Constant Time Recovery in Azure SQL Database

Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu,
Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda,
Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, Girish Mittur Venkataramanappa
Microsoft
One Microsoft Way
Redmond, WA 98052 USA

{panant, peterbyr, waync, cdiaconu, raghavt, hanumak, praspu,
aradu, ravellas, girishmv}@microsoft.com

ABSTRACT

Azure SQL Database and the upcoming release of SQL Server introduce a novel database recovery mechanism that combines traditional ARIES recovery with multi-version concurrency control to achieve database recovery in constant time, regardless of the size of user transactions. Additionally, our algorithm enables continuous transaction log truncation, even in the presence of long running transactions, thereby allowing large data modifications using only a small, constant amount of log space. These capabilities are particularly important for any Cloud database service given a) the constantly increasing database sizes, b) the frequent failures of commodity hardware, c) the strict availability requirements of modern, global applications and d) the fact that software upgrades and other maintenance tasks are managed by the Cloud platform, introducing unexpected failures for the users. This paper describes the design of our recovery algorithm and demonstrates how it allowed us to improve the availability of Azure SQL Database by guaranteeing consistent recovery times of under 3 minutes for 99.999% of recovery cases in production.

PVLDB Reference Format:

Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, Girish Mittur Venkataramanappa. Constant Time Recovery in Azure SQL Database. *PVLDB*, 12(12) : 2143-2154, 2019.

DOI: https://doi.org/10.14778/3352063.3352131

1. INTRODUCTION

Database recovery is a critical component of every DBMS, guaranteeing data consistency and availability after unexpected failures. Despite being an area of research for over three decades, most commercial database systems, including SQL Server, still depend on the ARIES [9] recovery protocol, an algorithm that leverages Write-Ahead Logging (WAL) and defines distinct recovery phases to eliminate ad-hoc recovery techniques.

Even though ARIES simplifies the recovery process and allows it to be generic for all transactional operations, recovering the database to a consistent state requires undoing all operations performed by uncommitted transactions which makes the cost of recovery proportional to the work performed by these transactions. This significantly impacts database availability since recovering a long running transaction can take several hours. A recent example that demonstrates the severity of this problem is a case where an unexpected failure occurred while one of our customers was attempting to load hundreds of millions of rows in a single transaction. The database required 12 hours to recover while the corresponding tables were completely inaccessible due to the exclusive locks held by the transaction performing the data load. Even though this is an extreme case, when a service like Azure SQL Database is responsible for millions of databases, long running recoveries are a common pain point.

To mitigate this issue, SQL Server has previously depended on targeted optimizations that speed up each phase of recovery, for example leveraging parallelism, but has not made any fundamental changes to the recovery algorithm itself. Although these optimizations proved to be adequate for on-premise environments, where the DBMS is running on high-end servers and failures are generally planned, the transition to the Cloud reset the expectations around database availability and recovery because of:

- The rapid growth in database sizes which also translates to longer transactions.
- The more frequent failures of commodity hardware used in Cloud architectures, resulting in frequent failovers that also depend on the recovery process.
- The fact that software upgrades and other maintenance tasks are now managed by the Cloud provider, making it hard to ensure that the user workload can adjust to minimize downtime, for example by preventing long running transactions during maintenance windows.

To meet our availability SLAs in the Cloud and improve the quality of our service, we designed a novel database recovery algorithm that combines ARIES with multi-version concurrency control (MVCC) to achieve recovery in constant time regardless of the user workload and the transaction sizes. Our algorithm

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352131

depends on Write-Ahead logging and the recovery phases defined by ARIES, but takes advantage of generating row versions for the most common database operations (e.g. inserts, updates, deletes) to avoid having to undo them when rolling back uncommitted transactions. This allows us to bring the database to a consistent state in constant time and release all locks held by uncommitted transactions, making the database fully accessible and eliminating the main disadvantage of ARIES. Using the same versioning technique, our algorithm also allows instant rollback of individual transactions and enables aggressively truncating the transaction log, even in the presence of long running transactions, therefore allowing large data loads and modifications using only a small, constant amount of log space.

This paper describes the overall design of "Constant Time Recovery" (CTR) in the upcoming release of SQL Server (also publicly known as "Accelerated Database Recovery" [7]) and demonstrates how it allows us to significantly improve the availability of Azure SQL Database. Section 2 begins with some background around the recovery process in the earlier releases of SQL Server. Section 3 outlines the architecture of CTR and provides the detailed design of its components. Section 4 presents our experimental results regarding the performance and resource usage of our recovery scheme. Finally, Section 5 covers the results and experience from enabling CTR in production in Azure SQL Database.

2. BACKGROUND ON SQL SERVER

This section provides a summary of the recovery process and MVCC implementation in the earlier releases of SQL Server which is required to better understand the architecture and design choices for CTR. More detailed information regarding both areas can be found in the public documentation of SQL Server [3].

2.1 Database Recovery

The database recovery process in SQL Server is based on the ARIES recovery algorithm. Data and log are stored separately, using WAL to log all operations to the transaction log before performing any modifications to the corresponding data pages.

All log records written to the transaction log are uniquely identified using their Log Sequence Number (LSN) which is an incremental identifier based on the physical location of the log record in the log file. Other than details about the operation performed (e.g. deletion of key X), each log record also contains information about the page modified (Page Id), the transaction that performed the operation (Transaction Id), and the LSN of the previous log record (Previous LSN) for the same transaction.

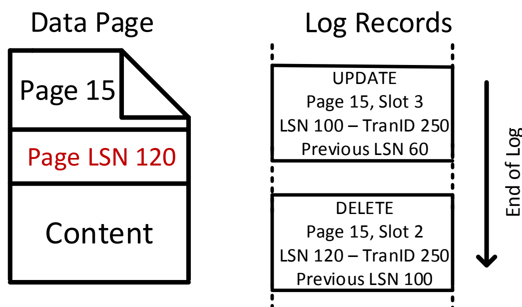

Figure 1. Representation of a data page and its log records.

Data is stored in pages of 8KB that reside in the data files. The Page Id that identifies each page is also based on the physical location of the page in the data file. Other than the data and other information about the page content, each page also maintains the LSN of the log record corresponding to the last modification to this page (Page LSN). Figure 1 provides an example of a data page and the corresponding log records that have updated it.

For performance reasons, data pages are cached in memory in the Buffer Pool. Transactions update the data only in-memory and a background checkpoint process is responsible for periodically writing all dirty pages to disk, after guaranteeing that the corresponding log records have also been flushed to disk to preserve WAL semantics. The checkpoint process additionally captures a) the state of all active transactions at the time of the checkpoint and b) the LSN of the oldest dirty page in the system (Oldest Dirty Page LSN), which will be used for the purposes of recovery. Following ARIES, the SQL Server recovery process has three distinct phases. Figure 2 demonstrates these phases and the portion of the log they process.

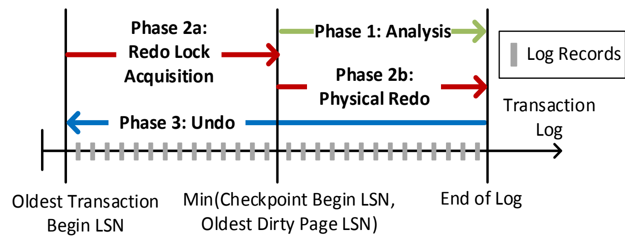

Figure 2. The phases of the recovery process.

