threaded cleaner, and user-assisted cleanup. The paper also presents the impact of deploying these changes across millions of SQL databases in Azure.

PVLDB Reference Format:

Jithendra Gandikota, Prashanth Purnananda, Chaitanya Sreenivas Ravella, Adrian-Leonard Radu, Hanuma Kodavalla, Max Garton, Allison Ohara, Wayne Chen and Raghavendra TK. Highly Scalable SQL Azure MVCC Version Cleaner. PVLDB, 19(12): 3916-3927, 2026.
doi: 10.14778/3827998.3828005

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi: 10.14778/3827998.3828005

1 INTRODUCTION

Multi-Version Concurrency Control (MVCC) [12] has become crucial for modern relational databases because it enables high concurrency without heavy blocking. By maintaining multiple versions of data rows, MVCC allows readers and writers to operate without blocking each other, which is ideal for today's highly concurrent workloads. This approach boosts performance and scalability in multi-user systems: for example, PostgreSQL [22], Oracle [13], and MySQL Inno DB [5] all rely on MVCC to handle large number of simultaneous transactions efficiently. Each transaction can see a consistent snapshot of the data without interfering with others, greatly reducing lock contention and improving throughput.

Like other database management systems, SQL Azure also supports MVCC concurrency control to prevent readers and writers from blocking each other. RCSI (Read Committed Snapshot Isolation) is the default isolation level in SQL Azure. Features like Constant Time Recovery [19] and Metadata Versioning [21] use MVCC.

However, MVCC's strength—keeping multiple data versions—comes with a cost. As transactions update and delete rows, obsolete versions accumulate in the database. When these old versions are no longer needed, they occupy space. Without effective stale version cleanup (VC), an MVCC-based system will suffer from space bloat. In short, version cleanup is critical in MVCC systems to reclaim space and maintain performance before obsolete versions overwhelm storage.

While SQL Azure supports MVCC to enable high concurrency, it is important to distinguish between two separate cleanup mechanisms: ghost cleanup and version cleanup. Ghost cleanup refers to the removal of only deleted rows, whereas version cleanup is specific to MVCC systems and deals with cleaning obsolete row versions generated by concurrent update transactions. Ghost cleanup existed historically and can be integrated into version cleanup as one background cleaner. Currently, all the improvements discussed in this paper apply exclusively to version cleanup. Ghost cleanup operates

independently and is not impacted by the enhancements described here.

Version cleanup in SQL Azure must be adaptive and scalable with workload intensity, cooperative with foreground activity, and sufficiently granular to avoid long-tail delays in reclaiming versions. These principles guide the design of our techniques and enable robust and efficient cleanup across diverse workloads and large-scale production environments.

The version cleanup task needs to identify old versions that are no longer needed and remove them without impacting ongoing workload. This is essential for maintaining consistent performance and scalability. Efficient version cleaner that frees outdated rows promptly keeps database size in check, ensuring that new transactions have room. In practice, MVCC version cleanup contributes directly to system throughput and stability by removing the stale versions and relieving pressure on storage and memory.

The previous implementation of version cleaner was single threaded. Cleanup slowed down significantly while handling very large databases and multi database instances. Another problem was in the presence of a conflicting workload; it failed to acquire necessary locks and delayed cleanup. This caused cleanup to fall behind. Also, when the undoing of transactions lagged, cleanup was held.

Addressing these challenges requires rethinking version cleanup beyond isolated fixes. While the improvements presented in this paper resolve specific limitations of the previous design, they are rooted in the broader design principles outlined above. This paper describes the key challenges observed in production telemetry from large-scale Azure SQL deployments and the corresponding enhancements to the version cleanup mechanism that have been developed over time.

The paper is structured as follows: Section 2 talks about the implementation of MVCC in SQL Azure, how versions are stored, and how version cleanup was implemented previously. Section 3 describes the challenges involved with the baseline version cleanup implementation, and the improvements made to address these. Section 4 describes our experience of deploying these improvements to millions of production databases. Section 5 mentions future improvements planned, and Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2 SQL AZURE BACKGROUND

This section describes how a transaction interacts with the MVCC system throughout its lifecycle in SQL Azure. At a high level, a transaction begins by establishing a snapshot, performs updates that generate new versions, and eventually commits or aborts, after which background cleanup determines when older versions can be reclaimed. The following subsections describe the key data structures and mechanisms involved in this lifecycle, providing a step-by-step view of how versions are created, accessed, and cleaned up in the previous implementation of the version cleaner.

A glossary of commonly used abbreviations is summarized in the Appendix.

2.1 MVCC in SQL Azure

SQL Azure employs Multi-Version Concurrency Control (MVCC) row versioning and timestamp-based mechanism. Under MVCC, readers do not block writers, and writers do not block readers – an essential property for high concurrency. Instead of using locks for reads, SQL Azure provides each transaction with a consistent snapshot of the data by keeping multiple versions of rows. This section provides definitions of a few data structures used in the implementation. This also explains how inserts, updates, deletes, and reads are implemented with MVCC.

2.1.1 Transaction Timestamp (XTS)

SQL Azure maintains a global monotonically increasing counter called XTS which is reset to 1 on server restart.. A versioned read begins by creating a snapshot, which captures the system visibility state for the read. Depending on the isolation level, a snapshot may be created at the start of a transaction or statement. Each snapshot maintains a read XTS, obtained by incrementing the current XTS, and a MinXTS, which is the minimum XTS of all active transactions when the snapshot is created.

2.1.2 Transaction Id (XdesID)

Transaction ID (XdesID) or XdesId is another monotonically increasing number. Write transactions are assigned a XdesID at the time of their first log generation. This uniquely identifies a write transaction.

2.1.3 Commit Sequence Number (CSN)

When a write transaction is committed, it is assigned a commit timestamp, commit XTS, by using the current XTS value in the system. This is also called the Commit Sequence Number (CSN). The same applies to aborted transactions.

2.1.4 XdesHistory

SQL Azure stores the mapping of every committed transaction's XdesID to its CSN through a data structure called XdesHistory.

2.1.5 Aborted Transaction Map

An Aborted Transaction Map (ATM) is a data structure containing all aborted transactions. When a transaction aborts, it will add its XdesID to the ATM.

2.1.6 MinUsefulXTS and MinUsefulStatementXTS

MinUsefulXTS—the minimum of MinXTS of all snapshots in the system—is a server-wide value maintained to indicate the minimum useful XTS in the system. This is used as a marker for version cleanup. Any versions generated by transaction whose CSN is less than MinUsefulXTS are eligible for cleanup. MinUsefulStatementXTS—the minimum of read XTS of all snapshots the system—is also a server-wide value maintained to indicate the minimum useful XTS across all the statements within active RCSI, SI transactions in the system. This is the new timestamp that was added to aggressive cleanup. As transactions start and complete, snapshots are continuously added and removed. These two values are updated on demand; however, updates do not occur more than once per defined threshold interval.

2.1.7 Inserts with MVCC

When a transaction inserts a row, a base version of the row is created. In addition, the transaction stamps its XdesID on all the rows that it inserts. With inserts, there is no old copy to maintain.

2.1.8 Updates with MVCC

When a transaction modifies a row, the database does not overwrite the original data in place immediately. Instead, it creates a new version of the row with the changes and preserves the old version for any transactions that started earlier and may still need to see the prior state. This is implemented by adding a small pointer to the modified row. This pointer is 8 bytes and stores the page and slot Id of the previous version of that row. The older data is copied into a special version storage area. In addition, the transaction stamps its XdesID, 6 bytes, on all the rows that it modifies. This is illustrated in Figure 1.

