Constant Time Recovery in Azure SQL Database

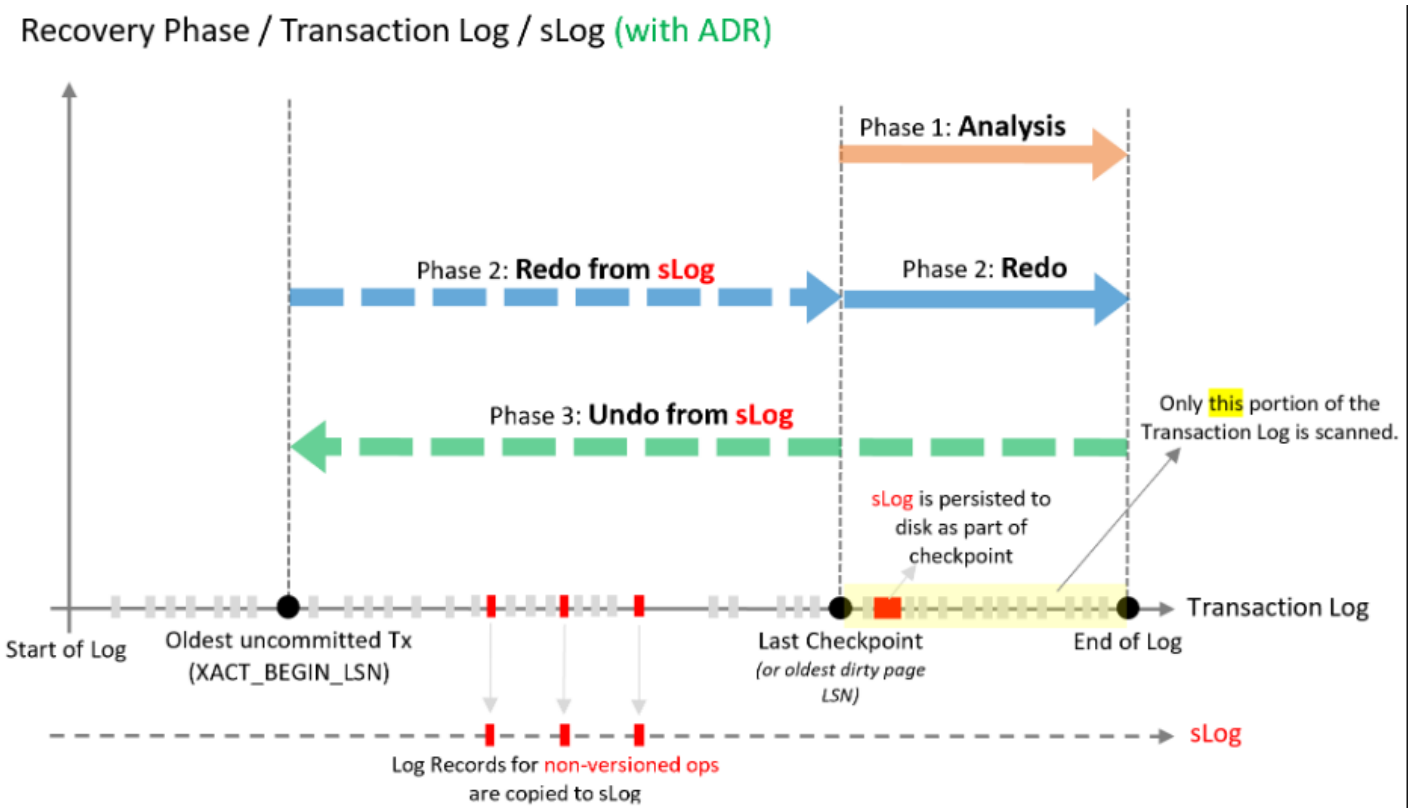
Last updated by | Subbu Kandhaswamy | Oct 13, 2022 at 1:28 PM PDT

Contents

Constant Time Recovery (CTR)

Constant Time Recovery (CTR), a novel database recovery algorithm that depends on ARIES but leverages the row versions generated for Multi-version Concurrency Control (MVCC) to support,