2.1.1 *Analysis*

Analysis is the first phase of recovery and is responsible for identifying a) any transactions that must be rolled back because they had not committed when the failure occurred and b) the LSN of the oldest dirty page in the system. Since the checkpoint process captured all active transactions and the oldest dirty page LSN at the time of the checkpoint, Analysis can scan the log starting from the minimum of the beginning of the last completed checkpoint (Checkpoint Begin LSN) and the Oldest Dirty Page LSN to reconstruct the required information.

2.1.2 *Redo*

The Redo phase is responsible for bringing the database back to the state it was at the time of the failure. This is achieved by processing the transaction log and redoing all operations that might not have been persisted to disk. Since Analysis has recomputed the Oldest Dirty Page LSN, Redo should only process the log from this point as any previous updates have already been flushed to disk. When processing a log record, Redo compares the Page LSN with the LSN of the current log record and only applies the operation if the Page LSN is lower, which indicates that the current image of the page is older.

Even though redoing this smaller portion of the log suffices to bring the data pages to the required state, SQL Server's Redo processes the log starting from the beginning of the oldest active transaction. This allows recovery to reacquire all the locks held by active transactions and make the database available at the end of Redo for improved availability. However, this causes the Redo process to be proportional to the size of the longest active transaction. To improve performance, in the latest releases of SQL Server, Redo has been parallelized, but still preserves the invariant that all operations corresponding to a specific page are applied in LSN order.

2.1.3 Undo

Undo is the last phase of recovery and is responsible for rolling back any transactions that were active at the time of the failure. As Redo has reacquired the locks required by these transactions, the Undo process can be performed while the database is available and user queries will be blocked only if they attempt to access the data modified by the transactions pending undo.

For each active transaction, Undo will scan the log backwards, starting from the last log record generated by this transaction and undo the operation performed by each log record. This makes the cost of Undo proportional to the size of uncommitted transactions. Undoing these operations is also logged using Compensation Log Records (CLR) to guarantee that the database is recoverable even after a failure in the middle of the Undo process. The log records for each transaction can be efficiently traversed using the Previous LSN stored in each log record. Once a transaction is rolled back, its locks are released.

2.2 Multi-version Concurrency Control

SQL Server introduced multi-version concurrency control in 2005 to enable Snapshot Isolation (SI). Versioning is performed at the row level: for every user data update, SQL Server updates the row in-place in the data page and pushes the old version of the row to an append-only version store, linking the current row version to the previous version. Further updates generate newer versions, thereby creating a chain of versions that might be visible to different transactions following the SI semantics. Each version is associated to the transaction that generated it, using the Transaction Id, which is then associated to the commit timestamp of the transaction. The versions are linked to each other using their physical locator (Page Id, Slot id). Figure 3 provides an example of a row linked to two earlier versions.

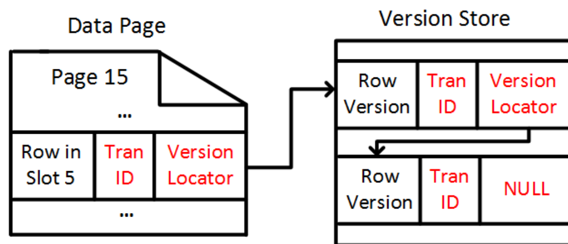

Figure 3. Example of MVCC row version chain.

Upon visiting a row, a transaction traverses the chain of versions and determines the visibility of each version by comparing the transaction's snapshot timestamp, established at the beginning of the transaction, with the commit timestamp of the transaction that generated the version. Once older snapshot transactions commit, older versions become permanently invisible and the version store can get truncated to free up space. Given that these versions are only used for the purposes of SI, the version store doesn't need to be preserved across restarts and is stored in SQL Server's "TempDB", a system database that is recycled every time the SQL Server process restarts. This allows for efficient version generation, as these operations are not logged.

3. CONSTANT TIME RECOVERY

3.1 Overview

Azure SQL Database and the upcoming release of SQL Server introduce Constant Time Recovery (CTR), a novel database recovery algorithm that depends on ARIES but leverages the row versions generated for MVCC to support:

- Database recovery in constant time, regardless of the user workload and transaction sizes.
- Transaction rollback in constant time regardless of the transaction size.
- Continuous transaction log truncation, even in the presence of long running transactions.

CTR achieves these by separating transactional operations into three distinct categories and handling their recovery using the most appropriate mechanism.

3.1.1 Data Modifications

Data modifications refer to regular DML operations that update user data. CTR leverages MVCC versions to instantly undo data updates without having to undo every operation independently using the transaction log. All data modifications are versioned, storing the earlier versions of each row in the version store that is now redesigned to be persistent and recoverable. Each version is marked with the Transaction Id of the transaction that generated it which allows us to identify the state of the corresponding transaction (active, committed or aborted). When a transaction rolls back, it is simply marked as "aborted", indicating that any new transactions should ignore the versions generated by this transaction and access the earlier committed versions. During database recovery, Analysis identifies the state of every transaction and Redo recovers the row and the version store content as of the time of the failure. Then Undo marks the uncommitted transactions as aborted making all updates by these transactions invisible. This allows Undo to complete in constant time, regardless of the transaction sizes.

3.1.2 System Operations

System operations refer to internal operations the DBMS uses to maintain its internal data structures, such as space allocation and deallocation, B-Tree page splits, etc. These operations cannot be easily versioned because they update system data structures that have been designed to be highly efficient, using bitmaps and other compacted data structures that do not allow maintaining the required versioning information. Additionally, these operations are usually tied to user data modifications and can be a significant percentage of the operations performed by a long-running transaction. For example, a large data load allocates a large number of pages. In CTR, these operations are always performed by short-lived, system transactions that update the internal data structures and immediately commit. Based on that, when a failure occurs, these operations will not be undone as part of a long-running user transaction. Instead, the allocated space and other updated data structures will be lazily reclaimed and fixed up in the background.

3.1.3 Logical and Other Non-versioned Operations

This last category refers to operations that cannot be versioned because they are either a) logical, such as lock acquisition operations that indicate that a certain lock must be acquired during recovery or cache invalidation operations that are responsible for invalidating in-memory caches when a transaction rolls back, or b) they are modifying data structures that need to be accessed when starting up the database, before recovery has started, and, therefore, must maintain a very specific format that does not allow versioning. Even though redoing and undoing these operations

still require the transaction log, CTR leverages an additional log stream, *SLog*, that allows tracking only the relevant operations and not having to process the full transaction log for the corresponding transactions. Given that such operations are generally associated with schema changes and not data manipulation, their volume is several orders of magnitude lower than the previous categories and can be practically redone/undone in minimal time.

3.2 Persistent Version Store

Persistent Version Store (PVS) is a new implementation of SQL Server's version store that allows persisting and recovering earlier versions of each row after any type of failure or a full database restart. PVS versions contain the same data and are chained in the same way as the ones stored in TempDB. However, since they are recoverable, they can be used for accessing earlier versions of each row after a failure. CTR leverages this to allow user transactions to access the committed version of each row without having to undo the modifications performed by uncommitted transactions.

