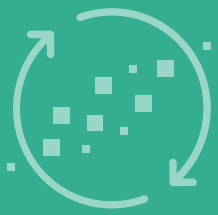


data Artisans Streaming Ledger

Serializable ACID Transactions on Streaming Data

Whitepaper

Patent pending in the United States, Europe, and possibly other territories

October 2018

Table of Contents

About data Artisans

data Artisans was founded by the original creators of Apache Flink®, a powerful open-source framework for stateful stream processing.

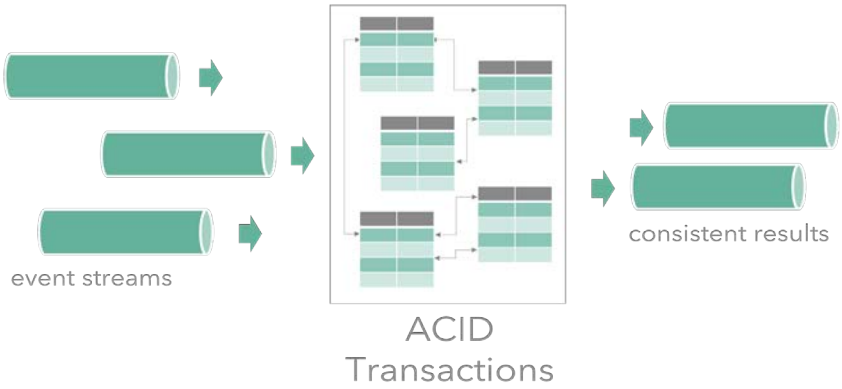
In addition to supporting the Apache Flink community, data Artisans provides data Artisans Platform, a complete stream processing infrastructure that includes open-source Apache Flink.

data Artisans Platform makes it easier than ever for businesses to deploy and manage production stream processing applications.

Introduction

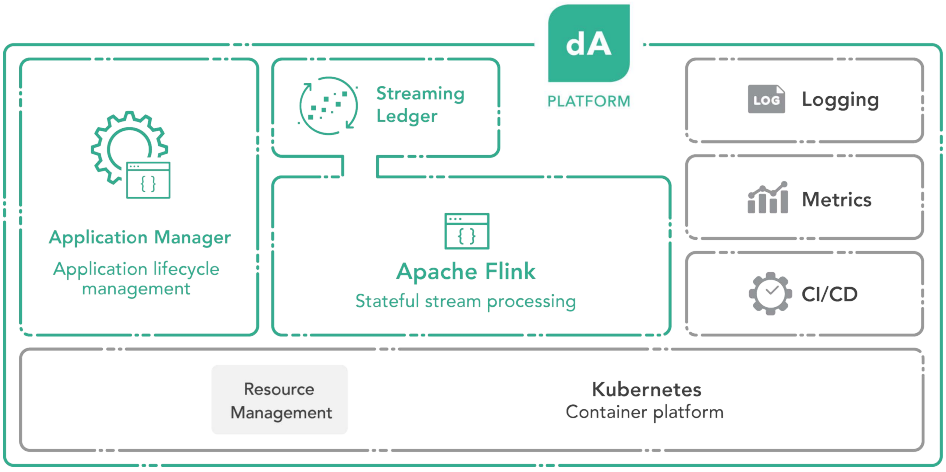
data Artisans Streaming Ledger, now available in the River Edition of data Artisans Platform, works as a library for processing event streams across multiple shared states/tables with Serializable ACID Semantics — the highest class of transactional consistency

With data Artisans Streaming Ledger, you define a set of tables, connect streams of events that drive the transactions, and define flexible transaction logic that processes the events and accesses/updates various rows across tables, under full serializable transactional consistency. Instead of operating on only a single key in a single operator at a time (like in vanilla Apache Flink and other stream processors), transactional functions operate consistently on multiple keys across multiple tables at a time. The same tables can be shared between various streams of events. All that without compromising performance or consistency.



Imagine going from a key/value store (with single row update consistency) to a relational database with full serializable transactions across rows and tables. In the same way, data Artisans Streaming Ledger goes beyond today's exactly-once stateful stream processing model and brings multi-row, multi-state, cross-stream transactions to data stream processing. This is not currently possible with any other stream processing technology.

With data Artisans Streaming Ledger, you can now move a new class of applications to stream processing: applications that previously needed to operate against relational databases for their more powerful consistency model (ACID transactions). These applications can now take full advantage of all the benefits of data stream processing and blend naturally into a streaming data architecture. Because data Artisans Streaming Ledger offers high throughput, larger scale applications can now be efficiently transformed to consistent streaming applications.



Streaming Data, Stateful Stream Processing, and its Limitations

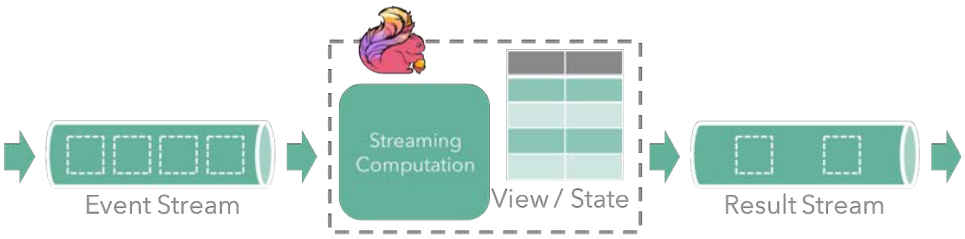
(Skip this section if you already understand stateful stream processing and Apache Flink's approach to stream processing)

Data Stream Processing (or event stream processing) is the paradigm of working on sequences of data events (i.e. streams). The events typically correspond to business actions (e.g. a customer putting an item in a cart, or a trade of assets being booked). The event streams can be consumed both in real time as well as in hindsight, making data stream processing a powerful paradigm for both real time online applications, as well as for lag-time and historic data analytics. "Stream Processors" are the systems that power data stream processing applications. Apache Flink is a powerful stream processor that runs some of the world's largest and most demanding stream processing applications.

Powerful stream processors, like Apache Flink, are able to persistently store and manage "state" that is derived from the sequences of events. This state is basically a view of the world resulting from what happened in the world (the events). Examples of such state can be:

- for each product, the number of users that viewed it in the last 5 minutes (analytics)
- the ongoing status of fraud detection patterns (complex event processing)
- for each user, the current status of their online interaction session (application state)
- the assets currently owned as the result of all trades (ground truth / master data)

In such applications, the stream processor manages what would otherwise typically be tables in a database. The input is the persistent sequence of events, stored either in a log, a message queue, a file system, or a combination thereof. You can draw parallels to database management systems here, where the persistent sequence of input events is the transaction write-ahead log, and the stream processor's state is the actual database tables — a view of the current state of things.



