T. Grust et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2019),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2019 453

Partial Reload of Incrementally Updated Tables in Analytic Database Accelerators

Knut Stolze[1] Felix Beier[1] Jens Müller[1]

Keywords: accelerator, data synchronization, replication, mvcc

Abstract:

The IBM Db2 Analytics Accelerator (IDAA) is a state-of-the art hybrid database system that seamlessly extends the strong transactional capabilities of Db2 for z/OS (Db2z) with very fast column-store processing in Db2 Database for Linux, Unix, and Windows. IDAA maintains a copy of the data from Db2z in its backend database. The data can be synchronized in batch with a granularity of table partitions, or incrementally using replication technology for individual rows.

In this paper we present the enablement of combining the batch loading of a true subset of a table's partitions for replicated tables. The primary goal for such an integration is to ensure data consistency. A specific challenge is that no duplicated rows stemming from the two data transfer paths come into existence. We present a robust and yet simple approach that is based on IDAA's implementation of multi-version concurrency control.

1 Introduction

The IBM® Db2® Analytics Accelerator for z/OS (IDAA, cf. Fig. 1) [Pa15] is an extension for IBM's® Db2® for z/OS® database system (Db2z) [IB18]. Its primary objective is the extremely fast execution of complex, analytical queries on a snapshot of the data copied from Db2z. Many customer installations have proven that the combination of Db2z with a seamlessly integrated IDAA delivers an environment where both, transactional workload and analytical queries, are supported without negatively impacting existing and new applications. In fact, many new use cases and applications have been developed specifically because of the existence of IDAA and the performance it delivers on core business data. The achieved query acceleration for analytical workloads is at least an order of magnitude, often even exceeding that.

After the initial version of IDAA became available in 2011, functional enhancements were continuously added to expand the product's scope. Fine-granular data synchronization mechanisms, like *partial reload* and *incremental update*, were among the first enhancements.

[1] IBM Germany Research & Development GmbH, {*stolze, febe, jens.mueller*}@de.ibm.com

doi:10.18420/btw2019-27

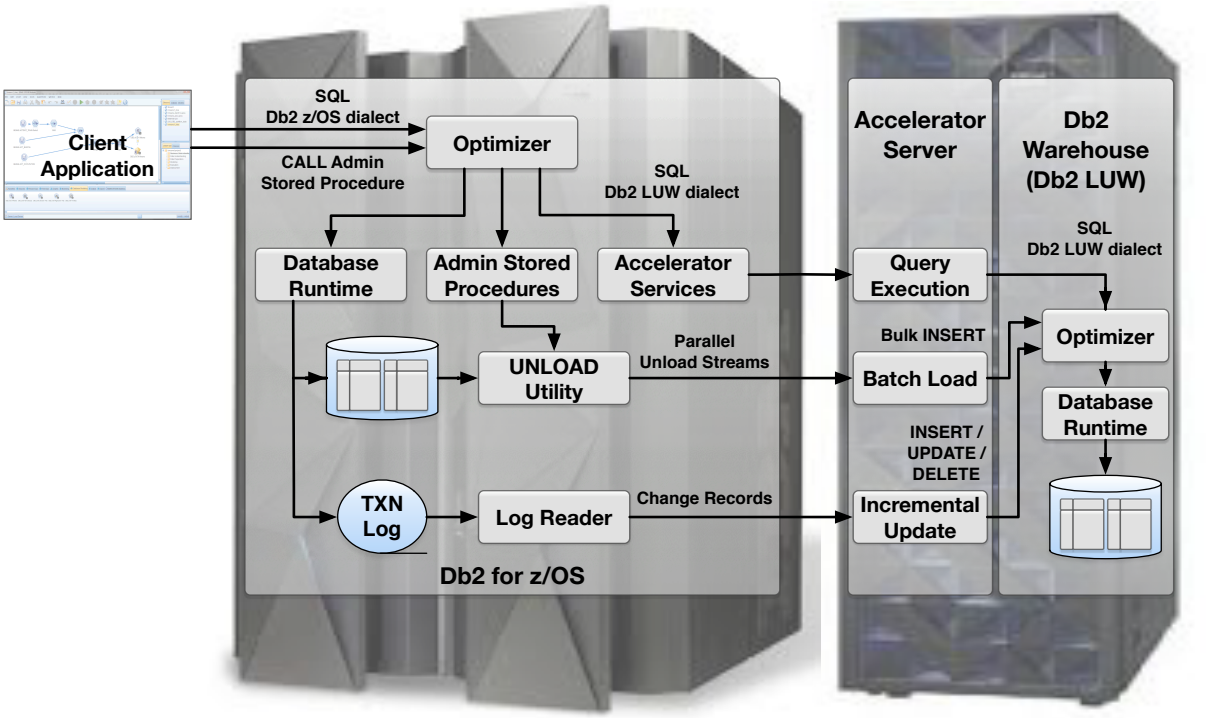

Fig. 1: System Overview of the IBM Db2 Analytics Accelerator for z/OS (IDAA)