PVS allows row versions to be recoverable by storing them in the user database and logging them in the transaction log as regular user data. Hence, at the end of Redo all versions are fully recovered and can be accessed by user transactions. In CTR, versions are needed for recovery purposes and have to be preserved until the committed version of each row has been brought back to the data page. This process occurs lazily in the background (described in Section 3.7) and, therefore, causes the size of PVS to be higher than the TempDB version store which is only used for SI and can be truncated aggressively. Additionally, logging the versions introduces a performance overhead for the user transactions that generate them. To address both the performance and the storage impact of PVS, we separated it into two layers.

3.2.1 In-row Version Store

The in-row version store is an optimization that allows the earlier version of a row to be stored together with the latest version in the main data page. The row contains the latest version, but also the required information to reconstruct the earlier version. Since in most cases the difference between the two versions is small (for example when only a few columns are updated), we can simply store the diff between the two versions. Even though computing and reapplying the diff requires additional CPU cycles, the cost of generating an off-row version, by accessing another page and logging the version as a separate operation, is significantly higher. This makes in-row versioning a great solution for reducing the storage overhead, but also the cost for logging the generated version, as a) we effectively log the version together with the data modification and b) we only increase the log generated by the size of the diff. At the same time, in-row versioning improves read access performance, since the earlier version of a row can be reconstructed without having to access another page that might not be in the cache.

Figure 4 demonstrates an example of a row that contains the current version of the row together with the diff and required information to reconstruct the earlier version. When a row is deleted, TempDB version store creates a copy of the row in the version store and replaces the row on the data page with a stub indicating that the row was deleted. With in-row versioning, we simply mark the row as deleted, while retaining the original content to be served as the earlier version. Similarly, updates modify the row in-place, but also append the byte-diff between the old and the new versions to allow reconstructing the earlier version.

Row Header	Latest Row Version	Tran ID	In-Row Version Metadata	Diff for Previous Version (incl. Tran ID and Version Locator)

Figure 4. Example of a row that contains in-row versioning information for an earlier version.

Despite its benefits in most common cases, in-row versioning can negatively impact the performance of the system if it significantly increases the size of rows in the data pages. This is particularly problematic for B-Trees as it can lead to page splits, which are expensive, but also deteriorate the quality of the data structure, making future accesses more expensive. To mitigate this issue, the size of in-row versions is capped both in terms of the size of the diff, as well as the size of the row it can be applied to. When these exceed certain thresholds, PVS will fall back to generating off-row versions, on a different page.

3.2.2 Off-row Version Store

Off-row version store is the mechanism for persisting versions that did not qualify to be stored in-row. It is implemented as an internal table that has no indexes since all version accesses are based on the version's physical locator (Page Id, Slot Id). Each database has a single off-row PVS table that maintains the versions for all user tables in the database. Each version of user data is stored as a separate row in this table, having some columns for persisting version metadata and a generic binary column that contains the full version content, regardless of the schema of the user table this version belongs to. Generating a version is effectively an insertion into the off-row PVS table, while accessing a version is a read using the version's physical locator. By leveraging regular logging, off-row PVS is recovered using the traditional recovery mechanisms. When older versions are no longer needed, the corresponding rows are deleted from the table and their space is deallocated. Details regarding the off-row PVS cleanup process are described in Section 3.7.

The off-row PVS leverages the table infrastructure to simplify storing and accessing versions but is highly optimized for concurrent inserts. The accessors required to read or write to this table are cached and partitioned per core, while inserts are logged in a non-transactional manner (logged as redo-only operations) to avoid instantiating additional transactions. Threads running in parallel can insert rows into different sets of pages to eliminate contention. Finally, space is pre-allocated to avoid having to perform allocations as part of generating a version.

3.3 Logical Revert

3.3.1 Overview

CTR leverages the persistent row versions in PVS to instantly roll back data modifications (inserts, updates, deletes) without having to undo individual row operations from the transaction log. Every data modification in CTR is versioned, updating the row in-place and pushing the previous version into the version store. Also, similar to MVCC (Figure 3), each version of the row is marked with the Transaction Id of the transaction that generated it.

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, but if the transaction is aborted, this version is definitely not visible and the query traverses the version chain to identify the version that belongs to a committed transaction and is visible.

This algorithm allows queries to access transactionally consistent data in the presence of aborted transactions; however, this state is not ideal in terms of performance since queries traverse multiple versions to access the committed data. Additionally, if a new transaction updates a row with an aborted version, it must first revert the effects of the aborted transaction before proceeding with the update. To address these and limit the time that aborted transactions are tracked in the system, CTR implements two different mechanisms for reverting the updates performed by aborted transactions:

- Logical Revert is the process of bringing the committed version of a row back to the main row in the data page, so that all queries can access it directly and versions in the version store are no longer required. This process compares the state of the aborted and committed versions and performs the required compensating operation (insert, update or delete) to get the row to the committed state. The operations performed by Logical Revert are not versioned and are executed in system transactions that are undone normally using the transaction log. Since these transactions only revert a row at a time, they are guaranteed to be short-lived and don't affect recovery time. Figure 5 provides an example of a Logical Revert operation. Logical Revert is used by a background cleanup process, described in detail in Section 3.7, to eliminate all updates performed by aborted transactions and eventually remove the aborted transactions from the system.
- When a new transaction updates a row that has an aborted version, instead of using Logical Revert on demand, which would be expensive, it can leverage an optimization to overwrite the aborted version with the new version it is generating, while linking this new version to the previously committed version. Figure 6 presents an example of this optimization. This process minimizes the overhead for these operations and allows them to be almost as fast as if there was no aborted version.

Using these mechanisms, both reads and writes can access or update any row immediately after a transaction that updated it rolls back. The same process applies during recovery, eliminating the costly Undo process that undoes each operation performed by uncommitted transactions. Instead, in CTR, the database is fully available, releasing all locks, while row versions are lazily cleaned up in the background.

It is also important to note that although CTR depends on MVCC for recovery purposes, it still preserves the locking semantics of SQL Server, for both reads and writes, and supports all isolation levels without any changes in their semantics.

3.3.2 Transaction State Management

As described in the previous section, each query decides whether a version is visible by checking the transaction state based on the Transaction Id stored in the version. For SI, visibility depends on the commit timestamp of the transaction that generated the version. Since SQL Server does not allow snapshot transactions to span server restarts, the commit timestamps can be stored in memory and need not be recovered. CTR, however, requires tracking the state of aborted transactions until all their versions have been logically reverted and are no longer accessible. This depends on the background cleanup process (Section 3.7) that performs Logical Revert for all aborted versions in the database and can be interrupted by unexpected failures. Because of that, the state of aborted transactions must be recovered after any type of failure or server restarts.

CTR stores the aborted transaction information in the "Aborted Transaction Map" (ATM), a hash table that allows fast access based on the Transaction Id. When a transaction aborts, before releasing any locks, it will add its Transaction Id to the ATM and generate an "ABORT" log record indicating that it was aborted. When a checkpoint occurs, the full content of the ATM is serialized into the transaction log as part of the checkpoint information. Since Analysis starts processing the log from the Checkpoint Begin LSN of the last successful checkpoint, or earlier, it will process this information regarding the aborted transactions and reconstruct the ATM. Any transactions that aborted after the last checkpoint will not be included in the checkpoint, but Analysis will process their ABORT log records and add them to the map. Following this process, Analysis can reconstruct the ATM as of the time of the failure, so that it is available when the database becomes available at the end of Redo. As part of the Undo phase, any uncommitted transactions will also be marked as aborted, generating the corresponding ABORT log records, and added to the ATM.

