MD-MVCC: Multi-version Concurrency Control for Schema Changes in Azure SQL Database

Panagiotis Antonopoulos, Mansi Chauhan, Shailender Dabas, Rajat Jain, Darshan Kattera, Wonseok Kim, Hanuma Kodavalla, Nikolas Ogg, Prashanth Purnananda, Rahul Ranjan, Alex Swanson, Divyesh Tikmani
Microsoft
Redmond, WA, USA
{panant, chauhanmansi, sdabas, rajatj, dakm, wonkim, hanumak, niogg, praspu, rahranjan, alswanso, ditikman}@microsoft.com

ABSTRACT

As applications and data evolve over time, the database schema must be adjusted to accommodate their needs. Schema changes in relational databases have traditionally required synchronization with concurrent read and write access, causing significant interruptions to user applications. Although, most commercial databases have optimized common schema changes to reduce their runtime, they have not fundamentally addressed the requirement for synchronization which can lead to data being inaccessible for minutes or even hours in the presence of long running queries. MD-MVCC is a new technology in Azure SQL Database that enables multi-version concurrency control for schema changes. This allows schema changes to occur without any synchronization with concurrent queries which can operate on the earlier version of the schema until ongoing operations are committed, following Snapshot Isolation semantics. Schema deployments can now occur with minimal impact, increasing data availability but also flexibility for application developers. This required a comprehensive redesign of the schema management and metadata components of the RDBMS that are now multi-versioned across all layers, from query execution and in-memory caches to the system tables where metadata is persisted. This paper presents the overall design of MD-MVCC and demonstrates how it fundamentally improves data availability during schema changes without incurring any performance overheads.

PVLDB Reference Format:
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 Azure SQL Database. PVLDB, 18(12): 4791 - 4803, 2025.
doi:10.14778/3750601.3750605

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750605

1 INTRODUCTION

Schema management has been a long-standing challenge for relational databases. As applications evolve over time, the database schema must be adapted for their needs and this imposes a continuous burden for applications and developers:

1) Coordinating schema changes with the corresponding application upgrades is complex and error prone.
2) Schema changes have traditionally required synchronization with concurrent read and write access to the objects being altered, leading to extensive blocking and data unavailability.

The first problem has been extensively researched under the areas of schema versioning and evolution [5, 6, 7, 20, 27, 28, 29], with some commercial RDBMSs also providing solutions [24] in this space. Despite the challenges, we have noticed that users have been able to work around it by deploying changes incrementally and performing manual versioning when necessary, e.g. for views or stored procedures.

In this paper, we focus on the second problem which is a fundamental synchronization limitation at the core of the RDBMS and cannot be mitigated externally. This problem is particularly impactful for:

- Mission critical applications which operate 24/7 and cannot afford regular maintenance windows for application and schema upgrades. This has become increasingly common for many of our financial, retail and gaming customers.
- Reporting systems where there are overlapping, long-running queries, blocking any window available for schema changes. In this case, ongoing queries must be drained or aborted, and the database must be effectively taken offline for schema changes to be deployed.

Even though scenarios related to schema changes are generally excluded from the availability SLAs advertised by database cloud services, they incur significant interruptions to user applications. According to our telemetry, **every day** there are:

- More than 500 databases experiencing at least 5 minutes of continuous blocking due to schema changes.
- More than 50 databases experiencing at least 1 hour of continuous blocking due to schema changes.

These data points, combined with several incidents where schema changes caused hour-long blocking to some of our most critical customer workloads, motivated us to analyse this problem further and pursue a solution.

There are two cases where the synchronization required for schema changes causes extensive blocking:

1) The schema change itself requires a long time to complete, blocking concurrent user activity.
2) Long running queries or transactions are blocking a schema change from proceeding, which, in turn, blocks other activity on the objects being altered. This is an artifact of the "first in, first out" (FIFO) design of database lock managers which intends to avoid starvation of operations that require exclusive locks, like schema changes.

It is important to note that the latter scenario does not only impact user workload on the primary replica but also on readable secondary replicas, which are frequently used for long-running, reporting queries. In fact, the impact on secondaries is higher because a blocked schema change completely stalls the log replay (redo) process on the replica. This a) impacts the freshness of the data available on the secondary, b) prevents the transaction log from being truncated, potentially leading to the database running out of log space, but also c) significantly extends the duration of a potential failover since the replica is not caught up with the latest log and will need to redo all the pending operations before becoming the new primary.

Over time, SQL Server (the database engine serving Azure SQL Database) and other RDBMSs have incrementally improved common schema changes, like adding or removing columns, to make them instantaneous by only modifying the table metadata and avoiding size-of-data operations [12, 22, 26]. For operations that require processing all data, like constructing an index, many RDBMSs have introduced the notion of "online" schema changes [2, 9, 21, 25] where the tables are accessible for the majority of the operation with the exception of short synchronization points. Although both solutions reduce the impact of long running schema changes, they don't fundamentally address the need for synchronization which leads to the second blocking scenario described above. In 2014, SQL Server introduced the concept of Lock Priorities [14] which allow users to control how the lock manager prioritizes requests to minimize their impact to concurrent user workload. Even though this capability has been widely used, it requires careful tuning and falls short in the presence of long running queries.

MD-MVCC is a new technology introduced in Azure SQL Database that enables multi-version concurrency control for schema changes. This allows schema changes to occur without any synchronization with concurrent queries which can operate on the earlier version of the schema until ongoing schema changes are committed, following Snapshot Isolation semantics. User data now remains fully accessible for reads, throughout the duration of Data Definition Language (DDL) operations that modify the database schema. Considering data modifications are short-lived for the majority of applications and, therefore, do not cause extended blocking, our solution significantly improves data availability but also flexibility for application developers who can now deploy the necessary schema changes without the need for maintenance windows. Finally, our solution addresses the log replication challenges, described earlier, since replaying schema changes on secondary replicas no longer needs to synchronize with user queries executing there.

MD-MVCC is currently enabled on select customer workloads and will be deployed worldwide by the second half of 2025. This paper describes the overall design of MD-MVCC and is organized as follows: Section 2 provides background on traditional schema management and multi-version concurrency control in SQL Server. Section 3 outlines the architecture of MD-MVCC and describes the detailed design of the main components involved. Section 4 presents our experimental results and the real-world benefits we have observed while testing with production customer workloads. Finally, Section 5 concludes with our future plans to extend this technology further.

2 BACKGROUND ON SQL SERVER

This section provides an overview of the main components involved in schema management and data versioning in SQL Server. We expect other RDBMSs to have a very similar design in these areas which will render MD-MVCC applicable to those systems as well.

2.1 Metadata Manager

Metadata Manager (MM) is the main component involved in schema management. MM stores and serves all properties of every object in the database, such as tables, indexes, columns, views and more. It is technically divided between the Relational Engine (RE), which deals with the metadata of logical objects, like tables, columns, views, etc. and the Storage Engine (SE), which deals with physical objects, like rowsets (B+ trees or heaps). However, the overall design is consistent across both layers and we will, therefore, describe it as a single component. Figure 1 demonstrates the various layers of the MM architecture.

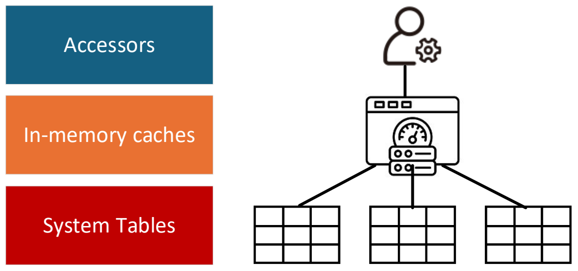


Figure 1. Metadata Manager Architecture.