The advantages of realizing such an architecture based on stream processing rather than monolithic database management systems are many:

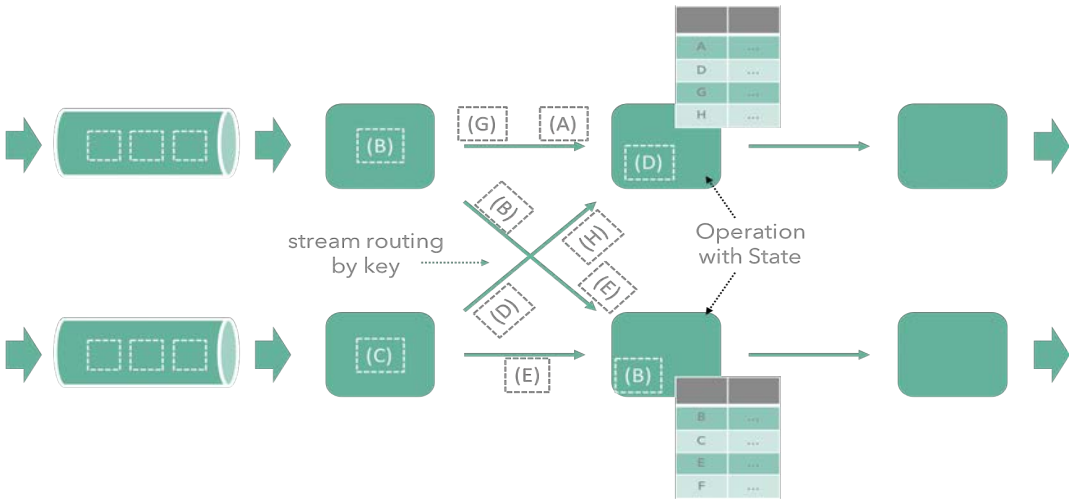
- Stream processors are much more flexible than SQL. They can compute, in real time, insights that databases cannot, and they interoperate easily with other libraries and frameworks, such as statistical and machine learning libraries.
- The fully asynchronous, event-driven nature of stream processing allows it in most cases to achieve a much higher throughput compared to what is possible with request/response against databases.
- Computed views can be added at any point by different organizations, by simply starting a new stream processing application that consumes the same source stream. New applications do not interfere with others and can be built without planning ahead.
- Stream processing computations can be flexibly derived from each other, thereby creating a simple and modular infrastructure architecture.

Parallel Distributed Stateful Stream Processing

The state of a stream processing application (the current computed view), typically stored in the form of key/value tables, can grow rather large. In the case of Apache Flink, some applications keep 10s of TBs of state in the stream processor.

Apache Flink shards this state in order to scale out to many parallel processes/machines, a strategy that is similar to many database systems and key/value stores. The key of a table's row determines on which process/machine it is stored.

The parallel operations (transformation functions) that interact with that state are always co-located with state: Parallel instances of the transformation functions are executed with each shard of the state, and the events are routed to the functions such that each function receives only the events that will interact with its shard. Consequently, all state access and modifications are shard-local.

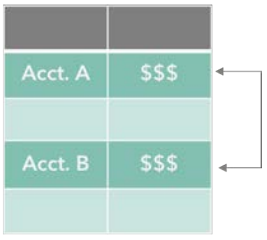


This execution model "*moves the computation to the data*": The events that trigger the computation are routed to the relevant state/table shard. In contrast to "pulling the relevant state to the compute", this model is very efficient, enables the high-throughput flow of events, and offers strong consistency. There is effectively an exclusive reader/writer per shard, so individual event interaction with its shard of state is trivially consistent. Effectively, this is an efficient way to do "single row transactions" (as offered by many key/value stores or cloud databases) from a data stream.

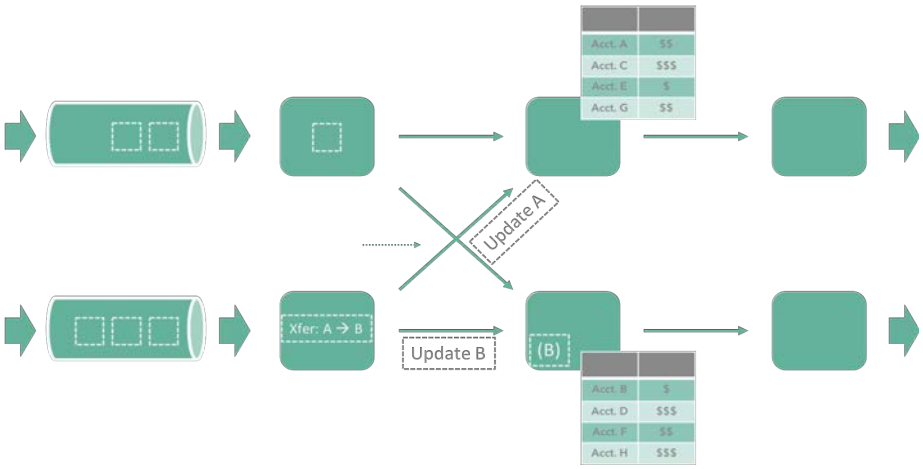
The Limits of Current Stream Processors

Note: This section is not specific to Apache Flink — the limitations discussed here apply to all parallel stream processors today. We use Flink here merely as an example, because it offers best-of-breed support for stateful operations — hence the limitations are at least as severe in other distributed stream processors today.

The limits of the one-key-at-a-time (or one-shard-at-a-time) model can be illustrated with the textbook example for ACID transactions in relational database management systems: A table with account balances and transferring money from one account to the other.



In this example, transactional correctness is a must: The transfer must change either both accounts or none (atomicity), the transfer must only happen if the source account has sufficient funds (consistency), and no other operation may accidentally interfere and cause incorrect result (isolation, no anomalies). Violation of any of these aspects inadvertently causes money to be lost or created, and accounts to have an incorrect balance.

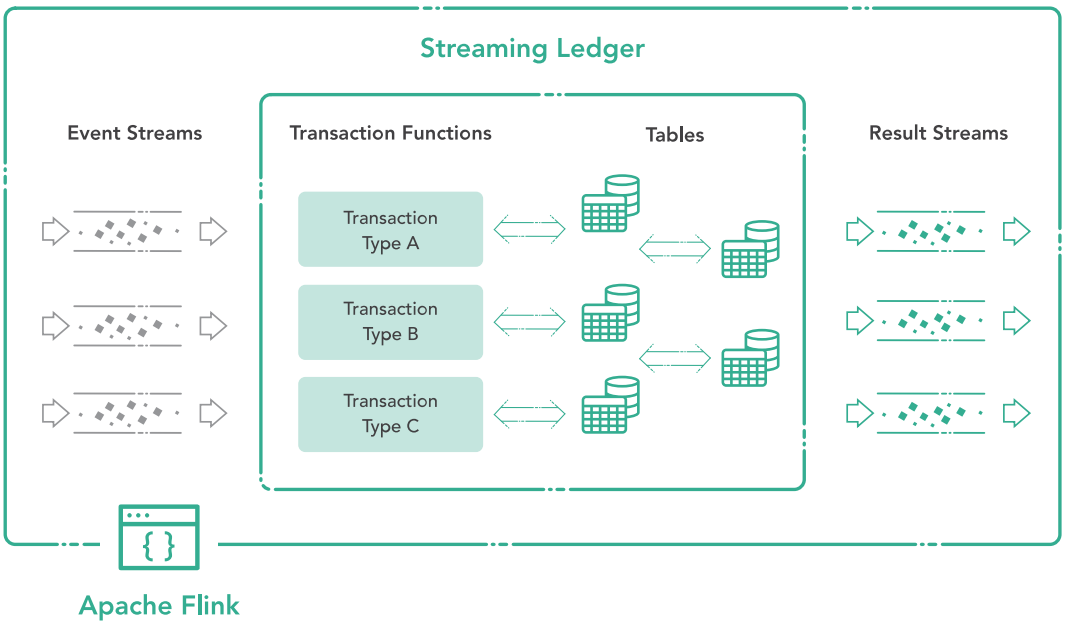


Implementing this in a stateful stream processor is illustrated in the figure above: The account ID is the key, and for each transfer, the two involved accounts will, most likely, be in different shards. The transfer event needs to send an update to both account IDs. When processing the update for account A, there is no way to assess whether all pre-conditions are fulfilled from Account B's perspective. Neither of the shards has a consistent view of the other's status. The example becomes more complex in many real-world scenarios, where additional constraints exist on more accounts (like total liquidity within a division or the transfer has to happen atomically together with a swap of digital assets in another table).