Once all versions generated by an aborted transaction have been reverted, the transaction is no longer interesting for recovery and can be removed from the ATM. Removing a transaction is also a logged operation, using a "FORGET" log record, to guarantee that the content of the ATM is recovered correctly.

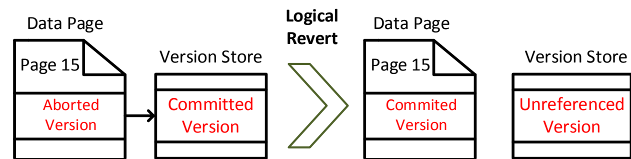

Figure 5. Example of a row before and after Logical Revert.

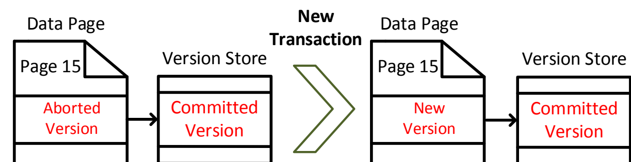

Figure 6. Optimization to overwrite an aborted version.

3.3.3 Short Transaction Optimization

Despite the benefits of Logical Revert, maintaining the Aborted Transaction Map and forcing queries to visit additional versions incur a performance penalty. This overhead is well justified for long running transactions that are generally rare, but significantly impact recovery time. However, it can be problematic for high-

volume, short OLTP transactions as they would significantly increase the size of the ATM, leading to high memory footprint and inefficient checkpoints serializing the ATM into the transaction log. At the same time, undoing such short-running transactions using the transaction log would only take a few milliseconds, while allowing the system to remove aborted versions and avoid performance impact to future queries.

To optimize for both scenarios, CTR dynamically decides, based on the transaction size, whether a transaction should be marked as aborted, using the CTR mechanisms, or undone using the transaction log. When a transaction attempts to roll back, we evaluate the number of operations it performed and the amount of log it generated and qualify it as "short" if these don't exceed certain thresholds. Short transactions will not go through the CTR rollback process, but use traditional undo, so that they are immediately removed from the system.

3.4 Non-versioned Operations

Although Logical Revert allows us to eliminate undo for any data modifications that are versioned, SQL Server has a wide variety of operations that cannot be versioned because they are:

- Logical, such as acquiring coarse-grained locks at the table or index level, invalidating various caches when a transaction rolls back or accumulating row and page statistics for Bulk operations.
- Updating system metadata in data structures that are highly compacted, such as information about which pages are allocated.
- Updating critical system metadata required for starting up the database, before recovery can reconstruct versioning information, such as updates to the "boot page", a special page that contains the core information required for initialization.

To handle these operations while guaranteeing recovery in constant time, we are leveraging two different mechanisms:

3.4.1 SLog: A Secondary Log Stream

SLog is a secondary log stream designed to only track non-versioned operations that must be redone or undone using information from the corresponding log records. This allows us to efficiently process relevant log records without having to scan the full transaction log. Given that such operations are generally associated with schema changes and other rare database operations, such as changing various database options, the volume of log records written to the SLog is several orders of magnitude lower than the total volume of the transaction log, allowing us to process these operations in minimal time. For example, when altering the data type of a column in a large table, the transaction will have to update millions of rows, but SLog will only contain a handful log records, for acquiring the exclusive lock and invalidating metadata caches.

SLog is used during Undo to roll back any outstanding non-versioned operations, but it is also used by the Redo phase to redo logical operations, such as reacquiring coarse-grained locks, without having to process the transaction log from the beginning of the oldest uncommitted transaction, as described in Section 2.1. Because of this, SLog must be stored in a data structure that can be recovered before Redo starts. This cannot be easily achieved using traditional database data structures (Heaps, B-Trees, etc.) as these depend on WAL and recovery. Instead, SLog is implemented as an in-memory log stream that is persisted to disk by being serialized into the traditional transaction log. In memory, SLog is stored as a linked list of log records, ordered based on their LSNs to allow efficient traversal from the oldest to the newest LSN during Redo. Additionally, to allow efficient processing during Undo, SLog records are also linked backwards pointing to the previous log record generated by the same transaction. Figure 7 provides an example of SLog with 5 log records corresponding to two transactions.

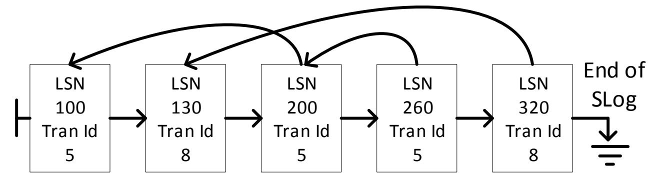

Figure 7. In-memory SLog structure.

As SQL Server generates the log records corresponding to non-versioned operations, they get written to the transaction log, as usual, but they also get appended to the in-memory SLog. When a checkpoint occurs, all in-memory records with LSN $\leq$ the Checkpoint Begin LSN get serialized and written into the transaction log. During recovery, the Analysis phase will process the transaction log starting from at least the Checkpoint Begin LSN of the last completed checkpoint, and can now additionally reconstruct the portion of the SLog that was serialized to the transaction log as part of the checkpoint. Furthermore, any log records with higher LSNs will also be visited by Analysis and will be appended to the in-memory SLog.

With this algorithm, at the end of Analysis, we have reconstructed the SLog as of the time of the failure and can use it for the Redo and Undo phases. The Redo process is split into two phases:

a) For the portion of the log between the Begin LSN of the oldest uncommitted transaction and the min(Oldest Dirty Page LSN, Checkpoint Begin LSN), Redo processes the log using the SLog since it only has to redo logical operations as all physical page operations had been flushed to disk. The number of log records in the SLog should be minimal and, therefore, redone almost instantly, regardless of the size of the oldest uncommitted transaction.

b) For the portion of the log after the min(Oldest Dirty Page LSN, Checkpoint Begin LSN) and until the end of the log, Redo will follow the regular process of redoing all operations from the transaction log. Since the database engine takes checkpoints frequently, regardless of the user workload, we can guarantee that this portion of the log will always be bounded and redone in constant time.

During Undo, since regular DML operations are recovered using Logical Revert, we only undo non-versioned operations using the SLog. Since the SLog only collects a small subset of operations, undoing them should be almost instant, regardless of the size of the uncommitted transactions that are being rolled back. If the undo process performs any compensating non-versioned operations, the corresponding Compensating Log Records (CLRs) will also be appended to the SLog with their backlinks pointing to the next log record to be undone. This guarantees that Undo will never reprocess the same log records twice and can make forward

progress even after repeated failures. As Undo completes for each uncommitted transaction, the transaction is added to the ATM and its locks are released.

Figure 8 demonstrates the recovery process in CTR which leverages the SLog to complete each phase of recovery in constant time. If a transaction rolls back during normal transaction processing, while the database is online, CTR will also use the SLog to roll back the corresponding non-versioned operations, therefore completing the rollback almost instantly.

