Resumable Online Index Rebuild in SQL Server

Panagiotis Antonopoulos, Hanuma Kodavalla, Alex Tran, Nitish Upreti,
Chaitali Shah, Mirek Sztajno
Microsoft
One Microsoft Way
Redmond, WA 98052 USA
{panant, hanumak, altran, niupre, chashah, mireks}@microsoft.com

ABSTRACT

Azure SQL Database and the upcoming release of SQL Server enhance Online Index Rebuild to provide fault-tolerance and allow index rebuild operations to resume after a system failure or a user-initiated pause. SQL Server is the first commercial DBMS to support pause and resume functionality for index rebuilds. This is achieved by splitting the operation into incremental units of work and persisting the required state so that it can be resumed later with minimal loss of progress. At the same time, the proposed technology minimizes the log space required for the operation to succeed, making it possible to rebuild large indexes using only a small, constant amount of log space. These capabilities are critical to guarantee the reliability of these operations in an environment where a) the database sizes are increasing at a much faster pace compared to the available hardware, b) system failures are frequent in Cloud architectures using commodity hardware, c) software upgrades and other maintenance tasks are automatically handled by the Cloud platforms, introducing further unexpected failures for the users and d) most modern applications need to be available 24/7 and have very tight maintenance windows. This paper describes the design of "Resumable Online Index Rebuild" and discusses how this technology can be extended to cover more schema management operations in the future.

1. INTRODUCTION

Over the last three decades, several database systems have added support for Online Indexing operations [5, 8, 12, 13, 16] that allow index management without impacting concurrent transactions accessing the data. However, these operations have not been designed to be resilient to failures and are completely rolled back when a failure occurs. With the rapid growth in database sizes, as well as the shift to Cloud architectures, built on commodity hardware where failures are common, fault-tolerance has become a critical property for any long running operation.

This is particularly important for indexing operations given:

- Their long duration which is proportional to the size of the index thereby increasing the probability of a failure.
- The large amount of resources (CPU, memory, disk and log space) they require.
- The short maintenance windows of modern business-critical applications.

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

Online Index Rebuild is a frequently used operation that allows rebuilding an existing index to defragment it or modify its properties (e.g. compression). Heavy, random updates performed by OLTP applications cause indexes to get severely fragmented and impact the performance of the overall application. Because of that, it is common for users to rebuild indexes on a weekly or even daily basis to guarantee consistent performance. In Azure SQL Database, there are approximately 40 million rebuild operations executed per week, with thousands of them taking between 1 and 14 hours. Based on that, improving the reliability of this operation is critical for the quality of the service.

Azure SQL Database and the upcoming release of SQL Server allow users to resume an Online Index Rebuild operation after any unexpected failure has occurred or after a user manually paused the operation to free up system resources. SQL Server is the first commercial DBMS to support pause and resume functionality for index rebuilds.

Any existing rowstore index can be rebuilt in a resumable fashion using Resumable Online Index Rebuild by simply specifying the "RESUMABLE = ON" option to the rebuild operation. E.g.:

```
ALTER INDEX <index_name> ON <table_name> REBUILD
WITH (ONLINE=ON, RESUMABLE=ON)
```

Once an Index Rebuild operation is started as "resumable", any type of exception, including transient errors, such as database failovers and server restarts, or errors that would require user intervention, such as running out of disk space, will get the operation in a "paused" state. The user can later resume this operation by issuing a RESUME command. E.g.:

```
ALTER INDEX <index_name> ON <table_name> RESUME
```

Additionally, users can manually pause a resumable operation at any time by executing a PAUSE command.

While the operation is running or when it is paused, SQL Server provides useful information to the users, such as the progress (%) of the operation, the current execution time, the space used, etc. that allows them to estimate the time remaining for the operation and make an informed decision regarding whether they should wait for it to complete, pause it or completely abort it.

This paper describes the overall design of "Resumable Online Index Rebuild" (ROIR) in the upcoming release of SQL Server and discusses our current plans to extend this technology to more indexing and schema management operations in the future. Section 2 begins with an overview of the online indexing operations in earlier releases of SQL Server. Section 3 outlines the architecture of ROIR and describes how it can split the overall operation into incremental and resumable steps. Section 4 covers the core

indexing algorithm and describes how the state of each row is maintained to handle concurrent updates. Some performance results are presented in Section 5. Section 6 concludes with our plans to extend this technology to more operations and other types of indexes.

2. BACKGROUND ON SQL SERVER

This section provides a summary of the design of online indexing operations in the earlier releases of SQL Server which is required to better understand the architecture and design choices behind the ROIR technology. More detailed information can be found in the white paper on the initial implementation in SQL Server 2005 [8] and the public product documentation [9].

2.1 Overview

At a high level, SQL Server's Online Indexing technology is a variation of the "No Side-File" (NSF) algorithm described by Mohan and Narang [10]. The operation begins by creating a new empty index (B^+-Tree) with the appropriate key columns and new properties. Then, the Index Builder process starts in the background by scanning the existing data and inserting it into the new index, sorting, if necessary, on the new index key columns to achieve improved performance and reduce fragmentation. Any concurrent updates to the table must maintain both the existing and the new index in order to guarantee that the new index is always updated with the most current data. On the other side, queries continue using only the existing indexes and are not affected by the new index until the operation completes (see Figure 1). Once the index build process has completed, we update the table metadata to point to the new index and, if this is an index rebuild operation, the old index is dropped.

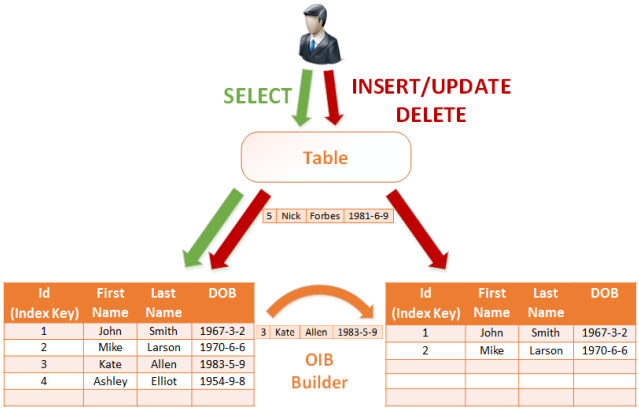

Figure 1. DML operations and Index Builder while an online index operation is running.

2.2 Terminology

In this paper, we will use *"source index"* to refer to the existing index that is used by the Index Builder to scan the data and *"new index"* to refer to the new index being constructed. We assume that each row in the index can be uniquely identified by its values for the key columns of the index.

2.3 Index Build Phases

As described in Section 2.1, the Online Index Build process is logically split into three distinct phases:

2.3.1 Preparation Phase

During this initial phase, the index build process creates the new empty index and associates it with the existing table so that DML operations can start updating it going forward. This phase drains any existing updates to the table by acquiring a Shared (S) lock on the table to update its metadata and force any future updates to recompile and start maintaining the new index. Even though this phase drains all updates to the table, it is only updating the metadata of the table and should complete almost immediately. The S lock is released as soon as this phase completes.