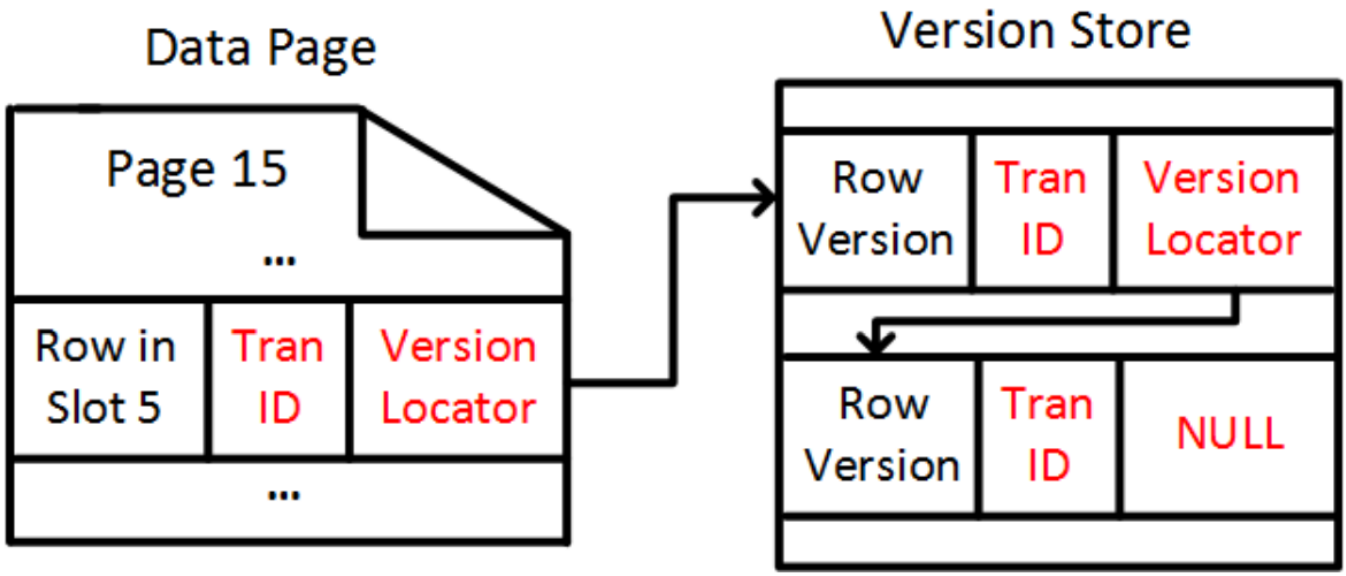
- Database recovery in constant time, regardless of the user workload and transaction sizes.
- Transaction rollback in constant time regardless of the transaction size.
- Continuous transaction log truncation, even in the presence of long running transactions.



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

Multi-version Concurrency Control (MVCC)

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). Below figure provides an example of a row linked to two earlier versions (MVCC Row version chain).



Persistent Version Store (PVS)