Similar to the transaction log, SLog is only needed for recovery purposes and can be truncated as transactions commit or abort and are no longer interesting for recovery. When a checkpoint occurs, SQL Server calculates the Begin LSN of the oldest active transaction in the system and this is the low watermark used to truncate any SLog records with lower LSNs. Additionally, since SLog is maintained in memory and it is critical to reduce its footprint, we have introduced an aggressive cleanup process that will scan the full content of SLog and remove any records generated by transactions that are no longer active, regardless of their LSNs. Both truncation and cleanup have been designed to run in parallel with the workload that might be appending new records to the SLog and, therefore, do not impact user activity.

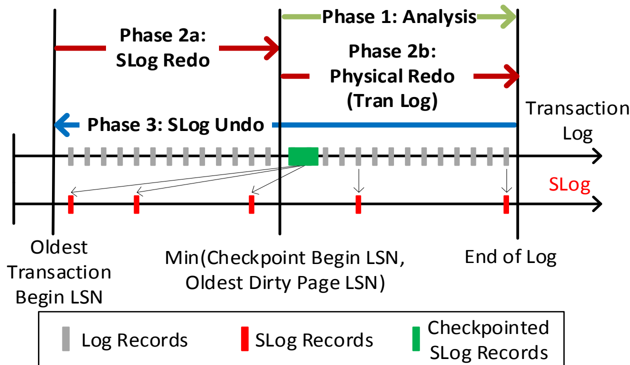

Figure 8. Recovery process in CTR.

3.4.2 Leveraging System Transactions

Even though SLog allows us to easily handle the recovery of any non-versioned operation, we want to minimize the amount of log records that are appended there because:

- Both Redo and Undo process the corresponding SLog records and, therefore, recovery time depends on the size of SLog.
- SLog is kept in expensive main memory.

Space allocations and deallocations are the most common non-versioned operations in the database, as they are tied to the user transaction sizes: the number of pages to be allocated or deallocated depends on the amount of data the user transaction inserts or deletes. In SQL Server, the information regarding whether a page is allocated and what object it belongs to is captured in special metadata pages that maintain a bitmap with the status of a range of pages in the file. Since the information is tightly packed, it is impractical to be versioned, making it impossible to handle their recovery using versioning and Logical Revert. To avoid using SLog, CTR performs all allocation and deallocations in system transactions that can commit immediately after performing the operation. These transactions don't depend on versioning and are normally redone and undone using the transaction log. Due to their very small size, these transactions can be recovered in minimal time and do not affect the overall recovery time.

More specifically, in CTR, all operations will allocate new pages in a system transaction that is committed immediately, but will also mark the page as "potentially containing unused space" so that they can be reclaimed if the user transaction rolls back. A background cleanup thread will periodically scan the database for such pages, evaluate whether they contain any data and deallocate them if they are empty. On the other hand, deallocations cannot be committed before the user transaction commits, since in case of rollback all deallocated pages contain valid data that should be accessible. CTR addresses this by deferring all deallocations until after the user transaction that deallocated the space is committed. The user transaction will only mark a large set of pages as "deferred deallocated", using an operation logged in SLog, while the actual deallocation will be performed by a background thread only after the user transaction has committed. The background thread deallocates the deferred pages in batches using short-lived system transactions that are recovered using the transaction log and do not depend on versioning or SLog. If the user transaction rolls back, as part of undoing the SLog, it will unmark the corresponding pages as "deferred deallocated" and, therefore, they will remain allocated as expected.

3.5 Redo Locking Optimization

As described in Section 2.1, in order to reacquire all the locks held by uncommitted transactions, the Redo phase normally processes the transaction log from the beginning of the oldest uncommitted transaction. This allows making the database available before the Undo phase and is important in ARIES recovery because Undo can take significant time. Since Undo in CTR is extremely efficient, we could technically defer making the database available until Undo has completed and let Redo not acquire any locks. However, there are special cases of recovery that require allowing user access without performing Undo and rolling back uncommitted transactions:

- Readable secondaries using physical replication [8] replay the transaction log generated by the primary and allow users to execute read queries. If a secondary crashes, it has to go through recovery to bring the database to a consistent state, but it will stop after the Redo phase since the transactions are still active on the primary. To allow queries after recovery, Redo reacquires any locks held by currently active transactions.
- Unresolved distributed transactions are transactions where the database engine failed to contact the distributed transaction coordinator to retrieve their outcome and, therefore, cannot yet be declared as committed or aborted. To make the database available during this time, which is generally unbounded, Redo reacquires the locks held by these transactions.

To address these scenarios, SLog is used to track and reacquire low-volume, coarse-grained locks, such as table or metadata object locks, during Redo, while a new locking mechanism is introduced at the transaction level to handle granular locking, at the page and row level, where the volume can be extremely high. More specifically, at the end of Redo, each uncommitted transaction will acquire an exclusive "Transaction" lock on its

Transaction Id. As described in Section 2.2, each row version is marked with the Transaction Id of the transaction that generated it. When a new transaction attempts to access a row version, it will request a shared lock on the Transaction Id of this version and block if this transaction is still in recovery. Once the uncommitted transaction is committed or aborted, it will release its Transaction lock and allow any conflicting transactions to access the corresponding rows. This mechanism allows us to achieve row and page level locking during recovery without having to track and reacquire the locks for individual rows and pages. When all transactions that were in recovery have been committed or aborted, the database state is updated and any new row accesses will no longer request a Transaction lock, eliminating the performance overhead.

3.6 Aggressive Log Truncation

Despite the significant differences compared to ARIES recovery, CTR still depends on WAL and uses the transaction log for recovering the database. ARIES recovery uses the log to Undo uncommitted transactions, therefore, requiring it to be preserved from the Begin LSN of the oldest active transaction in the system and making the log size proportional to the transaction size. This is problematic because it requires users to carefully provision the log space based on their workloads and is particularly complex in the Cloud where the system automatically allocates the appropriate space without having visibility into the user workload.

By leveraging Logical Revert and the SLog, CTR only uses the portion of the log after the beginning of the last successful checkpoint to redo any updates; hence CTR no longer needs to Redo or Undo the log from the beginning of the oldest uncommitted user transaction. The log must still be preserved for undoing system transactions without versioning, but it can now be aggressively truncated up to the minimum of a) the Checkpoint Begin LSN of the last successful checkpoint, b) the Oldest Dirty Page LSN and c) the Begin LSN of the oldest active system transaction. Since system transactions are guaranteed to be short-lived and checkpointing is managed by the DBMS and can occur at consistent intervals, the transaction log can be truncated continuously regardless of the size of the user transactions. This enables performing large data loads or modifications in a single transaction using only a small, constant amount of log space.

Finally, to enable the "short transaction optimization" described in Section 3.3.3, if there are any active user transactions, CTR will allow the log to be preserved for at least 200 MBs, so that it can be used to roll back any "short" transactions that qualify for the optimization. When a transaction attempts to roll back, together with the criteria described earlier, we evaluate whether the required log is still available. If it is, we will follow the traditional, log-based rollback process, otherwise the transaction is marked as "aborted" and will be cleaned up lazily using Logical Revert.