At the lowest level, MM uses system tables to store the properties of each object. System tables are almost identical to regular, user tables. They are stored and accessed using the same storage and access methods with the exception that they are not accessible to users and have hard-coded schema that can only be modified through code deployments. The metadata describing all the properties of an object is serialized as a set of rows in various system tables. For example, each table will be represented by a row storing the table name and basic properties, multiple rows storing each of its columns and indexes and additional rows for other advanced properties. All of this information can then be read and deserialized to construct the full objects and their properties in memory.

Given that every query that uses an object needs to read its metadata, it would be highly inefficient to read all this disparate information across system tables and deserialize it for every access. Therefore, MM has a layer of in-memory caching where the loaded object metadata is stored and reused across operations. The information read from disk is deserialized into a class that is specific to the object type, e.g. table, function, view, etc. Complex objects, like tables, then contain sub-objects, like columns or indexes. Each of the top-level objects is stored in a hashtable-based cache, at the database level, which allows efficiently retrieving the objects based on their IDs, as the primary key, but also secondary keys, such as their names. The caching layer guarantees the transactional consistency between the in-memory and the on-disk states, even when transactions roll back. It is also responsible for invalidating the metadata caches of secondary replicas when schema changes occur on the primary. This is done by generating special "cache invalidation" log records that instruct the secondaries to refresh the corresponding cache entries.

Finally, on top of the in-memory cache lies a layer of accessors that expose the information of each object to external components, such as the query optimizer, while abstracting all the implementation details regarding caching, synchronization and the object's lifetime. Specifically, MM exposes all database metadata as an object model that external components can navigate, starting from top-level items, like a table, and then drilling into sub-objects, like columns or indexes. The accessor layer exposes a clean and safe interface for other components to interact with the objects managed by MM and has been critical for the health of the MM component, preventing external callers from accidentally corrupting its state or any shared objects.

2.2 Synchronization and Locking

Since MM has traditionally maintained a single version of metadata for each object, any schema changes must be fully synchronized with operations accessing the metadata of the objects being altered. The same is true for operations accessing the data of an object, such as rows of a table, since data access relies on the corresponding metadata to "parse" the physical format of each row and it is, therefore, unsafe for metadata to change in the middle of ongoing data access.

To guarantee the above synchronization, SQL Server relies on:

- A fully exclusive lock mode called "Schema Modification" (SCH-M), that is acquired by DDL operations on an object.
- A shared lock mode called "Schema Stability" (SCH-S) that is acquired by operations accessing the metadata of an object.

SCH-M is the most exclusive lock mode and incompatible with all modes, including SCH-S and SCH-M, but also the ones acquired for data access, such as Intent Shared (IS), used by SELECT queries, or Intent Exclusive (IX), used by data modifications. In this way, SCH-M allows DDL operations to fully synchronize with all schema and data access. On the contrary, SCH-S is compatible with all lock modes other than SCH-M. It is important to note that these locks are acquired at the object level (table, view, stored procedure, etc.) and only synchronize access for this specific object. [17] provides all the details about the various lock modes and their behavior.

The accessor layer of the MM component is responsible for acquiring these locks based on the specified intent (access or modification) from the caller. This allows MM to fully abstract the synchronization semantics from the calling components and enforce the necessary synchronization without relying on the caller.

2.3 Query Processing

Query Processing (QP) can be divided into two parts: Query Compilation (QC) and Query Execution (QE). QC is responsible for translating the user query into a physical execution plan. This plan can be cached to avoid repeated compilations of the same query when executed multiple times. To generate the query plan, QC needs to bind all objects referenced in the query to the corresponding database objects, such as tables or views. For that purpose, it must access the metadata of these objects to resolve their names but also load their properties, such as columns and indexes. All metadata information is accessed through the interfaces exposed by MM which will also acquire the necessary SCH-S locks. This prevents any concurrent schema changes, thereby stabilizing the metadata that QC will retrieve across the entire duration of the compilation process. Every object in metadata also tracks a timestamp of when it was last updated. As part of the query plan generation, QC will collect the timestamp for each object referenced by the query.

When query execution starts, QE will take the necessary locks for each referenced object (SCH-S or higher depending on the operation type and isolation level) and validate its current timestamp matches what is captured in the query plan. Since all locks are released at the end of query compilation, objects could have been modified between the time the query was compiled and executed. If that occurs, the query plan must be recompiled before execution begins. Note that the query plan could have been compiled much earlier and cached for efficiency, providing a large window for potential schema changes.

The locks acquired at the beginning of query execution are held until the end of the query or even the overall transaction, depending on the operation type and isolation level. This guarantees that the metadata of all objects is stable and the query can be executed safely. However, it also introduces significant blocking since a) queries must wait for ongoing schema changes to complete before they can acquire the necessary locks and b) schema changes are also blocked until ongoing queries, or transactions, complete and release their locks.

2.4 Multi-version Concurrency Control

Multi-version concurrency control was introduced in SQL Server 2005 to support Snapshot Isolation (SI). Versioning is performed at the row level: for every row update, SQL Server updates the row in-place and pushes the previous 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 with the transaction that generated it, using the Transaction Id, which is then associated with the commit timestamp of the transaction. Figure 2 provides an example of a row linked to two earlier versions.

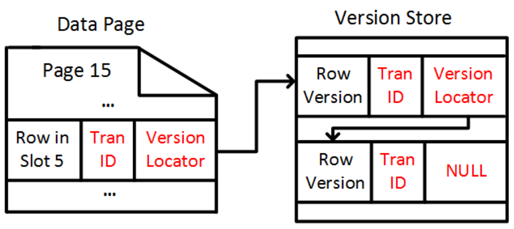


Figure 2. Example of MVCC row version chain.

Upon visiting a row, a snapshot 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. All this is done under page latches without the need to acquire page or row locks which are needed for synchronization in other isolation levels. This allows full concurrency between snapshot queries and data modification operations. It is important to note that Read Committed Snapshot Isolation (RCSI) is the default isolation level in Azure SQL Database and, therefore, the improved concurrency benefits the vast majority of queries.

Versioning was traditionally supported exclusively for user data. All MM layers only managed a single version of metadata for each object. None of the system tables, caches or accessors were versioned. Queries would always read metadata "as of now" and synchronize with any schema changes by acquiring, at least, a SCH-S lock to guarantee that the schema remains stable for the duration of the query. This was true even for SI or Read Uncommitted isolation which do not require locking and synchronization at the data level (pages/rows). In the case of SI, in particular, if a schema change occurs on an object between the time the transaction snapshot is established and the time the transaction accesses the metadata for this object, MM will detect this and raise an error to abort the transaction.

3 MD-MVCC

3.1 Overview

MD-MVCC is a new technology in Azure SQL Database that leverages multi-version concurrency control on metadata (MD) to allow full concurrency between queries and schema changes. It guarantees that "readers never block writers" (and vice versa) which earlier only applied to data access but not schema changes. This introduces significant benefits:

- Schema changes are no longer blocked by long-running queries, which have traditionally been the main source of blocking, and can proceed immediately without impacting new user requests. Schema deployments can, therefore, occur with minimal impact, increasing data availability but also flexibility for application developers who can now roll out their changes without downtime or the need for maintenance windows.
- Schema changes that are not supported as instant metadata updates or as "online" operations, such as adding a foreign key constraint, still allow read access for the full duration of the operation, both on the primary and secondary replicas.
- Replaying schema changes on secondary replicas is no longer blocked by ongoing queries. This a) avoids transaction log space growth, which often leads to outages due to running out of disk space and b) keeps the replicas up to date to guarantee data freshness and, more importantly, instant failover in the event of a failure, since the replica is fully caught up with the latest log.

To achieve this, we introduce the concept of versioned metadata and extend all layers of SQL Server interacting with it to make them version aware. Our locking scheme is also revamped to enable the new synchronization semantics required for the various isolation levels. In this section, we discuss the detailed design of our solution.

