Short Paper

Snapshot Isolation for Transactional Stream Processing

Philipp Götze
Technische Universität Ilmenau
Ilmenau, Germany
philipp.goetze@tu-ilmenau.de

Kai-Uwe Sattler
Technische Universität Ilmenau
Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Transactional database systems and data stream management systems have been thoroughly investigated over the past decades. While both systems follow completely different data processing models, the combined concept of transactional stream processing promises to be the future data processing model. So far, however, it has not been investigated how well-known concepts found in DBMS or DSMS regarding multi-user support can be transferred to this model or how they need to be redesigned. In this paper, we propose a transaction model combining streaming and stored data as well as continuous and ad-hoc queries. Based on this, we present appropriate protocols for concurrency control of such queries guaranteeing snapshot isolation as well as for consistency of transactions comprising several shared states. In our evaluation, we show that our protocols represent a resilient and scalable solution meeting all requirements for such a model.

1 INTRODUCTION

Emerging application domains such as cyber-physical systems, IoT, and healthcare have fostered the convergence of stream and batch processing in data management. This can be observed not only by market trends such as real-time warehousing but also by architectural patterns like the lambda architecture aiming at combining batch and online processing on Big Data platforms.

This convergence means that the input to the system is treated as a continuous stream of data elements whose processing is implemented as a stream processing pipeline (query) with one or more persistent states/tables as sinks. Updates on these tables trigger further processing implemented again as stream processing pipeline. In addition, tables can be also queried in an ad-hoc fashion, e. g., for snapshot reports. Hence, the query part of an application is a mix of stream and ad-hoc queries while the data objects are streams and tables. However, concurrency and the need for providing fault tolerance require transaction support: at least stream queries writing to or reading from a table should run within a transaction context with ACID guarantees. This is necessary for recovery in case of failures and to provide a consistent view on (persistent) subsets of the stream (such as a window). Here, we refer to this processing style as *transactional stream processing* [2, 13] meaning that (a) a stream query writing to tables represents a sequence of transactions and (b) stream or batch queries on such tables require transaction isolation. Supporting such *queryable states* raises several requirements:

① state representations (tables) have to be queryable at all,
② the isolation property for concurrently running stream queries updating the state and ad-hoc queries on these states has to be guaranteed, and
③ consistency among multiple states of the same stream query is required even in the case of transaction aborts.

Figure 1 sketches a possible smart metering scenario which could benefit from transactional stream processing. Here, it is getting data from private households and the global infrastructure which is checked against respective specifications. It consists of three continuous and one ad-hoc query accessing various (shared) states whose semantic we explain in more detail in Section 3.

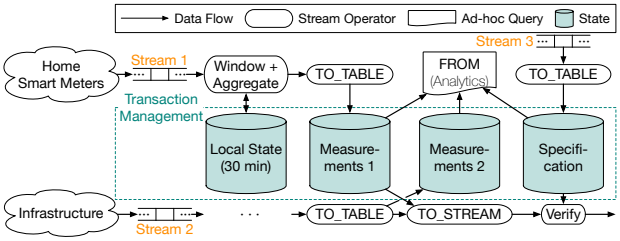

Figure 1: Possible smart metering use case.

In this paper, we present techniques to address the requirements above. Based on the work introduced in [2, 13], we propose a data-centric transaction model for data streams, discuss why and how to realize snapshot isolation on queryable states together with a consistency protocol. With the help of a micro benchmark, we show the suitability and scalability of our approach.

2 RELATED WORK

The first major investigation of a possible combination of relational and stream processing took place in the STREAM project [14]. However, when designing the system the team focused only on continuous queries instead of a hybrid query set of continuous and single point in time queries. To model the transformations of streams and relations, STREAM provides operations based on the whole relation (*RStream*), inserts (*IStream*), or deletes (*DStream*).

A more recent development of a system supporting transactions and data streaming is S-Store [13] which is built upon the main memory OLTP system H-Store [9]. S-Store inherits the ACID properties of H-Store by implementing data streaming concepts like windows and streams as time-varying H-Store tables with a timestamp as ordering property. Continuously running stream queries are implemented as stored procedures. Together with an input batch, which is the unit of a stream with the same timestamp, the execution of the stored procedure over such a batch forms a transaction over stream data.

Transactional Stream Processing [2] is a different notion to combine streaming and pure transaction processing. Their approach defines a unified transaction model which assigns a timestamp to each transaction and transforms continuous queries into a series of one-time queries, where each one-time query reads all required data before writing the result set. Transactional Stream Processing is based on a storage manager which transfers data processing into a producer-store-consumer scenario and identifies different properties to provide the proper storage for a producer-consumer combination.