2.3.2 Build Phase

This is the main phase of the index build process which scans the source index, sorts the data (if necessary) and finally inserts it into the new index. During this phase all updates are allowed to the table and they need to maintain both the existing indexes, as well as the new index. It is important to note that all updates to the new index, by both the Index Builder and concurrent DML operations, perform full index maintenance and therefore guarantee the B-Tree stability at all times. SELECT queries only use the existing indexes since the new index is not yet fully populated. The duration of this phase is proportional to the size of the index since it needs to scan all rows and copy them to the new index.

2.3.3 Final Phase

Once the new index has been fully built, the index build process acquires an Exclusive (X) lock on the table to drain any table access, updates the metadata of the table to point to the new index and drops the existing index if it is no longer needed (for example for index rebuild). As part of the metadata update, all query plans are invalidated so that all future access to the table will only use the new index. This indicates the end of the operation. Even though the final phase requires exclusive access to the table, this phase only updates the metadata of the table and should complete almost immediately.

2.4 Concurrency

The algorithm described in the previous section explains how the Index Builder and concurrent DML activity operate, but doesn't address race conditions between these two independent entities that can result in them finding a row in an unexpected state while the index operation is running. These cases are described in detail in [10], but we will also briefly describe them here for reference since they are important for presenting the algorithms of this paper:

a) A delete operation might delete a row that has already been read by the Index Builder, but before it has been inserted into the new index. In this case, the delete would fail to locate the row in the new index and simply ignore it. Then, the Index Builder would re-insert the row it has read, therefore corrupting the new index.

b) An insert operation will insert a new row to both the source and the new index. When the Index Builder visits this row, it will attempt to copy it to the new index as well. In this case, one of the two operations, depending on their ordering, will hit a duplicate key error since the row already exists.

c) Given that concurrent DML operations might rollback at any time, the index build process must guarantee that, by the end of the operation, the new index only contains committed data.

SQL Server's Online Indexing algorithm uses a different mechanism than the NSF solution to address these issues:

- Instead of using special locking and logging techniques, as presented in [10], to guarantee that the new index only contains committed data, SQL Server's Index Builder uses a snapshot scan to read a consistent version of committed data as of the time the operation started. This simplifies reasoning about the version of the data that the Index Builder read but also improves concurrency since the Index Builder doesn't need to acquire locks. On the other hand, all row modifications to the new index, by both the Index Builder and any concurrent DML operations, acquire exclusive row locks to make sure that updates to the same row are synchronized and all state transitions for this row are transactional. To improve concurrency, the Index Builder copies the rows from the source to the new index in batches, committing the corresponding transactions to release all locks.
- In order to address the delete problem (see (a) above), SQL Server uses a special row state called "delete anti-matter" (similar to the pseudo-deletes of [10]): when a delete operation does not find the corresponding row in the new index, it will insert a row with the same key and the "delete anti-matter" state set. When the Index Builder attempts to insert a row to the new index and finds the same row in this special state, it will remove this "stub" from the new index. Given that the Index Builder uses a snapshot scan, it is guaranteed to read all the rows as of the time the indexing operation started and, therefore, remove all rows marked as "delete anti-matter" by the time the operation finishes.
- Given that the Index Builder uses a snapshot scan, it will never see any new rows inserted by concurrent Inserts. However, an Update of an existing row will also insert the updated row to the new index and when the Index Builder attempts to insert the original row there it will fail since the row already exists. This leads back to the duplicate insert problem (b). To address this, SQL Server uses a mechanism similar to the one described in [10], where the Index Builder will ignore the duplicate row. The difference, however, is that SQL Server uses an additional "insert anti-matter" state that allows the Index Builder to distinguish between false duplicates, introduced by concurrent DML operations updating existing rows of the index, and real duplicates which indicate a violation of the uniqueness of the index. This capability is particularly important to support unique index creation, where the Index Builder needs to validate the uniqueness of the new index as it is being built. The "insert anti-matter" state is used whenever an insert occurs on top of a row that was earlier marked as "delete anti-matter" to indicate that this row already existed when the index build operation started and, therefore, can be safely ignored. Updates are also treated as a delete followed by an insert and will also introduce an "insert anti-matter". New row insertions, on the other side, will insert regular rows, not marked as "insert anti-matter". When the index builder tries to insert a row to the new index and finds a row in this state, it will clear up the state, converting it into a regular row. If it sees a regular row, it means that there is a unique key violation and the operation will fail. Since the "insert anti-matter" state is only used for rows that existed when the operation started, the Index Builder is guaranteed to process all of them and eventually clear up this state from all rows.

Figure 2 illustrates the state machine for the state of each row during an Online Index Build operation. One invariant that the algorithm maintains is that at the end of the operation we should not have any rows in the delete or insert "anti-matter" state.

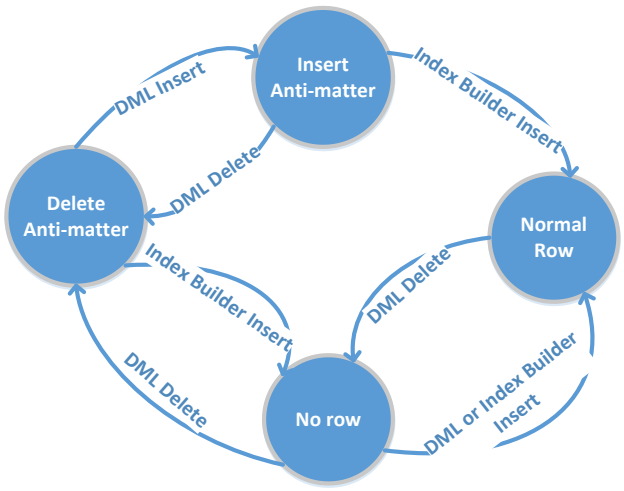

Figure 2. SQL Server's Online Index Build row state machine.

3. RESUMABLE ONLINE INDEX REBUILD

Azure SQL Database and the upcoming release of SQL Server introduce Resumable Online Index Rebuild (ROIR) that provides fault-tolerance and allows users to resume these long running operations after a failure or after a user-initiated pause. It is important to note that the operations can be resumed after any system failure, including cases where the database has failed over to a different replica. Additionally, the underlying tables are fully available while these operations are "paused", without consuming any additional resources (memory, temporary space, etc.) other than the data space that is occupied by the partially built index.

ROIR leverages the existing infrastructure for SQL Server's Online Indexing operations, described in Section 2, extending it to allow the operation to resume after any type of failure where all in memory state will be lost abruptly.

The main idea behind making these long running operations resumable is to:

- Split the overall operation into incremental units of work so that each unit can be completed within a small amount of time.
- Periodically commit the progress of the operation, using internal transactions, to harden the work completed so far.
- Persist the state that is required for the operation to resume from this point in case of failure.

This logic must be applied to each step of the operation to make the overall process resumable. Given that our goal is to provide resumability even in catastrophic scenarios (e.g. loss of power), the

required state needs to be persisted to disk and replicated to any replicas of the database.

In the following sections, we describe how the various operations have been modified for the purposes of our design.

3.1 The Rebuild Operation

For simplicity, we will first describe the design of the index rebuild operation in the absence of concurrent DML operations that might be updating the content of the original index.