3.2 Multi-version Metadata Manager

In Section 2.1, we described the various layers of SQL Server's Metadata Manager. Each of these layers must be extended to support multiple versions of metadata.

3.2.1 MVCC for System Tables

Given that in-memory caches are transient (e.g. eviction due to memory pressure), system tables are the ultimate source of all object metadata. Therefore, their content must be versioned to support storing and serving different versions of the schema, as modifications occur. Since system tables use the same storage and access methods as user tables, we can reuse the data MVCC infrastructure, introduced in Section 2.4, to version their data. As schema changes modify object metadata, they update system tables, generating new versions associated with the transaction performing the schema change. Similarly, queries that need to retrieve the metadata of various objects can identify the right version based on the snapshot of their transaction. Although we can leverage our existing MVCC infrastructure, there are some challenges that are specific to versioning system tables:

1) **Synchronization and isolation semantics:** Before a user application can move from Read Committed (RC) to Snapshot Isolation (SI), it is critical to address any dependencies on a) the blocking semantics of RC or b) the need to access the latest committed data, both of which are not provided by SI. Traditionally, system table access had always used RC (or Read Uncommitted) isolation, regardless of the user transaction isolation level. Therefore, as we enabled SI for system tables, we had to go through the same evaluation for SQL Server's code base and make sure that we addressed any dependencies on the above properties of RC. Although most scenarios worked as expected, there were special cases where a) we had to force RC access to guarantee that the latest data is read, for example when reading the current value of a sequence object that must always be the latest or b) introduce additional locking to compensate for the lack of synchronization in SI, for example for metadata updates that previously relied on row level synchronization on the system tables.

2) **Recursive access:** Some of the storage engine system tables are used when accessing the version store, to load its metadata. Given that these system tables are now versioned, accessing their data might lead to accessing the version store, which introduces the possibility of recursive access to their pages, causing latching problems. To address that, we

handle version store metadata differently from regular tables: a) we preload all version store metadata during the database initialization and b) we maintain its metadata in memory, in all cases, to avoid having to access the problematic system tables.

3.2.2 In-memory caches

One of the most intricate areas of our work was introducing version support for the in-memory metadata caches, which had previously only stored the latest version of each object. As described in Section 2.1, the metadata cache stores complex objects that contain multiple sub-objects, such as a table with multiple columns. Based on that, we had the option to maintain versions of a) the top-level objects or b) individual sub-objects. The latter is theoretically optimal since it:

1) Allows for concurrent modifications to different sub-objects of the same top-level object which would otherwise cause a "fork" in the version chain of a top-level object and require some form of reconciliation when the modifications commit.

2) Requires storing multiple versions only for the sub-objects that are being modified, and not the full objects.

Despite its benefits, this approach requires special handling for each type of object and is prohibitively expensive to implement and test given the large variety of metadata objects and DDL operations supported by SQL Server (and most commercial DBMSs). Instead, we decided to generate versions at the top-level object granularity, such as a table or a view. This is an important design decision because it allowed us to introduce the concept of versioning directly in the cache infrastructure, without having to reason about individual object types and scenarios. We believe this is the right approach given that:

1) The vast majority of schema changes are serialized at the object level, i.e. there can only be one schema change for a specific object at a time. For these, versioning at lower granularity does not introduce any concurrency benefits. Cases of concurrent modifications to sub-objects are very limited (e.g. creating statistics on different columns of a table) and we handle them in a targeted way by versioning at the sub-object granularity, as needed.

2) The size of metadata is small (less than 100KB per object at P99) and schema changes do not occur at high frequencies to generate a large number of versions. So, the memory overhead for versioning top-level objects should be minimal.

Based on this design decision, we modified the cache structure to introduce the notion of "containers" that store a list of multiple versions for each object, each associated with the ID of the transaction that generated them. This is done for the main cache, that stores the metadata objects indexed on their primary key (object ID), but also for its secondary indexes, which index objects based on their name or other properties that can also be altered. An additional lock is introduced for each container to synchronize concurrent requests accessing or modifying versions of the object. Since read access is orders of magnitude more common than modifications, we opted for reader-writer locks which allow concurrent read access. Figure 3 provides an example of our caching structure after a schema change has occurred, creating a second version for an object.

At the high level, the versioning logic is simple: when a schema change occurs, instead of modifying the cached object in-place, it first generates a new version in the container, associated with the current transaction. It then modifies the newly created version which is invisible to any concurrent transactions until the schema change gets committed. If the transaction rolls back, any versions it generated are marked as "ghosted" and will be discarded by new transactions until the cleanup process eventually removes them (more details in Section 3.2.4). Similar logic applies when invalidating the caches of secondary replicas. The schema changing transaction generates special "cache invalidation" log records that notify the secondary replica to generate a new version associated with this transaction, instead of updating the earlier version in-place. This allows concurrent queries to safely access the earlier versions of the object metadata both on the primary and any secondary replicas.

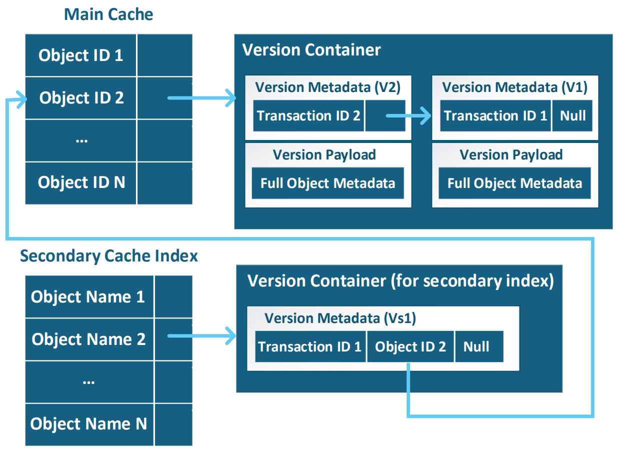


Figure 3. Example of the caching structure after a schema change has occurred.

Metadata retrieval also benefits from our decision to version top-level objects. The version lookup logic is introduced at the caching layer, returning the appropriate version of the entire object. Its sub-objects and properties can, then, be accessed directly without additional versioning logic. Specifically:

- When retrieving an object based on its primary key (object ID), we lookup directly into the main cache based on the key and locate the corresponding container (cases of cache misses are described below). We traverse the version chain and identify the visible version based on the snapshot of the transaction performing the retrieval and the corresponding version information. This logic is effectively identical to the MVCC version traversal described in Section 2.4.

- When retrieving an object based on a secondary key, such as the name, we first perform a version lookup on the secondary index based on that key. This retrieves the corresponding container and identifies the appropriate version using the same version traversal logic. From there, we retrieve the primary key of the object and perform the lookup on the main cache. It is important to note that the container of a specific secondary key might contain versions pointing to different primary keys, for example if there were two different objects with the same name at different points in time.

Since cached entries are transient and can be evicted under memory pressure, the cache eviction and reload logic must also be extended to support versioning. Specifically, each container must maintain information for all versions of the object to allow transactions accessing it to evaluate which version is visible according to their snapshot. This information must be preserved even if the full object metadata for a specific version is evicted. For that purpose, we decouple the "version metadata", which is needed for visibility checks, from the "version payload", which stores all the schema information for this version. When the system is under memory pressure, the payload of individual versions can be evicted, while the version metadata remains in the version chain. This is important because version metadata only contains the transaction ID and the lookup keys for the entry (e.g. the name and ID) whereas the version payload contains the full schema information and is significantly larger. The payload for evicted versions can be reloaded dynamically when requested by a transaction that must retrieve this version based on its snapshot. Figure 4 demonstrates the state of a container containing two versions when the payload of the older version has been evicted.