3.7 Background Cleanup

In CTR, all data modification operations generate row versions which must be eventually cleaned up to free up space, but also eliminate the performance overhead for queries that traverse multiple versions to access the committed version of each row. In the case of committed transactions, earlier versions are not interesting for recovery purposes and can be immediately removed once they are no longer needed for SI. On the other hand, for aborted transactions, the committed data resides in the earlier versions, requiring Logical Revert to be performed before the older versions can be safely removed. Even though new user transactions participate in the cleanup process when an aborted version is updated, we need a mechanism to continuously remove unnecessary versions. CTR introduces a background cleanup task that is responsible for:

- Logically reverting updates performed by aborted transactions.
- Removing aborted transaction from the ATM once all their updates have been logically reverted.
- Cleaning up in-row and off-row versions from PVS once they are no longer needed for recovery or SI.

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

3.7.1 Logical Revert and In-row Version Cleanup

Both Logical Revert and in-row version cleanup are performed by accessing all data pages in the database that contain versions to logically revert the rows corresponding to aborted transactions and remove any unnecessary in-row versions. Once all versions generated by aborted transactions have been reverted, the corresponding aborted transactions can be safely removed from the ATM.

To efficiently identify data pages requiring cleanup and avoid scanning the entire database, CTR introduces additional metadata for each data page indicating whether it contains versions that might be eligible for cleanup. This metadata is stored in special system pages that SQL Server maintains and are known as Page Free Space (PFS) pages since they are mainly used for tracking whether each page has space available. In CTR, before any data modification occurs on a data page, a bit indicating that this page contains versions is set in PFS. It is important for this to happen before the actual operation on the data page occurs, since, in case of failure, we must guarantee that the cleanup process is aware that this page might contain uncommitted data and must be cleaned up. 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 we only access the PFS page if the state for this page is changing.

The cleanup process wakes up periodically, every few minutes, and proceeds as follows:

- Takes a snapshot of the Transaction Ids for all the aborted transactions in the ATM.
- Scans the PFS pages of the database and processes all data pages that are marked as containing versions. For every such page, a) performs Logical revert for any rows that have a version corresponding to an aborted transaction and b) removes any in-row versions that belong to earlier committed transactions and are no longer needed for SI. If all versions were successfully removed from the page, the bit indicating that this page contains versions is unset. The cleanup process operates at a low level, cleaning up pages incrementally, and does not conflict with concurrent user activity.
- After successfully processing all the pages containing versions, the aborted transactions snapshotted at the beginning are removed from the ATM. The snapshot established at the beginning is necessary to make sure

that we remove only the aborted transactions for which we have already logically reverted all their versions and not any new transactions that might have rolled back while the cleanup was active.

Although the cleanup process can take a significant amount of time when a large portion of the database has been updated, it is important that it can remove all snapshotted aborted transactions in one pass. This allows us to execute the cleanup lazily in the background, providing it with only a limited amount of resources (CPU and IO/sec), to avoid impacting concurrent user workload.

3.7.2 Off-row Version Cleanup

In contrast to the Logical Revert cleanup which is necessary to remove aborted transactions from the system and eliminate the overhead of traversing additional versions, the off-row cleanup is only responsible for deleting earlier versions and freeing up space in the database. As described in Section 3.2, the off-row PVS stores all generated versions as rows in an internal table. Versions generated by any transaction and for all user tables in the database get inserted there. This simplifies the cleanup process because the pages containing off-row versions are effectively the pages that belong to this table. Additionally, since this table is optimized for inserts, it is designed to be append-only, allocating new pages to store newer versions. Hence the cleanup process 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, CTR uses a hash map that tracks the status of each page containing off-row versions based on their Page Id. When a new page is allocated, a corresponding entry is added to the hash map. 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, we leverage the fact that SQL Server generates monotonically increasing Transaction Ids and only store the highest Transaction Id, which is indicating 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 SI or recovery (if they still contain committed data for rows that were updated by aborted transactions). CTR globally maintains the lowest Transaction Id that is still accessible by ongoing snapshot transactions in the database and the lowest Transaction Id across all aborted transactions in the ATM. If a page's aggregated Transaction Id is lower than both of these, the page can be deallocated.

As the page gets filled with versions from various transactions, the hash map entry is updated to maintain the highest Transaction Id. When the page is full, the hash map entry indicates that the page is now eligible for cleanup. The cleanup process periodically visits all entries in the hash map and deallocates all pages that are eligible for cleanup and their Transaction Id indicates that they are no longer needed.

Since the hash map is stored only in memory, in case of a failure, recovery is responsible for reconstructing its content. The off-row PVS pages are easily discoverable by identifying all pages that belong to the PVS internal table, an operation SQL Server supports natively [3]. Then, instead of attempting to recompute the exact Transaction Id for each page, recovery simply uses the highest Transaction Id that Analysis identified in the system. 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. In this way, the off-row cleanup process can free up space in the database even in the presence of failures or restarts.

4. EXPERIMENTAL RESULTS

This section presents experimental results regarding the performance of the system when CTR is enabled. All our experiments are executed on a workstation with 4 sockets, 40 cores (Intel® Xeon® Processor E7-4850, 2.00GHz) and 512GB of RAM. External storage consists of two 1.5TB SSDs for data and log respectively.

4.1 Recovery and Transaction Rollback

Our first set of experiments evaluates the performance of recovery and transaction rollback for transactions of different types and sizes. We simulate a long running transaction that inserts, updates or deletes a large number of rows in a table with a clustered index when a failure occurs. Figure 9 presents the recovery and rollback times for CTR and traditional recovery. As we expected, in traditional recovery, both recovery and rollback times are proportional to the size of the transaction for all types of operations. Analysis is performed in constant time in all cases since it only processes the transaction log from the beginning of the last successful checkpoint. On the other hand, Redo, Undo and transaction rollback scale linearly to the size of the transaction as they need to process all operations it performed. When CTR is enabled, recovery completes in constant time for all cases. Analysis and Redo only process the log from the beginning of the last successful checkpoint and, therefore, complete in a small, constant amount of time. The small variance noticed among the experiments only depends on the exact time a background checkpoint occurs before the crash and is not related to the size or type of the transaction. Undo and transaction rollback are practically instant as all DML operations leverage Logical Revert and have no log records to be undone (SLog remains empty as there are no logical operations or updates to system metadata).

During these experiments, we also measure the disk space usage for the transaction log and the version store. We measure the size of the version store as the combined overhead of in-row and off-row versions in the database. Table 1 demonstrates the space usage after inserting, updating or deleting 10 and 50 million rows. The rows are 200 bytes wide and the update modifies an integer column of 4 bytes. Without CTR, the transaction log size grows linearly to the size of the transaction for all operations, consuming significant space. With CTR, however, once it reaches the limit of 200MBs required by the short transaction optimization (Section 3.6), it starts getting truncated, as checkpoints occur, and remains

Table 1. The transaction log and the version store sizes with and without CTR.