To overcome this, one needs to communicate status between the shards, with mechanisms to provide a consistent view on state, manage concurrent modifications, and ensure atomicity of changes. That is precisely what distributed databases implement in their concurrency protocols. Doing this efficiently is hard, and the reason why many systems do not support these types of operations.

Serializable Multi-key Multi-table Transactions on Streaming Data

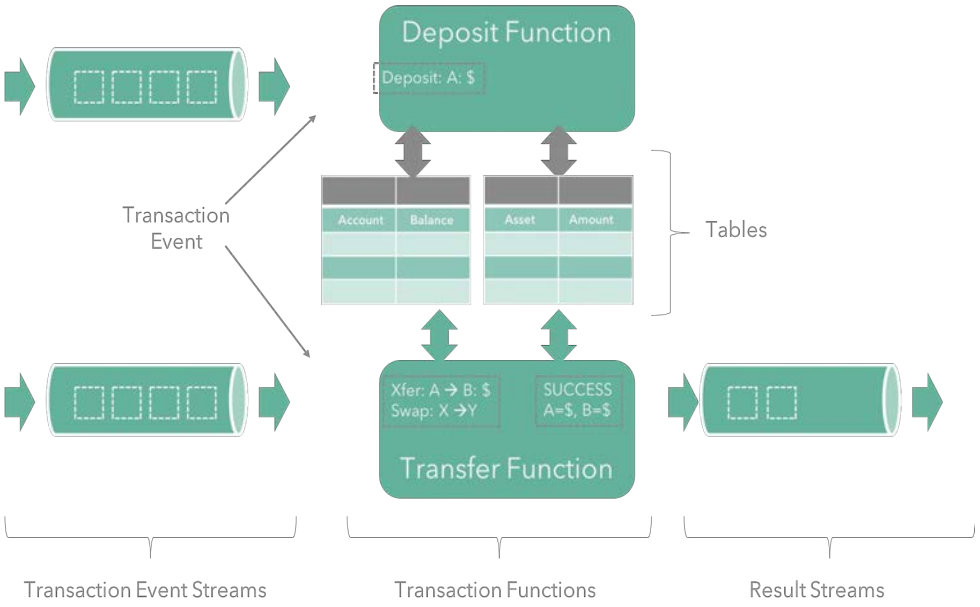
data Artisans Streaming Ledger overcomes these limitations by extending Apache Flink with the ability to perform serializable transactions from multiple streams across shared tables and multiple rows of each table. Think of it as the data streaming equivalent of running multi-row transactions on a key/value store or even across multiple key/value stores. Similar to serializable ACID transactions in a relational database management system (RDBMS), each function modifies all tables with ACID semantics under isolation level "serializable", which guarantees full data consistency the same way as the best relational databases do today. This makes it possible to move a whole new class of applications to a data streaming architecture. To the best of our knowledge, no other parallel stream processor exists today that has these capabilities.



data Artisans Streaming Ledger embeds transparently in Apache Flink. It uses Flink's state to store tables, therefore no additional systems or storage needs to be configured. Unlike classical lock-based transactions, the implementation does not rely on distributed locking, and is fast and scalable. Unlike optimistic concurrency control with timestamps, transactions cannot fail due to conflicts, so pathological degradation to situations where the system is only busy with retrying transactions will never happen. Instead, the implementation uses a unique combination of timestamps, watermarks, and out-of-order scheduling to achieve serializable transaction isolation.

Tables, Event Streams, and Transaction Functions

The core building blocks of data Artisans Streaming Ledger applications are Tables, Transaction Event Streams, Transaction Functions, and optional Result Streams. To explain each of them, we look at the example of transferring money and digital assets between accounts and books.



Tables

The state of a streaming ledger application is maintained in one or more key/value Tables, which are transactionally updated by the Transaction Functions. A table is a collection of rows, each row uniquely identified by a key.

The Tables are stored in Apache Flink's state with all the typical properties: Tables are persisted as part of checkpoints. They can be stored purely in memory or in RocksDB, depending on the configured state backend. The table keys and values can be any type (structured row, object, blob, etc.).

The example above has two tables: Accounts and Asset Ledger, in each case identifying rows by a unique ID.

Transaction Event Streams

Following the stream processing paradigm, transactions are driven by events, here called *Transaction Events*. Events flow in parallel streams, triggering transactions in parallel as they are received. Multiple streams with different event types can feed into the streaming ledger — typically one stream per type of transaction. In our example here, we have a stream with Transfer Events and one with Deposit Events. The events carry the "parameters" to the transaction, e.g., the rows needing to be modified, as well as some information about the type of modification. In the example above, the Transfer Events hold the account IDs of the source and target accounts, the IDs of the ledger entries to be updated, plus the amount to be transferred and possibly a precondition (such as minimum account balance, etc.)

Transaction Functions

There is one Transaction Function per Transaction Event Stream, and the Transaction Function contains the transaction's business logic. Transaction functions are like transformation functions in Apache Flink, triggered when a transaction event is received, and given access to the relevant rows. The main difference with core Apache Flink functions is that different Transaction Functions share access to the same Tables and can access and modify multiple rows/keys at a time while maintaining strict consistency.

The transaction functions can contain complex business logic, implemented in Java/Scala. That makes the streaming ledger usable in cases where complex logic is needed to consolidate an event with the current state. You can think of the Transaction Functions as powerful stored procedures that execute directly in the system that manages the data, rather than pulling the data out and pushing updates back in.

In our example, there would be two Transaction Functions: One to process Deposits, one to process Transfers. The functions are triggered with the respective event types and adjust the rows in the tables according to their logic. The Transfer Function might also decide to not perform any modifications to the table, for example if the source account does not have a minimum balance, or if the asset to swap is not available.

Result Streams

A transaction function may optionally emit events into a result stream. This can be used to communicate success or failure, for example based on satisfying the preconditions that the Transaction Function checks.

Result streams can also send the updated values out in order to create a mirror of the Tables in an external system. In that case Apache Flink and data Artisans Streaming Ledger perform the hard task of ensuring consistency in the transactions, while another system (for example a key/value store makes the state accessible to other consumers.

In the example here, the Transfer Function would send out a result event in order to indicate whether the transfer happened or was rejected.

ACID Semantics

The implementation of the transaction obeys ACID semantics with serializable transaction isolation:

A — Atomicity: The transaction applies all changes in an atomic manner. Either all of the modifications that the transaction function performs on the rows happen, or none.

C — Consistency: The transaction brings the tables from one consistent state into another consistent state.

I — Isolation: Each transaction executes as if it were the only transaction operating on the tables. Databases know different isolation levels with different guarantees. data Artisans Streaming Ledger here offers the best class: serializability.

D — Durability: The changes made by a transaction are durable and are not lost. Durability is ensured in the same way as in other Flink applications — through persistent sources and checkpoints. In the asynchronous nature of stream processing, durability of a result can only be assumed after a checkpoint (for details, see the later section on the consistency model).

Use Cases and Examples

We have built data Artisans Streaming Ledger following a set of real-world use cases that we observed during our joint work with leading adopters of stream processing. data Artisans Streaming Ledger solves challenges at the intersection of streaming event data and consistent consolidated state. Below is a selection of such use cases.