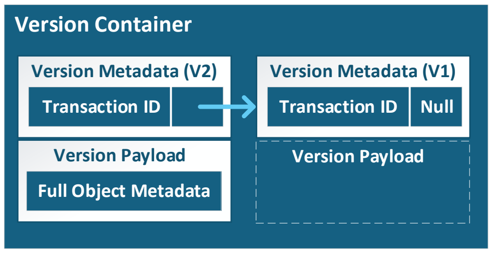


Figure 4. The state of a container with multiple versions under memory pressure.

Finally, since under heavy memory pressure, a container can be fully evicted, including all versions and their payload, we need the ability to reload all versions of an object from disk. To achieve that, we introduce a new access methods API that allows traversing the version chain of a specified system table row and retrieving the version metadata (Transaction ID, lookup keys) for all versions in the chain. When there is a cache miss for a specific object, this API is called on the system table row that corresponds to this object. It retrieves the necessary version information and populates the newly created container with all the versions, including their metadata but not their payload. This allows any transactions accessing this object to traverse the version chain, in-memory, and identify the version that is visible to them. Finally, the payload of the visible version is dynamically loaded to serve the transaction.

3.2.3 Accessors

Even though the MM component is now multi-versioned, across all its layers, we do not want to expose this complexity to external components that only intend to access or modify the schema of an object. From their perspective, they should simply interact with the object and its properties, without having to reason about its multiple versions and their visibility. The layer of indirection provided by the MM accessor interfaces allows us to abstract all this complexity.

When a caller expresses their intent to access metadata, they must provide the transaction under which the access will be performed. This protocol predates our work because the MM accessors need to acquire the necessary locks in the context of the provided transaction. We now leverage this to retrieve the transaction snapshot and identify the correct version of the objects the caller is requesting. The version lookup is only done when retrieving top level objects, such as a table, from the cache. Any sub-objects or properties are strictly associated with this version and can be directly accessed without any additional version traversals.

3.2.4 Version Garbage Collection

As older snapshot transactions commit or abort, older schema versions become permanently invisible and can be garbage collected. For row versions in system tables, this occurs automatically, leveraging the storage engine garbage collection (GC) mechanism of the MVCC infrastructure that was earlier only responsible for cleaning up user tables. This identifies the oldest active snapshot in the system and then visits each versioned row to evaluate what is the version that is visible to it. It then removes any older versions since they are guaranteed to be invisible to all active transactions.

For the in-memory MM caches, we introduce a new GC process. This wakes up periodically, every few seconds, and visits all containers in the cache. Similar to the storage engine GC, it traverses the version chain in each container and identifies the version that is visible to the oldest snapshot transaction in the system. Any older versions can be safely discarded. This process is also responsible for removing versions that correspond to aborted transactions. These versions are clearly marked as "ghosted" and can be safely removed.

3.3 Synchronization and Locking

As part of our Constant Time Recovery (CTR) [1] work, any schema changes that modify the underlying data, such as altering the type of a column, were adjusted to perform these modifications in a versioned manner, leveraging the pre-existing MVCC infrastructure. With MD-MVCC, metadata is also versioned, allowing snapshot queries to access a stable version of the schema, even in the presence of ongoing schema changes. By combining these two technologies, snapshot queries can now safely access the data versions that are visible to their transaction and interpret them using the corresponding snapshot of metadata. This enables full concurrency between snapshot queries and schema changes but the locking scheme must be adjusted to allow that. Our locking scheme needs to satisfy the following requirements:

1) Only one schema change should be allowed for an object at any point in time.
2) Only snapshot queries and query compilation should be allowed to execute while there are ongoing schema changes on the object being accessed. All other isolation levels must be blocked. Data modifications must also be blocked.
3) Some operations cannot be versioned and must block even snapshot queries. For example, shrinking the database files which temporarily affects the physical structure of indexes.

To satisfy these requirements, we preserved the existing SCH-S and SCH-M lock modes but also introduce SCH-A ("Schema Access") and SCH-C ("Schema Create"). SCH-C is an exclusive lock that conflicts with all lock modes other than SCH-A. SCH-A is a shared lock that only conflicts with SCH-M. Table 1 demonstrates the compatibility matrix for these lock modes.

Table 1. Lock mode compatibility matrix. (C) indicates conflict and (N) indicates no conflict.

	SCH-A	SCH-S	Other modes	SCH-C	SCH-M
SCH-A	N	N	N	N	C
SCH-S	N	N	N	C	C
Other modes	N	N	…	C	C
SCH-C	N	C	C	C	C
SCH-M	C	C	C	C	C

These lock modes are used as follows:

- Snapshot queries and query compilation acquire a SCH-A lock on the objects they access.
- Queries using other isolation levels acquire SCH-S or more restrictive locks on the accessed objects. Data modifications also acquire more restrictive locks, like IX.
- Schema changes acquire a SCH-C lock on the objects they modify. This allows them to run concurrently with snapshot queries but not queries using other isolation levels, data modifications or other schema changes.
- Non-versioned operations acquire SCH-M on the associated objects to fully synchronize with all other operations.

This logic allows us to meet our requirements and achieve the necessary synchronization depending on the operation types and isolation levels.

3.4 Snapshot Management

In this section, we discuss how snapshots are managed in different stages of query execution.

3.4.1 Query Compilation

Since compiling a query involves retrieving the schema of multiple objects, some of which might be connected to each other, such as tables linked with a foreign key, query compilation must have a consistent view of the schema of the objects it accesses. Otherwise, the schema inconsistencies might lead to access violations or unexpected results since references between the objects could be broken. For example, in a foreign key relation, if the referenced column is dropped while the query is compiling, we could attempt to retrieve a column that no longer exists.

Traditionally, schema locks acquired during query compilation protected us from such inconsistencies. However, since MD-MVCC allows schema changes to occur while the corresponding metadata is accessed, we must address this problem differently. Specifically, query compilation will now establish a snapshot that is used throughout the compilation process and retrieve all metadata using that. This guarantees that all properties and relations for all objects referenced by the query are accessed as of a consistent point in time. If the compilation occurs in the context of an SI transaction, the transaction snapshot is used for the query compilation to guarantee that we compile the query with the same snapshot as the one that will be used during execution. In the case of RCSI or other isolation levels, query compilation will establish its own snapshot that will only be used for compiling the query.

It is important to note that, with MD-MVCC, query compilation will always use snapshot semantics, even for isolation levels that might not use snapshot when executing the query. This design choice was made for two reasons:

1) Query compilation only accesses metadata and its semantics are not impacted by the isolation level that will be used during query execution. Therefore, using snapshot allows for improved concurrency, since no query compilation will need to synchronize with ongoing schema changes.
2) In many cases, the isolation level that will be used at query execution is not known during compilation. Therefore, we do not want query compilation to block behind a schema change for a query that will later use snapshot and execute successfully.

3.4.2 Query Execution

To safely access a table undergoing schema changes, a transaction must have a consistent view between its data and metadata, so that the data can be interpreted correctly. For that purpose, we establish a single snapshot at the beginning of query execution and use that throughout the query. In the case of SI, we first check if a snapshot has already been established for the transaction and use that. If not, a new snapshot is established and associated with the transaction for the entirety of its lifetime. In the case of RCSI, a new snapshot is established for every query and only used for the duration of that query.

As described in Section 2.3, at the beginning of query execution, we first check whether the timestamp of each referenced object matches the one captured in the query plan we are attempting to use. If not, the query must be recompiled. This check must occur using the same snapshot that will be used for executing the query. This guarantees that we perform the checks based on the exact version of the schema that we are going to use to access the data during query execution.

3.5 Isolation Level Support

Similar to other RDBMSs, SQL Server supports a variety of isolation levels and even allows users to combine them by using isolation level hints [16]. Over the years, this capability has become critical for a large number of applications, whose correctness depends on the specified isolation semantics, both in terms of synchronization (e.g. blocking in Read Committed (RC)), as well as data visibility (e.g. latest committed version in RC). Because of that, preserving these semantics in all scenarios is a key requirement for our solution.