Operation	Log size w/ CTR (MB)	Log size w/o CTR (MB)	PVS size w/ CTR (MB)
10M Inserts	99	5101	0
50M Inserts	185	25873	0
10M Updates	162	3110	173
50M Updates	78	15680	908
10M Deletes	341	7771	0
50M Deletes	147	38255	0

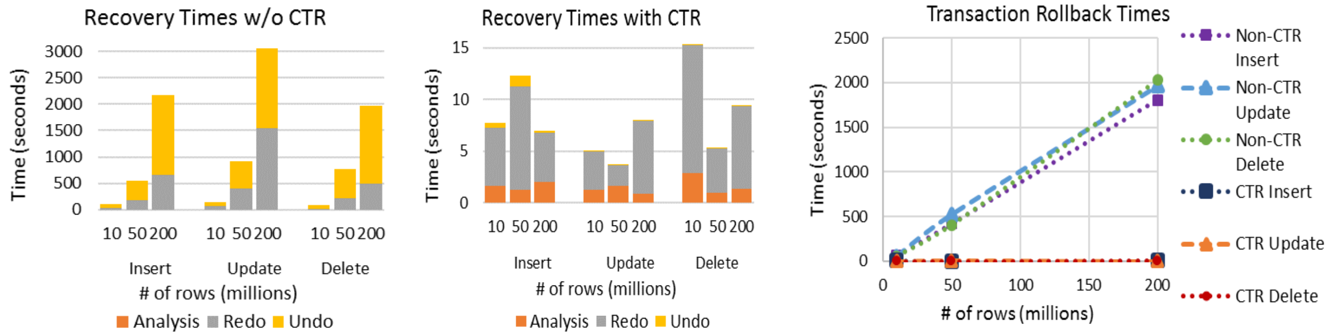

Figure 9. Recovery and rollback times with and without CTR for different operations and number of rows.

stable regardless of the transaction size. The version store is empty for inserts as they do not generate additional versions. The same applies for deletes since, by leveraging in-row versioning, the information that a row is deleted and the Transaction Id of the transaction that deleted it are encoded as part of the existing versioning information and do not consume additional space. In-row versioning also reduces the space overhead in the case of updates. The space is proportional to the number of rows updated, but by storing only the diff between the old and the new version, the version store size remains low. In fact, the aggregate space used for the transaction log and the version store is significantly lower than the log space required for traditional recovery in all cases.

4.2 User Workload Performance

Having analyzed the benefits of CTR around recovery, we also evaluate how it affects the performance of the system during online transaction processing. CTR clearly introduces some overhead by logging the generated versions to make them recoverable, checking for aborted transactions in the Aborted Transaction Map to determine row visibility and having background cleanup processes that consume resources. At the same time, we have performed various optimizations, such as in-row versioning and the ability for future transactions to overwrite aborted versions. To validate these optimizations, we measure the throughput and the latency for the user workload when CTR is enabled.

4.2.1 Throughput

Since most of the overhead introduced by CTR is around generating versions and accessing rows that were recently updated by transactions that might have aborted, we evaluate the throughput of the system using update intensive OLTP workloads. Table 2 presents the throughput degradation introduced by CTR for a TPCC-like workload that is extremely update intensive and should be a worst-case scenario for CTR, and a TPCE-like workload that represents a more common ratio between reads and writes. These numbers are relative to the throughput of the system when using traditional versioning in TempDB. In these workloads we explicitly rollback 1% of transactions to simulate a realistic case where users abort and exercise the rollback and cleaner code paths. We also measure the throughput of the system when in-row versioning is disabled to evaluate its impact.

These results show that the throughput degradation will be negligible for the vast majority of applications, which are generally not as update intensive as TPCC. As we anticipated, the overhead introduced by CTR is greater for more update intensive workloads that generate versions at a higher rate. According to the profile data we collected, CTR is spending more CPU cycles compared to versioning in TempDB when generating the diff for in-row versions, as well as inserting and logging off-row versions in the PVS table. The cleanup processes also consume additional resources (2-3% CPU) and contribute to this difference, however, their impact is relatively small since they are single-threaded and run lazily in the background. In-row versioning significantly improved the throughput for both workloads by allowing versions to be generated as part of the row modification, without having to perform additional inserts to the off-row version store. We are currently analyzing the profile data further to optimize all relevant code paths and reduce the overhead introduced by CTR.

Table 2. Throughput degradation for TPCC and TPCE with and without in-row versioning.

Workload	With in-row	Without in-row
TPCC	13.8%	28%
TPCE	2.4%	3.4%

4.2.2 DML Latency

In this section, we evaluate the impact of aborted transactions on future DML operations that update rows with aborted versions. This scenario is particularly interesting because it is common for an application to retry the same operation after a failure. Even though CTR allows the database to be available in constant time, we must guarantee that the performance of the system is similar to its original performance when there were no aborted versions for the updated rows. Table 3 demonstrates the latency of insert, update and delete operations targeting rows in a clustered index when the same operation rolled back earlier and is being retried. We compare this to the latency of the initial operation when there were no aborted versions for these rows. We measure only the cost of performing the operation after having located the row to be updated and exclude the cost of committing the user transaction that would be dominant and add approximately 0.35ms.

Table 3. Latency of retrying an index row modification.

Operation	Initial Latency	Retry Latency	Difference
Insert	6.1 μs / row	5.8 μs / row	-5%
Update	4.9 μs / row	6.0 μs / row	22%
Delete	3.7 μs / row	4.3 μs / row	16%
Bulk Insert	4.2 μs / row	6.3 μs / row	50%

In all cases, the latency of the retried operations is not significantly higher than the initial latency. This validates that the optimization which enables new operations to overwrite the aborted versions without performing Logical Revert (Section 3.3) allowed us to minimize the overhead for cleaning up the aborted versions. Insert becomes, in fact, slightly faster, as the initial operation already allocated the space for the row, allowing the retried operation to simply use this space without similar overhead. Bulk Insert, on the other hand, is 50% slower due to the fact that the initial operation could just allocate and fill up completely new pages, without having to locate the page in the index corresponding to each row. Since the pages have already been added to the index, the retried operation has to first locate each page before performing the insert. Despite that, this overhead should still be acceptable given the availability benefits of CTR that allow the table to be immediately accessible compared to going through a long running recovery where the table is exclusively locked by the bulk operation that is rolling back.

4.3 Background Cleanup Performance

As described in Section 3.7, it is important for the background cleanup processes to complete regularly to guarantee that the version store size remains stable and the number of aborted transactions in the system is bounded. Figure 10 presents the time required by the cleanup processes to logically revert all rows corresponding to a long running transaction and cleanup all unnecessary versions for different operations and number of updated rows.

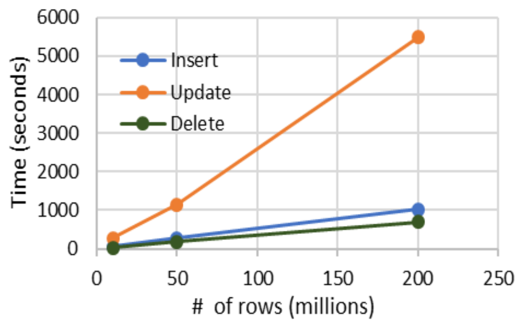

Figure 10. Background cleanup duration.

As we expected, the time required for the cleanup is proportional to the number of rows to be processed. Inserts and deletes can be reverted very efficiently because they only need to mark/unmark the row as deleted to compensate for the aborted operation. Updates, on the other hand, must perform a full update of the row to bring the content of the committed version back on the page, hence more expensive. We ran these experiments for a total database size of 200GB, but the overall database size does not affect the duration of the cleanup. Using the mechanisms described in Section 3.7, the pages that contain versions can be efficiently identified without having to scan the whole database.