Streaming Transactions across Tables

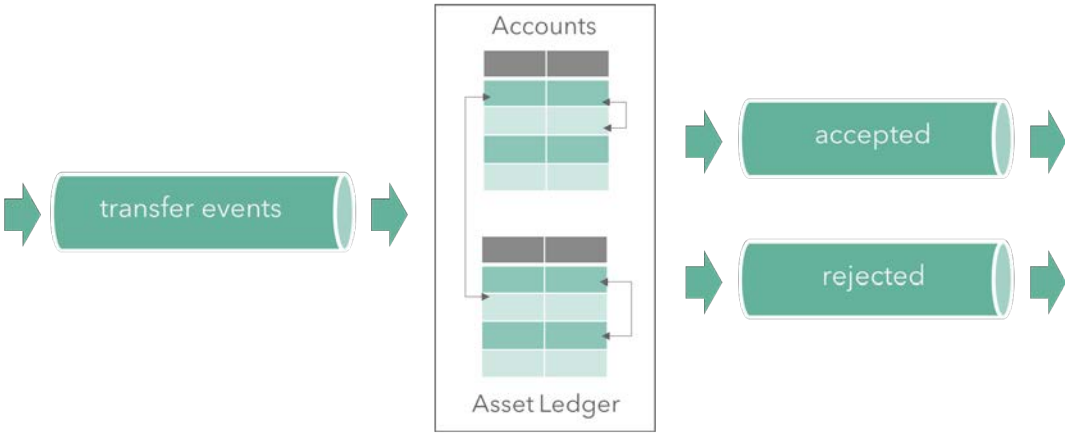
As illustrated in the section above, a typical use case for data Artisans Streaming Ledger is to work with multiple states in a stream processing application. As with ACID transactions in databases, changes should be applied atomically (all or none), and the view of the state that the transaction sees should be consistent and isolated, so that no anomalies can occur.

This is straightforward in a relational database with serializable ACID transactions, but not in today's stream processors. data Artisans Streaming Ledger gives you exactly that as its core abstraction: Multiple tables that are transactionally modified as the result of streaming events and sets of modifications that are isolated against each other following serializable consistency.

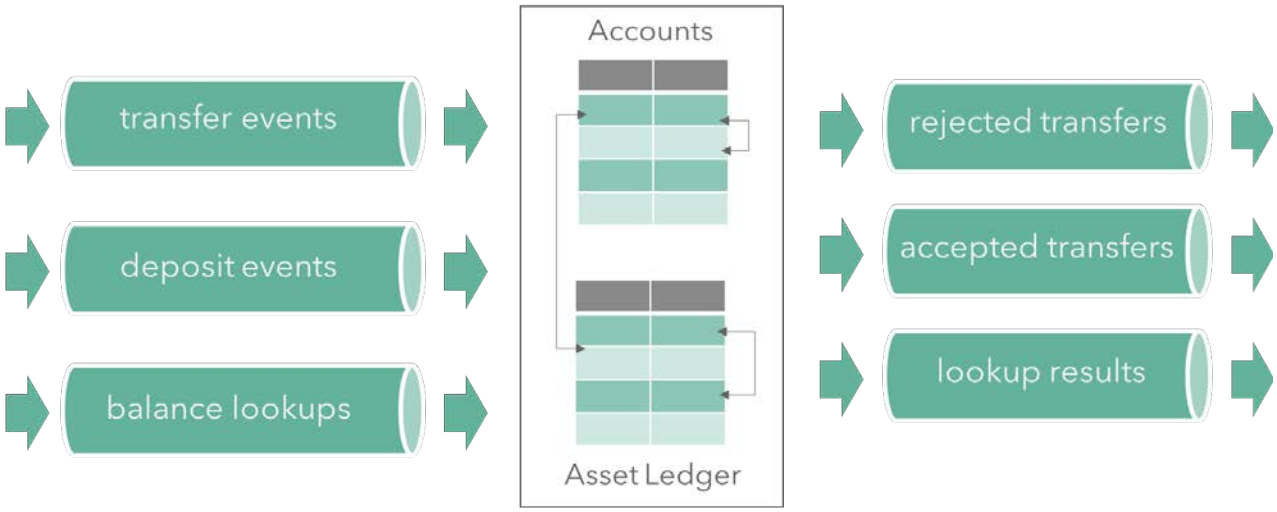
Example: Wiring money and assets between accounts and ledger entries.

In this example, the streaming ledger maintains two tables: "accounts" and "asset ledger". The streaming ledger consumes a stream of events that describe transfers between accounts, ledger entries, or both, possibly with additional preconditions.

When an event comes in, the transaction function accesses the relevant rows, checks the preconditions, and decides to process or reject the transfer. In the former case, it updates the respective rows in the tables. In both cases, the transaction function may optionally produce result events that indicate whether the transfer was accepted or rejected.



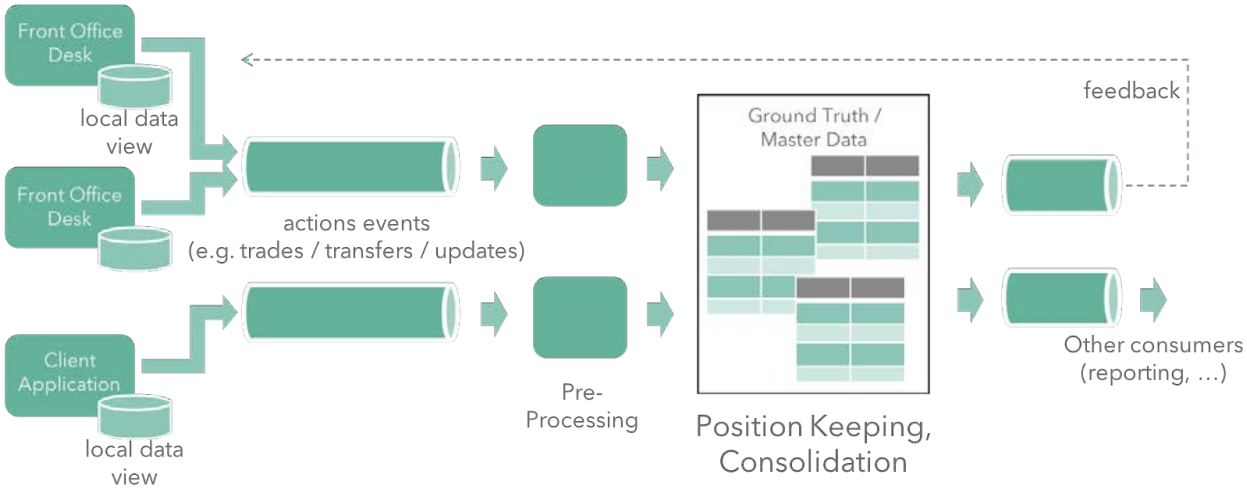
The example is easily extended to processing multiple streams with different actions: Deposits, Transfers, or Lookups. The streaming ledger consumes the different event streams and applies different transaction functions for each event type and writes to different result streams for each event type. Balance Lookup Events, for example, would not modify data, and simply read the balance from the indicated table and row and write it to a result stream.



Source-of-truth Position-keeping / Data Consolidation

The streaming ledger can be used to consolidate the actions and views of multiple independent actors into a common source-of-truth state. The actions are received as event streams, and the truth is maintained in streaming ledger tables and can optionally be mirrored out through a result stream.

The transactional logic implements the rules by which the events and the current state are consolidated. The serializable consistency semantics ensure that no effects are lost, and no decisions and changes are made inconsistently.