3.5.1 Session Isolation Level

The majority of applications configure the isolation level at the session (connection) level, changing that for different transactions and stored procedures to meet their needs. The session level setting remains consistent for query compilation and execution, allowing us to compile and execute the query with the same semantics. The locking scheme described in Section 3.3 guarantees the necessary blocking semantics: Only snapshot transactions (SI/RCSI) will acquire a SCH-A lock and execute concurrently with schema changes, whereas transactions in other isolation levels will be blocked. In terms of data visibility, MD-MVCC is only employed for snapshot transactions. Transactions in other isolation levels will access the latest version of the schema, as they have historically done.

3.5.2 Table Level Hints

Isolation level hints at the table level introduce additional complexity since they only control the isolation semantics for individual tables and not the overall query and its compilation, which occurs based on session level setting. This introduces the risk that the version of the schema used to compile and execute the query might be different than the version of the data the query will access in a specific table, leading to correctness issues. Table 2 indicates the problematic combinations of isolation levels set at the session level and through query hints. Specifically, the risk is present when the session isolation level uses different snapshot semantics than the query hint. Even between SI and RCSI, the snapshots are established at different points in time and can, therefore, lead to inconsistencies.

Table 2. Problematic combinations of session isolation levels and query hints.

	Isolation Level Hint		
Session Isolation Level	SI	RCSI	Other iso. levels
SI	-	X	X
RCSI	X	-	X
Other isolation levels	X	X	-

To eliminate this problem, when we detect such isolation level mismatch, we perform additional checks at the beginning of query execution to verify that the version of the schema as seen by the snapshot access is the same as the latest version of the object, as seen by the non-snapshot access. This is done by comparing the latest timestamp of each object to the timestamp visible by the snapshot. If the former is higher, it means the object has been altered after the snapshot was established and, therefore, access is unsafe. In the case of SI, this results in an error that aborts the transaction. In the case of RCSI, however, since the snapshot is established at the query level, we can internally retry the query without any user impact by establishing a new snapshot. This is important because RCSI is the default isolation level, used by more than 70% of queries, and this optimization can significantly improve the user experience.

3.6 Deferred Data Deallocation

With MD-MVCC, a snapshot transaction can potentially access a table (or index) that has been dropped if the transaction snapshot was established before the transaction dropping the table was committed. When a table is dropped, the underlying data pages must be deallocated, so that the space can be reclaimed and reused for other purposes. Once a page is allocated to a new table, it gets formatted to a clean state and any earlier data is permanently erased. If pages were synchronously deallocated when a table is dropped, earlier snapshot transactions that can still access the table might attempt to access data pages that are no longer valid.

To address that, we leverage the concept of "deferred data deallocation". When a table is dropped, the underlying data pages are not synchronously deallocated but, instead, the table gets registered for deferred deallocation. A background task periodically checks for tables that are pending deallocation and evaluates if they might still be visible to active snapshot transactions. This is done by comparing the commit timestamp of the transaction that dropped each table to the minimum snapshot timestamp across all active transactions in the system. Once older snapshot transactions commit or abort, the minimum snapshot timestamp advances and, eventually, the background task can safely deallocate the pages of any dropped tables.

3.7 Query Plan Cache

As mentioned in Section 2.3, once a query is compiled, it can be cached and reused for multiple executions. Traditionally, both query compilation and execution always accessed the latest version of the schema. Based on that, the query plan cache maintained only one query plan for each query. With MD-MVCC, however, the schema version used by query compilation or execution might not be the latest. In fact, two concurrent snapshot transactions with different snapshots might be using different versions of the schema. When these transactions attempt to execute the exact same query, they need to use a different query plan and, if the right plan is not available in the cache, they would need to recompile, leading to increased recompilations.

One way to address this scenario would be to allow the plan cache to preserve multiple query plans for every query, each reflecting a different version of the schema. Although this solution minimizes query recompilations, it would require a broader redesign of the plan cache to introduce a version chain for each plan, similar to what was done for the metadata caches. Furthermore, since a query can reference multiple objects, each of which has a different version history, we would need to globally order the versions of the plan based the referenced object versions. Snapshot transactions would then identify the appropriate plan version by checking the commit timestamps associated with every object referenced by the query. This would introduce significant complexity but also increase the memory footprint of the plan cache, which is already one of the highest memory consumers in our service. To avoid that, we decided to retain the current behavior where the cache only stores a single plan for each query. This could theoretically cause repeated recompilations, if two transactions with different snapshots repeatedly executed the same query. However, in reality, this should be an extremely rare case because:

- In RCSI, which is the default isolation level, snapshots are established at the beginning of the query and are, therefore, very recent, reducing the probability of a version mismatch. Even if a mismatch is identified and the query is recompiled, a new snapshot will be established allowing the query to use the latest version and proceed.
- In SI, although there is a possibility of repeated recompiles among two transactions, it is very uncommon that two

long-lived snapshot transactions would repeatedly execute the same query.

Based on this rationale and the fact that we have not identified any problematic cases as we enabled MD-MVCC in production, we believe that our approach is sufficient to address this scenario without introducing additional complexity.

3.8 Security

Although our goal is to apply snapshot isolation across schema and data to improve concurrency, we must guarantee that the latest security configuration is always applied for all transactions, regardless of when their snapshot was established. Otherwise, users with old active transactions could, maliciously or accidentally, execute queries bypassing the latest security settings. The MD-MVCC design needs to account for these semantics for the following reasons:

1) Schema properties of tables and other objects affect the authorization checks applied when they are accessed. For example, a table might have Row Level Security (RLS) [15] or Dynamic Data Masking [13] enabled, which limit the data each user can access. Similarly, objects have a specific user assigned as their owner. Owners can grant other users indirect access to objects they own through views or stored procedures. This capability is known as Ownership Chaining [18]. Ownership information is also stored as part of the object metadata.

2) Various security related entities and configurations, including users, roles, permissions and auditing settings, are stored as database metadata, fully managed by the Metadata Manager component, as any other schema information.

By default, MD-MVCC would version all this information and incorrectly apply snapshot semantics for authentication, authorization or auditing purposes, violating the principle that all transactions should access the latest version of any security information. A naïve solution to this problem would be to detect any modifications to security information and abort snapshot transactions that attempt to access it. However, this would lead to a large number of transaction aborts for scenarios that can simply access the latest metadata and execute successfully. For example, changes to role membership or the permissions of a role are common and would cause unnecessary aborts when accessing the latest version of this information is perfectly safe.

To address this issue, we allow the Metadata Manager to suppress snapshot isolation when exposing security related information and return its latest committed version. This, however, creates the risk that a transaction might see an inconsistent state between elements that are accessed with different isolation semantics. We avoid this problem by introducing the ability to selectively abort snapshot transactions based on what modifications have been made to the entities they are accessing. Specifically, for each schema or security modification, we decide whether it should abort snapshot transactions based on the following logic:

1) We distinguish regular, versioned objects, like tables, where snapshot isolation applies, from non-versioned security metadata, where snapshot isolation should not apply.

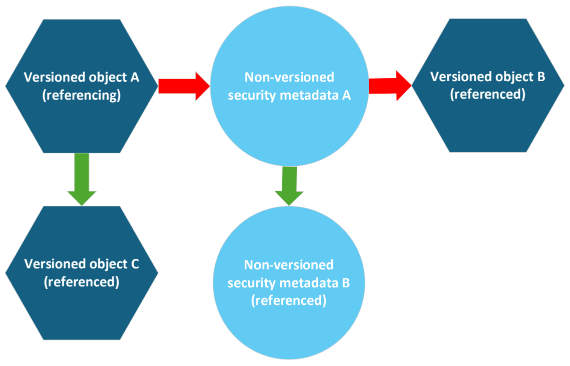
2) We construct a directed acyclic graph (DAG) which has the above entities as nodes and their relations as edges. The referencing entities point to the referenced ones. For example, a table (versioned) references an RLS policy (non-versioned) which in turn references a function (versioned) that applies the necessary security logic. Figure 5 demonstrates a sample graph with all the possible relationships between versioned and non-versioned entities.