In the context of scalable data stream processing platforms, Apache Flink Streaming is one of the most advanced approaches to consistent states and fault tolerance [3]. Their mechanism is

based on distributed snapshots where stream barriers are moving along with the data through the dataflow. At operator level, the flow is aligned until all barriers with the same ID arrive and subsequently the current state is persisted in the state backend (e. g., HDFS). Whenever a failure occurs, the last complete snapshot is used for restoring a consistent state per streaming pipeline.

In [15] the authors present a platform combining OLTP, OLAP, and stream processing in a single system. For that, they merge Apache Spark with an in-memory transactional store and enhance it by additional features such as approximate processing.

MVCC is one of the most widely used concurrency control protocols and has been implemented in, e. g., Postgres [18], HyPer [10], Hekaton [5], and SAP HANA [12]. No precise standard exists that describes how MVCC should be implemented. There are several possibilities and adjustments that strongly depend on the expected workload. In [20] the authors present an extensive study of key design decisions when implementing MVCC in an in-memory DBMS. In particular, these are concurrency control protocol, version storage, garbage collection, and index management. We have used the findings of this study to design our own MVCC approach (see Section 4).

3 TRANSACTION MODEL

Addressing the scenario and requirements described in Section 1 requires to distinguish two basic concepts: *tables* for representing states and *streams*. Tables represent a finite collection of data structured in rows and columns, as known from relational DBMS. Streams define a potentially infinite sequence of tuples of data, where tuples carry an implicit or explicit ordering. While a table requires a physical storage representation, streams are volatile.

Linking operators. As already proposed in the query language for STREAM [1], two classes of query operators are required to link these objects: stream-to-table and table-to-stream. Here, we call them `TO_TABLE` and `TO_STREAM`.

- `TO_TABLE` inserts, deletes, or updates tuples from a stream in a table, e. g., to update an operator state, and
- `TO_STREAM` produces a stream of tuples from a table.

Whether a stream tuple is inserted or updated in a table depends on the presence of a table tuple with the same key. A delete occurs if the tuple is outdated (e. g., from a window) or explicitly removed by a *delete tuple*. `TO_STREAM` unifies the two cases, where stream tuples are just incrementally processed and, on the other hand, table wide operations are executed before new tuples can be emitted. Whenever a certain condition on a table is fulfilled, `TO_STREAM` is executed and emits a new (set of) tuple(s) to a stream. In addition to these operators, a standard ad-hoc query operator `FROM` is required to either attach to a stream, i. e., read all tuples of the stream starting at the point of attachment, or to read data of a table. Figure 2 illustrates these operators.

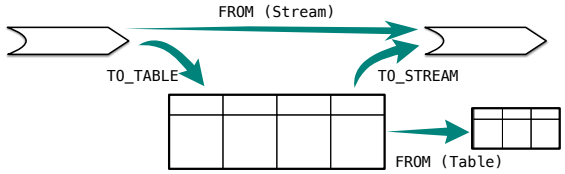

Figure 2: Linking operators.

Transaction boundaries. A first question is how to define transaction boundaries for data streams. Apart from the rather trivial case where each stream element represents its own transaction (aka "auto-commit"), two basic approaches can be distinguished. In the *data-centric* approach, transaction boundaries (`BOT`, `COMMIT`, `ROLLBACK`) are marked by dedicated stream elements, whereas the other stream elements are interpreted as insert or update operations. *Punctuations* [19] or *control tuples* are useful concepts for this purpose. This way each transaction can be defined over a consecutive number of stream tuples. Thereby, a transaction span can range from the length of an entire stream to the length of a sub-stream or each tuple is considered a single transaction. An alternative strategy is the traditional *query-centric* approach meaning that transaction boundaries are specified as part of the query or dataflow program. Obviously, this approach is better suited for ad-hoc and not for stream queries.

Unified tables for queryable states. For our system model, we rely on a unified table model taken from the DBMS world to cover all relevant aspects needed for transactional stream processing. However, stateful stream operators such as windows or aggregates also require structures for maintaining operator related data. Such operators exploit tables as internal structures to publish their state as a table allowing to share their content with other queries or to query the content. Besides sharing operator states, this approach also provides the advantage of re-using persistence and recovery mechanisms.