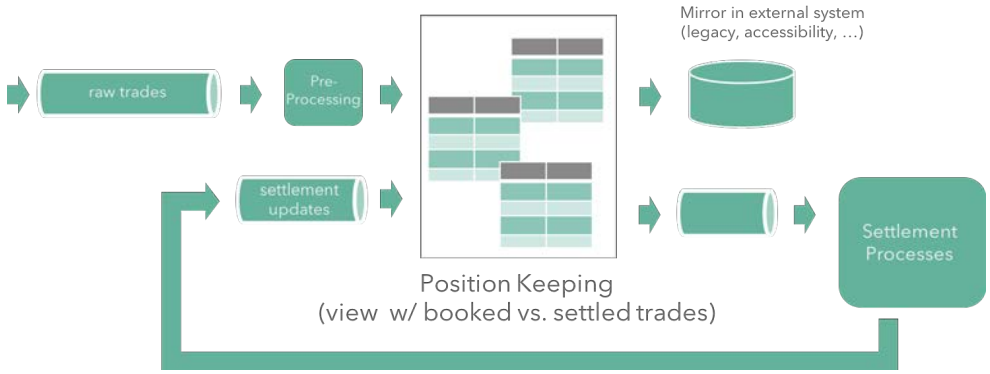
Example: Real time position keeping on streams of trades

Different applications (like trading desks, clients, etc.) have a fast-local view on a subset of the position data and work based on that for performance, compliance, and simplicity. Trades and other events that affect the position are written to streams. Before processing the event streams with the streaming ledger, the events can be preprocessed, which could happen within the same Apache Flink streaming application.

The streaming ledger here maintains the source of truth position in the tables it manages. It takes the event streams and transactionally updates the tables according to the events. The application logic in the transaction function applies the rules to determine how to update the position — the logic can check preconditions (and for example reject/cancel trades based on that and decides how to update the tables according to the event and the current consistent view on the position). The ability to update multiple rows and tables consistently is key to arriving at a reliable ground truth state. The transactional logic has the full flexibility of Java/Scala application code and hence supports highly complex business logic, as well as the use of libraries.

The results of each update can be written to a result stream that is consumed by an external system, in order to "mirror out" the ground truth and make it available in another system (database) that does not need to support serializable transactions, support similarly flexible business logic, or handle a high volume of transactional updates.

Additionally, the streaming ledger application can also receive additional streams of events from even later stages of the trade processing (such as trade settlement) and maintain a consistent view across trades that were booked versus trades that settled or failed to settle.



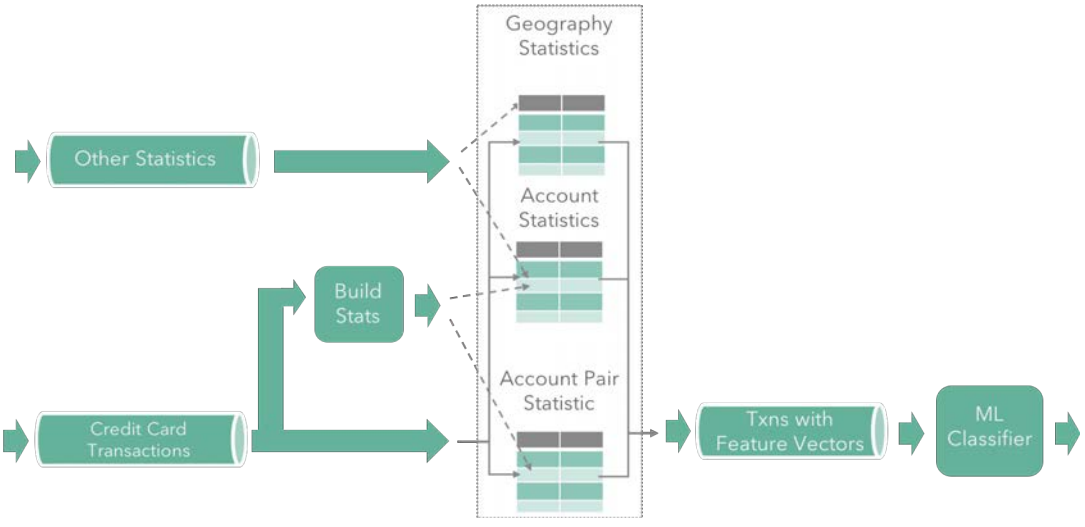
Multiway Joins / Feature Vector Assembly

data Artisans Streaming Ledger's support to share tables between streams and access to multiple rows across these tables makes it a perfect building block for multiway joins across shared tables.

Some streams can be used to build and update tables, other streams join together the rows of different tables to assemble joined results.

Example: Streaming Feature Vector Assembly for Machine Learning Applications

Machine Learning Models that classify events (such as labeling credit card transactions valid or fraudulent) typically need a set of "features" to be attached to the event, as input to the evaluation. In the case of credit card fraud detection, these features could, for example, be the frequency of use in recent days, the typical times of day when the card is used, the geographies where it is used, or the accounts that are paid to. The real time computation of such features, as well as the assembling individual features into a 'feature vector' is an excellent use case for the data Artisans Streaming Ledger.



In this example, the streaming ledger maintains three tables with different statistics about geographies, accounts, and pairs of accounts. The statistics are updated by the credit card transaction events themselves, as well as by another stream of updates produced by a different system. Each transaction event carries keys for relevant features to be collected and joins with all tables, building the feature vector that goes into the Machine Learning Model.

A Look at the Abstraction and APIs

Developing against data Artisans Streaming Ledger is simple. The APIs are designed to feel natural both to users that have used stream processing before, and to users that are familiar with databases.

We illustrate this with an example (the full code is in Appendix A and on GitHub). The use case is similar to the "Streaming Transactions across Tables" example discussed above:

- There are two tables: **Accounts** and **Book Entries**
- We have two streams of events: **Deposits** and **Transfers**
- Deposits put values into Accounts and the Book
- Transfers atomically move values between accounts and book entries, under a precondition

1. Create a Flink DataStream Program and Establish the Event Sources

The data Artisans Streaming Ledger API blends into the Apache Flink DataStream API. The first step is to have a DataStream API program with a StreamExecutionEnvironment and source event streams:

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
DataStream<DepositEvent> deposits = env.addSource(...);
DataStream<TransferEvent> transfers = env.addSource(...);
```

2. Define the Transactional Scope and Tables

The next step is to define the scope and tables. For simplicity, our tables are using strings as keys (account ID, entry ID), and longs for the current value/balance.