2.1.9 Deletes with MVCC

When a transaction deletes a row, row data is not physically deleted immediately. Row is just deleted logically by marking it as a ghost. This is done by pushing the old version of the row to version store and a ghost stub is created in place. In addition, the transaction stamps its XdesID on all the rows that it deletes.

2.1.10 Reads with MVCC

Versioned read traverses the version chain till it finds the version whose transaction's CSN is less than read XTS. It is the most recent version generated by the latest transaction that was committed before the read snapshot.

2.2 Persisted Version Store

In SQL Azure's implementation, old row versions are stored in a dedicated version store that is part of the database. Initially, SQL Azure's row versioning stored these in the TempDB system database. However, with the introduction of Constant Time Recovery or Accelerated Database Recovery, which is ON by default in SQL Azure, the version store has moved into the user database itself as a Persistent Version Store (PVS). In practice, this means that when a row is updated, the latest copy is maintained on the page, and the previous version is kept either with the latest row on the same page (in-row version) or kept off row in a separate version store (off-row version). In the case of in-row versioning, the earlier version is embedded within the current version as a diff. This approach can improve access efficiency by avoiding additional page reads. Further details on in-row versioning can be found in prior work on constant-time recovery [19]. These versions are leveraged to allow user transactions to access the appropriate snapshot version of the row.

2.2.1 In-row Versioning

The in-row version store is an optimization that enables storing the previous version of a row directly alongside the current version within the same data page. While the row primarily holds the latest version, it also includes information necessary to reconstruct the earlier version. Since updates typically affect only a few columns, the system can store just the difference (diff) between versions. This approach enhances read performance by removing the need to fetch extra pages for reconstructing older versions, while also lowering page IO and logging overhead during writes. However, generating in- row versions for every update can increase row size and negatively impact performance. This is particularly problematic for B-Trees as it can lead to page splits—an operation that is costly. Page splits not only increase overhead but also degrade the index structure, resulting in more expensive access patterns in the future. To address that, the size of in-row versions is capped both in terms of the diff size and size of the row [19].

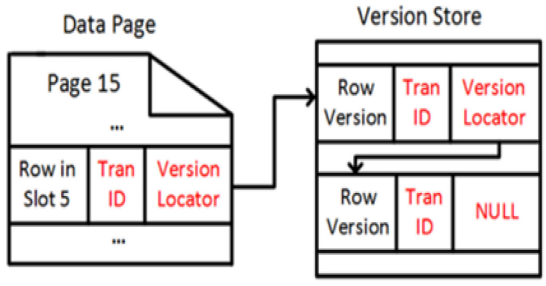

Figure 1: Example of MVCC row version chain.

2.2.2 Off-row Versioning

The off-row version store handles versions that exceed the limits for in-row storage. It is implemented as an internal, index-free table where all version lookups are performed using the version's physical locator (Page ID, Slot ID). Each database maintains a single off-row PVS table that stores versions for all user tables. Every version is represented as a separate row in this table, containing metadata columns and a generic binary column that holds the full version content, independent of the schema of the originating user table. Creating a version corresponds to inserting a row into the off-row PVS table, while retrieving a version involves reading it via its physical locator.

In SQL Azure, both in-row and off-row versions are maintained. For simplicity, the paper refers to off-row versions throughout but same applies to in-row versions as well.

2.3 Version Cleanup Implementation

This section describes the prior version cleanup mechanism in SQL Azure. This baseline serves as the foundation for understanding the challenges discussed in Section 3 and the optimizations introduced to address them.

Every data modification with MVCC is versioned, updating the row in-place and pushing the previous version into the version store or in-row. These row versions serve two purposes: 1. To remove partial updates of the rolled back transactions and replace them with the committed. 2. To serve concurrent read queries (MVCC). When a query accesses a row, it first checks the state (active, committed or aborted) of the transaction that generated the latest version, based on the Transaction Id, and decides whether this version is visible. If the transaction is active or has been committed, visibility depends on the query isolation level and read XTS, but if the transaction is aborted, this version is not visible, and the query traverses the version chain to identify the version that belongs to a committed transaction and is visible. If these versions are not interesting for rollback or MVCC, they become stale and are eligible for cleanup.

SQL Azure uses a background task to perform version cleanup continuously. This task wakes up periodically (roughly every few minutes) to sweep the database for expired versions. The cleanup is automatic.

This background cleanup task wakes up periodically and iterates over all databases one at a time and performs the following steps.

1) Create a copy of the aborted transactions in the ATM before starting cleanup.
2) Identify data pages with versions by scanning Page Free Space (explained in detail in Sec 3.1) metadata pages.
3) Logically revert (or Undo) updates performed by aborted transactions.
4) Remove aborted transactions copied in step 1 from the ATM once all their updates have been logically reverted.
5) Clean up off-row versions from PVS once they are no longer needed for rollback or MVCC.
6) This process is repeated for the next database on the server.

The task of performing Logical Revert is different from the one that cleans up off-row PVS versions and, therefore, is described separately.

2.3.1 Identifying Versioned Pages

Before the sweep begins, the ATM is copied. This snapshot allows the cleaner to precisely identify which aborted transactions should be removed at the end of the sweep. By doing so, the cleaner avoids incorrectly removing transactions that abort after the sweep starts, since their versions are not guaranteed to be cleaned up in this cycle.

To efficiently identify data pages requiring cleanup and avoid scanning the entire database, SQL Azure introduced additional metadata for each data page indicating whether it contains rows that need to be logically reverted or in-row versions that need to be removed. This metadata is stored in already existing special system pages that SQL Azure maintains and are known as Page Free Space (PFS) pages since they are mainly used for tracking whether each page has space available. Each PFS page tracks metadata for 8088 data pages with one byte for each page. Before updating a data page, its corresponding bit in PFS is set indicating that this page has versions [29][28]. To avoid repeatedly accessing the PFS pages to modify the version bit, the same state is also maintained on the data page itself, so that the PFS page is accessed only if the state for this page is changing. The version cleaner scans the PFS pages sequentially for pages marked as needing cleanup. This will find data pages in user tables needing cleanup. For each such page, it performs logical revert or in-row cleanup.

2.3.2 Logical Revert and In-row Version Cleanup

Logical revert is the process of restoring the latest committed version of a row that was last modified by an aborted transaction back to its main location in the data page. This allows queries to access the correct data directly.

During cleanup, the version cleaner takes intent exclusive locks [30] on the page and table to which page belongs. Using the XdesID stamped on the row and cross-referencing it with the ATM, version cleaner inspects every row to determine if it was modified by an aborted transaction. If a match is found, a logical revert operation is performed. In addition, version cleaner removes any in-row versions that belong to earlier committed transactions and are no longer needed for MVCC. If all versions were successfully removed from the page, its corresponding version bit in PFS is unset.

After successfully processing all the pages containing versions, the aborted transactions copied at the beginning are removed from the ATM.

Version cleaner is designed to be non-intrusive and to avoid blocking user transactions. In the presence of conflicting workload which prevents version cleaner from acquiring necessary table or page level locks, version cleaner will often skip pages it cannot lock immediately and defer cleanup to a later cycle. However, in sustained high-concurrency scenarios, this deferment can lead to a backlog of stale versions accumulating, increasing storage consumption, and potentially degrading performance.