Partial reload allows batch reloading of a subset of the table on a granularity of horizontal partitions. A table can be partitioned either by-range or by-growth. Contrary to that, incremental update operates on individual rows. It reads the transaction log of Db2 to detect data changes and replicates those to IDAA's backend database system.

Starting with IDAA Version 6, a strategic business decision heavily influenced IDAA and basically led to a new product: the backend database system changed to IBM Db2 Database[2] (for Linux, UNIX, Windows platforms) [IB16]. Db2 LUW does not provide multi-version concurrency control (MVCC) and snapshot semantics as Netezza [Fr11] did. In this paper, we present our work to implement MVCC on top of Db2 LUW. We developed a mechanism to allow Incremental Update to operate most efficiently, while allowing partial reload on such tables as well. The interaction of both data synchronization processes posed several challenges, and the solution to them is explained.

The remainder of the paper is structured as follows. The architecture of the IBM Db2 Analytics Accelerator is briefly touched on in section 2, including the simulation of MVCC in that environment. IBM's Change Data Capture (CDC) [Be12] is being employed as replication technology to detect and extract data changes in Db2z and apply them to IDAA's backend database. Section 3 outlines how CDC deals with IDAA's MVCC setup. How we bring CDC and partial reload together is explained in section 4. Finally, the paper concludes with a summary in section 5.

[2] IBM Db2 Database for Linux, UNIX, and Windows is called Db2 LUW henceforth.

2 IBM Db2 Analytics Accelerator

The IBM Db2 Analytics Accelerator (IDAA) [Pa15] is a hybrid system, which uses Db2® for z/OS® [IB18] for transactional workload with excellent performance characteristics, proven time and again in many customer scenarios. A copy of the Db2z data resides in the accelerator, which deals with analytical workload in an extremely high-performing way.

The benefits of this system are a reduced complexity of the necessary IT infrastructure on customer sites for executing both types of workloads and its tight integration into the existing Db2z system, which results in overall cost reductions for customers. A single system is used and not a whole zoo of heterogenous platforms needs to be operated and maintained.[3] Aside from the system management aspects, existing investments into a business' applications is protected, which is crucial for companies of a certain size. With IDAA, existing applications can continue to use Db2z unchanged. At the same time, the analytics capabilities of IDAA can be exploited without any (or just with minuscule) changes.

IDAA provides the data storage and SQL processing capabilities for large amounts of data. Fig. 1 already showed the high-level architecture. Key is the seamless integration of all components with Db2z to leverage the appliance as analytical backend. Db2z is the central entry point for queries and administrative functions. Its optimizer decides whether to execute queries locally in Db2z – or to route them to the accelerator. In the second case, a new SQL statement in the dialect of Db2 LUW will be generated by Db2z and sent to the Accelerator Server, which passes it on to Db2 LUW for execution. Db2 LUW itself optimizes the statement and executes it on its own copy of the data. The copy is created (and maintained) by the admin stored procedures (running in Db2z) that are provided as part of the IDAA solution. Thus, the interface for customers to work with IDAA is Db2z only.

2.1 Data Replication Strategies

IDAA offers three options for refreshing the data that has been shadow-copied from Db2z. Entire tables or individual table partitions can be refreshed in the accelerator batch-wise. The latter is called *partial reload*. Fig. 2 and 3 illustrate both batch-loading scenarios conceptually. The Db2z table is on the left side, and the IDAA table on the right. The sketch above the arrow shows which pieces of the Db2z table are copied to the IDAA table.