3) Modifications to properties of an entity (versioned or not) that do not affect its relations with other entities should not abort any transactions. For example, users can safely modify the nullability of a column (versioned) or the destination of audit records (non-versioned). For the following scenarios, we only consider modifications that affect relations of an entity, such as dropping the entity the relation relies on.

4) Modifications affecting relations between two versioned or two non-versioned entities should not abort any transactions since entities of the same type would be accessed using the same isolation semantics.

5) Modifications to a non-versioned entity that is referenced by a versioned entity must abort snapshot transactions that access an earlier version of the versioned entity since it is unsafe for them to access the latest state of the non-versioned entity. For example, dropping the RLS policy (non-versioned, security metadata) of a table (versioned) should abort transactions accessing the table with a snapshot established before this modification occurred.

6) Modifications to a versioned entity that is referenced by a non-versioned entity must abort snapshot transactions that access an earlier version of the versioned entity since it is unsafe for them to access its earlier state. For example, dropping a table (versioned) that is referenced by permissions granted to various users (non-versioned) should invalidate snapshot transactions that started before the table was dropped since it is unsafe to access the inconsistent state between the two entities.



Versioned object A (referencing)
Non-versioned security metadata A
Versioned object B (referenced)
Versioned object C (referenced)
Non-versioned security metadata B (referenced)

Figure 5. A DAG representing various schema and security entities and their relationships. Relationships that can trigger snapshot transaction aborts are indicated in red whereas safe relationships are indicated in green.

To selectively abort the necessary transactions, we maintain the commit timestamp of the last operation that performed an unsafe modification (per the rules above) for each entity. When a snapshot transaction accesses an entity, we check if the snapshot timestamp is lower than the timestamp for this entity. If so, the

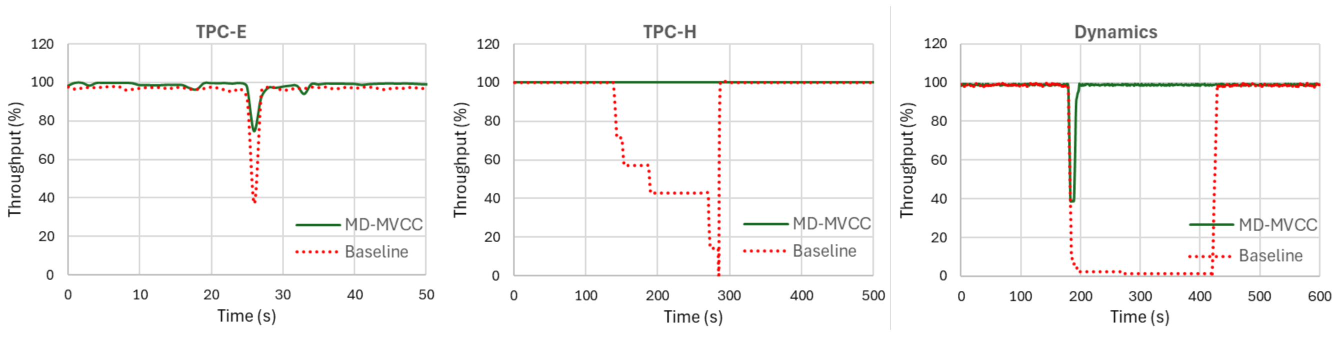


Figure 6. The impact of a DDL operation with and without MD-MVCC for different types of workloads.

transaction is aborted. It is important to note that the transaction is fully aborted only in the case of SI. In RCSI, which is more commonly used, transaction aborts can be avoided. The timestamp checks occur before the query execution begins and, if there is a need to abort, we establish a new snapshot and retry the operation without any visible interruptions to the users.

4 EXPERIMENTAL AND PRODUCTION RESULTS

This section presents experimental results regarding the performance of the system when MD-MVCC is enabled. All our experiments are executed on a workstation with 2 sockets, 52 cores (Intel® Xeon® Platinum 8171M Processor, 2.60GHz) and 575GB of RAM. External storage consists of six 1TB SSDs.

4.1 Concurrency During Schema Changes

In our first set of experiments, we evaluate how MD-MVCC improves concurrency during schema changes and reduces their impact to the user workload. Given that all common DDL operations are either instant, through a metadata update, or implemented as "online" operations, they only acquire an exclusive, SCH-M, lock for a very short window of time, in the order of few milliseconds. As described in Section 1, most of the blocking is introduced due to long running queries that cause the DDL to wait, further blocking new requests. Based on that, the exact type of DDL does not affect this experiment and we will demonstrate the concurrency improvements using an ADD COLUMN operation, which is the most common operation according to our production telemetry. However, the results would be identical for any other DDL, like dropping a column or creating/dropping indexes.

On the contrary, the concurrency benefits heavily depend on the shape of the workload, the ratio between reads and writes and the duration of these operations. Based on that, we evaluate MD-MVCC using three different types of workloads:

1) An operational workload that simulates TPC-E [33] and involves short read and write operations. While the workload is running, we add one column to the "TRADES" table which is heavily used by most operations.
2) An analytical workload that simulates TPC-H [34] and only executes long-running read queries. While the workload is running, we add one column to the "LINEITEM" table which is used by several queries.
3) A production workload from Microsoft Dynamics, one of the biggest customers of Azure SQL Database. This is a workload from the Finance and Operations service of Dynamics that combines operational reads and writes with reporting queries. While the workload is running, we add a column to one of the tables that is used by both the operational workload as well as reporting queries.

Figure 6 demonstrates how each workload was impacted by the DDL operation with and without MD-MVCC:

- In the TPC-E workload, the improvement with MD-MVCC is marginal. All operations are short (under 1 second) and do not introduce extensive blocking. The throughput drops for a small window of time when the DDL is executed but quickly recovers, as ongoing queries complete, allowing the DDL to acquire the exclusive lock and complete the operation. Since MD-MVCC allows read queries to proceed while the DDL is waiting and executing, the drop is less pronounced there.
- In the TPC-H workload, MD-MVCC introduces a significant improvement. Given that the workload only includes read queries, MD-MVCC allows them to execute concurrently with the DDL, eliminating any impact to the user workload. In the baseline run, however, long-running queries cause the DDL to wait, blocking all new requests for over 150 seconds until all ongoing queries complete.
- In the Dynamics workload, we also see a significant improvement with MD-MVCC. There is still a drop in throughput when the DDL is executed, but it is not as deep and only lasts less than 10 seconds, compared to more than 200 seconds in the baseline run. This is because of the relatively short duration of data modifications as compared to long-running reporting queries. With MD-MVCC, all read queries were able to continue uninterrupted without any blocking. Data modifications blocked the DDL for a short time, also blocking new requests, but, then, the DDL completed quickly and the workload recovered in seconds. On the contrary, in the baseline run, reporting queries caused the DDL to wait for more than 3 minutes, blocking any new requests for an extended amount of time.

Although the exact improvements vary depending on the profile of the workload, in all cases, MD-MVCC improves data availability and minimizes the impact of schema changes to the user workload.

4.2 User Workload Performance

As a next step, we evaluate how MD-MVCC affects the performance of the system during online transaction processing. Metadata is accessed during query compilation and execution, so any additional overheads could negatively impact performance. We measure the throughput and latency of the user workload with and without MD-MVCC to assess if our versioned algorithms introduce any substantial overheads or scalability bottlenecks that would impact the workload.