Additionally, concurrent updates that trigger row movements such as page splits—may cause the cleaner to overlook rows that have shifted to pages it has already processed. This can lead the version cleaner to assume all aborted versions have been removed, resulting in the incorrect removal from ATM of all the previously copied aborted transactions while leaving some aborted versions behind in the database. To maintain correctness, the system tracks a value called MinXdesIdInMove, representing the minimum XdesId stamped on all rows moved during the sweep. At the end of a successful cleanup cycle, version cleaner only removes aborted transactions with XdesIds lower than MinXdesIdInMove in the copied ATM. While this is conservative, it ensures that none of the aborted transactions are removed whose versions got missed because of row movement.

2.3.3 Off-row Version Cleanup

Off-row cleanup is only responsible for deleting stale versions stored in off-row PVS store and freeing up space in the database. As described earlier, the off-row PVS stores versions as rows in an internal table. Versions generated by transactions (for all user tables in the database) get inserted there. This simplifies the cleanup task because the pages containing off-row versions are effectively the pages that belong to this table. Additionally, this table is optimized for inserts; it is designed to be append-only, allocating new pages to store newer versions. Hence, the cleanup task can track the status of the generated versions at the page level, as each page gets filled with versions, and deallocate complete pages.

To achieve that, Accelerated Database Recovery uses an in-memory sorted list called PVSPageTrackerList that tracks status of each PVS page containing off-row versions based on their Page Id. Since the versions are generated by different transactions, their lifetime is also different and requires us to maintain information about all transactions that inserted versions in each page. To minimize the information stored per page, SQL Azure leverages the fact that Transaction IDs generated are monotonically increasing and only stores the highest Transaction Id, which indicates the newest transaction

that inserted a version in this page. This is used to evaluate whether a page contains versions that might be still required for MVCC or logical revert (if they still contain committed data for rows that were updated by aborted transactions). As mentioned in Sec 2.1.6, SQL Azure maintains MinUsefulXTS, which represents the earliest timestamp before which all versions in the system are guaranteed to be unused by any active transaction. In addition, it also maintains the lowest transaction Id (OldestActiveXdesId) across all active transactions and the lowest transaction Id (OldestAbortedXdesId) across all aborted transactions in the system. For cleanup, a combination of these three is used as watermarks. If a page's maximum Transaction Id PageMaxXdesId is lower than this OldestActiveXdesId and OldestAbortedXdesId and if the CSN of this PageMaxXdesId is less than MinUsefulXTS, the page can be deallocated.

When the page is full, the entry in the PVSPageTrackerList indicates that the page is now eligible for cleanup. The cleanup thread periodically scans all pages in the list and deallocates those that are eligible for cleanup after checking the PageMaxXdesId and CSN as mentioned before. Even though this PVSPageTrackerList is only in-memory, in case of failures this can be easily constructed during recovery.

The off-row PVS pages are easily discoverable by identifying all pages that belong to the PVS internal table; an operation SQL Azure supports natively [4]. Rather than recalculating the exact PageMaxXdesId for each page during recovery, the system conservatively uses the highest Transaction ID identified during the analysis phase [11]. Even though this is too conservative, as the database starts accepting new transactions and logical revert cleanup removes earlier aborted transactions, all these pages will quickly become eligible to be deallocated. This ensures that the off-row cleanup task can continue to reclaim space effectively, even in the event of failures or restarts.

While the cleanup task can be time-consuming, it is crucial that version cleaner successfully logically reverts all versioned records associated with copied aborted transactions in a single pass. If it fails to do so, it can hinder the removal of aborted transactions from the ATM, slowing the advancement of the OldestAbortedXdesId. This delay can obstruct the cleanup of PVS pages, ultimately resulting in database space bloating.

3 VERSION CLEANUP CHALLENGES AND IMPROVEMENTS

The previous implementation of version cleanup mentioned in section 2.3 faced several challenges. Each subsection below describes a specific challenge, and the corresponding solutions adopted to address it. The challenges described in this section were identified by observing production telemetry from the largest consumers of version store in SQL Azure databases. They are slow cleanup sweeps in large databases, interference from hot update workloads, and delayed watermark advancement caused by aborted or long-running transactions. While MVCC systems exhibit a wide range of theoretical challenges, our work focuses on addressing the dominant contributors to PVS growth observed in real customer workloads in Azure. This section describes these challenges and corresponding optimizations to address those challenges.

3.1 Scalability

In SQL Azure, version cleanup was typically handled by a single-threaded background task that scans and removes obsolete row versions. While this approach may work well for isolated databases, it can become a bottleneck in the following cases.

- Elastic Pools [2]—shared resource environments where multiple databases compete for CPU and I/O within a single SQL Azure instance. A single-thread cleaner may struggle to keep up with the volume of versioned data across pooled databases. Given that elastic pools represent a significant portion of SQL Azure, the need for parallelized cleanup becomes critical.
- Even in standalone databases, the single-threaded version cleanup mechanism can become a bottleneck—particularly in very large databases. This issue is especially evident with SQL Azure Hyperscale [20], where database size can exceed 100TB. Although the cleaner only scans a subset of data pages, the large size of the database proved overwhelming for a single thread. Under high activity, a full sweep of the database sometimes took over a week to complete.
- In our production environment, it is not uncommon to encounter workloads where the same row is repeatedly updated within a single transaction. This leads to the creation of long version chains. In such scenarios, version cleaner must traverse numerous versions to reach the committed one, which can be slow especially when storage is remote, as is often the case in Azure. This latency can trigger additional problems, such as latch timeouts.
- Another issue with slow version cleanup is that if the current sweep is delayed, the work for the next sweep can grow significantly, especially when the workload consistently generates a high number of versions. This creates a serious risk of unbounded growth in the PVS size, potentially impacting performance and resource stability. To meet the increasing demand for Hyperscale and maintain system stability, scaling version cleanup operation became essential.

To tackle the overall slowness in version cleaner, the following improvements are made.

3.1.1 Read Ahead

From section 2.3.1, version cleaner scans each PFS page to identify all data pages with the version bit set. Since each PFS page tracks metadata for 8088 data pages, version cleaner can optimize its I/O by first collecting the list of all pages marked with the version bit and issuing a read-ahead for those pages. This reduces I/O wait times by preloading the relevant pages into memory, allowing version cleaner to process them more efficiently without incurring disk latency for each individual page read. These read-aheads are done in a round robin manner across the files to not overwhelm one single file.

3.1.2 Parallel Cleanup

Introducing dynamic parallelism into the version cleanup significantly alleviated the bottlenecks observed in large SQL Azure Elastic Pool and Hyperscale Tiers. However, enabling

parallel cleanup introduces resource consumption, which requires careful control mechanisms. In our design, parallelism is applied adaptively. This parallelization kicks in only when there are enough resources in the system and avoids unnecessary resource consumption for smaller PVS footprints, which can be handled by the default single-threaded cleanup mechanism. This is done not only across the databases but also within the database. When the PVS size is above a certain threshold, cleanup sweep is parallelized dynamically within the database. Once it goes below the threshold, the subsequent cleanup sweeps are not parallelized.

With dynamic parallelism, the system was able to reduce the duration of full cleanup cycles from several days to just a few hours. This improvement not only accelerated the advancement of the cleanup watermark but also prevented excessive growth of the PVS, which had previously threatened resource stability. Parallel cleanup allowed the system to keep pace with version generation. To prevent excessive CPU usage and disruption to user workloads during parallelized version cleanup, a CPU cap is enforced. Additionally, parallelizing only based on the need and available resources ensured that smaller resource instances were not strained by cleanup overhead.