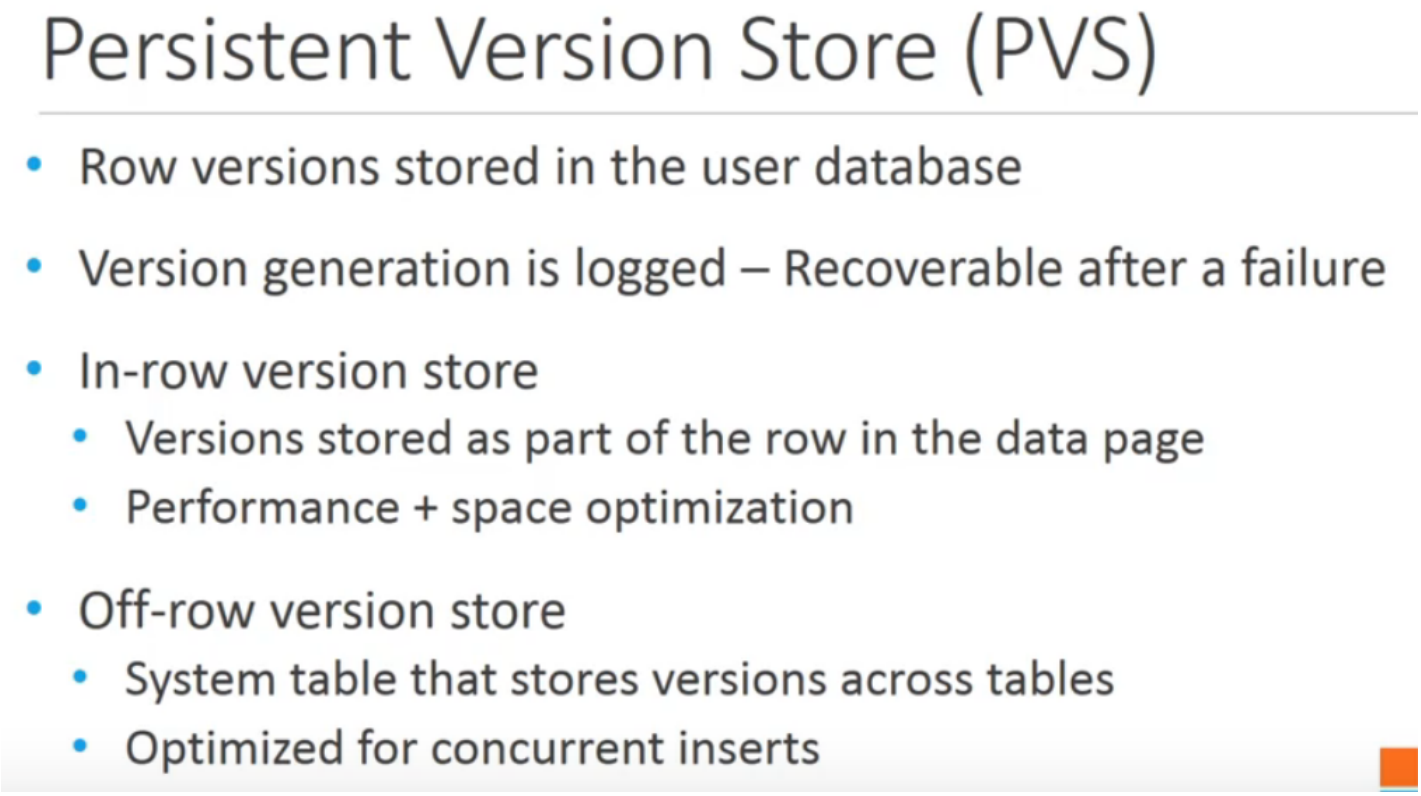
Persistent Version Store (PVS) is a version store that allows persisting and recovering earlier versions of each row after any type of failure or a full database restart. PVS versions contain the same data and are chained in the same way as the ones stored in TempDB. However, since they are recoverable, 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.

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.

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 processes 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.