For the purposes of ROIR, the three phases of the Online Indexing infrastructure, described in Section 2.3, now occur in separate transactions so that they can commit their work independently. Additionally, when we transition from one phase to the next, we persist this state transition in one of SQL Server's system tables so that the operation can identify the phase it needs to resume from. The state transition is persisted in the same transaction as all the other operations of the current phase we just executed so that they are atomic. Finally, the Preparation phase is extended for ROIR to persist all the required metadata for the operation (index options, etc.) so that the system is aware of the operation and can use the correct information to resume after a failure.

Since the Preparation and the Final phases only involve metadata modifications and should be extremely short, each of these phases can be considered a single unit of work, occurring in a single transaction. On the other hand, the duration of the Build phase is proportional to the size of the index and must be separated into small, incremental units that will allow us to resume at a much smaller granularity.

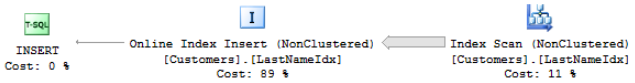

Figure 3. Sample query plan of an Index Rebuild operation. The "Online Index Insert" operator is inserting the data to the new index over a simple "Index Scan".

The Build phase involves scanning the table and inserting the corresponding data into the new index. Generally, the data might need to be sorted before the insertion, for example when creating a new index with different keys, but for the purposes of index rebuild, we can safely assume that there is an existing index (the one that is being rebuilt) that already provides the correct ordering and, therefore, no sorting should be required. Internally, the build process is accomplished by issuing an INSERT…SELECT query that is compiled and executed similar to regular, user queries, but has additional context indicating it is targeting an index build operation. Figure 3 demonstrates a serial execution plan for this query. For the rest of this section, we will assume serial execution of the index build process.

Based on the query plan of Figure 3, we can simply split the Build phase into smaller units of work by reading batches of N rows at a time and inserting them into the new index in a separate transaction that will be committed to persist our progress. This guarantees that we will never lose more than a single batch (N rows) in case of a failure, but we still need a mechanism that will allow us to resume from the last batch we processed and avoid duplicate work. For this purpose, we will use an ordered scan on the source index to guarantee that the data is retrieved in a deterministic order and use the key of the last row inserted, essentially a "high watermark", in order to determine the range of rows that have already been processed. When the operation resumes after a failure, this key will be used to position a new (ordered) scan starting from the row that is right after the last row we processed in the previous execution, therefore resuming the operation without duplicating any work. Figure 4 provides an example of this process.

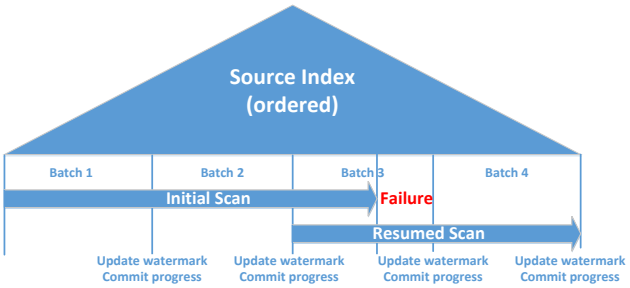

Figure 4. Tracking progress at batch boundaries and resuming.

The key of the last row processed will be persisted in the system tables for durability and only needs to be updated when committing the batch we are currently processing, since any failure before this point would anyways force us to start from the beginning of this batch. It is important that all insertions to the new index as well as the "watermark" update occur in the same transaction as we need these operations to be atomic for correctness. All this logic is internally applied by the "Online Index Insert" operator (see Figure 3) that was extended for the purposes of ROIR to persist the progress based on the key of the last row in each batch.

In the case of index rebuild, the data is read using the same, existing index and, therefore, it is read in the same order as the one it gets inserted to the new index. However, this is not important for the correctness of our algorithm and the same logic can be applied for creating new indexes where the order of the source and target indexes can differ.

3.2 Parallelism

Given the high cost and long duration of Index Build operations, it is critical for them to be efficiently parallelized, leveraging all the available resources of modern multicore machines. The described index build algorithm can be extended to allow for parallelism.

SQL Server generally supports several ways of parallelizing different parts of the query plan (including plans for index build operations) based on the cardinality of the data, as well as the requirements of each operator, such as the sort order of its input. For example, a parallel scan can distribute the data by using round-robin, hashing or by splitting it in ranges. However, all index build plans need to maintain one important invariant: insertions to the new index (or at least to a contiguous range of the index) need to be ordered based on the key columns of the new index. This is important to avoid fragmentation as well as random access to different pages of the new index that could impact the performance of the index build process. To satisfy this requirement, we achieve parallelism by assigning a contiguous, disjoint range of the new index to each execution thread so that the insertions are not completely ordered, but are still partially ordered within each disjoint range that is processed by a single thread. All other operators (scans, sorts, etc.), below the final insertion to the new index, can generally perform any kind of parallelism they consider optimal and only need to scatter their output to the thread that is processing the corresponding range for each row.

For ROIR, we decided to narrow down the space of possible options and enforce a specific type of parallel plan, called "Range partitioning", which is a common choice for large index builds in SQL Server. This type of parallelism assigns a disjoint range of the new index to each thread but the corresponding thread is responsible for both scanning this range of rows from the source index, as well as inserting it to the new index. Essentially, this technique partitions the whole query plan horizontally and each thread processes its assigned range, without requiring any type of synchronization during the whole process. The lack of synchronization is an important reason why we selected this type of parallelism for ROIR:

- Each thread gets assigned a contiguous range of the new index, which is essentially a range of values based on the key columns of this index.
- As each thread scans the data within its range from the source index and inserts the rows to the new index, it tracks the progress it has made by persisting the key of the last row processed as described in Section 3.1. Given that each thread processes a disjoint range of rows, performing its own scan on the source index, the progress can be maintained independently from any other threads.
- The assigned range and the progress of each thread is persisted in a system table so that it can be reloaded at resume time and continue processing the correct range from the point where each thread stopped.

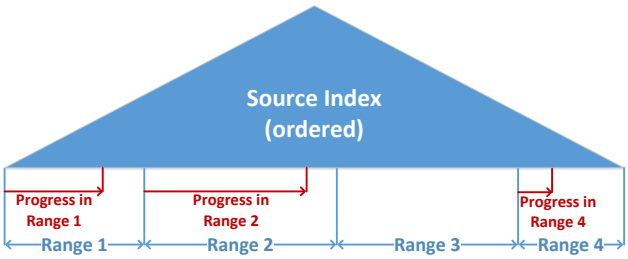

Figure 5. Range partitioning and tracking progress for each range.

Figure 5 provides an example of a parallel rebuild operation, visualizing the data that different threads will process. Specifically for index rebuild, the source index and the new index have the same key columns, and, therefore, both the progress and the range of each thread are defined on the same columns, with the progress falling within the range of this thread. However, this logic can be extended without modifications to cases where the columns might be different, such as when creating a new index.