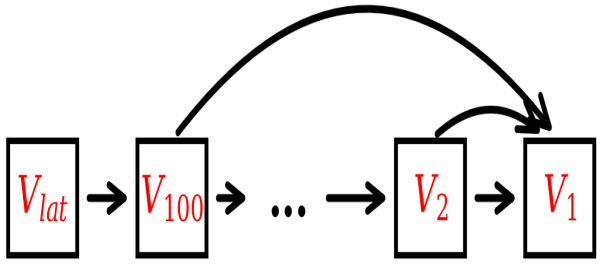

Figure 2: Version chain linkage for traversal optimization.

3.1.3 Version Traversal Optimization for Logical Revert

In the case of the same row getting updated repeatedly within a single transaction, the intermediate versions need to be retained to support partial rollback via savepoints. In cases where transactions that generated multiple versions aborts later, the version cleaner must traverse long version chains to get to the committed version. This pattern is not uncommon in SQL Azure. To handle this, when a version is pushed off-row, the pointer to the row's first committed version is stored along with the existing pointer to the previous version in the chain. While this shortcut-based traversal significantly improves efficiency, introducing such a pointer naively would increase the storage footprint of each version record. To avoid this overhead, we encode the additional pointer by reusing an existing metadata field that previously stored redundant information, ensuring no increase in per-version storage cost. Figure 2 illustrates a version chain of length 100. With this optimization, to get to the committed version instead of traversing the entire version chain reading 100 versions, version cleaner first reads latest version (v_{lat}) on the data page. From v_{lat}, it fetches the next version in the chain which is present off-row, v_{100}. Any off-row version should have a pointer to the first committed version of the record. From v_{100} it directly goes to v_1. This made sure that the length of the version chain traversed is not more than 3.

3.2 Contention Due To Hot Workload

For the cases where the workload is hot—characterized by frequent updates—version cleaner struggled to keep pace with the rate of version generation. This is especially problematic when the cleaner attempts to acquire locks on tables and pages that are constantly in use. The cleaner is designed to run with low priority and to avoid competing with user transactions for locks. Rather than waiting, it will often skip pages it cannot lock immediately and defer cleanup to a later cycle. However, in sustained high-concurrency scenarios, this deferment can lead to a backlog of stale versions accumulating in the PVS, increasing storage consumption, and potentially degrading performance. The inability to acquire locks consistently under heavy load prevents the cleaner from reclaiming space efficiently. Additionally, concurrent updates could trigger page splits in B-Trees. This could cause aborted rows to get moved to PFS ranges already visited by the cleaner. As mentioned in Sec 2.3.3 earlier, it is crucial that version cleaner successfully eliminates all versioned records associated with copied aborted transactions in a single pass. Otherwise, the removal of aborted transactions from the ATM gets delayed. To tackle the interference from the hot workload the following improvements were made.

3.2.1 Side List

All the pages which version cleaner skipped because it failed to acquire locks are tracked separately in a dedicated side list. Additionally, operations such as page splits can move aborted rows to pages that the cleaner has already processed, potentially causing the version cleaner to overlook them. All such pages to which the aborted rows are moved are also added to the side list. In this case, the transaction splitting the page tracks these pages in the side list. After version cleaner completes its sweep of the entire database, the main cleanup thread revisits the pages in this side list, retrying cleanup in a loop for a configurable time. In scenarios where the workload remains persistently conflicting, locks may still not be acquired despite multiple attempts. To ensure robustness under extreme contention, the side list itself is bounded. If the number of skipped pages exceeds a predefined threshold, the system avoids adding further pages to the side list. In such cases, the sweep is invalidated and deferred to the next iteration. However, such cases are rare. This approach of retrying these pages in a loop significantly improves version cleaner's chances of cleaning these pages effectively in the presence of a conflicting workload and supports achieving cleanup in a single pass.

3.2.2 User-Assisted Cleanup

Additionally, there are a few cases in production where cleanup continued to fail even when using the side list approach, primarily because the cleaner was unable to acquire the necessary table-level lock. This typically occurred under sustained workloads that hold conflicting locks for extended periods. To address this, stale version cleanup is delegated to user transactions that hold the conflicting lock. Version cleaner maintains a separate hash map to track pages where cleanup could not be performed due to failure in acquiring table-level locks. The key in this hash map is the lock resource ID, mapped

to a list of page IDs that were skipped by the cleaner for that lock resource. At the end of each statement, user transaction checks whether the lock resource ID of the table it has locked is present in the hash map. If so, it proceeds to clean up a threshold number of those pages.

Incorporating cleanup into user transactions without increasing requires foreground latency requires carefully bounding the work performed; this is addressed by capping the amount of cleanup per transaction, ensuring that the additional work does not significantly impact user-facing execution.

Additionally, the number of pages tracked in user transaction cleanup hash table is bounded; if this limit is reached, further pages are not processed by the user transactions and are instead deferred to the side list for retry at the end of the sweep. This design ensures that cleanup remains effective while avoiding unbounded resource usage or impact on transaction execution.

Cleanup operations performed within a transaction are safe under aborts, as they are idempotent and are retried later through the side-list mechanism, ensuring no correctness issues.

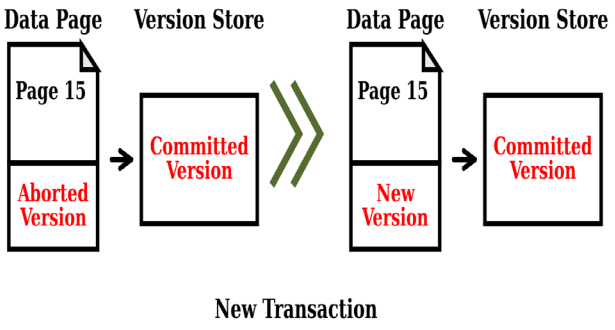

Figure 3: Optimization to overwrite an aborted version.

It is worth noting that this approach complements an already existing optimization. When a user transaction updates a row that contains an aborted version, instead of relying on the logical revert mechanism on demand, it can apply an efficient strategy: directly overwrite the aborted version with the new one being created, while linking this new version to the last committed version. As illustrated in Figure 3, this technique significantly reduces the operational overhead and enables the update to perform nearly as efficiently as if no aborted version had existed. These approaches, when combined with the side list mechanism, significantly enhance cleanup effectiveness, even under contending workloads.

3.3 Cleanup Watermark Not Moving Faster

When the cleanup watermark—typically influenced by the OldestActiveXdesId, the OldestAbortedXdesId, and MinUsefulXTS—stalls, the version cleaner in SQL Azure cannot efficiently remove stale versions. This watermark serves as a boundary for determining which versions are still required for snapshot visibility or undo.

If long running or idle transactions linger, they can delay the advancement of MinUsefulXTS.

Additionally, if the cleanup sweep is lagging, it can prevent removal of aborted transactions from the ATM. This can lead to slow movement of the OldestAbortedXdesId.

Together, these factors force the cleaner to retain versions that would otherwise be safe to delete. As a result, the PVS continues to expand, consuming excess storage. This delay in watermark progression contributes to version bloat and performance degradation, particularly in heavy workloads with long running transactions.