For tables having a higher update frequency and where a high data concurrency is desired, the Incremental Update feature (cf. Fig. 4) is suitable. This feature uses IBM® InfoSphere® Change Data Capture (CDC) [IB13a, Be12], which reads Db2z transaction logs and extracts all changes to accelerated Db2z tables. Unlike the batch-load strategies, CDC replicates only changed rows while unchanged data is not affected. A draw-back is the dependency

[3] Other systems like [KN11] may also provide an integrated solution. However, customers have made significant investments over the past decades into Db2z and applications on top of it. That's why IDAA has been developed as an enhancement of that platform right from the start.

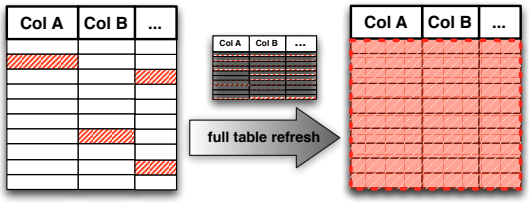

Fig. 2: Full Table Reload

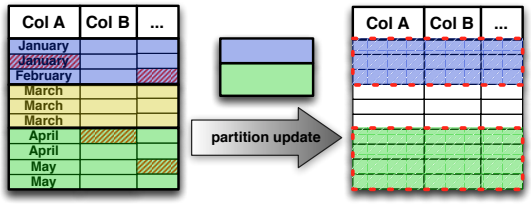

Fig. 3: Partial Reload

on the transaction log: any data maintenance operation that bypasses the log cannot be replicated. For example, batch-loading data into Db2z is often done using the `LOAD` utility, which does not log.

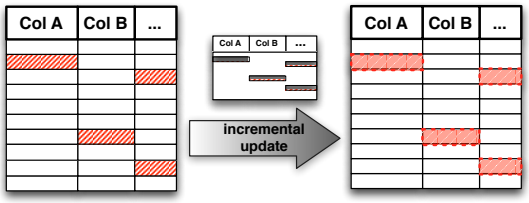

Fig. 4: Refreshing Table Data with Incremental Update

The batch-load strategies into IDAA offer great throughputs but require to copy large data volumes and, hence, are high-latency operations. The incremental update strategy replicates lower data volumes with larger overheads per tuple, but with low latency. It is the responsibility of the database user/administrator to trigger the refresh of the data in each accelerator or to set up incremental update where appropriate [IB13b]. Since many customers have tens of thousands of tables in Db2z and also in IDAA, it is important to understand query access patterns to the individual tables and analyzing the accelerated workload with the help of monitoring tools.

2.2 Simulating Multi-Version Concurrency Control using Views

Multi-version concurrency control (MVCC) [WV02] is a well-known means in database systems to enable highly parallel data access for read-only and read/write operations. The basic idea is to not apply data modifications in-place but rather to create a new version of the row on every such modification. The old row versions are still available for concurrently running transactions. The following Fig. 5 illustrates the physical storage of such rows. The arrows indicate the chain of versions that is created for a sample Row 1.

Each INSERT operation produces a new row, of course. But also each UPDATE effectively produces a new version of the row. The prior existing row remains, but it is marked as logically deleted. Likewise, DELETE operations only mark a row as logically deleted, but the row remains in existence.

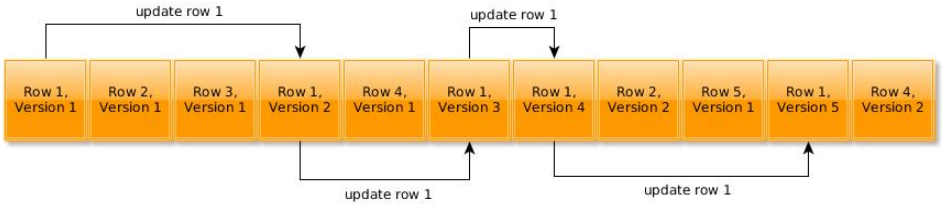

Fig. 5: Example for MVCC storage of rows

A key element of this approach is to determine which versions of the rows shall be visible to data access operations to provide consistent snapshots. For example, a query may access the table with the (physical) rows shown in Fig. 5. If the query (or rather its transaction) was started before any of the updates took place, it should see Row 1 in Version 1 – but not any other version of that row, which may have been created by concurrently running data modifications. Likewise, if the query starts after the 2nd update of Row 1, it should see that row in Version 3, but not any of the prior existing versions.