In order to balance the load across threads and achieve better parallelism, the number of rows in each range must not differ drastically. For this purpose, the Query Optimizer generates (or uses existing) statistics on the column and selects the appropriate ranges based on the data distribution and the number of available threads. In the existing Online Indexing operations, there is 1-1 mapping between the ranges and the available threads in the system. For ROIR, we modified the way ranges are defined in order to allow the user to change the Degree of Parallelism (DOP) when the operation is resumed and enable scenarios where the user can scale up or down the database resources, allowing for improved elasticity that is particularly important in the Cloud. To achieve that, we still leverage statistics to generate evenly sized ranges, but use a larger number of ranges compared to the number of available threads to account for cases where the number of threads might change when the operation is resumed. We then generate a special query plan that allows us to dynamically assign ranges to the available threads.

Figure 6 demonstrates an example of such a plan. The "Constant Scan" contains the identifier of each range starting from 1 to N where N is the number of ranges. A parallel Nested Loop Join feeds the ranges to the main index build plan (Insert over a Scan) so that each thread picks up a specific range. Using the range identifier, the Scan operator will internally load the range boundaries, as well as the progress that was made earlier (if any) and start the scan from the appropriate row within this range. Given that for rebuild operations the source and target index have the same key column, we can easily position to the beginning of the range by simply seeking in the source index. This makes transitioning between ranges very efficient and allows us to use a larger number of ranges without significant cost.

3.3 Concurrent DML Operations

In the previous sections, we discussed how the Index Rebuild operation can be re-designed to be resumable in the absence of any concurrent activity. However, given that this operation needs to be "online", we need to establish how concurrent updates to the index will operate, while guaranteeing the index consistency at the end of the operation.

Since ROIR depends on the existing Online Indexing infrastructure in SQL Server, concurrent user updates will maintain both the original and the new index. This logic is enforced through special query plans that are specifically generated for DML operations that are executed while an index build operation is in progress.

The main difference of ROIR, compared to the existing algorithm, is the fact that the operation can now pause and resume at any time, potentially after a full server restart, and, therefore, we can no longer depend on a consistent snapshot to guarantee consistency (see Section 4 for more details). Because of that, the way concurrent DML activity maintains the new index needs to be modified to account for the fact that the index build process might now see some of these concurrent updates.

Given the complexity of DML query plans, that need to maintain both the old and the new index, and the number of issues SQL Server had to deal with over the years in this area, we decided to avoid any significant changes to the query plans of concurrent DML operations. Instead, we re-designed the underlying algorithm for tracking the state of each row, leaving all DML plans unchanged. Surprisingly, under this new algorithm, the state machine becomes significantly simpler. Our algorithm is described in detail in Section 4.

One important thing to note is that concurrent DML operations need to maintain the new index even when the index build operation is paused. This is required because, when the operation resumes, it will start processing from the point it previously stopped and, therefore, the portion of the index that had already been processed will not be updated. This introduces an extra overhead to all updates, even when the operation is paused. Theoretically, the portion of the index that has not yet been processed does not need to be updated and we could optimize updates that are targeting this portion of the index to avoid the extra overhead. However, this would require synchronization between the index builder and concurrent DML operations regarding the ranges that have already been processed and is particularly complex when we have

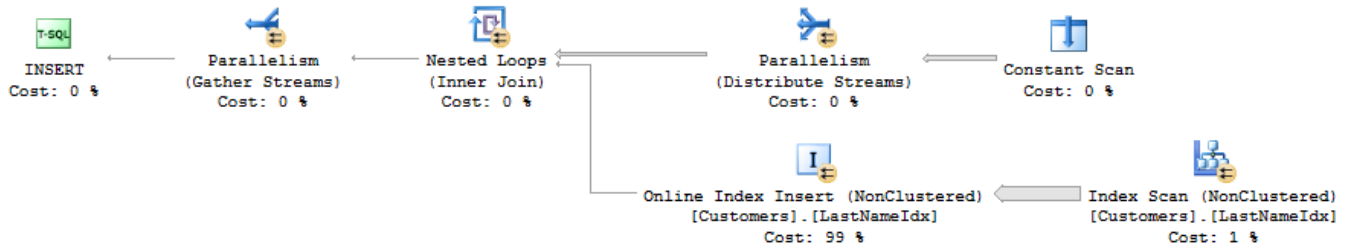

Figure 6. Parallel plan for ROIR.

parallelism, where there are multiple, partially built, ranges. Based on our experiments and customer interactions so far, this overhead should be acceptable and we should not need further optimizations.

3.4 Other Schema Modifications

Even though Online Index Build operations allow concurrent updates to the underlying tables, schema modifications, such as adding or altering a column, are not permitted while these operations are executing. Online Index Build currently depends on schema locks, held for the duration of the operation, to prevent such modifications until the operation completes. For ROIR, we continue using schema locks while the operation is executing, but additionally set the table in a special, persistent state that will prevent any schema modifications even when the operation is paused and locks have naturally been released (e.g. after a restart).

3.5 Log Management

Log management is an important aspect of any long running operation. Currently, all Index Build operations are performed in a single transaction that is active for the duration of the operation. This transaction is used to hold all the necessary locks, but also clean up the database state in case of failure. According to ARIES [11], in the event of a failure, the transaction will be rolled back using the database transaction log and, therefore, SQL Server cannot truncate this portion of the log until this transaction is committed or rolled back. Given the duration of these operations, but also the fact that they need to perform data modifications proportional to the size of the index that is being built, the required log space is also proportional to the size of the index. Moreover, since these operations are online, concurrent activity in the database can generate an additional amount of log.

This has been a significant problem for our users that currently need to provision an extraordinary amount of log space compared to what is normally needed for their workload. This is particularly cumbersome in Azure SQL Database, where users don't have direct control of the corresponding log space and can accidentally run into out-of-log situations when building large indexes.

An important design goal for ROIR is to enable rebuilding large indexes using only a small, constant amount of log space. To achieve that, we eliminated the need for a long running transaction by performing all database modifications in short, internal transactions that persist the appropriate state so that the operation can resume accordingly. In case of failure, each of these short transactions is designed to bring the database back to a consistent state that can be handled correctly when the operation attempts to resume. Any locks that need to be held for the duration of the operation are now held by a read-only transaction that does not block log truncation. With that, the transaction log can get naturally truncated as the short, internal transactions commit and, therefore, the operation can complete using only a small amount of log space.

3.6 Statistics

As part of the Index Build process, SQL Server builds statistics for the key columns of the index. Since the Index Build process already accesses all rows of the index, it can build full statistics (meaning that they were built using a full scan of the data), at very low cost. Generating statistics involves building a histogram and computing cardinalities and is currently performed completely in memory, persisting the result of the process to metadata at the end of the operation.

This works perfectly for the existing Index Build operations where the in-memory state is preserved for the duration of the operation, but would be problematic for ROIR where the operation must resume after any failure, including a full server restart. To address that we would need to make the process of building statistics resumable. This is not an insurmountable problem. Instead of aggregating statistics for the whole index at once, we would build partial statistics for batches of N rows, persist them to disk and eventually merge them to generate the final statistics. SQL Server already supports merging statistics without significant loss of quality, currently used for merging per-partition statistics for partitioned tables.

