WHITE PAPER

MODERN STREAM PROCESSING USING STREAMING SQL

Democratizing Real-Time Data Access Across the Enterprise

STREAMING SQL

Table of Contents

Overview

During the technology boom of the last two decades, a trend has come into focus: data is the lifeblood of a modern company and nearly every company is, essentially, a data company. Organizations have realized that investing in data systems and infrastructure can make them more competitive in their marketplaces and allow for new and exciting innovations.

In the last few years, this paradigm has exploded as it has become clear that streaming, or real-time, data inherently has maximum value. Organizations can add streaming data systems to their arsenal and massively boost their ability to provide highly differentiated, competitive, and compelling user experiences.

Overlapping this trend is the popularity to generate data using machines and cloud computing. The Internet of Things (IoT) is an obvious example, but lesser known is the trend to instrument everything in your business—every web click, visitor session, page view, purchase, interaction, chat message, all of it. In 1995, companies stored who the customer was and what they purchased. In 2021, companies store every single interaction they have with the business—from the manufacturing of the product to data being generated as the product is used. In fact, companies are being created because of streaming data, and without it, they wouldn't exist.

But, the velocity and volume of streaming data is massive, which has dictated that new architectures in data systems be invented. Ingesting the firehose of data, distilling down the useful parts, and routing it to this new breed of applications requires new and specialized designs. Because these streaming systems are not siloed, they are typically used alongside and in conjunction with more traditional technologies like relational database systems (PostgreSQL, MySQL, Oracle), data lakes/data warehouse (Hive, Impala), or even NoSQL systems (MongoDB, Elastic).

Further, the modern enterprise is on a collision course between the desire to capture data at an increasing rate and the ability to process that data. Gaining access to streaming data for immediate processing requires special skills in Java, Scala or similar languages. Plain old Structured Query Language (SQL) can be used to query such data once it has landed into a SQL-compatible store. Given the popularity of SQL skills across the industry over other specialized skills, it would be ideal to query real-time data with just SQL while it was still in the streaming system. And thus, Streaming SQL was born. Streaming SQL solves this problem by continuously running the SQL processes on the boundless stream of business data.

Streaming Architectures Today

A modern stream processing architecture consists of a myriad of components—including systems that provide robust ingestion, schema definition and lineage, security and governance, and more. In general, the three key parts of such an architecture are:

- Data ingestion that acquires the data from different streaming sources and orchestrates and augments the data from other enterprise sources
- A messaging system that will guarantee delivery and track consumption of messages by various consumers
- A stream processing system that will allow for creating computations using these messages

While the first tenet (data ingestion) is fairly easily done with powerful engines like Apache NiFi, the real challenge lies in how that data is consumed in the enterprise in real-time. For this purpose, we will focus on the other two tenets in this paper.

The Messaging System

Append-only distributed log data systems or messaging systems like Apache Kafka and Apache Pulsar have provided relatively simple, scalable, and high-performance architectures for managing the input/output and storage for streams of data. Architectures differ, but a common design trait is that these systems allow for the persistence of data at a very high volume and concurrency, but they give up things like transactional semantics. Typically, they allow for various durability guarantees and allow capabilities from "at least once" to "exactly once" processing. They generally do this via an append only, message based paradigm.

In the case of Kafka, it provides a massively scalable and redundant architecture coupled with straightforward APIs. Data is organized by a namespace called a topic and supports highly concurrent writes and reads. High performance and scalability are provided using a partitioning scheme. Programs can write (produce) and read (consume) data via language-specific drivers. The data can be in various formats with Apache AVRO and JSON being two common ones.

Messaging systems give us part of a solution for most organizations: APIs to write and read data in an insanely fast, yet durable, manner.

Now, let us look at how this data is processed by stream processing engines.

The Stream Processing System

"SQL is inherently suitable for stream processing as the semantics and grammar of the language are naturally designed to ask for data in a declarative way."

The stream processing paradigm was invented to perform continuous parallel computations on streams of data. Stream processors are programs that interact with the streams of data—performing a computation or mutating the data in the flow of data. You don't have to use stream processors in combination with a messaging system, but when you do, they are massively powerful and unlock amazing data processing potential.