5. RESULTS AND EXPERIENCE FROM PRODUCTION

As of May 2019, we have enabled CTR worldwide for over three million databases in Azure SQL Database. The results have been very promising, clearly indicating a significant improvement in recovery times and database availability.

Table 4 demonstrates the difference in recovery times with and without CTR. In the common case, where there are no long running transactions at the time of the failure, the difference is negligible. However, CTR brings significant benefits for the cases where long running transactions would have normally caused recovery to take many minutes or even hours. Although the percentage of impacted databases might seem small, the absolute numbers are quite high given the millions of databases hosted in Azure. This causes major unavailability for the affected users, without any mitigation possible. The other advantage that is visible in our telemetry is that the log space usage has decreased dramatically for all databases that use CTR, as it no longer depends on the user workload. This has made it easier for our service to provision and manage the required space, but also improves the customer experience by eliminating out of log errors for long running transactions.

Table 4. Recovery times with and without CTR.

Configuration	99.9 percentile	99.99 percentile	99.999 percentile
With CTR	60 seconds	1.3 minutes	3 minutes
Without CTR	68 seconds	> 10 minutes	> 1 hour

Despite its benefits, CTR has presented a new set of challenges that must be addressed to guarantee the best possible experience for all user workloads. The main issues we have faced so far are around cases where the cleanup processes get blocked by user transactions, allowing the PVS size to grow and eventually run out of disk space. Since these processes are designed to be online and run in parallel with the user activity, it is natural to compete for system resources (CPU, I/O), as well as locks and latches for the objects that are being cleaned up. While keeping the cleanup processes minimally invasive, we must also ensure that they can successfully complete regularly, deallocate PVS space and remove aborted transactions. To achieve that, we have been fine tuning the resources allocated to the cleanup threads, while making additional optimizations to allow the cleanup processes to retry for any objects they failed to obtain the required locks for. Furthermore, we are introducing a new flavor of off-row version cleanup that aggressively removes versions which are not associated to aborted transactions and are, therefore, not needed for recovery purposes. This allows us to decouple the off-row cleanup from the Logical Revert cleanup and reclaim PVS space more aggressively.

Based on our experiments, these improvements significantly reduce the cases where the cleanup processes cannot keep up, minimizing the risk of running out of disk space. However, we are continuously monitoring the service to identify potential issues and address them accordingly.

6. RELATED WORK

Database recovery has been an area of research for over three decades. While latest research has focused on redesigning recovery to leverage modern hardware, such as non-volatile memory (e.g. [1], [2], [4], [12], [15]), most commercial DBMSs still depend on the ARIES recovery algorithm developed by Mohan et al. [9] in the early 1990s. This is mainly because the latest proposals depend on specialized hardware and require making fundamental changes to the storage engine and transaction

management components of the DBMS that are already complex and highly optimized.

IBM DB2 [5] and MySQL [10] strictly follow the ARIES algorithm, going through the Analysis, Redo and Undo phases. The database becomes available at the end of Redo to improve availability, but locks are held until the end of Undo to prevent concurrent access to the uncommitted data. Amazon Aurora [14] eliminates the need for Redo by continuously redoing all operations at its distributed storage layer which remains available even in the presence of failures for the user database. However, Undo remain unchanged and, therefore, proportional to the user transaction sizes. Oracle [11] introduces an optimization where, if a new transaction attempts to access an uncommitted row while the corresponding transaction is rolling back, the new transaction will undo the operations on this page, on demand, to allow the row to be accessed without having to wait for the overall transaction to roll back. This optimization achieves similar results with CTR but introduces an overhead to new transactions that must first undo earlier changes. As described in Section 3.3, CTR allows new transactions to simply ignore and overwrite aborted versions without having to undo their changes, achieving performance similar to when there are no aborted versions. POSTGRES [13] has been designed from the beginning to leverage versioning and not generate Undo log. All database operations are versioned and new transactions can access earlier versions of each page/row to retrieve the committed data, after a transaction rolls back. Based on that, recovery no longer requires an Undo phase and the database is fully available after Redo. This achieves recovery in constant time, however leveraging this technique for an existing database engine that depends on Undo for a large number of logical and non-versioned operations would be extremely challenging. CTR combines traditional ARIES with MVCC versioning to improve recovery time without having to fundamentally redesign the storage engine. In-row versioning allows versions to be efficiently stored within the data pages, reducing the performance and storage overhead, similar to the version compression scheme applied in Immortal DB [6]. Finally, CTR introduces a novel mechanism for tracking and reverting aborted row versions in the background, without blocking the concurrent user workload or affecting its performance.

7. ACKNOWLEDGEMENTS

We would like to thank all members of the CTR team for their contributions to the project. Without their commitment and hard work, the technology described in this paper would not have been possible. Additionally, we would like to thank our leadership team for sponsoring the project and continuing to invest in our work in this area.

8. REFERENCES

[1] Arulraj, J., Pavlo, A., and Dulloor, S. R. Let's talk about storage & recovery methods for non-volatile memory database systems. *SIGMOD*, 2015, Pages 707-722.

[2] Coburn, J., Bunker, T., Schwarz, M., Gupta, R., and Swanson, S. From ARIES to MARS: Transaction support for next-generation, solid-state drives. *In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, *SOSP*, 2013, Pages 197–212.

[3] Delaney, K., Randal, P. S., Tripp, K. L., Cunningham, C., Machanic, A. *Microsoft SQL Server 2008 Internals*. Microsoft Press, Redmond, WA, USA, 2009.

[4] Gao, S., Xu, J., He, B., Choi, B., Hu, H. PCMLogging: Reducing transaction logging overhead with pcm. *CIKM*, 2011, Pages 2401–2404.

[5] IBM, IBM DB2, Crash recovery. https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.ha.doc/doc/c0005962.html

[6] Lomet, D., Hong, M., Nehme, R., Zhang, R. Transaction time indexing with version compression. *PVLDB*, 1(1):870-881, 2008.

[7] Microsoft, Accelerated Database Recovery. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-accelerated-database-recovery

[8] Microsoft, Offload read-only workload to secondary replica of an Always On availability group. https://docs.microsoft.com/en-us/sql/database-engine/availability-groups/windows/active-secondaries-readable-secondary-replicas-always-on-availability-groups?view=sql-server-2017

[9] 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.

[10] MySQL, InnoDB Recovery. https://dev.mysql.com/doc/refman/8.0/en/innodb-recovery.html

[11] Oracle, Using Fast-Start On-Demand Rollback https://docs.oracle.com/cd/B10500_01/server.920/a96533/instreco.htm#429546

[12] Oukid, I., Booss, D., Lehner, W., Bumbulis, P., Willhalm ,T. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. *DaMoN*, 2014.

[13] Stonebraker, M., Rowe, L. A. The design of POSTGRES. *SIGMOD*, 1986.

[14] Verbitski, A., Gupta, A., Saha, D., Corey, J., Gupta, K., Brahmadesam, M., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvilli, T., Bao, X. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. *SIGMOD*, 2018, Pages 789-796.

[15] Wang, T., Johnson, R. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865-876, 2014.