4.2.1 Throughput

Since all query execution must access metadata and will, therefore, exercise the MD-MVCC logic, we evaluate the throughput of the system using two high-concurrency, OLTP workloads that are simulating TPC-C [32] and TPC-E. In these experiments, we do not perform any schema changes as we want to evaluate the throughput of the system in steady state. Given that OLTP workloads are highly repetitive, all query plans and row accessors should be cached and reused across executions. However, metadata must still be accessed to validate that the query plan and accessors are consistent with the visible version of the schema. Thus, these workloads will stress the MD-MVCC cache retrieval and synchronization logic.

Table 3 presents the throughput degradation introduced by MD-MVCC for our TPC-C and TPC-E-like workloads. Since TPC-C queries are extremely short, metadata access will be very frequent and we consider this a worst-case scenario, whereas TCP-E represents a more common type of workload. Given that there is some variability in the measured throughput, we also capture the standard deviation for our runs. The impact of MD-MVCC is minimal for both workloads and well below the standard deviation. This indicates that our algorithms do not introduce any visible overheads or scalability bottlenecks.

Table 3. Throughput degradation for TPC-C and TPC-E.

Workload	Degradation	Std Deviation
TPC-C	0.2%	1.8%
TPC-E	0.18%	1.5%

4.2.2 Latency

For this set of experiments, we measure the impact of MD-MVCC on the latency of query compilation and execution. Given that metadata is accessed in the same way for all objects referenced by the query, regardless of the query complexity or how objects are used, we execute a simple query that performs a "UNION ALL" among N tables containing ten columns and only one row each. Our goal is to capture a worst-case scenario where the cost of compiling and executing the query is minimal and metadata access represents a bigger portion of the overall execution time. Before executing the queries, we perform a schema change on the tables so that the query must traverse two versions, further increasing the cost of metadata access.

Figure 7 demonstrates the latency of query compilation and execution with and without MD-MVCC for a varying number of tables and depending on whether the metadata cache is already populated. As we expected, compilation and execution times are proportional to the number of tables used in the query. MD-MVCC does not introduce any noticeable overhead when the metadata cache is warm. The right version of the object is quickly identified in the in-memory version chain and the query executes as usual. When the metadata cache is cold, we can see that there is an additional overhead of ~200us to load the necessary metadata for each table. This increases by another ~20us when MD-MVCC is enabled which is caused by the additional cost of performing a snapshot scan on the system tables and loading the "version metadata" for the different versions of the object, as described in Section 3.2.2. Overall, even in this worst-case scenario, the overhead of MD-MVCC is negligible in all cases.

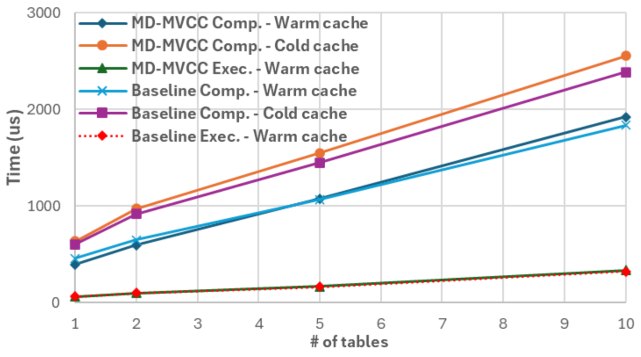


Figure 7. Latency of query compilation and execution for varying number of tables and state of the metadata cache.

4.3 Garbage Collection Performance

As described in Section 3.2.4, as older snapshot transactions commit/abort, earlier schema versions become permanently invisible and are lazily cleaned up by a background garbage collection (GC) process. In this experiment, we evaluate the performance of our GC process in the presence of heavy volume of DDL operations. Specifically, we use a common pattern we have observed in production where users will frequently create and drop tables that are used as temporary tables by the workload. We establish 32 connections that create and drop such tables in a loop and measure the number of versions in our metadata caches over time. Given that we do not introduce any long-running snapshot transactions, we expect GC to periodically collect all earlier versions.

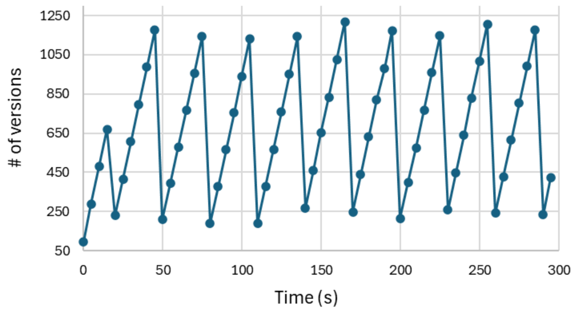


Figure 8. Number of metadata versions over time under heavy DDL workload.

Figure 8 presents the number of metadata versions over time under this heavy DDL workload. As we anticipated, the GC process periodically wakes up and removes all invisible versions, guaranteeing that the memory footprint of the cache remains bounded.

5 FUTURE WORK

5.1 Concurrency with Data Modifications

Although our initial goal is to enable concurrency between schema changes and read queries, our longer-term plan is to leverage the MD-MVCC capabilities to also support concurrency with data modifications. This would allow schema changes to be deployed without any data unavailability or impact to the user workload. In the current implementation of MD-MVCC, schema changes a) synchronize with data modifications using locks and b) introduce write-write conflicts between data modifications and schema changes. Although schema changes and data modifications are conceptually both updating the same object, some of the most common schema changes do not technically perform conflicting modifications with data updates. For example, adding or dropping a column are only updating the metadata of the table and would not conflict with concurrent updates, neither physically nor semantically. The same is true for creating or dropping an index. Dropping an index only removes the index from the table metadata and enqueues it for deferred deallocation (see Section 3.6). Even though creating an index needs to insert all rows to the new index, hence performing some row updates, per our Online Index Build algorithm [2], this is only done for rows that have not been updated since the index build started and, therefore, no conflict can occur. Based on that, we plan to extend MD-MVCC to allow concurrency with data modifications for some of the most popular schema changes needed by applications.

5.2 AS OF Queries

The ability to execute queries "as of" a certain point in time has been a common ask from our users. This is typically needed for analytical queries that are used to generate financial or other reports as of a specific point in time (e.g. last day of the calendar or fiscal year). By leveraging our data MVCC infrastructure, we can retrieve the version of the data at the specified point in time. However, as the schema of the tables evolves, the latest version of the schema might no longer be compatible with that version of the data. For example, columns could have been added, dropped or altered, making it impossible to interpret an earlier version of the data. MD-MVCC allows us to access the right version of the schema for the point in time the query specifies. In this way, schema and data can be consistent and safely serve any query. Based on that, we plan to leverage MD-MVCC to enable AS OF query capabilities.

6 RELATED WORK

Database schema management has been an active area of research for over three decades with main focus on:

a) Reducing the blocking introduced by schema modifications [10, 11, 19, 30, 31, 35, 37].

b) Schema evolution which enables different versions of the application to operate on different versions of the schema, simplifying the deployment of new application code that requires schema changes [5, 6, 7, 20, 27, 28, 29].

Although there has been extensive work in the academia around schema evolution, Oracle is the only commercial RDBMS that currently provides a native capability for managing schema versions [24]. This is largely because application developers have managed to work around this challenge by carefully orchestrating the deployment of their application code together with the corresponding schema changes.

On the contrary, the data unavailability caused by schema changes which block concurrent user workload is a fundamental problem that the RDMBS needs to address. Over the years, there have been significant advancements across all database products to either a) support common operations, like adding/dropping a column, as an instant metadata update [12, 22, 26] or b) make schema changes "online", where the operation must still process all data but can allow user access for the majority of its duration [2, 9, 11, 21, 25]. Although these solutions reduce the blocking introduced by schema modifications, they still need to synchronize with concurrent access, at least for a small window of time. This introduces extensive blocking in the presence of long-running queries which force the schema changes to wait and, in the meantime, block all new requests.