However, in practice, users generally avoid creating full statistics for large tables in all cases other than index builds where this happens automatically. Additionally, as tables get updated, SQL Server will automatically update the corresponding statistics in the background using sampling, even for statistics that were originally built using a full scan. Therefore, most query plans on large tables already depend on sampled statistics and their performance is generally acceptable. Based on that, for ROIR, we decided not to generate full statistics, but create sampled statistics, that should be built quickly even for large tables, at the end of the operation. This has been an acceptable compromise for all our users so far, but is an area we are considering to improve in the future.

3.7 The cost of "checkpointing"

To provide resumability even in catastrophic scenarios (e.g. loss of power), the state we maintain in order to resume the operation must be persisted to disk and replicated to database replicas. Given that the cost of this process is not trivial, there is a trade-off between the "Resumability SLA", i.e. how much progress is lost in the event of a failure, and the overhead that this "checkpointing" will incur to the operation. For ROIR, we measured the overhead of checkpointing at different frequencies (batch size) and decided to persist progress every 100K rows that the Index Builder processes. This should allow us to avoid increasing the cost of the operation while providing an acceptable Resumability SLA (< 1 minute of progress would be lost in most cases). Section 5 presents some experimental results regarding the performance of the index build operation for different batch sizes.

3.8 No Side-File vs. Side-File

As described in Section 2, SQL Server's Online Indexing operations use an algorithm similar to the No Side-File (NSF) algorithm where DML operations have to maintain both the source and the new index. There are several tradeoffs between the NSF and Side-File (SF) approaches, some of which are presented in [10], but NSF was considered the right solution mainly because it guarantees that the Index Build operation will eventually complete with minimal down time, regardless of the number of concurrent updates that occur to the table. This is achieved by forcing all updates to maintain both the existing and the new indexes, so that the Index Builder doesn't have to process any updates at the end of the operation, but it comes at a cost, both in terms of complexity, as well as performance of DML operations.

When designing ROIR, we reconsidered this choice in the context of the new scenarios that we want to support. SF is attractive because it reduces the performance overhead of concurrent DML operations, that would only need to append the update to a side file, instead of having to update an additional index. This is particularly important when the operation is paused, since we want to prioritize the normal user activity and minimize any overhead. On the other hand, if the operation stays paused for a long time, the delta that would be accumulated from concurrent updates could become significant, even comparable to the size of the index. In this case, resuming the operation could take significantly longer and the whole purpose of resumability becomes void. Additionally, the complexity of NSF is no longer a concern. As described in Section 3.3, all this logic has been built and stabilized for many years in SQL Server.

Based on these arguments, we decided that NSF is still the right design choice for ROIR.

4. CORE INDEXING ALGORITHM

As described in Section 2.4, the existing Online Indexing algorithm depends on a snapshot scan of the input data to guarantee the consistency of the new index in the presence of concurrent DML operations. SQL Server implements snapshot scans using row-level versioning. When versioning is enabled, all updates generate a new version for each updated row and move the older version into TempDB (SQL Server's temporary storage), creating a link between the two versions. Scans can traverse the version chain of each row and retrieve the correct version based on their snapshot timestamp, as well as the commit timestamp of the transaction that generated each version, following the Snapshot Isolation [1] semantics. Since TempDB is refreshed when the server restarts, all row versions disappear and, therefore, it is not possible to have a consistent snapshot that spans server restarts.

This is problematic in the case of ROIR since we must resume the operations after any potential failure, including random server restarts, in which case all transient state would be lost. Without a snapshot, the Index Builder might:

- Read rows that were inserted by DML operations after the index build operation started.
- Miss rows that were deleted by DML operations after the index build operation started.

To address these scenarios, we have to modify the core indexing algorithm to allow correctly tracking the state of each row without depending on a consistent snapshot of the input data.

4.1 The ROIR Algorithm

The main idea behind the proposed algorithm is that the state for each row (whether the row was deleted or inserted) is maintained in a persistent fashion for the lifetime of this row. This allows the Index Builder to identify whether each row was previously inserted or deleted by a concurrent DML operation and take the right action based on that. The logic of our new algorithm is the following:

- Any deletion in the new index will insert a "delete anti-matter" row for this key. This needs to happen regardless of whether the row was already present in the index, in which case it will simply be updated with the appropriate state, or whether it was not found, in which case we will insert a new row and mark it accordingly. This is different compared to the original Online Indexing algorithm, where "delete anti-matters" were only inserted when the row was not found in the new index.
- Any insertion (by DML operations or the Index Builder) to the new index will insert a normal row, without any special state. If a "delete anti-matter" row already exists in the new index, this row will be overwritten by the inserted regular row that contains the updated content.

This allows us to store the state of each row in a persistent manner that does not depend on whether the Index Builder has already read or copied the row and, therefore, is not impacted by the lack of a snapshot scan.

Since all DML operations are updating both the source and the new indexes, any rows that can be found in the target index are guaranteed to be up to date. As the Index Builder is copying rows from the source to the new index, it can check if the row exists and what its state is and apply the following logic:

- If the row was deleted by a concurrent DML operation, the Index Builder will find a "delete anti-matter" and the row can be removed completely.
- If the row was inserted by a concurrent DML operation, the index builder will find a regular row and can safely ignore it since the version that has been inserted by the DML operation is guaranteed to be the newest version of the row.

Figure 7 demonstrates the state machine for our algorithm.

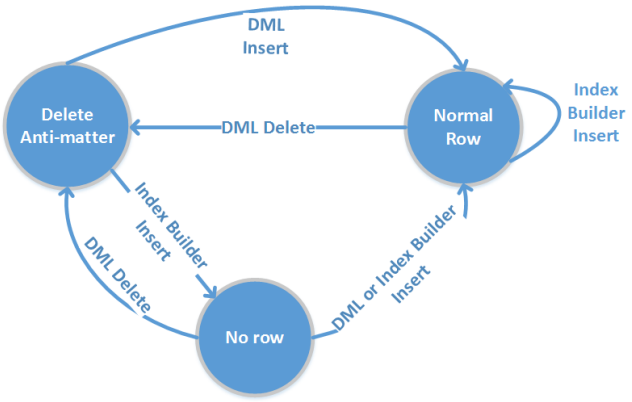

Figure 7. The modified row state machine for ROIR.

For the purposes of our algorithm, we still assume that the Index Builder only reads committed data. This can be achieved either through locking or by using a snapshot scan. In our system, we

decided to use a snapshot scan for improved concurrency. However, our algorithm does not depend on a consistent snapshot for the duration of the operation. The snapshot is only used for reading committed data and we can always re-establish a new snapshot if the server restarts and the operation is resumed. It is also important to note that, similar to the existing Online Index operation, row locks are used to synchronize DML operations and the Index Builder when accessing and updating the new index.

The updated algorithm eliminated the "insert anti-matter" state and, therefore, is simpler than the original Online Index Build algorithm described in Section 2.4. As described in this section, the "insert anti-matter" state was mainly introduced to support unique index creation, where the uniqueness of the index needs to be validated by the Index Builder. To achieve this, however, the original algorithm also depended on having a consistent snapshot to guarantee that any concurrent updates will not be visible to the Index Builder. Given that this capability is not available for ROIR, this state is no longer useful and can be eliminated for simplicity. In order to support unique index creation in the future, we are planning to use an additional data structure that will allow us to validate the uniqueness of the new index that is being built.