3.3.1 Transaction Level Version Tracking

As mentioned in Section 2.3.3, each PVS page maintains the highest transaction ID that inserted versions into it. As part of the version cleanup, this highest transaction ID is compared against both the OldestAbortedXdesId and OldestActiveXdesId. In scenarios where the sweep of the database is prolonged, removal of aborted transactions from the ATM is delayed, causing the OldestAbortedXdesId marker to advance slowly. This delay hinders the cleanup of all PVS pages, even those that no longer contain versions required for logical revert.

Let us look at this simple scenario with two transactions T1 and T2. T1's XdesId is 100 and has generated off row versions on PVS page P1. P1's highest transaction ID is updated to 100. T1 is aborted at XTS 300. It gets added to ATM and the OldestAbortedXdesId is updated to 100. T2's XdesId is 200 and has generated off row versions on PVS page P2. P2's highest transaction ID is updated to 200. T2 committed at XTS 350. At this point there are no other active transactions in the system, MinUsefulXTS would have advanced beyond 350. Figure 4 illustrates how the system's state looks at this instant.

3.3.1.1 Cleanup behavior with old approach

With T1 being the OldestAbortedXdesId, watermark remains at 100. In this case, versions on P1 and P2 cannot be cleaned up until logical revert is done, and OldestAbortedXdesId moves. This is because the XdesId on PVS pages is not less than OldestAbortedXdesId. While it is expected that versions on P1 cannot be cleaned up until logical revert/undo of T1 is done, versions on P2 can be cleaned up as they belong to committed transactions and not needed for undo. The presence of aborted transactions or delay in movement of OldestAbortedXdesId can block cleanup and leave committed versions unnecessarily occupying space.

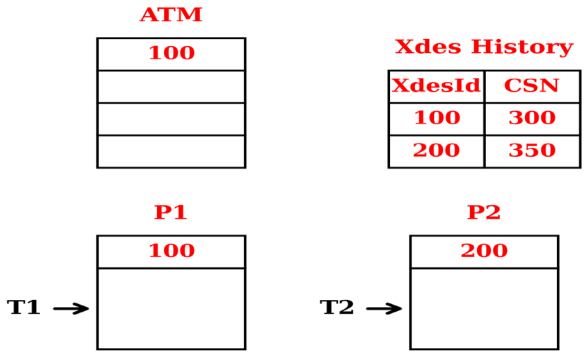

Figure 4: Internal state along with relevant data structures.

To address this issue, a transaction-level tracking mechanism for PVS pages was introduced. Each transaction now maintains a list of all PVS pages where it inserted versions. When generating subsequent versions, the transaction first attempts to reuse pages from this list; if none is suitable, it allocates a new page. With transaction level PVS, presence of bursty short transactions can lead to excessive version store growth and space inefficiencies. To address this, efficient reuse of PVS pages from completed transactions. Efficient reuse of PVS pages requires correctly handling pages that may contain a mix of committed and aborted versions.

Upon committing, the transaction transfers its associated PVS pages that are full to the PVSPageTrackerList dedicated to committed transactions. The last page in this list may have free space and can be reused by other transactions. If the transaction is aborted, its full pages are moved to a separate PVSPageTrackerList for aborted transactions. For the last PVS page used, a bit called HasAbortedTranVersions is set in the corresponding tracker entry to indicate the presence of aborted versions. Additionally, transactions cache PVS pages that are not full upon commit or rollback, allowing reuse by other transactions. For cleanup, the approach differs based on the tracker list.

- Aborted transactions: PVS pages in the aborted list continue to use OldestAbortedXdesId and OldestActiveXdesId as markers for cleanup, as before.
- Committed transactions: If the HasAbortedTranVersions bit is set, cleanup still relies on both these markers because the page contains versions from aborted transactions. If the bit is not set, cleanup no longer depends on OldestAbortedXdesId, since none of the versions on these pages are needed for logical revert.

Let us revisit the same example and see how cleanup happens.

3.3.1.2 Cleanup Behavior With New Approach

The system now tracks which PVS pages belong to each completed transaction, and accordingly it classifies pages into a committed list or an aborted list. Page P1 from T1 is on the aborted list and the HasAbortedTranVersions bit not set (assuming no free space), and Page P2 from T2 is on the committed list and does not have HasAbortedTranVersions bit set. P2 has no free space as well and eligible for cleanup. For cleanup of Page P1, there is no change in criteria. Cleaner waits for the undo of T1 to finish and ATM to get truncated to cleanup P1. This is ensured by using OldestAbortedXdesId 100 as marker. But for cleaning up P2, HasAbortedTranVersions is checked first. This is not set in this case because all versions on P2 belong to the committed transactions. So P2 can be cleaned up without comparing with OldestAbortedXdesId. Note that OldestActiveXdesId and MinUsefulXTS are much larger values as there are no other active transactions in the system.

Transaction-level version tracking incurs modest overhead because PVS pages, which were previously maintained globally, are now tracked per transaction and later transferred into two separate global lists—one for committed transactions and one for aborted ones. It enables earlier cleanup of committed-transaction pages, reduces unnecessary PVS growth, and prevents committed pages from being stalled behind slow-moving aborted-transaction watermarks. If the per-transaction PVS page lists are lost due to a crash, the system recovers them from durable storage — the PVS pages themselves are persistent as they are allocated in the database's data files and tracked by metadata pages as mentioned in section 2.3.3. On restart, during recovery system performs a scan of the PVS table to reconstruct the full set of PVS pages that existed before the crash. All such pages are conservatively placed into PVS pages corresponding to aborted transaction list. Then background version cleaner resumes normal page-level cleanup using both oldestAbortedXdesId and oldestActiveXdesId.

As a result, this approach ensured that even if the logical revert sweep is slow, PVS pages linked to committed transactions are still cleaned up efficiently. In systems with fewer aborted transactions, which is the common case, this strategy significantly improves cleanup performance and helps maintain PVS size in check. It is important to note that if an active scan needs versions on these PVS pages, they will not be cleaned up. This is done by comparing the CSN of the highest transaction ID that inserted version with MinUsefulXTS.

3.3.2 Statement Level Cleanup

Currently, whether the versions on a PVS page are needed for read transactions is determined by comparing CSN of the highest transaction ID that inserted versions into the PVS page with MinUsefulXTS. Under RCSI, which is the default isolation level of SQL Azure, this approach can be overly conservative because RCSI creates snapshots at the statement level. This means versions only need to be retained as of the beginning of each statement, not the start of the entire RCSI transaction—which is what MinUsefulXTS represents. This optimization is only applicable when all transactions that inserted versions into the PVS page have finished and are not needed by any read queries. Since only the highest XdesId that inserted versions into the PVS page, but not all XdesIds are tracked, it can only be applied to pages where all versions were generated by a single transaction. This is the common case with transaction-level PVS Section 5 outlines the future improvement of this feature.

With the implementation of transaction-level PVS, PVS pages containing versions from a single transaction can be seen more frequently. This is because each transaction maintains a local list of PVS pages and for subsequent versions generated it uses pages from this list. In the PVSPageTrackerList, a bit is used to indicate whether all versions on a PVS page belong to the same transaction which would be the highest transaction ID. For these pages, cleanup is first attempted by comparing the CSN of the highest transaction ID with MinUsefulXTS. If that fails, cleanup is retried using MinUsefulStatementXTS. This enhancement, combined with transaction-level PVS, has significantly improved cleanup efficiency.