MD-MVCC leverages MVCC and the semantics of Snapshot Isolation to eliminate all synchronization between schema changes and queries, and support full concurrency. MVCC has been studied extensively since the 80s [3, 4, 36] but mainly in the context of user data. MD-MVCC exploits the same ideas but for metadata to address concurrency for schema changes. Tesseract [8] employs a similar technique to allow both reads and writes in the presence of ongoing schema changes but is not integrated in a commercial RDBMS and, therefore, does not address the complexities introduced by the large variety of DDL operations, security and other features in these environments. Oracle is the only other commercial RDBMS to support concurrency between queries and schema changes [23]. However, it does not address the scenarios around different isolation levels and how their semantics can be preserved, given the more limited isolation level support in their system as compared to SQL Server and other popular RDBMSs. Unfortunately, details about the Oracle implementation are not available to further compare the two solutions and their trade-offs. Finally, our algorithm for reasoning about security, presented in Section 3.8, is a novel contribution that has not been covered by prior work in this space and is critical for correctly applying access control under MVCC.

ACKNOWLEDGMENTS

We would like to thank all members of the MD-MVCC 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.

REFERENCES

[1] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant time recovery in Azure SQL database. Proc. VLDB Endow. 12, 12 (August 2019), 2143–2154. https://doi.org/10.14778/3352063.3352131

[2] Panagiotis Antonopoulos, Hanuma Kodavalla, Alex Tran, Nitish Upreti, Chaitali Shah, and Mirek Sztajno. 2017. Resumable online index rebuild in SQL server. Proc. VLDB Endow. 10, 12 (August 2017), 1742–1753. https://doi.org/10.14778/3137765.3137779

[3] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In Proceedings of the 1995 ACM SIGMOD international conference on Management of data (SIGMOD '95). Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/223784.223785

[4] 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

[5] Souvik Bhattacherjee, Gang Liao, Michael Hicks, and Daniel J. Abadi. 2021. BullFrog: Online Schema Evolution via Lazy Evaluation. In Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21). Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/3448016.3452842

[6] Carlo A. Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2010. Update rewriting and integrity constraint maintenance in a schema evolution support system: PRISM++. Proc. VLDB Endow. 4, 2 (November 2010), 117–128. https://doi.org/10.14778/1921071.1921078

[7] Kai Herrmann, Hannes Voigt, Andreas Behrend, Jonas Rausch, and Wolfgang Lehner. 2017. Living in Parallel Realities: Co-Existing Schema Versions with a Bidirectional Database Evolution Language. In Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17). Association for Computing Machinery, New York, NY, USA, 1101–1116. https://doi.org/10.1145/3035918.3064046

[8] Tianxun Hu, Tianzheng Wang, and Qingqing Zhou. 2022. Online schema evolution is (almost) free for snapshot databases. Proc. VLDB Endow. 16, 2 (October 2022), 140–153. https://doi.org/10.14778/3565816.3565818

[9] IBM, IBM DB2, Rebuild Index Online Utility. https://www.ibm.com/support/knowledgecenter/en/SSEPEK_11.0.0/ugref/src/tpc/db2z_utl_rebuildindex.html

[10] Jørgen Løland and Svein-Olaf Hvasshovd. 2006. Online, non-blocking relational schema changes. In Proceedings of the 10th international conference on Advances in Database Technology (EDBT'06). Springer-Verlag, Berlin, Heidelberg, 405–422. https://doi.org/10.1007/11687238_26

[11] Meta, Online Schema Change for MySQL. https://www.facebook.com/notes/10157508558976696

[12] Microsoft, Add columns as an online operation in SQL Server. https://learn.microsoft.com/en-us/sql/t-sql/statements/alter-table-transact-sql?view=sql-server-ver16#adding-not-null-columns-as-an-online-operation

[13] Microsoft, Dynamic data masking. https://learn.microsoft.com/en-us/sql/relational-databases/security/dynamic-data-masking?view=sql-server-ver16

[14] Microsoft, How It Works: SQL Server Locking WAIT_WITH_LOW_PRIORITY. https://techcommunity.microsoft.com/blog/sqlserver/how-it-works-sql-server-locking-wait-with-low-priority/3122713

[15] Microsoft, Row-level security. https://learn.microsoft.com/en-us/sql/relational-databases/security/row-level-security?view=sql-server-ver16

[16] Microsoft, Table hints (Transact-SQL). https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-table?view=sql-server-ver16

[17] Microsoft, Transaction locking and row versioning guide. https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver16

[18] Microsoft, Tutorial: Ownership Chains and Context Switching. https://learn.microsoft.com/en-us/sql/relational-databases/tutorial-ownership-chains-and-context-switching?view=sql-server-ver16

[19] C. Mohan and Inderpal Narang. 1992. Algorithms for creating indexes for very large tables without quiescing updates. In Proceedings of the 1992 ACM SIGMOD international conference on Management of data (SIGMOD '92). Association for Computing Machinery, New York, NY, USA, 361–370. https://doi.org/10.1145/130283.130337

[20] Hyun J. Moon, Carlo A. Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. 2008. Managing and querying transaction-time databases under schema evolution. Proc. VLDB Endow. 1, 1 (August 2008), 882–895. https://doi.org/10.14778/1453856.1453952

[21] MySQL, InnoDB and Online DDL. https://dev.mysql.com/doc/refman/8.4/en/innodb-online-ddl.html

[22] Oracle, Database Administrator's Guide, Adding Table Columns. https://docs.oracle.com/cd/B28359_01/server.111/b28310/tables006.htm#ADMIN11005

[23] Oracle, Database Concepts, Data Concurrency and Consistency. https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/data-concurrency-and-consistency.html

[24] Oracle, Edition-Based Redefinition. https://www.oracle.com/docs/tech/ebr-technical-deep-dive-overview.pdf

[25] Oracle, Online Data Reorganization & Redefinition. https://www.oracle.com/gr/database/technologies/high-availability/online-ops.html

[26] PostgreSQL, ALTER TABLE, Notes for add and drop column operations. https://www.postgresql.org/docs/current/sql-altertable.html

[27] Ian Rae, Eric Rollins, Jeff Shute, Sukhdeep Sodhi, and Radek Vingralek. 2013. Online, asynchronous schema change in F1. Proc. VLDB Endow. 6, 11 (August 2013), 1045–1056. https://doi.org/10.14778/2536222.2536230

[28] John F. Roddick. 1992. Schema evolution in database systems: an annotated bibliography. SIGMOD Rec. 21, 4 (Dec. 1992), 35–40. https://doi.org/10.1145/141818.141826

[29] John F. Roddick. 1992. SQL/SE: a query language extension for databases supporting schema evolution. SIGMOD Rec. 21, 3 (Sept. 1992), 10–16. https://doi.org/10.1145/140979.140985

[30] M. Ronstrom. 2000. On-line schema update for a telecom database. In Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073). 329– 338.

[31] Betty Salzberg and Allyn Dimock. 1992. Principles of Transaction-Based On-Line Reorganization. In Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 511–520.

[32] TPC, TPC-C. https://www.tpc.org/tpcc/

[33] TPC, TPC-E. https://www.tpc.org/tpce/

[34] TPC, TPC-H. https://www.tpc.org/tpch/

[35] Lesley Wevers, Matthijs Hofstra, Menno Tammens, Marieke Huisman, and Maurice Van Keulen. 2015. Analysis of the Blocking Behaviour of Schema Transformations in Relational Database Systems, Vol. 9282. 169–183.

[36] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. Proc. VLDB Endow. 10, 7 (March 2017), 781–792. https://doi.org/10.14778/3067421.3067427

[37] Ling Zhang, Matthew Butrovich, Tianyu Li, et al. 2021. Everything is a Transaction: Unifying Logical Concurrency Control and Physical Data Structure Maintenance in Database Management Systems. In Conference on Innovative Data Systems Research (CIDR).