IDAA's Db2 LUW backend database system does not provide MVCC out of the box. Using SQL transactions to apply all data changes atomically is also not an option, due to limitations on the available transaction log space and lock contention that can occur. (Those are traditional issues with long-running transactions in database systems.) IDAA uses views to define data visibility and, thus, simulates MVCC. That permits small, arbitrary transactions to populate data in the backend and even committing the changes in between. All data ingested into the target table in IDAA via LOAD processing will be versioned along the way. Data visibility is not handled on individual rows but rather per *partition*. An artificial column (named *mapped partition ID* or *backend partition ID*) is added to the table in IDAA's backend to store the *mapped* value for a Db2z partition.

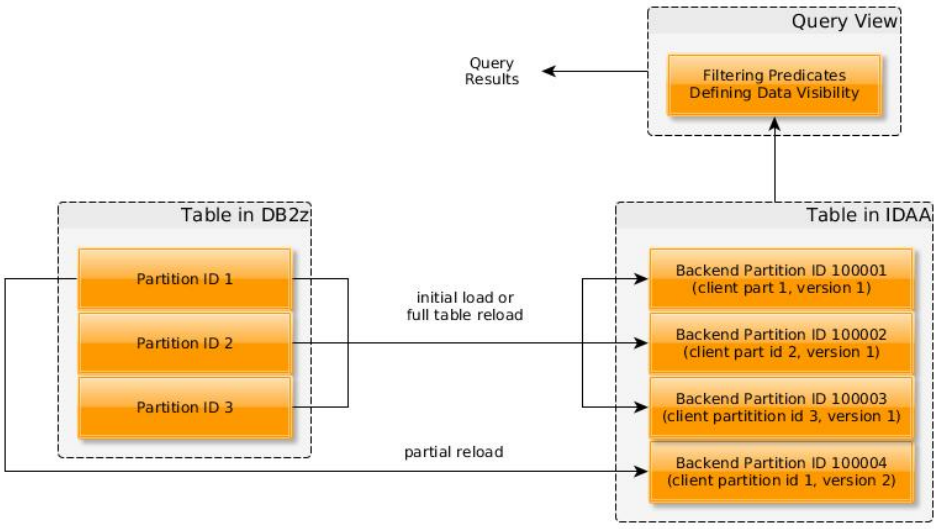

Fig. 6: Mapping of Db2z-side partitions to IDAA table

The view definition will use a simple predicate to ensure correct data visibility. The generalized pattern for the predicate is the following. The NOT IN predicate is only added if there are actually any invisible partitions.

```
mappedPartitionId <= maximum-mapped-id-of-all-visible-partitions AND
mappedPartitionId NOT IN ( list-of-mapped-ids-of-invisible-partitions )
```

For our specific example in Fig. 6, the predicate becomes:

```
mappedPartitionId <= 100004 AND mappedPartitionId NOT IN ( 100001 )
```

All LOAD operations are serialized on a table, which means that no concurrent LOADs can happen and potentially interfere. The values used for the `mappedPartitionId` column in the target table start at 100,000 and are monotonously growing. Thus, any rows that a later LOAD operation inserts into the table will automatically be invisible, whether those rows are committed already in the backend database or not. Once the LOAD operation finishes, it changes the predicates in the view definition, sets the new value for the maximum mapped partition ID, and updates the list of invisible partitions IDs. Fig. 7 visualizes this. Subsequent SQL statements, like queries, will pick up the new view definition while currently running SQL statements continue to use the view definition as it was in effect when the SQL statement started its execution.

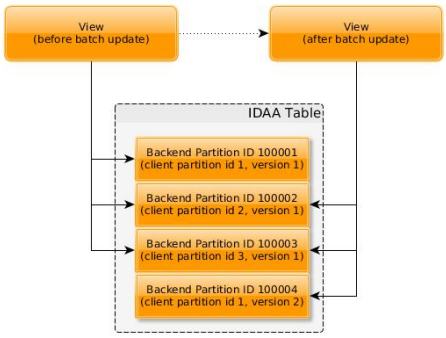

Fig. 7: View Update for Partial Reload

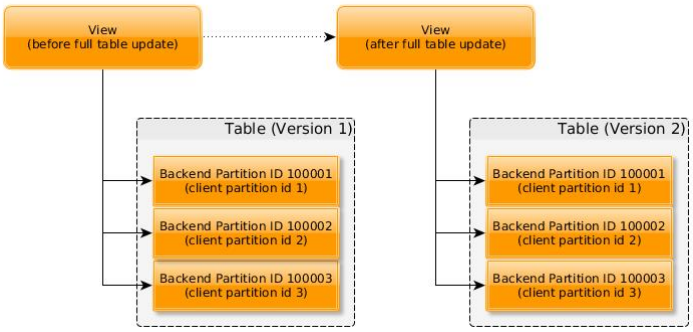

Fig. 8: View Update for Full Reload

Changing the definition of a view is an atomic and fast operation. Furthermore, view resolution takes place during the compilation of SQL statements. The compiled SQL statement remains unchanged during its execution – even if the view definition changes concurrently. Thus, stable and consistent results for the SQL statement are guaranteed.