Transactional semantics. Inserts, deletes, or updates on states/tables need to run in a transactional context to guarantee atomicity for writes to a table and isolation for reads. The only way to modify a table in our model is provided by using the `TO_TABLE` operator which has to guarantee atomicity based on the transaction boundaries. Because a single stream query might contain multiple operators maintaining a persistent state, consistency among multiple states is a further requirement. This means that all states updated within the same stream query should be updated atomically with each commit, i. e., another query reading these states should see updates from the same (most recent) committed transaction. For reads of the `FROM` operator, we have to consider isolation properties. This also applies if `FROM` provides access to a data stream: here different isolation levels should provide different levels of visibility. For reads in `TO_STREAM`, we have to consider the trigger policy in addition to the isolation property. Trigger policy means the condition when a stream element is produced (`TO_STREAM`) or modifications in form of one or more transactions are generated into a back-to-the-table-directed stream. Here, possible policies are to consider each tuple modification or to rely on transaction commits.

4 TRANSACTIONAL STATE MANAGEMENT

From the transactional semantics described in Section 3, the following requirements can be derived that correspond to the ACID principle applied to the transactional stream processing model. First, there is atomicity, which occurs in several aspects. As mentioned before, the scope of a transaction that should be executed atomically must be marked via certain transaction boundaries. In addition, the individual operations must also be performed atomically in order to ensure both fault tolerance and consistency in case of concurrent access. This leads directly to the next two requirements: persistence and isolation. The latter property must ensure that both continuous and ad-hoc queries do not influence each other in terms of correctness and consistency. Furthermore, consistency must be ensured even if a query accesses multiple states, also in the event of transaction aborts. The persistence requirement means that the results of successfully committed transactions are still available after a system restart or crash. This goes hand in hand with recoverability, which must ensure that

the states are brought back or always stay in a consistent form. Taking these requirements into consideration, we have realized a snapshot isolation approach comprising three components:

- multi-versioned data structures for queryable states,
- a transaction protocol to access these states, and
- a protocol guaranteeing consistency among multiple states.

These components are prototypically implemented as part of our data stream processing framework PipeFabric[1]. We opted for an MVCC approach as it has proven to be the most scalable and widely used concurrency control protocol in the literature for DBMSs [4, 16, 20]. We expect it to behave similarly in a transactional stream processing environment. However, this still has to be revealed, which we will pursue in the following sections.

4.1 Data Structures

As transactional state representation, we have designed a table wrapper as shown in Figure 3. For the base table, any existing backend structure with a key-value mapping can be used. Therefore, every state type can use a suitable underlying structure making our design extremely versatile. Here, each key is mapped to an MVCC object. An entry in this object corresponds to the typical structure for MVCC [17, 20] ($\equiv <[cts, dts], value>$). The commit and deletion timestamp (`CTS/DTS`) indicate the lifetime of the value version. With the help of a bit vector (`UsedSlots`) the available free version slots are managed.

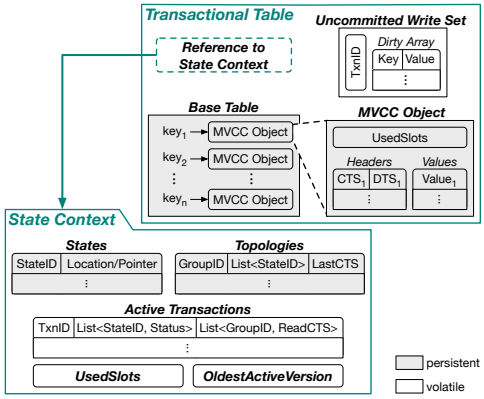

Figure 3: Transaction components.

Before new versions become visible to readers the changes are transiently stored as `Uncommitted Write Set`. This enables simple and fast aborts and also prevents the mixing of committed and uncommitted versions. Furthermore, the transactional table has a reference to the global context which contains all necessary runtime information. It consists of metadata about the states, topologies, and active transactions. For the states it contains general information such as their ID and physical location (e. g., file system path). In PipeFabric a query is written by defining a so-called *Topology*. It can be seen as graph where each node is an operator and the edges represent their subscribed streams. Each state is part of at least one topology for which we track the states that must be written together atomically. This is necessary for the correct application of the consistency protocol (see Section 4.3) for which the last committed transaction (`LastCTS`) per group is recorded. For recovery purposes, this information needs to be persistent. At the beginning of each transaction, it is assigned a unique timestamp (`TxnID`). All timestamps are logical and generated by a global atomic counter. For currently running transactions, we save a list of accessed states containing their ID and status (`Active`, `Abort`, or `Commit`) as well as the global commit timestamp at the time of reading for each topology. Again, we use a bit vector[2] to atomically manage the available slots. For garbage collection, we clean up old versions on demand (using `OldestActiveVersion`), i. e., if a new version has to be created and no space is available in the version array. The state context of the transaction management is completely latch-free and solely uses atomic instructions.