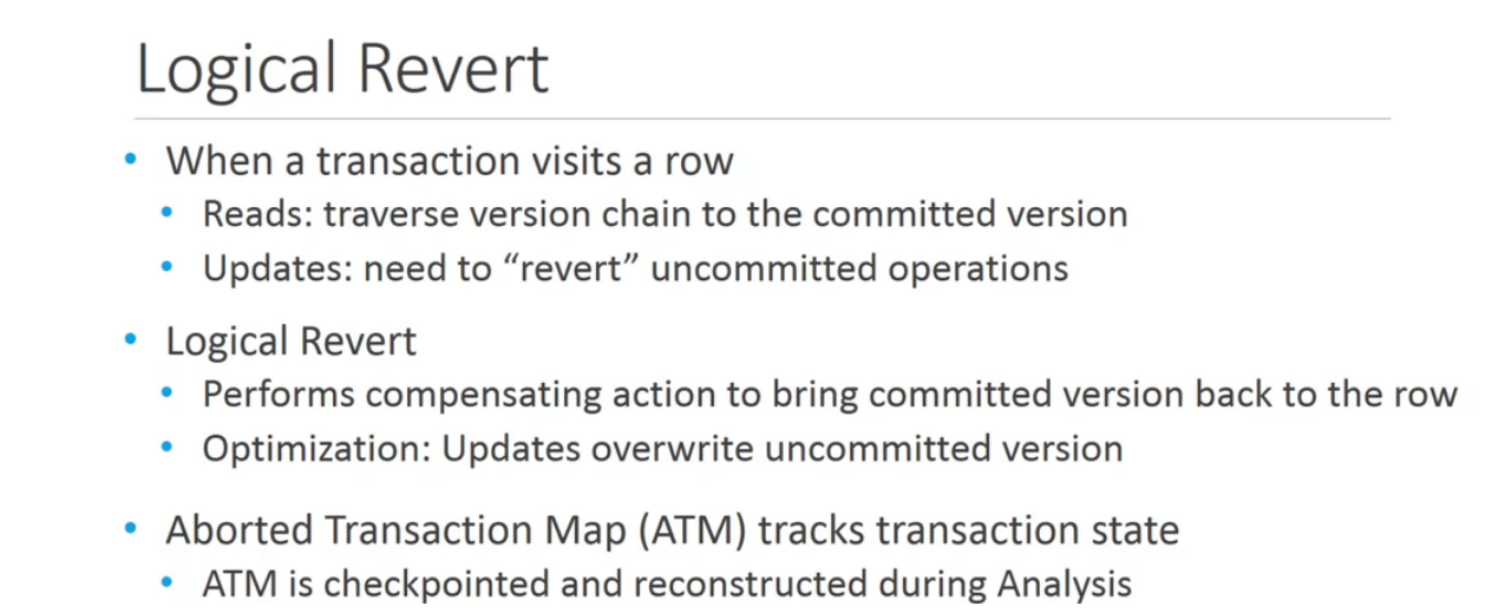


Logical Revert

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.



Transaction State Management (TSM)

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 onthe 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.

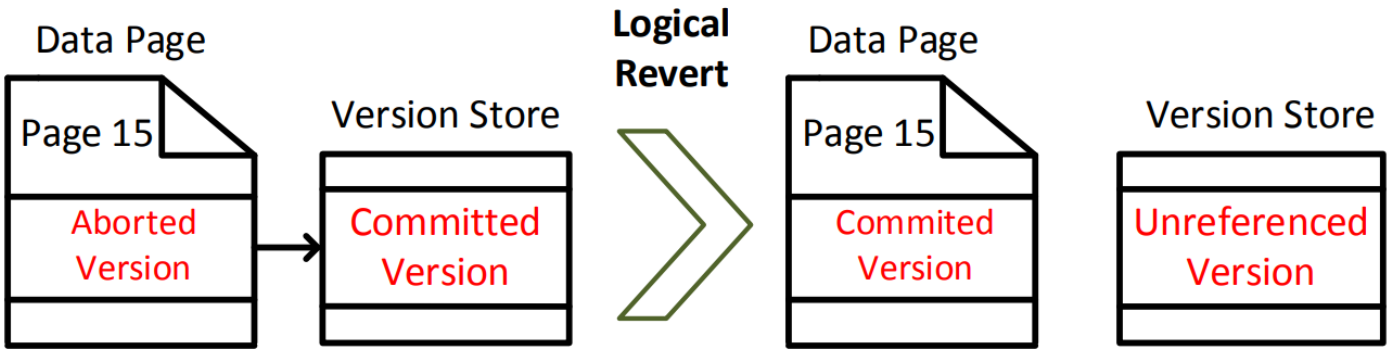


Figure 5. Example of a row before and after Logical Revert.

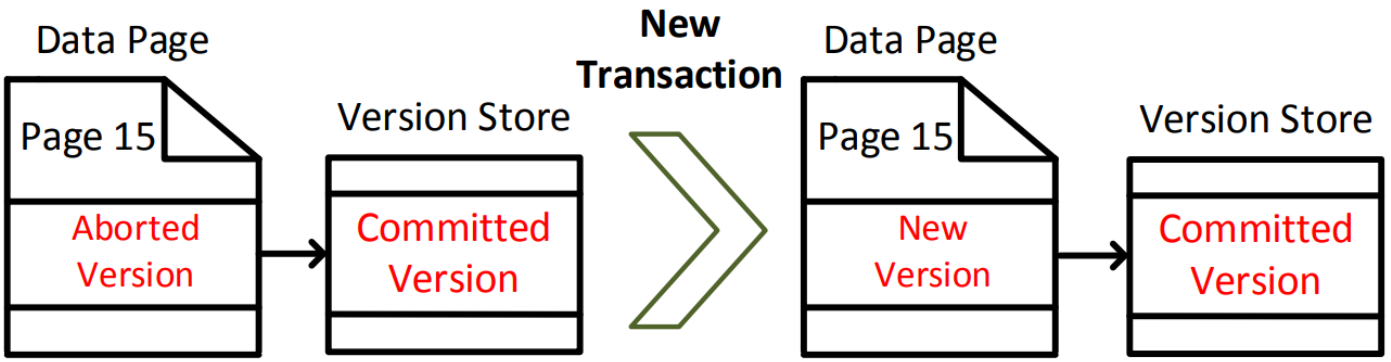


Figure 6. Optimization to overwrite an aborted version.

Reference

- Why & How CTR Works by SQL Engineering

How good have you found this content?