Db2 LUW keeps track of which currently running SQL statements are using which version of the view definition. Due to that, it is possible to determine whether a specific version of the view definition is no longer in use. Once an old version of a view definition is no longer used, those invisible rows will not be accessed henceforth and can be physically purged from the table. In IDAA, this purging is implemented in asynchronous reorganization jobs that periodically check view usage and drop outdated data using a `DELETE` statement.

A special case occurs if all partitions of a table are to be reloaded. In such a *full table reload*, it is possible to create a new target table into which all rows are copied, and the view definition is changed to refer to that new table (cf. Fig. 8). This optimization drastically simplifies the physical cleanup of old data, because a `DROP TABLE` statement is sufficient.

3 Replication

CDC, the replication technology for IDAA's Incremental Update feature, is not aware of the mapping of the Db2z-side partition ID as described in Sec. 2.2. It would have been possible to extend CDC to perform that mapping, but that raises 2 major concerns: synchronization and performance.

Whenever a LOAD operation in IDAA happens, this mapping has to be updated. CDC has to be aware of the new mapping, and it has to be aware of it at the right point in time. Replication runs asynchronously, but changing the mapping has to happen synchronously. While this can be solved with well-known techniques, implementing them reliably and robustly requires a significant effort.

Such a synchronization of the partition ID mapping indirectly causes heavy performance impacts for the replication itself. A lookup via that map would have to occur for each row being replicated. Achieving throughput rates that can keep up with transactional workload occurring on Db2z-side is virtually impossible with such a bottleneck.

IDAA's Incremental Update takes a different route: No mapping of the partition ID occurs. Instead, all replicated rows get the Db2z-side partition ID assigned. Those partition IDs are in the range of 0 to 4096, which is lower than the starting value for mapped partition IDs. Due to the filtering predicate in the view definition (`mappedPartitionID <= x`) new replicated rows will be immediately visible once the Db2 LUW update transaction commits, which also physically deletes any old rows right away. Extending the example from Fig. 6 to include rows produced by replication in the target table leads to the situation in Fig. 9.

4 Handling Replicated Rows during Partial Reload

A partial reload on a replicated table runs into the issue that each loaded row carries a mapped partition ID greater than 100000. Any existing version of that row with a mapped partition ID assigned by IDAA during a prior full table load or partial reload does not pose a problem. Such a row will become invisible once the view definition is changed at the end of the LOAD operation. However, that very row may exist already in the target table with the Db2z-side partition ID – which is what the replication tool would generate.