```
// start building the transactional streams
StreamLedger tradeLedger = StreamLedger.create("simple trade example");

// define the transactional states
StreamLedger.State<String, Long> accounts = tradeLedger.declareState("accounts")
        .withKeyType(String.class)
        .withValueType(Long.class);

StreamLedger.State<String, Long> books = tradeLedger.declareState("bookEntries")
        .withKeyType(String.class)
        .withValueType(Long.class);
```

3. Define the Transaction Functions and the Keys

Next, we specify which transaction function to apply to each stream, and for each table that will be used, how to select the keys for the rows that will be accessed.

The `'.apply(…)'` functions themselves contain the transactions' business logic (see below).

For each row being accessed, we add a call that specifies the access: `'.on(table, key, name, type)'`:

- `'table'` indicates the table that is accessed
- `'key'` is a function through which the key for that row is derived from the input event
- `'name'` is a logical name for that particular row (used later)
- `'type'` qualifies read-only, write-only, or read-write access to the row. This is an optimization, where `READ_WRITE` is the most generic option

```
// define transactors on states
tradeLedger.usingStream(deposits, "deposits")
       .apply(new DepositHandler())
       .on(accounts, DepositEvent::getAccountId, "account", READ_WRITE)
       .on(books, DepositEvent::getBookEntryId, "asset", READ_WRITE);
// define the transfes that update always four rows on different keys.
// We store a handle to the result stream for later use.
OutputTag<TransactionResult> transactionResults = tradeLedger.usingStream(transfers,
"transactions")
       .apply(new TxnHandler())
       .on(accounts, TransactionEvent::getSourceAccountId, "source-account", READ_WRITE)
       .on(accounts, TransactionEvent::getTargetAccountId, "target-account", READ_WRITE)
       .on(books, TransactionEvent::getSourceBookEntryId, "source-asset", READ_WRITE)
       .on(books, TransactionEvent::getTargetBookEntryId, "target-asset", READ_WRITE)
       .output();
```

4. Implement the Transaction Function

A transaction function contains the business logic that decides whether and how to update the table rows provided to it, and what to emit as a result.

These Transaction Functions are passed a state access object for each row that is being read or updated. To correlate the state accesses with the rows and keys, they are annotated with the name defined in the prior step.

For simplicity, we only show the implementation of the `'TransferFunction'`

```
private static final class TxnHandler extends
TransactionProcessFunction<TransactionEvent, TransactionResult> {
    private static final Supplier<Long> ZERO = () -> 0L;
     @ProcessTransaction
    public void process(
            final TransactionEvent txn,
            final Context<TransactionResult> ctx,
            final @State("source-account") StateAccess<Long> sourceAccount,
            final @State("target-account") StateAccess<Long> targetAccount,
            final @State("source-asset") StateAccess<Long> sourceAsset,
            final @State("target-asset") StateAccess<Long> targetAsset) {
```

```
        final long sourceAccountBalance = sourceAccount.readOr(ZERO);
        final long sourceAssetValue = sourceAsset.readOr(ZERO);
        final long targetAccountBalance = targetAccount.readOr(ZERO);
        final long targetAssetValue = targetAsset.readOr(ZERO);

        // check the preconditions
        if (sourceAccountBalance > txn.getMinAccountBalance()
               && sourceAccountBalance > txn.getAccountTransfer()
               && sourceAssetValue > txn.getBookEntryTransfer()) {

               // compute the new balances
               final long newSourceBalance = sourceAccountBalance - txn.getAccountTransfer();
               final long newTargetBalance = targetAccountBalance + txn.getAccountTransfer();
               final long newSourceAssets = sourceAssetValue - txn.getBookEntryTransfer();
               final long newTargetAssets = targetAssetValue + txn.getBookEntryTransfer();

               // write back the updated values
               sourceAccount.write(newSourceBalance);
               targetAccount.write(newTargetBalance);
               sourceAsset.write(newSourceAssets);
               targetAsset.write(newTargetAssets);

               // emit result event with updated balances and flag to mark transaction as
        processed
               ctx.emit(new TransactionResult(txn, true, newSourceBalance,
        newTargetBalance));
               }
                else {
               // emit result with unchanged balances and a flag to mark transaction as
        rejected
               ctx.emit(new TransactionResult(txn, false, sourceAccountBalance,
        targetAccountBalance));
                }
           }
        }
```

5. Optionally obtain the resulting streams

As an optional step, we can grab the stream of result events produced by the `'TransferFunction'`. We use the `'result'` tag which was created when we defined the transaction in step (3).