4.2 Concurrency Protocol

Before considering global consistency, we look at the basic operations required, namely *read*, *write*, *commit*, and *abort*. For a better separation of the protocols, we first assume that only a single state needs to be accessed consistently at a time. We further assume a beginning punctuation to signal the start of a transaction that assigns a timestamp and registers it in the context. To synchronize the actual access of MVCC blocks a lightweight locking strategy with read-write locks (latches) can be used.

The *read* operation starts by checking whether the accessing transaction has already written a new value (`Uncommitted Write Set`) and returns it. Otherwise, the latest visible version will be looked up using the commit and deletion timestamps. To achieve snapshot isolation, the first read version timestamp of this transaction must be transiently stored and used for subsequent reads (`readCTS`). The found value is finally returned to the caller.

When a transaction wants to *write* a new value, it is merely appended to its write set (`Dirty Array`). In case of a single writer, no exclusive locks are required and writes never block. For multiple writers, it could be checked if write sets overlap and then prematurely abort/restart the later transaction. Alternatively, this could be done only at commit time to prevent slower writes.

As all uncommitted changes are stored transiently together, it is enough for the *abort* operation to simply clear the corresponding write set and release the memory. The *commit* operation, on the other hand, is a bit more complex since all changes must be applied atomically and isolated from other readers. For each modification the MVCC object is loaded, the position of the new value is determined, and the position of the current value is looked up. The changes are then applied in memory. If no free insert position could be found, the garbage collection is performed at this point. Subsequently, the changes are populated atomically and isolated into the base table. As a final step, the global commit timestamp of the table is changed to the committing transaction's ID. By atomically setting this timestamp, the protocol can guarantee that the changes of a transaction are either visible completely or not at all. In the case of multiple writers, additional write locks are introduced and the order of the commits must be checked. If the current version is greater than the timestamp of the transaction, it must abort (First-Committer-Wins rule).

The way we designed our operations eliminates the need for undo operations within the actual table. In addition, read operations are generally not blocked by write operations and vice versa. Only during the commit time, a short synchronization is required. Therefore, we expect the performance to remain stable even for high contention situations.

4.3 Consistency Protocol

If now a continuous query needs to update multiple states, they must become visible together to maintain consistency. Assume a simplified case where a data stream query writes two states and

[1] PipeFabric: https://github.com/dbis-ilm/pipefabric

[2] In fact, it is a 64-bit integer, which is updated by CAS operations.

a second (ad-hoc) query reads from both states. We coordinate the operators with the help of the state context (see Figure 3). Whenever a commit arrives the corresponding status flag for this transaction and state is set to `Commit`. The modifications are not persisted until all states registered for this transaction are ready for commit. The operator that sets the last status flag to `Commit` becomes the coordinator and is responsible for the global commit. In addition, a transaction must be aborted globally as soon as `Abort` has been flagged for at least one state. In this respect, this is a modified version of the 2-Phase-Commit protocol [11] relying on proven concepts which adds almost no overhead in our case.

Readers can see the last completed transaction using `LastCTS` for each topology which is set at the end of a commit (instead of the transaction ID as described before). They must also check in the context which states are written together. For these, the version must be the same, otherwise, a commit has been executed in the meantime. Therefore, the read version is noted within the context (`ReadCTS`) and is only set at the first read per topology. Thus, every operation reads from the same snapshot and interleaved commits do not pose a problem. If there is an overlap when reading multiple topologies with different versions (`LastCTS`), the older version must be read to guarantee consistency.

5 EVALUATION

In this section, we present a micro benchmark to substantiate the suitability and scalability of our approach. For this, we evaluate our MVCC protocol against a simple strict two-phase locking (S2PL) [6] and a backward-oriented optimistic concurrency control (BOCC) [8] protocol. We have made every effort to implement optimized versions of each protocol. All concurrency control protocols use fundamentally the same consistency protocol for multiple states as described in Section 4.3.

5.1 Setup and Workloads

The experiments were run on a 2-socket Intel Xeon E5-2630 each 6 cores à 2 threads, 128 GB DDR3, Linux kernel 4.15, and GCC 7.3. As a base table, we use a persistent key-value store, namely RocksDB[3]. It has a log-structured merge-tree (LSM) design and provides many configuration options for a wide range of requirements. We kept the default configuration and only set the *sync* option to true to guarantee failure atomicity. As a benchmark, we use a scenario having one stream continuously writing to two states and multiple ad-hoc queries reading from these states. Both are initialized with a table size of one million key-value pairs (4 Byte key, 20 Byte value). During the experiments, we vary the number of parallel ad-hoc queries and the contention rate using a Zipfian distribution ($\theta = 2.9 \mathrel{\hat{=}} 82\%$ the same key) [7].