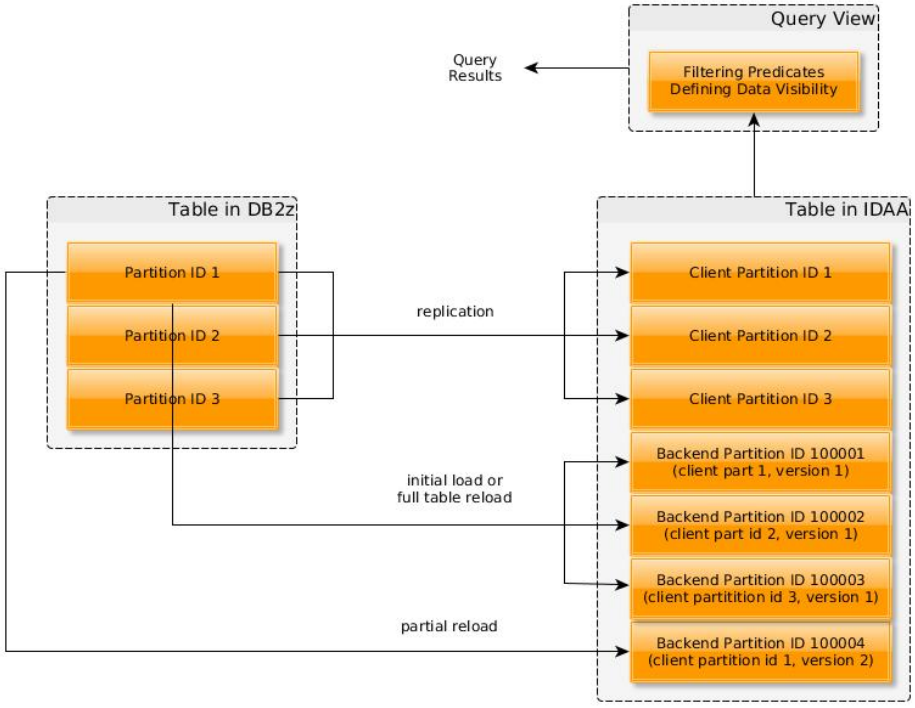

Fig. 9: Example for MVCC storage of rows

The filtering predicate in the view definition will always treat all rows with a Db2z-side partition ID as visible. That is, the Db2z-side partition ID must not be included in the NOT IN list and it must not be filtered out by any other means. The reason is that data changes occur in Db2z after the partial reload, and then those changes are replicated to the accelerator, using the Db2z-side partition ID. Propagating those changes again via a partial reload must not create duplicates, which implies that formerly replicated rows must be made invisible as part of the partial reload. Fig. 10 shows the desired result.

Several ideas came to our mind to solve this problem of hiding rows replicated before the partial reload but still allowing rows replicated after the partial reload to be visible. The three most promising are discussed in the following.

4.1 Deleting Replication Rows

The first idea to get rid of the old replicated rows is to delete them by a SQL statement like `DELETE FROM ... WHERE mappedPartitionId = ...`. That statement could be executed during the partial reload. However, that is not a viable option. Concurrently running queries may still need to access those rows. In fact, such queries may run longer than the partial reload itself. Physically deleting the rows would lead to incorrect query results.

4.2 Using Separate Tables for Partial Reload

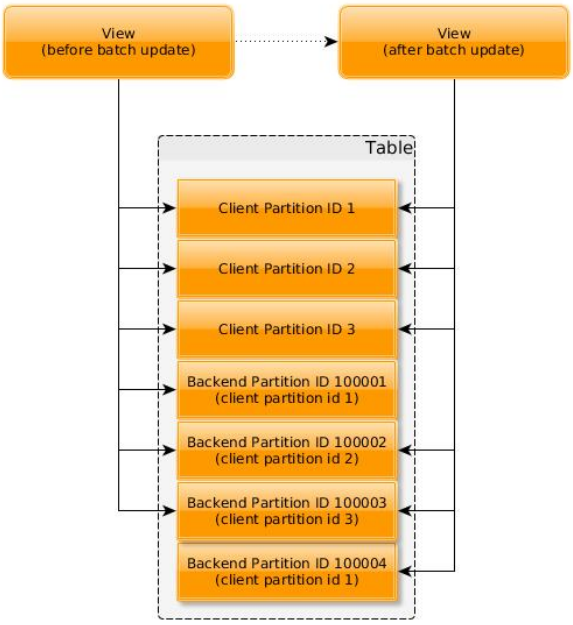

Fig. 10: View Update on Partial Reload with Replicated Data

Similarly to the optimization for a full table reload, partial reload could use separate tables for the newly loaded data. The view definition could be used to pick the data for each partition from the correct table and combine all of them with a `UNION ALL` clause. Correctness is not jeopardized, but this approach increases the number of base tables by several orders of magnitude. It also impacts SQL statements accessing the view and increases the pressure on the Db2 LUW optimizer to come up with a good access plan. Since IDAA is used by our customers to drive their business, maintainability is at least as important as correctness and performance. This idea was yet not further pursued since the next one has proven to meet our expectations with much less overhead and was easy to implement.

4.3 Moving Replicated Rows into Loaded Partitions

As part of the partial reload, all previously replicated rows of the affected partition(s) are updated. The value in the `mappedPartitionId` column are changed from the Db2z-side partition ID to the IDAA-defined partition ID – as it was before the update. Referring to Fig. 10, all rows of partition 1 (mapped to 100001) are being reloaded (now mapped to 100004). All rows that have partition ID 1 originate from the same Db2z-side partition. So the following SQL statement moves those rows to the old mapped partition ID 100001.

```
UPDATE ...
SET    mappedPartitionId = 100001
WHERE  mappedPartitionId = 1
```

Visibility of the rows being updated is not impacted – the rows are visible before the filtering predicates in the view are changed, regardless of whether the partition ID being 1 or 100001. After the filtering predicates are changed at the end of the batch update, those updated rows become invisible, exactly as intended. All the way, concurrently running queries can access the rows.