4 VERSION CLEANUP IN PRODUCTION

This section describes the deployment of these improvements across all the databases in SQL Azure and quantifies the impact of these improvements. Before deploying, these improvements were validated to not introduce performance regressions on critical OLTP benchmarks such as TPCC and TPCE. Because transactions in these benchmarks are short lived, the PVS size remains small and cleanup thresholds are not reached;

consequently, they are not suitable for evaluating cleanup effectiveness or space reclamation. TPCC and TPCE was used solely for regression validation, while relying on production telemetry to evaluate the impact of the proposed techniques under workloads that generate sustained version store pressure.

4.1 Impact of Parallel Cleanup

In SQL Azure, parallel cleanup is enabled only when there are sufficient resources, and when the PVS sizes are larger than 100 GB. Figure 5 demonstrates the difference in cleanup sweep times with and without parallel version cleanup on all the databases where PVS size was above 100 GB over a 30-day period. For the common cases, the sweep times with and without parallel cleanup feature do not differ significantly. But in cases where cleanup times are larger, parallelizing cleanup sweep reduces sweep time noticeably.

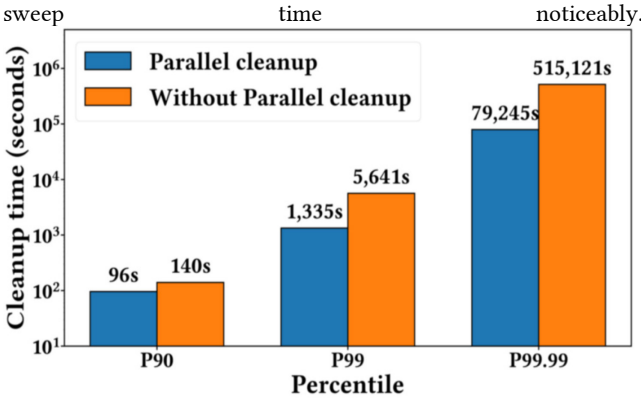

Figure 5: Sweep time in seconds with and without parallel cleanup on databases with at least 100 GB PVS.

Figure 5 shows that with the parallel cleanup feature enabled, the 99.99th percentile sweep time is approximately 1 day, whereas with parallel cleanup disabled, databases at the same percentile required approximately 6 days to complete. Although the percentage of databases that have larger sweep times might seem small, the absolute numbers and the resulting DevOps incidents were high given the millions of databases hosted in Azure.

4.2 Impact of Transaction Level Version Tracking

In case of longer sweeps, versions from committed transactions are aggressively removed by the transaction level tracking feature. This can be seen clearly from our production telemetry. As illustrated in Figure 6, each data point represents an aggregate of PVS cleaned in TB across databases in all regions with sweep times exceeding 4 hours at a given time interval. Over a 30-day period, on these scenarios where sweep time is at least 4 hours, transaction-level version tracking accounted for approximately 60% of the total PVS reclaimed. Across all regions, in a 30-day period without transaction level cleanup, this 60% of the total PVS would remain allocated until at least the next full sweep, each of which takes on average four hours to complete. By allowing these pages to be reclaimed earlier, the feature not only accelerates space recovery but also reduces customer Cost of Goods Sold (COGS) by shrinking billable storage footprint sooner, preventing unnecessary bloat.

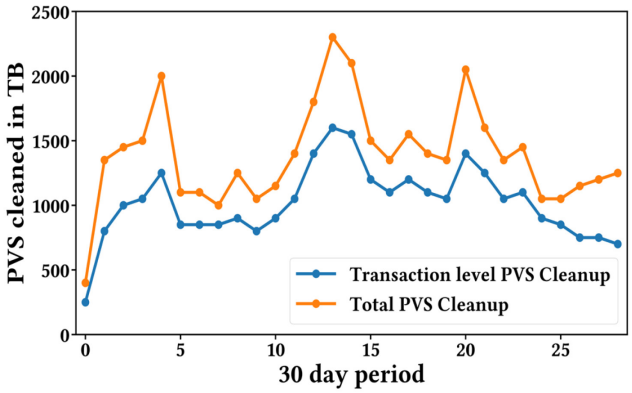

Figure 6: PVS cleaned by transaction level cleanup vs Total PVS cleaned in TB.

4.3 Impact of Statement Level Cleanup

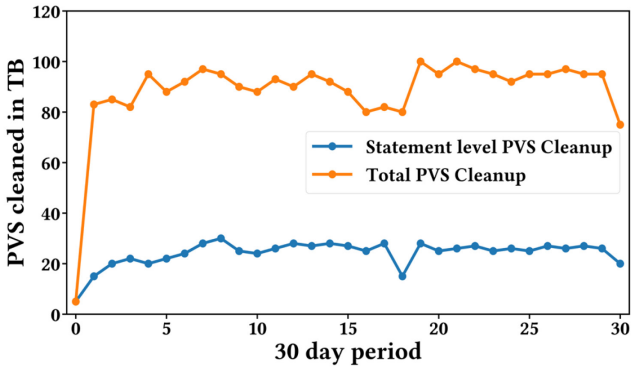

Figure 7: PVS cleaned by statement level cleanup vs Total PVS cleaned in TB.

For RCSI, statement level cleanup feature attempts to aggressively clean up by moving the cleanup marker when PVS was held by MinUsefulXTS. In Figure 7, each data point represents an aggregate of PVS cleaned in TB across databases in all regions where PVS holdup exceeded 50 GB due to MinUsefulXTS at each time interval. In these scenarios, statement-level cleanup contributed approximately 27% of the total PVS reclaimed. Across all regions, in a 30-day period without statement level cleanup, this 27% PVS would have remained allocated until the long running snapshot transaction finishes. By allowing these pages to be reclaimed earlier, the feature not only accelerates space recovery but also reduces customer COGS by shrinking billable storage footprint sooner, preventing unnecessary bloat.

4.4 Impact Under Hot Workload

In a 10-day period, there were approximately 385M cleanup sweeps across the fleet. Of these, around 3.76M sweeps got affected by some conflicting workload. Among the sweeps affected, 3.23M sweeps succeeded eventually. This indicates that

over 98.6% of the sweeps under hot workload finished successfully because of the retrying logic implemented through side list or cleanup happening through user transactions. While the unsuccessful sweeps are only 1.4%, but they represent around 52K in absolute number, which is still high. Currently, the User-Assisted Cleanup feature operates only when exclusive locks are acquired on the table. Extending this capability to transactions holding shared locks could help in reducing these further.

Overall, these improvements significantly reduced the size of the PVS by enabling more aggressive and timelier cleanup. They also reduced situations where cleanup fell behind workload demand, lowering the risk of storage pressure. From a user perspective, this translated directly into lower storage costs and less memory consumption for tracking these PVS pages, contributing to a decrease in customer cost of goods sold. In some cases, user queries also showed modest performance improvements, particularly when large transactions that inserted significant data were aborted; faster cleanup of these stale versions reduced unnecessary page reads. We also observed a noticeable drop in customer-reported incidents, reflecting improved system reliability and reduced operational overhead—even as the number of databases in Azure continued to grow.

5 FUTURE IMPROVEMENTS

All the improvements mentioned above for version cleanup have been enabled in SQL Azure over the past few years. More than 12 million databases in Azure have been running with these improvements. With these, bottlenecks in version cleanup have significantly reduced.

Currently, ghost cleanup is another cleanup task that runs independently scanning the database on its own schedule, reading the pages into the buffer pool, and taking locks on them separately. Both version cleaner and ghost cleaner can be combined into one cleaner. With this, all the redundant IO and locking can be avoided; the full capabilities of the improved version cleanup can be leveraged for ghost cleanup as well.