Another important difference is that the updated algorithm fails to maintain the invariant that at the end of the indexing operation, all "delete anti-matter" rows are guaranteed to be removed. This invariant is impossible to maintain without a consistent snapshot since the Index Builder can now miss rows that were deleted by concurrent DML operations after the index operation started and, therefore, these rows will stay in the new index forever. Based on that, at the end of the operation, the new index might contain several "delete anti-matter" rows that need to be invisible to new scans on the table since they logically represent deleted rows. To achieve that, but also make sure that these rows are eventually removed from the index to free up the corresponding space, we mark all "delete anti-matter" rows as "ghosts". The infrastructure around ghost rows already exists in SQL Server to support row locking and snapshot isolation and is extended here for our purposes. By marking these rows as ghosts, we ensure that they are visible for the duration of the index build process, where they are meaningful, but are no longer visible to any scans after the index build operation completes. Additionally, these rows will be eventually deleted by a background process, known as "Ghost cleanup" that is identifying and removing such rows from the database. Given that "delete anti-matters" are critical for the correctness of our algorithm, the Ghost Cleanup process needs to be blocked for the new index while the index build operation is running. This guarantees that we will not accidentally remove any "delete anti-matters" until the operation completes.

4.2 Correctness

Let I_{source} be the source index of the index build operation and I_{new} the new index. Each row can be uniquely identified by its keys K_{source} and K_{new} in the source and the new index respectively. For the purposes of Index Rebuild, both indexes have the same key columns and ordering ($K_{source} = K_{new} = K$ for all rows), but the correctness of our algorithm does not depend on this property.

To prove the correctness of our algorithm, we need to guarantee that, at the end of the index build operation, the new index is consistent with the source index maintainining the following invariants:

- Every row with key K_i in I_{source} exists exactly once in I_{new}.
- Every row with key K_i in I_{new}, except "delete-antimatter" rows that are logically deleted, exists in I_{source}.
- The payload (non-key columns) of all rows that are common between the two indexes is identical.

4.2.1 Index Builder

The Index Builder scans all rows in I_{source} and copies only committed versions to I_{new}, either by locking them or using a snapshot scan (our implementation uses the latter). Also, since the Index Builder uses a scan, it is guaranteed to visit every row in I_{source} but also not process any row twice, therefore maintaining all invariants described above. Even in the event of a failure, since the operation is transactional, the consistency of the index is guaranteed, and the last batch of processed rows will be rolled back. When the operation is resumed, the scan will reposition right after the row that was last processed, based on the algorithm of Section 3, so all invariants are still maintained.

4.2.2 DML operations

DML operations can be separated into two main categories:

- The DML operation is updating a row that has already been processed by the Index Builder (meaning that it has been successfully inserted into I_{new}).
- The DML operation is updating a row that has not yet been processed by the Index Builder.

In the first case, since the Index Builder has already processed the updated row, any modifications are not visible to the Index Builder and, therefore, the index build operation is not affected. Since the Index Builder has already inserted all rows from I_{source} and DML operations will update both I_{source} and I_{new}, we are guaranteed that this portion of the index will always be consistent between I_{source} and I_{new}. Deletes might have introduced additional "delete anti-matters" in I_{new}, but these can be safely ignored.

In the second case, the Index Builder has not yet processed this portion of the index and, since we are no longer using a consistent snapshot, it might see some of these updates. If the DML operation is inserting a row, according to our algorithm, the row will be transactionally inserted to both I_{source} and I_{new}. If the Index Builder reads this row from I_{source}, it should also be able to locate the same row in I_{new} and can simply ignore the row. Since the row has been inserted there by a DML operation, it is guaranteed to contain the most updated values, consistent with the source index. If the DML operation is deleting a row, it will also transactionally update both I_{source} and I_{new}, inserting a "delete anti-matter" to I_{new}. If the Index Builder scans the row after the deletion, the row will simply not be part of the scan. The "delete anti-matter" will stay in I_{new} until the end of the operation, but this does not violate our invariants. If, on the other hand, the Index Builder had read the row before it was deleted, it will attempt to insert the row to I_{new} and identify that a "delete anti-matter" already exists there. In this case, the Index Builder can remove the "delete anti-matter" and proceed to the next row.

As we can see in all cases, the Index Builder should copy all initial rows from I_{source} to I_{new}, while DML operations are updating the corresponding rows in both indexes, therefore maintaining all the required invariants.

4.3 TLA+ Verification

In order to verify the correctness of the proposed algorithm, we modeled our algorithm and state machine using the TLA+ formal specification language [6] and verified that it maintains the expected invariants by using the corresponding TLA+ toolkit [17].

The TLA+ model for our state machine can be found at: https://github.com/panant/Resumable_Online_Index_Rebuild

Our model contains the following state transitions:

- The Index Builder (sequentially) reads a row from the source index.
- The Index Builder inserts a row it previously read into the new index, ignoring it if it finds an existing row (since this would have come from a DML operation and is guaranteed to be updated).
- A DML operation inserts a random row by updating both the source and the new index, potentially overwriting a "delete anti-matter" in the new index. The "randomness" of the row is important in order to simulate inserts that occur both in the portion of the index that has already been processed by the Index Builder, as well as the rest of the index.
- A DML operation deletes a random row by updating both the source and the new index, inserting a "delete anti-matter" to the new index.

Each state transition needs to be atomic and our system guarantees that through row locking on the new index. Given that the Index Builder doesn't hold any locks to guarantee the atomicity of reading and inserting a row, these two operations had to be modeled separately.

Finally, the model verifies the following invariant:

At the end of the Index Build operation, which is defined as the state where the Index Builder has processed all rows of the source index and inserted them to the new index, the content of the new index (key and payload of each row) should be identical to the content of the source index, except for "delete anti-matter" rows that are simply representing deleted rows and can be ignored.

This invariant essentially encapsulates the invariants described in Section 4.2 and verifies the index consistency at the end of the index build operation.

It is interesting to note that an earlier version of our algorithm contained an incorrect state transition where the deletion of a row that already exists in the new index would simply delete the row instead of introducing a "delete anti-matter". This would eventually cause the Index Builder to re-insert a deleted row and corrupt the index. Our team was able to identify this issue, but it required several iterations and significant amount of time. On the other hand, TLA+ was able to identify that our invariant gets violated within five seconds and provide a simple series of events that would result in this inconsistency, thereby proving that it is a powerful tool for verifying concurrent and distributed algorithms.

5. EXPERIMENTAL RESULTS

This section presents experimental results regarding the performance of ROIR, as well as the latency and the throughput of concurrent DML workloads, executed while an index is being built. All our experiments are run on a workstation with 2 sockets, 24 cores (Intel® Xeon® Processor E5-2673 v3, 2.40GHz) and 192GB of RAM. External storage consisted of two 480GB Samsung SSDs for data and log respectively.