This `UPDATE` statement is executing concurrently to the insertion of the rows being reloaded with mapped partition ID 100004. The statement is actually slightly more complex to limit the number of rows being updated by one statement execution in order to inject intermediate `COMMIT`s. Using a sequence of small-sized intermediate transactions significantly reduces the amount of logging and locking that has to be done on the target table. In particular, lock escalations are avoided. For example, the following statement, which is executed in a loop until no more rows are found and modified, limits updates the first 100,000 rows only:

```
UPDATE ( SELECT ROW_NUMBER() OVER () as rowNumber, mappedPartitionId
         FROM   ...
         WHERE  mappedPartitionId = 1 ) AS table
SET    mappedPartitionId = 100001
WHERE  rowNumber BETWEEN < 100000
```

Evaluation We found that the `UPDATE` statement has about the same run time as an `INSERT` statement to process the same number of rows. Since the `INSERT` and the `UPDATE` can run concurrently, there is no measurable impact. Only when the system reaches its limits in terms of CPU and I/O resources, the `INSERT`s need to be throttled [SBM17].

5 Summary

In this paper we have presented recent enhancements of the IBM Db2 Analytics Accelerator to support reloading a selected subset of a table's partitions, which is also synchronized between Db2 for z/OS and the accelerator using the Incremental Update feature. Data consistency has to be ensured to avoid that duplicate rows become visible on accelerator-side due to the two different data synchronization paths. The solution turned out to be really simple but very effective at the same time: an `UPDATE` statement moves the replicated rows into the old mapped partition, which is subsequently made invisible by changing the filtering predicate of the view definition. Since this can be parallelized with the data transfer and insertion that has to take place as part of the partial reload anyway, no performance impact on a used system could be observed.

The next step for our work is the further investigation of reloading partition data into separate tables (see section 4.2). We strive to evaluate how well the Db2 LUW optimizer handles this on all possible cases. Functionality-wise, it is not yet clear to us if replicating the deletion

of a row would try to scan all legs of the UNION ALL or if the Db2 LUW optimizer can directly determine which of the base tables need to be processed and which ones can be skipped. A similar question arises for insertion of new (replicated) rows through the view and whether they will be applied to the correct target base table. It may become necessary to employ `INSTEAD OF` triggers on the view to ensure the correct row distribution, whose impact on replication performance and throughput needs to be evaluated.

Trademarks

References

[Be12] Beaton, A.; Noor, A.; Parkes, J.; Shubin, B.; Ballard, C.; Ketchie, M.; Ketelaars, F.; Rangarao, D.; Tichelen, W.V.: . Smarter Business: Dynamic Information with IBM InfoSphere Data Replication CDC. IBM Redbooks, 2012.

[Fr11] Francisco, P.: . The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. IBM Redbooks, 2011.

[IB13a] IBM: . IBM InfoSphere Data Replication V 10.2.1 documentation, 2013.

[IB13b] IBM: Synchronizing Data in IBM DB2 Analytics Accelerator for z/OS. Technical report, IBM, 2013. http://www-01.ibm.com/support/docview.wss?uid=swg27038501.

[IB16] IBM: . IBM Db2 Database for Linux, UNIX, and Windows Version 11.1, 2016. https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.welcome.doc/doc/welcome.html.

[IB18] IBM: . Db2 12 for z/OS documentation, 2018. https://www.ibm.com/support/knowledgecenter/en/SSEPEK_12.0.0/home/src/tpc/db2z_12_prodhome.html.

[KN11] Kemper, Alfons; Neumann, Thomas: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany. pp. 195–206, 2011.

[Pa15] Parziale, Lydia; Benke, Oliver; Favero, Willie; Kumar, Ravi; LaFalce, Steven; Madera, Cedrine; Muszytowski, Sebastian: . Enabling Real-time Analytics on IBM z Systems Platform. IBM Redbooks, 2015.

[SBM17] Stolze, Knut; Beier, Felix; Müller, Jens: Autonomous Data Ingestion Tuning in Data Warehouse Accelerators. Datenbanksysteme für Business, Technologie und Web (BTW) 2017, March 2017.

[WV02] Weikum, Gerhard; Vossen, Gottfried: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.