In addition, the current design of the User Assisted Cleanup feature would clean up pages only if the user transaction took an X lock on the table. If the transaction is running with a shared lock, this feature would not help. This feature can be extended to support such cases. Additionally, when the cleaner cannot obtain the necessary locks, it can read the page under a shared latch to compute the minimum XdesId stamped on rows associated with aborted transactions. Rather than invalidating the entire sweep, this approach enables a best-effort cleanup by removing aborted transactions whose XdesIds are lower than the computed minimum across all pages where lock acquisition failed.

Similarly, the Statement-Level Cleanup feature is currently restricted to PVS pages where all stored versions were from a single transaction. This is because when a PVS page contains versions from multiple transactions, just using maxXdesId that inserted versions cannot guarantee the commit order of all the transactions involved. Extending this to cleanup pages with non-unique XdesIds is possible by maintaining the LatestXdesId that inserted versions in addition to maxXdesId.

6 RELATED WORK

Beyond commercial system implementations, research of in-memory databases has explored both scalable MVCC execution. Silo [31] focused on high-performance transaction processing using optimistic concurrency control and an epoch-based commit protocol, with an emphasis on scalability and recovery rather than version cleanup itself. Scalable garbage collection [3] showed that in-memory MVCC systems can suffer from significant garbage-collection overhead under long-running queries, and proposed integrating more aggressive, eager cleanup into transaction processing to mitigate this bottleneck. Similarly, Hybrid garbage collection in MVCC [10] examined version growth in SAP HANA under mixed OLTP/OLAP workloads and introduced a hybrid garbage collection approach that combines multiple strategies to efficiently reclaim obsolete versions. Our work is complementary to these efforts. Instead of proposing a new concurrency control protocol or an in-memory garbage collection design, we focus on version cleanup in a large-scale cloud database service. In this setting, cleanup must operate reliably across highly diverse workloads and production environments, including very large databases, resource contention, and delays in advancing global visibility thresholds.

A brief survey was done on how versions are maintained and how stale version cleanup is implemented in other commercial database systems such as PostgreSQL, MySQL Inno DB, and Oracle. This section compares how cleanup is done in these systems with SQL Azure.

Like Version Cleaner in SQL Azure, the Autovacuum process in PostgreSQL (PG) [26], runs periodically in the background to clean up dead tuples. Autovacuum employs multiple processes to parallelize cleanup, when necessary, whereas Version Cleaner uses a multi-threaded approach. A key distinction is that in PG, cleanup is not parallelized at table level, only single process cleans up for one table or index at any time [26] [1], while in SQL Azure, multiple threads can concurrently clean up a large table with numerous versions. Like Version Cleaner, Autovacuum leverages metadata to avoid scanning all pages. Despite some similarities, cleanup mechanisms differ between PG and SQL Azure because there are fundamental differences in PG MVCC implementation. In PG, rows are updated by invalidating the old tuple and inserting a new tuple. All the older versions of the row are maintained along with the latest copy within the main table itself and not in a separate version store which is different from SQL Azure. Autovacuum makes a first pass to gather all tuple Ids corresponding to dead tuples, cleans up indexes and makes another pass rereading some pages to cleanup table [27]. Autovacuum is non-intrusive and defers cleanup when conflicting workloads are present, which can cause it to lag [23]. Version cleaner is non-intrusive as well but makes use of user transactions to assist with cleanup under contention. Any long running transactions would prevent the cleanup of dead tuples and cause the vacuum operation to lag. In PG, where Transaction ID is 32- bit, vacuuming is also responsible for avoiding the Transaction ID wraparound issues [25]. If vacuum is falling behind, the risk of wraparound can cause downtime for vacuum maintenance. In SQL Azure, there is less risk of vacuum maintenance due to the Transaction ID wraparound, as it uses an internal 48-bit transaction ID and transaction ID is assigned for only write transactions while PG assigns them for both read and write transactions [24].

The cleanup mechanism in Inno DB [9] relies on a background purge task that continuously removes stale data from undo logs. Inno DB stores old row versions outside the data row, whereas SQL Azure maintains versions both in-row and off-row. Keeping versions in-row can not only improve read performance by eliminating the need to access additional pages to reconstruct older versions but also reduce page IO and logging overhead during writes. However, storing all versions in-row has drawbacks (see Section 2.2.1), which SQL Azure mitigates by moving older versions off-row. In essence, SQL Azure strikes an optimal balance between in-row and off-row storage of row versions. Another difference is that Inno DB only maintains a diff as opposed to maintaining a full copy of the row. This maintains versions in a space efficient manner; SQL Azure does this for in-row versions, but full data row is stored in case of off-row versions. Like SQL Azure, Inno DB cleanup is held if some transaction is interested in the undo logs needed for reading older versions [8]. Purge threads make use of metadata as well to keep track of which records to purge. This makes purge task to not scan the entire table – it is directly pointed to the specific undo entries and records that are ready to be cleaned up. In effect, the cleanup of old versions in Inno DB is incremental and continuous, interleaved with transaction processing, rather than a big periodic sweep. This is because purge threads do not perform rollback. In Inno DB, the rollback is performed synchronously by the user transaction and not by purge thread [6]. In SQL Azure, undo is logically instantaneous and asynchronous. It leverages version cleaner to remove the older versions physically (logical revert). This causes reduced unavailability during crash recovery and avoids locks being held until undo [7][19].

Oracle's implementation is like Inno DB; rollback is synchronous and is done by the user transaction itself [15] and is not deferred to background thread. Once the rollback is done and if the versions are no longer needed by anyone, System Monitor (SMON) [14] background thread cleans up the undo space, which is where oracle stores older versions. In case of crash recovery, oracle fast-tracks undo by making use of background thread and user transactions [16]. Oracle also maintains only diff needed to reconstruct the old version. These versions are maintained only off-row. This is different from the SQL Azure implementation, which uses a mix of in-row and off-row to improve the read query performance. Another difference is the way oracle handles cleanup of old row versions is through an undo retention policy [17] rather than an aggressive deletion process. The database will attempt to keep undo records for at least a threshold duration after they are produced, to satisfy queries. Once undo data is older than the retention period and no active query or open transaction still needs it, it is considered expired and the space can be reused for new undo records. The cleanup of undo is thus on-demand reuse: if a new transaction needs space and the undo tablespace is full, the database will start overwriting the oldest expired undo data. This is done by the normal operation of transactions as needed. The flip side of automatic reuse is that if a query is long running, the undo data needed for a consistent read might be overwritten before the query finishes as new transactions start [17][18]. This causes the transactions to be terminated. In some cases, if the undo data cannot be overwritten it causes new transactions to fail. SQL Azure does not use undo retention period. While this can cause the version store size to increase in the presence of long transactions, it will not terminate them.

Overall, while PostgreSQL, Inno DB, Oracle, and SQL Azure share the common goal of reclaiming space and maintaining versioned data efficiently, their approaches reflect fundamental architectural differences. PostgreSQL relies on periodic vacuuming to mitigate tuple bloat and transaction ID wraparound, whereas Inno DB and Oracle adopt continuous or on-demand purging tied to undo logs and retention policies. SQL Azure distinguishes itself by combining in-row and off-row version storage with asynchronous logical undo and multi-threaded cleanup, reducing contention and improving recovery times.

In the bigger picture, the optimizations in this paper target a long-standing pain point in MVCC systems: cleanup tends to be affected by slowest or most heavily contended transactions. Instead of changing MVCC semantics, our approach focuses on making cleanup less fragile by reclaiming committed versions sooner, scaling cleanup when pressure builds, and, when necessary, letting user transactions help make progress under contention.