```
ResultStreams resultsStreams = txStreams.resultStreams();
DataStream<TransferResult> output= resultsStreams.getResultStream(result);
```

That's it!

You can find further information in the official docs …

Consistency Model

data Artisans Streaming Ledger executes transactions with ACID semantics under the isolation level 'serializable'. That is the strongest isolation level known in database management systems. The semantics are as if the transactions execute serially: each transaction executes individually, and the next transaction only starts once the previous one is complete, seeing all changes made by the previous transaction.

The challenge is to provide these semantics without actually executing the transactions one after the other, which would not be scalable. data Artisans Streaming Ledger achieves this through a combination of logical timestamping and conflict-free scheduling of the individual operations that a transaction comprises.

Durability

Stream processing is highly asynchronous by nature, and many stream processing systems perform durability asynchronously as well, for performance reasons. Apache Flink, for example, performs asynchronous checkpoints for persistence / recovery.

Because data Artisans Streaming Ledger is a library on Apache Flink, its state updates are properly durable once a checkpoint completes. Hence the durability semantics rely on the type of sink operator that is used with the streaming ledger application:

- Using an *exactly-once* sink, hard durability is guaranteed once a result event is read from a result stream, but the sink typically introduces additional latency.

- Using an *at-least-once* sink, results of transactions are always based on a consistent view, but a result may be subsumed by another result created during a retry (the duplicate in the at-least-once case). The new result will also be based on a consistent view, but may be different from the first result, because it followed a different serializable schedule.

Serializable, Linearizable, Strictly Serializable

The isolation level *"serializable"* is always guaranteed by the implementation of data Artisans Streaming Ledger. Under common conditions, users can even assume "*strictly serializable*" concurrency semantics.

Strictly serializable combines the properties of being "*serializable*", with the semantics of *"linearizable"* updates: Linearizable here means that if a transaction event B is put into the stream after transaction event A's result is received from the result stream, then transaction B's modifications on the data happen after transaction A's modification. Simply speaking, the order of transactions obeys the real-world order of triggering the actions.

Due to the asynchronous nature of stream processing, linearizability can only be relied on if one of the two following conditions is fulfilled:

(1) Transaction A's changes are durable before transaction B is put into the source stream. This can easily be achieved by using an exactly-once sink for the result streams.

(2) Transactions A and B will be replayed in the same order during recovery. This can be easily achieved with log-style data sources like Kinesis, or Apache Kafka. When relying on linearizable semantics between two transactions, the events that trigger them would need to be added to the same partition or shard, to maintain order upon replay during failure/recovery scenarios.

Performance Characteristics

To give you an impression of the data volumes that data Artisans Streaming Ledger can handle, this section discusses some performance considerations.

The implementation of the streaming ledger has not undergone any specific performance optimization work, yet. Despite that, the data Artisans Streaming Ledger is already able to push an impressive number of transactions.

All experiments below were made on a Google Cloud Setup with the following parameters:

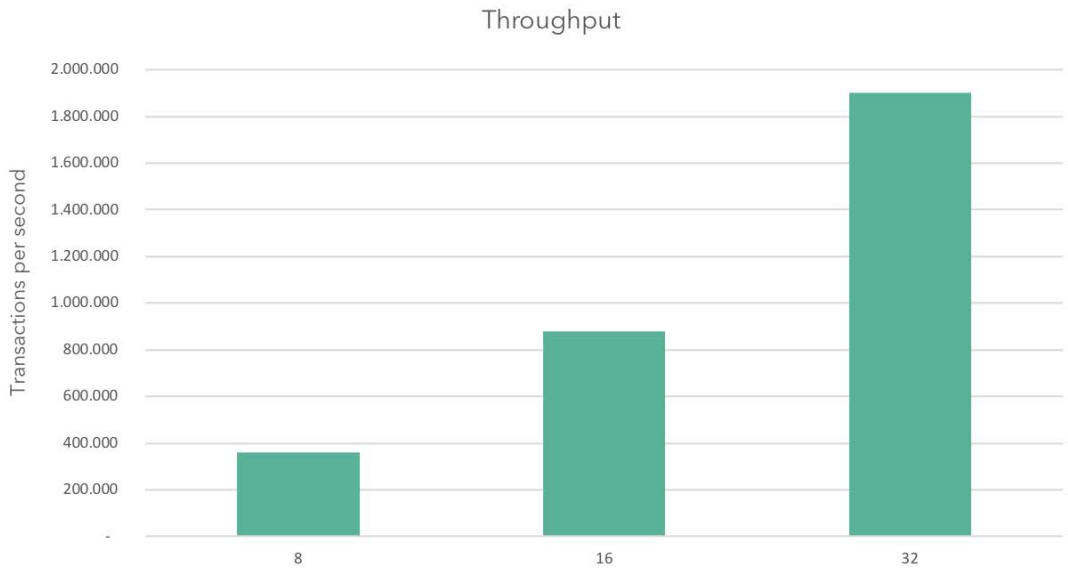
- 4 vCores per machine / TaskManager
- 10 GB Java VM heap size
- Apache Flink 1.6-SNAPSHOT
- State Backend: Java Heap Objects with snapshots to Google Cloud Storage

Use case and data for these experiments were:

- Transactions across 2 different tables (see first use case: Streaming Transactions across Tables)
- 100% update queries: each transaction reads and writes four rows (two per table)
- 100 million rows per table, (200 million rows total)

Scalability

The chart below shows the throughput with increasing parallelism. At a parallelism of 32, the system pushes nearly two million transactions per second. Please recall again that this is under a workload of 100% update



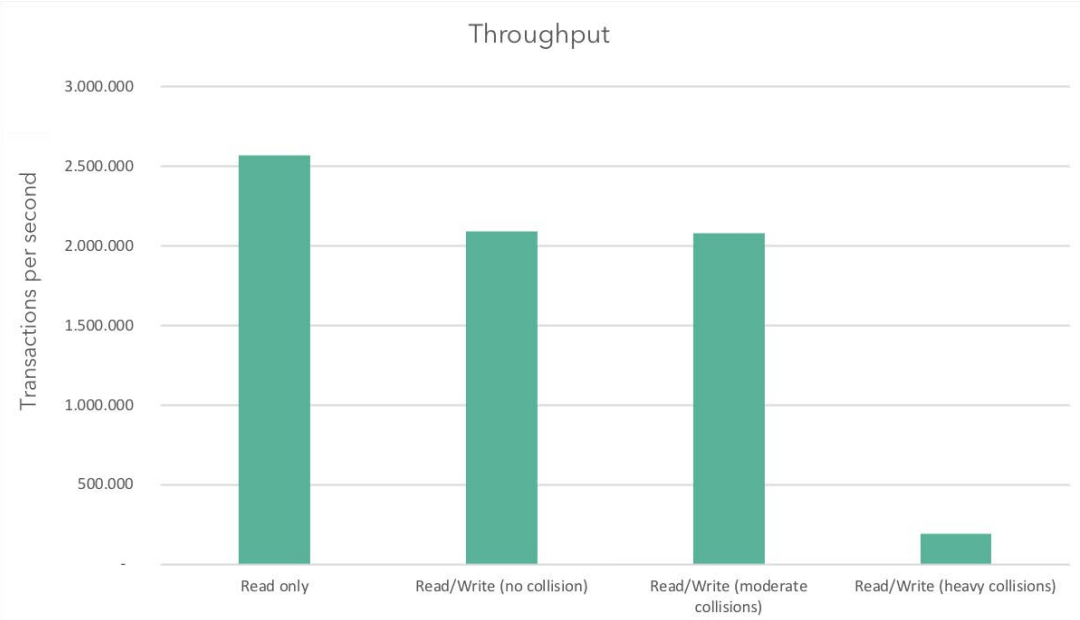
Contention

As explained in the previous section, processing transactions with Serializable Consistency means that the transactions follow the same consistency and semantics as if they were processed one after the other, across all tables and parallel processes. The implementation parallelizes what is possible to parallelize and maintains dependencies between the transactions where needed.

By definition, this means that the amount of parallelization depends on the overlap in the row accesses between concurrent transactions. When certain rows become contended, access to them may become a bottleneck.

The chart below shows the throughput of transactions under different contention. All experiments were executed with a parallelism of 32:

- Read only: The transactions read the four rows, but does not modify them.
- Read/Write (no collision): Transactions are generated such that they access different sets of rows within the tables.
- Read/Write (moderate collisions): Transactions modify overlapping sets of rows across the tables.
- Read/Write (heavy collisions): All transactions work on the same subset of 1000 rows within the tables and hence overlap heavily in the rows they access and modify.



Note that even in the synthetic degenerate case of very heavy collisions, the system makes still progress and does not go into deadlocks or retry-loops as other techniques for serializable transactions are often prone to do.

data Artisans Streaming Ledger versus Relational Database Management Systems

While bearing a lot of similarities to the ACID transactions of relational database management systems, data Artisans Streaming Ledger's serializable streaming transactions have some notable differences:

1. Transactions are not triggered by a client in a request/response manner. Instead, transaction events are put into one of the transaction event streams that the streaming ledger consumes. This follows the asynchronous/reactive processing paradigm behind data stream processing and allows data Artisans Streaming Ledger to achieve high throughput and to blend naturally into stream processing pipelines.

It is possible to build a client request/response abstraction on top of this, with a request and response message queue, and "correlation IDs" on the transaction events and responses.

2. Transactions are not specified in SQL, but as program code. This gives the streaming ledger a different abstraction compared to relational databases, but makes it possible to run sophisticated business logic asynchronously/reactively on the events and data in the tables.

3. Transaction types need to be defined up front. This is similar to some database systems that require transactions to be defined up front as "Stored Procedures".

4. The keys to the rows that a transaction accesses and updates must be derivable from the transaction event. The keys cannot be dynamically computed within the transaction's business logic. Concretely, that means that the business logic cannot look at some rows/values and make it dependent on those values which other rows to access and update. To achieve that sort of dynamicity, a transaction function would need to specify the set of all rows it might want to read/update and then leave some unchanged.

This restriction is essential to the underlying asynchronous scheduling / execution model, and makes is possible to achieve high consistency under high throughput. Future versions might relax this.

How does data Artisans Streaming Ledger achieve its performance?

The technology underlying the data Artisans Streaming Ledger is different from the transaction processing techniques used by traditional database management systems. The implementation uses neither distributed locks nor multi-version concurrency control with conflict detection. Distributed locks are very expensive and lock-based implementations need extra guards against deadlocks. Optimistic concurrency control with timestamps and conflict detection has the problem that transactions may need to be aborted and retried. In certain situations, retries can eat up the majority of the time resources, causing such a system to make very little progress through the load of transactions.

data Artisans Streaming Ledger is built to achieve high pipelined throughput, building on the strengths of the stream processing paradigm, and Apache Flink in particular:
(1) Transactions are pushed as functions into the system and triggered by events inside the stream processor when ready. This approach is fully asynchronous, and event driven and stands in contrast to having many threads and connections that spend most of their time waiting, as one typically has in request/response interaction sessions with database systems.

(2) Table row accesses as well as modifications are not managed by distributed locks, but by scheduling the accesses in a conflict-free way.

(3) The implementation uses logical clocks that are synchronized in a lightweight manner. Correctness is never affected by delays or clock drift, only the latency in row access scheduling is affected by that. Flink's watermarks are used to establish minimal guaranteed time progress when building the access schedule.

(4) The flexibility of Flink's checkpoints and state management is used to implement the out-of-order processing logic that guarantees serializability while maintaining persistence and exactly-once semantics.

The feasibility of this implementation is a testament to capabilities Apache Flink as a stream processor.

Appendix A: Complete Code Example