5.2 Performance Study

In the following, we compare the performance of all concurrency control protocols for transactions of medium length (10 operations each). An excerpt of our measurements is shown in Figure 4. Due to the synchronous writing, the readers (mostly only accessing memory) contribute almost exclusively to the total throughput. While the other two protocols are dropping, the MVCC protocol provides consistently a good performance. Interestingly, the BOCC protocol is slightly faster (~5%) than MVCC with little contention and many concurrent ad-hoc queries. However, this is logical as it is designed for scenarios with few conflicts. If the number of threads and the contention increases, it brings the

[3]RocksDB (version 5.15.10): https://github.com/facebook/rocksdb

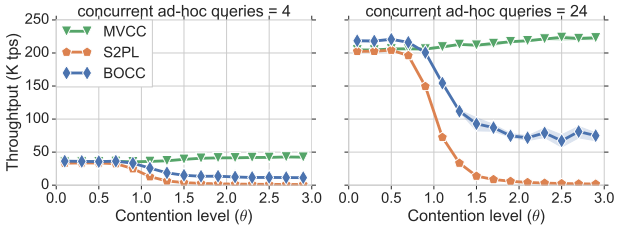

Figure 4: Contention and scalability check with persistent synchronous writes and medium-sized transactions.

S2PL and BOCC to their knees relatively quickly. It is noticeable that at least for MVCC caching effects are visible with a higher contention. Overall, it shows that our MVCC approach combined with the consistency protocol is highly scalable and resilient, making it well suited for transactional state management.

6 CONCLUSION

Transactional stream processing systems itself offer a wide range of research opportunities. In this paper, we presented a data-centric transaction model and investigated concurrency and consistency aspects within this model. We have found that our snapshot isolation approach meets all our initial requirements. We designed a versatile state representation which is queryable by both continuous and ad-hoc queries. Even under high parallelism and contention, the ACID properties could always be maintained with great performance even when involving multiple states.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG) within the SPP2037 under grant no. SA 782/28.

REFERENCES

[1] A. Arasu et al. 2003. CQL: A Language for Continuous Queries over Streams and Relations. In *DBPL*. 1–19.
[2] I. Botan et al. 2012. Transactional Stream Processing. In *EDBT*. 204–215.
[3] P. Carbone et al. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *PVLDB* 10, 12, 1718–1729.
[4] M. J. Carey and W. A. Muhanna. 1986. The Performance of Multiversion Concurrency Control Algorithms. *TOCS* 4, 4, 338–378.
[5] C. Diaconu et al. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*. 1243–1254.
[6] K. P. Eswaran et al. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11, 624–633.
[7] J. Gray et al. 1994. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD*. 243–252.
[8] T. Härder. 1984. Observations on Optimistic Concurrency Control Schemes. *Inf. Syst.* 9, 2, 111–120.
[9] R. Kallman et al. 2008. H-Store: A High-performance, Distributed Main Memory Transaction Processing System. *VLDB* 1, 2, 1496–1499.
[10] A. Kemper and T. Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*. 195–206.
[11] B. Lampson and H. E Sturgis. 1979. Crash Recovery in a Distributed Data Storage System. In *XEROX Research Report*.
[12] J. Lee et al. 2013. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.* 36, 2, 28–33.
[13] J. Meehan et al. 2015. S-Store: Streaming Meets Transaction Processing. *PVLDB* 8, 13, 2134–2145.
[14] R. Motwani et al. 2003. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *CIDR*.
[15] B. Mozafari et al. 2017. SnappyData: A Unified Cluster for Streaming, Transactions and Interactice Analytics. In *CIDR*.
[16] A. Pavlo and M. Aslett. 2016. What's Really New with NewSQL? *SIGMOD Record* 45, 2, 45–55.
[17] D. P. Reed. 1983. Implementing Atomic Actions on Decentralized Data. *TOCS* 1, 1, 3–23.
[18] M. Stonebraker and L. A. Rowe. 1986. The Design of Postgres. In *SIGMOD*. 340–355.
[19] P. A. Tucker et al. 2003. Exploiting Punctuation Semantics in Continuous Data Streams. *TKDE* 15, 3, 555–568.
[20] Y. Wu et al. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB* 10, 7, 781–792.