7 CONCLUSION

In this paper, the earlier implementation of the version cleaner, bottlenecks that limited its scalability as well as the solutions were presented. The challenges led to engineering version cleanup to make it more scalable and robust. These improvements are now enabled by default in SQL Azure DB. As evidenced by the production telemetry, these improvements enhanced version cleanup significantly and resolved the key bottlenecks. Also, the number of cases where cleanup was not keeping up went down substantially, minimizing the risk of space bloats. Together with these enhancements, version cleanup has evolved into a scalable and efficient system that meets the demands of our growing fleet in SQL Azure.

ACKNOWLEDGMENTS

We would like to thank all members of the Accelerated Database Recovery team for their contributions. Without their commitment and hard work, the improvements described in this paper would not have been possible. Additionally, we would like to thank our leadership team for sponsoring this work and continuing to invest in our work in this area. This work was performed while Raghavendra TK was employed at Microsoft. Raghavendra TK has since moved to Snowflake. The authors used Microsoft Copilot, an AI-based assistance tool, to help with grammar and structure. All content was reviewed and approved by the authors.

REFERENCES

[1] Amazon Web Services Documentation. Aurora Postgre SQL Vacuuming. (Accessed: July 7, 2026) https://aws.amazon.com/blogs/database/parallel-vacuuming-in-amazon-rds-for-postgresql-and-amazon-aurora-postgresql/

[2] Microsoft Documentation. Azure SQL Elastic Pools. (Accessed: July 7, 2026) https://learn.microsoft.com/en-us/azure/azure-sql/database/elastic-pool-overview?view=azuresql

[3] Böttcher, Jan & Leis, Viktor & Neumann, Thomas & Kemper, Alfons. (2019). Scalable garbage collection for in-memory MVCC systems. Proceedings of the VLDB Endowment. 13. 128-141. 10.14778/3364324.3364328.

[4] Delaney, K., Randal, P. S., Tripp, K. L., Cunningham, C., Machanic, A. Microsoft SQL Server 2008 Internals. Microsoft Press, Redmond, WA, USA, 2009.
[5] MySQL Documentation. The InnoDB Storage Engine. (Accessed: July 7, 2026) https://dev.mysql.com/doc/refman/8.4/en/innodb-storage-engine.html
[6] MySQL Documentation. InnoDB MVCC. (Accessed: July 7, 2026) https://dev.mysql.com/doc/refman/9.1/en/innodb-multi-versioning.html
[7] MySQL Documentation. InnoDB Recovery. (Accessed: July 7, 2026) https://dev.mysql.com/doc/refman/8.0/en/innodb-recovery.html
[8] MySQL Documentation. InnoDB Transaction Management. (Accessed: July 7, 2026) https://dev.mysql.com/doc/refman/9.5/en/optimizing-innodb-transaction-management.html
[9] MySQL Documentation. InnoDB Undo Management. (Accessed: July 7, 2026) https://dev.mysql.com/doc/refman/9.1/en/innodb-undo-tablespaces.html
[10] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid garbage collection for multi-version concurrency control in SAP HANA. In Proceedings of the 2016 international conference on management of data. 1307-1318.
[11] Mohan, C., Haderle, D. J., Lindsay, B. G., Pirahesh, H., Schwarz, P. M. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM TODS, 17(1):94–162, 1992.
[12] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. ACM Trans. Database Syst. 8, 4 (Dec. 1983), 465–483. https://doi.org/10.1145/319996.319998
[13] Oracle Documentation. Data Concurrency and Consistency. (Accessed: July 7, 2026) Data Concurrency and Consistency
[14] Oracle Documentation. SMON and Background processes. (Accessed: July 7, 2026) https://docs.oracle.com/en/database/oracle/oracle-database/19/refrn/background-processes.html
[15] Oracle Documentation. Transaction Rollback. (Accessed: July 7, 2026) https://docs.oracle.com/en/database/oracle/oracle-database/18/sqlrf/ROLLBACK.html
[16] Oracle Documentation. Rollback during Recovery. (Accessed: July 7, 2026) https://docs.oracle.com/cd/B10500_01/server.920/a96533/instreco.htm#429546
[17] Oracle Documentation. Undo management. (Accessed: July 7, 2026) https://docs.oracle.com/en/database/oracle/oracle-database/26/admin/managing-undo.html
[18] Oracle Support. ORA-01555: Snapshot too old. (Accessed: July 7, 2026) https://support.oracle.com/knowledgefs?docId=40689
[19] P. Antonopoulos et al. "Constant time recovery in SQL Azure database," Proceedings of the VLDB Endowment, Volume 12, Issue 12, August 2019, pp. 2143–2154.
[20] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19). Association for Computing Machinery, New York, NY, USA, 1743–1756. https://doi.org/10.1145/3299869.3314047
[21] Panagiotis Antonopoulos, Mansi Chauhan, Shailender Dabas, Rajat Jain, Darshan Kattera, Wonseok Kim, Hanuma Kodavalla, Nikolas Ogg, Prashanth Purnananda, Rahul Ranjan, Alex Swanson, and Divyesh Tikmani. MD-MVCC: Multi-version Concurrency Control for Schema Changes in SQL Azure Database. PVLDB, 18(12): 4791–4803, 2025. doi:10.14778/3750601.3750605
[22] PostgreSQL Documentation. (Accessed: July 7, 2026) https://www.postgresql.org/docs/current/
[23] PostgreSQL Documentation. Autovacuum. (Accessed: July 7, 2026) https://www.postgresql.org/docs/current/routine-vacuuming.html#AUTOVACUUM
[24] PostgreSQL Documentation. Transaction IDs. (Accessed: July 7, 2026) https://www.postgresql.org/docs/current/transaction-id.html
[25] PostgreSQL Documentation. Transaction ID Wraparound. (Accessed: July 7, 2026) https://www.postgresql.org/docs/current/wraparound.html
[26] PostgreSQL Documentation. Vacuum. (Accessed: July 7, 2026) https://www.postgresql.org/docs/current/routine-vacuuming.html
[27] PostgreSQL Documentation. Vacuum Phases. (Accessed: July 7, 2026) https://www.postgresql.org/docs/current/progress-reporting.html#VACUUMPHASES
[28] Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, and Girish Mittur Venkataramanappa. 2020. Concurrent updates to pages with fixed-size rows using lock-free algorithms. Proc. VLDB Endow. 13, 12 (August 2020), 3195–3203. https://doi.org/10.14778/3415478.3415544.
[29] Microsoft Documentation. SQL Server page and extent architecture guide. (Accessed: July 7, 2026) https://learn.microsoft.com/en-us/sql/relational-databases/pages-and-extents-architecture-guide?view=sql-server-ver17
[30] Microsoft Documentation. SQL Server Transaction Locking and Row Versioning Guide. (Accessed: July 7, 2026) https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver17
[31] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, page 18–32, New York, NY, USA, 2013.

8 APPENDIX

8.1 Glossary of Terms and Abbreviations

This section contains glossary of terms used throughout the paper.

Table 1: Frequently used acronyms with full forms.

Acronym	Full form
MVCC	Multi Version Concurrency Control
RCSI	Read Committed Snapshot Isolation
SI	Snapshot Isolation
PVS	Persistent Version Store
XTS	Transaction Timestamp
XdesID	Transaction ID
CSN	Commit Sequence Number
PFS	Page Free Space
ATM	Aborted Transaction Map