```
/*
 *  Copyright 2018 Data Artisans GmbH
 *
 *  Licensed under the Apache License, Version 2.0 (the "License");
 *  you may not use this file except in compliance with the License.
 *  You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing, software
 *  distributed under the License is distributed on an "AS IS" BASIS,
 *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
 */
package com.dataartisans.streamledger.examples.simpletrade;


import org.apache.flink.runtime.state.StateBackend;
import org.apache.flink.runtime.state.filesystem.FsStateBackend;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.util.OutputTag;
import com.dataartisans.streamledger.examples.simpletrade.generator.DepositsGenerator;
import com.dataartisans.streamledger.examples.simpletrade.generator.TransactionsGenerator;
import com.dataartisans.streamledger.sdk.api.StateAccess;
import com.dataartisans.streamledger.sdk.api.StreamLedger;
import com.dataartisans.streamledger.sdk.api.StreamLedger.ResultStreams;
import com.dataartisans.streamledger.sdk.api.TransactionProcessFunction;
import java.net.URI;
import java.nio.file.Paths;
import java.util.function.Supplier;
import static com.dataartisans.streamledger.sdk.api.AccessType.READ_WRITE;

    /**
     * A simple example illustrating the use of stream ledger.
     *
     * <p>The example here uses two states (called "accounts" and "bookEntries") and
    modifies two keys in each state in one
     * joint transaction.
     */
    public class SimpleTradeExample {
        private static final Supplier<Long> ZERO = () -> 0L;

     /**
         * The main entry point to the sample application. This runs the program with a
         * built-in data generator and the non-parallel local runtime implementation for
         * the transaction logic.
         *
         * @param args The command line arguments.
         */
        public static void main(String[] args) throws Exception {

               // set up the execution environment and the configuration
               StreamExecutionEnvironment env
               =StreamExecutionEnvironment.getExecutionEnvironment();
               // configure Flink
               env.setParallelism(4);
               env.getConfig().enableObjectReuse();

               // enable checkpoints once a minute
               env.enableCheckpointing(60_000);
               URI uri = Paths.get("./checkpoints").toAbsolutePath().normalize().toUri();
               StateBackend backend = new FsStateBackend(uri, true);
               env.setStateBackend(backend);
```

```
        // start building the transactional streams
        StreamLedger tradeLedger = StreamLedger.create("simple trade example");
        // define the transactional states
        StreamLedger.State<String, Long> accounts =
        tradeLedger.declareState("accounts")
                .withKeyType(String.class)
                .withValueType(Long.class);
        StreamLedger.State<String, Long> books =
        tradeLedger.declareState("bookEntries")
                .withKeyType(String.class)
                .withValueType(Long.class);
        // produce the deposits transaction stream
        DataStream<DepositEvent> deposits = env.addSource(new DepositsGenerator(1));
        // define transactors on states
        tradeLedger.usingStream(deposits, "deposits")
                .apply(new DepositHandler())
                .on(accounts, DepositEvent::getAccountId, "account", READ_WRITE)
                .on(books, DepositEvent::getBookEntryId, "asset", READ_WRITE);
        // produce transactions stream
        DataStream<TransactionEvent> transfers = env.addSource(new
        TransactionsGenerator(1));
        OutputTag<TransactionResult> transactionResults =
        tradeLedger.usingStream(transfers, "transactions")
                .apply(new TxnHandler())
                .on(accounts, TransactionEvent::getSourceAccountId, "source-
                account", READ_WRITE)
                .on(accounts, TransactionEvent::getTargetAccountId, "target-account",
                READ_WRITE)
                .on(books, TransactionEvent::getSourceBookEntryId, "source-asset",
                READ_WRITE)
                .on(books, TransactionEvent::getTargetBookEntryId, "target-asset",
                READ_WRITE)
                .output();

        //  compute the resulting streams.
        ResultStreams resultsStreams = tradeLedger.resultStreams();

        // output to the console
        resultsStreams.getResultStream(transactionResults).print();

        // trigger program execution
        env.execute();
    }


    /**
    * The implementation of the logic that executes a deposit.
    */
    private static final class DepositHandler extends
    TransactionProcessFunction<DepositEvent, Void> {

            private static final long serialVersionUID = 1;

            @ProcessTransaction
            public void process(
                    final DepositEvent event,
                    final Context<Void> ctx,
                    final @State("account") StateAccess<Long> account,
                    final @State("asset") StateAccess<Long> asset) {

            long newAccountValue = account.readOr(ZERO) + event.getAccountTransfer();

            account.write(newAccountValue);

            long newAssetValue = asset.readOr(ZERO) + event.getBookEntryTransfer();
            asset.write(newAssetValue);
     }
}
```

```
/**
    * The implementation of the logic that executes the transaction. The logic is
    given the original
      * TransactionEvent plus all states involved in the transaction.
      */
    private static final class TxnHandler extends
    TransactionProcessFunction<TransactionEvent, TransactionResult> {
        private static final long serialVersionUID = 1;
        @ProcessTransaction
        public void process(
                final TransactionEvent txn,
                final Context<TransactionResult> ctx,
                final @State("source-account") StateAccess<Long> sourceAccount,
                final @State("target-account") StateAccess<Long> targetAccount,
                final @State("source-asset") StateAccess<Long> sourceAsset,
                final @State("target-asset") StateAccess<Long> targetAsset) {

            final long sourceAccountBalance = sourceAccount.readOr(ZERO);
            final long sourceAssetValue = sourceAsset.readOr(ZERO);
            final long targetAccountBalance = targetAccount.readOr(ZERO);
            final long targetAssetValue = targetAsset.readOr(ZERO);

           // check the preconditions
           if (sourceAccountBalance > txn.getMinAccountBalance()
                    && sourceAccountBalance > txn.getAccountTransfer()
                    && sourceAssetValue > txn.getBookEntryTransfer()) {

           // compute the new balances
                   final long newSourceBalance = sourceAccountBalance -
                   txn.getAccountTransfer();
                   final long newTargetBalance = targetAccountBalance +
                   txn.getAccountTransfer();
                   final long newSourceAssets = sourceAssetValue -
                   txn.getBookEntryTransfer();
                   final long newTargetAssets = targetAssetValue +
                   txn.getBookEntryTransfer();

           // write back the updated values
                sourceAccount.write(newSourceBalance);
                targetAccount.write(newTargetBalance);
                sourceAsset.write(newSourceAssets);
                targetAsset.write(newTargetAssets);

           // emit result event with updated balances and flag to mark
           transaction as processed
           ctx.emit(new TransactionResult(txn, true, newSourceBalance,
           newTargetBalance));
            }
            else {
           // emit result with unchanged balances and a flag to mark transaction
           as rejected
           ctx.emit(new TransactionResult(txn, false, sourceAccountBalance,
           targetAccountBalance));
            }
        }
    }
}
```

Disclaimer

APACHE FLINK, FLINK AND APACHE ARE EITHER REGISTERED TRADEMARKS OR TRADEMARKS OF THE APACHE SOFTWARE FOUNDATION IN THE UNITED STATES AND/OR OTHER COUNTRIES, AND ARE USED WITH PERMISSION. THE APACHE SOFTWARE FOUNDATION HAS NO AFFILIATION WITH AND DOES NOT ENDORSE, OR REVIEW THE MATERIALS PROVIDED WITH THIS DOCUMENT.