5.1 Index Build

Since ROIR follows the same process of building the new index as the existing Online Index Rebuild, we expect the performance of the two operations to be the same. However, there are two important differences that can affect the performance of the resumable operation:

Firstly, ROIR periodically checkpoints the progress of the index build by persisting the key of the last row processed to a system table. This introduces a constant overhead for every batch that we process. Therefore, the batch size must be carefully selected to achieve the right balance between performance and resumability.

Table 1. Performance of ROIR based on the batch size

Operation/Configuration	Duration (sec)
Resumable with batch size = 100K rows	132
Resumable with batch size = 10K rows	476
Resumable with batch size = 1K rows	4802
Resumable with batch size = 100 rows	8642
Non-resumable	140

Table 1 presents the time required to rebuild a non-clustered index of 1 billion rows (~32GB) using ROIR with different batch sizes, as well as the duration of the traditional Online Index Rebuild operation for the same table. A small batch size can significantly impact the performance of the operation since the overhead of updating the system tables and committing the corresponding transactions becomes very high compared to the overall cost of the operation. Based on our experiments, we decided that a batch size of 100K rows is the right balance between performance and resumability, achieving equivalent performance to the traditional Online Index Rebuild operation.

Additionally, as described in Section 3.2, the parallel execution plan of the Index Build has been modified to split the index into a larger number of ranges and allow threads to dynamically pick up new ranges as they complete their previous work. This process introduces a small additional cost, since each thread needs to reposition to the new range, but more importantly can increase contention amongst the Index Build threads building consecutive ranges and updating common pages of the new index. This contention also exists for the traditional Online Index Build, but disappears as soon as pages get split and the threads are no longer modifying the same region of the index. Due to the large number of ranges for ROIR, threads will frequently move to a new, not yet populated, range and, therefore, continue to collide with their neighbors. This problem was noticeable in the original implementation of our feature, impacting its scalability.

To address that, we reversed the order in which ranges get assigned to threads from ascending to descending. For example, in Figure 5, Range 4 will be processed first, Range 3 second, etc., while each thread processes the rows within its range in ascending order. Hence, whenever a thread starts processing a new range (for example Range 3), it will attempt to insert the first row of the range and this will have lower key than any row belonging to the previous range (Range 4). Since each range is filled in ascending order, there is very high probability that the thread that was processing the

previous range (Range 4) has moved out of the first page that contains the range for these keys. Additionally, we modified the Index Build operator to commit the first batch of each range immediately after the first page is split in order to release any locks on the first page as soon as possible. With these optimizations, we were able to minimize contention between threads that are processing contiguous ranges of the index.

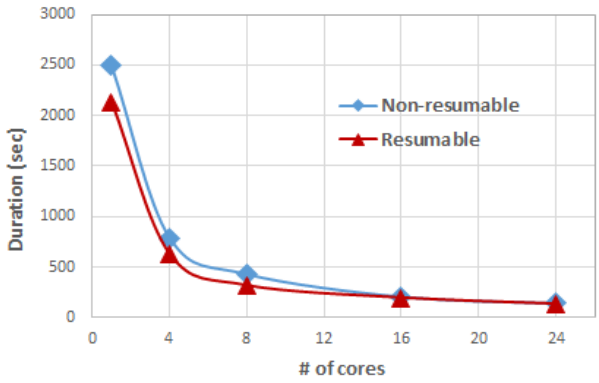

Figure 8. Scalability of ROIR.

Figure 8 demonstrates the scalability of the ROIR compared to the traditional, non-resumable operation. ROIR exhibits excellent scalability, similar to the non-resumable operation, but it is also interesting that ROIR is 10-15% faster than the traditional operation. Based on profiling of the SQL Server process, this difference is caused by the fact that the traditional operation also collects full statistics for the table, as described in Section 3.6, and, therefore, requires additional processing per row, whereas ROIR will only build sampled statistics at the end of the operation.

Finally, it is important to call out that ROIR only used a small, constant amount of log space (<1GB) to rebuild this large index, in contrast to the traditional non-resumable operation that required space proportional to the size of the index (>30GB). This is a significant improvement since provisioning the appropriate log space for index management operations has been a constant problem for our users.

5.2 DML Performance

As described in the previous sections, DML operations executed while an index build is running (or when paused) need to update both the source and the new index. Based on that, we need to evaluate the performance for different types of DML operations in the presence of an index build. To avoid the DDL and DML operations competing for resources on the system, we will pause the Index Build operation to allow DML operations to run uninterrupted. Since the traditional Online Index Rebuild operation cannot be paused, we introduce an internal "knob" that will allow the rebuild operation to suspend and not consume resources.

5.2.1 DML Latency

In this section, we evaluate the latency of each type of DML operation in the presence of an Online Index Rebuild. For this purpose, we used a set of micro-benchmarks to insert, update or delete a single row in an existing, large table with variable number of indexes. Figure 9 presents our experimental results for each scenario. As we can see, the latency of DML operations is roughly the same between the resumable and the regular index rebuild which is expected since they perform the same index maintenance. Moreover, the latency in both cases is roughly equivalent to having an additional index on the table, since DML operations also need to maintain the new index that is being rebuilt.

It is important to note that the results presented in Figure 9 only include the cost of each DML operation, excluding the cost of starting and committing the corresponding transaction. Our experiments indicated that, if each DML operation is performed in an independent transaction, the overall cost of the operation is increased by 0.35ms, since the transaction log needs to be flushed to the disk when the transaction commits. In this scenario, the cost of the transaction dominates the cost of the actual operation and, therefore, the overhead introduced by the index rebuild becomes relatively small (<10%).

5.2.2 Throughput

Having analyzed the impact of the index build on individual DML operations, we also need to evaluate how it affects the overall throughput of the system. This is particularly important for ROIR since we want to enable users to run their regular workloads while the index build operation is paused.

Since SELECT queries are unaffected by the index build operation (the new index is invisible to them), read-mostly workloads should not be impacted by the index build. On the other hand, all updates need to maintain the new index and, therefore, introduce an additional overhead that can affect the throughput of the system for update-heavy workloads. Based on that, we decided to use a TPC-C workload, that is extremely update-intensive and has a simple schema with only one or two indexes per table, to measure the throughput of the system in the worst case. Additionally, we performed the same experiments with a TPC-E workload which should represent a more common ratio between reads and writes. Since each of these workloads is updating different tables at different frequencies, we measured the throughput as it relates to the index that is being rebuilt.

Table 2 demonstrates the throughput degradation introduced by the index build operation for each workload. We measured the performance when rebuilding different indexes in the schema and calculated the average and the worst-case degradation. For TPC-C, the worst-case scenario is representing the case where the clustered index of the "Stock" table is being rebuilt. The Stock table is updated multiple times (one per "line-item") for each "order" that the workload processes and, therefore, has a higher impact on the overall performance. Similarly, for TPC-E, the worst-case is representing the scenario where the clustered index of the "Trade" table, which is one of the most frequently updated tables in TPC-E, is being rebuilt.

Table 2. Throughput degradation for TPC-C and TPC-E relative to the index being rebuilt

Workload	Average case	Worst case
TPC-C	5.4%	32%
TPC-E	4%	13%

Based on these experiments, we consider the performance degradation introduced by the index build to be acceptable for most cases and, therefore, users can continue executing their regular workload while the operation is paused. However, in the cases where the in-build index is heavily updated, the overhead of maintaining the new index can significantly impact the throughput

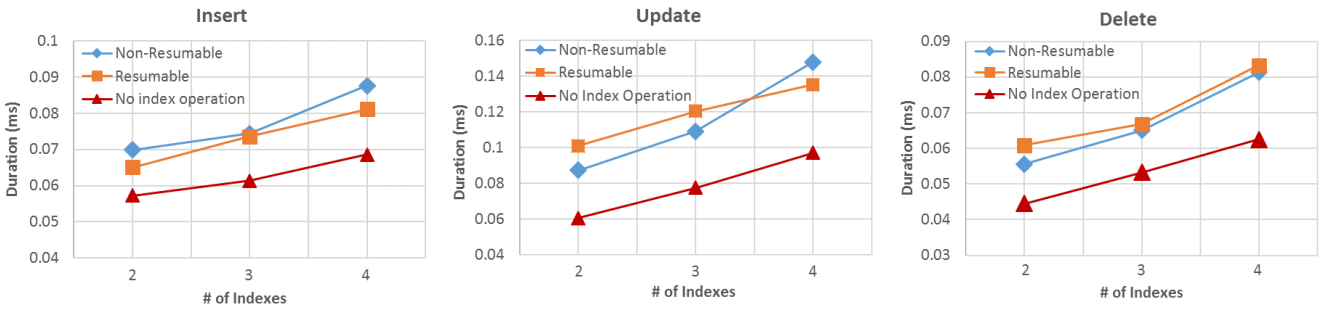

Figure 9. Latency of single row insert/update/delete operations while an index operation is running.

of the system. In these scenarios, users can still leverage the resumable operation in order to be able to resume after a failure, but should avoid leaving these operations paused for extended periods of time, since it can impact the throughput of their system.

6. WORK IN PROGRESS

Due to the SQL Server release timeline, we had to limit the scope of this feature and only support Index Rebuild operations for rowstore indexes, which are generally the most frequent operations and are causing issues for our users on a regular basis. Even though this is an important first step, there are many other index and schema management operations that are also suffering from the issues described in this paper. We are currently working on extending our algorithms to support resumability for more operations in the future releases of SQL Server.

Our immediate goal is to introduce support for resumable index creation. Index creation follows a similar process as index rebuild with the difference that the data needs to be sorted before being inserted to the new index. Sorting is important when building an index in order to reduce fragmentation and avoid random page access which can significantly impact the performance of the index build. Our plan is to reuse the infrastructure introduced by ROIR and additionally introduce a resumable SORT operator as part of the index build query plan that will allow the sorting process to resume after a failure.

Moreover, we are planning to extend the support for resumable operations to Columnstore indexes. Even though the fundamental logic remains the same and the Columnstore technology allows us to maintain the state of each row (as described in Section 4) in the additional "Delta Store" structures [7], we need to modify the way we track progress and resume the operation, since Columnstore indexes are not sorted based on the index key.

Finally, we are considering introducing resumability for a wider range of schema management operations, such as Online Alter Column. These operations are currently based on the Online Index Build infrastructure of SQL Server and, therefore, can leverage the ROIR technology.

7. RELATED WORK

Index management has been an active area of research. In the early 1990s, Mohan and Narang [10], as well as Srinivasan and Carey [15], published different algorithms that allow updates to be performed on a table while an index is being built, a technology that is generally known as Online Index Build. Mohan and Narang additionally describe the possibility of checkpointing the progress of the Index Builder in order to resume after failure, which is essentially the basis of the work presented in our paper.

A decade later, Graefe [2, 3] presented a mechanism for supporting incremental sorting and index builds using "Partitioned B-Trees" that allow building parts of the index independently and using them for query processing even before the overall index has been created. In 2011, Graefe et al. [4] published a paper describing the complexity of implementing "pause" and "resume" functionality for index build operations in a commercial DBMS and presented a high level design to support that. Our paper addresses those complexities and requirements.

Several commercial DBMSs currently support Online Index Build operations, while some of them also allow the operation to be resumed when specific failures occur. Oracle Database [13] supports building indexes online and allows users to resume the operation if there is a space allocation issue, but does not allow users to manually pause the operation or resume it after any other type of failure. IBM DB2 [5] and MySQL [12] also support Online operations, but don't provide resumability. Finally, Sybase [14, 16] allows for Online Index Rebuild, as well as resumable index reorganization, but it doesn't support rebuilding the index from scratch in a resumable fashion which also prohibits resumable index creation.

SQL Server is the first commercial DBMS that supports resuming index rebuild operations after any type of failure, including server restarts. Additionally, it is the first to allow users to pause and resume these operations at any time in order to free up system resources. The proposed design allows building a completely new index in a resumable fashion and therefore can be extended to support new index creation in the future.

8. ACKNOWLEDGEMENTS

We would like to thank all members of the ROIR 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.

9. REFERENCES

[1] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. A Critique of ANSI SQL Isolation Levels. *SIGMOD*, 1995. Pages 1-10.

[2] Graefe G. Sorting And Indexing With Partitioned B-Trees. *CIDR*, 2003.

[3] Graefe G. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, Vol. 38, No. 3, September 2006.

[4] Graefe G, Guy W, Kuno H. 'Pause and resume' functionality for index operations. *Data Engineering Workshops (ICDEW)*, 2011 IEEE 27th International Conference on 2011 Apr 11 (pp. 28-33). IEEE.

[5] 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

[6] Lamport, L. The Specification Language TLA+, Logics of Specification Languages. *Springer*, 2008, 616-620.

[7] Larson, P., Clinciu, C., Fraser, C., Hanson, E. N., Mokhtar, M., Nowakiewicz, M., Papadimos, V., Price, S. L., Rangarajan, S., Rusanu, R., Saubhasik, M. Enhancements to SQL server column stores. *SIGMOD*, 2013, Pages 1159-1168.

[8] Microsoft, Online Indexing Operations in SQL Server 2005, https://technet.microsoft.com/en-us/library/cc966402.aspx

[9] Microsoft, Online Index Operations in Books Online for SQL Server 2012, https://msdn.microsoft.com/en-us/library/ms191261.aspx

[10] Mohan, C, Narang, I. Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates. *SIGMOD*, 1992, Pages 361-370.

[11] Mohan, C., Haderle, D. J., Lindsay, B. G., Pirahesh, H., and Schwarz, P. M. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1):94–162, 1992.

[12] MySQL, Overview of Online DDL. https://dev.mysql.com/doc/refman/5.7/en/innodb-create-index-overview.html

[13] Oracle, Online Data Reorganization & Redefinition. http://www.oracle.com/technetwork/database/features/online-ops-087977.html

[14] Ponnekanti N, Kodavalla H. Online Index Rebuild. *SIGMOD*, 2000, Pages 529-538.

[15] Srinivasan, V., Carey, M. Performance of On-Line Index Construction Algorithms. *EDBT*, 1992.

[16] Sybase Adaptive Server Enterprise, Reorg command. http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.infocenter.dc36272.1600/doc/html/san1393051052682.html

[17] TLA+ Tools. http://lamport.azurewebsites.net/tla/tools.html