Stream processing frameworks present an API for writing processors that run in parallel to perform computations and to aggregate and package data in a usable format for a business or application to consume. Oftentimes these processors are called jobs. Stream processing frameworks like Apache Flink, Samza, and Storm do things like manage state, handle interprocess communications, provide high availability/restart-ability, and scalability of groups of jobs. Jobs can also be created as independent processors using APIs like Kstreams where Kafka itself is used for many of these functions.

Stream processing jobs are typically written using the specific API of the processing framework itself, and Java and Scala are prevalent languages for this. The APIs tend to be fantastically powerful and rich. For instance, in the case of Apache Flink, there are multiple APIs with various degrees of functionality and complexity. They range from very low-level operations (datastream API), up to a higher level (SQL API).

Stream processors tend to process data from an input (source) to an output (sink). Typical sources are Apache Kafka or AWS Kinesis. Typical sinks can be anything from Kafka to traditional relational database systems (RDBMS) like PostgreSQL, or distributed stores like Apache Druid or Hive, and even distributed file systems like Amazon S3. Jobs are chained in process (parsed to a DAG or Directed Acyclic Graph) or extra-process by using a sink as a source for an entirely different job. Creating chains of these processors is called a data pipeline.

Streaming SQL

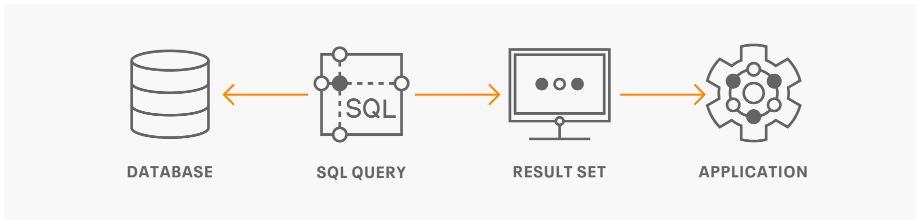
Streaming SQL—sometimes called Continuous SQL, StreamSQL, Continuous Query, Real-Time SQL, or even "Push SQL"—is the usage of SQL for stream processing. SQL is inherently suitable for stream processing as the semantics and grammar of the language are naturally designed to ask for data in a declarative way. Moreover, relational SQL has the characteristic of using a set of tuples with types (called relations or tables) to express the schema of the data. These relations fundamentally differ from traditional RDBMS relations; because, as streams, they must have a time element. Because of SQL's rich history, it is widely known and easy to write. Developers, data engineers, data scientists, and others do not need to use complicated, low-level APIs to create processors. They can create them in SQL. More importantly, they can issue SQL against the stream of data and iterate building up statements in development mode. This allows them to explore and reason about the data stream itself using a familiar paradigm.

Streaming SQL should be familiar to anyone who has used SQL with a RDBMS, but it does have some important differences.

In an RDBMS, SQL is interpreted and validated, an execution plan is created, a cursor is spawned, results are gathered into that cursor, and then iterated over for a point in time picture of the data. This picture is a result set, it has a start and an end.

The phasing is described as parse, execute, and fetch:

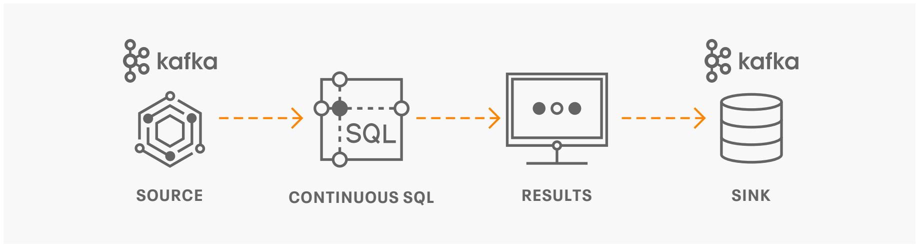
- **Parse**—Validate the SQL statement, create an execution plan, give feedback to the user
- **Execute**—Run the SQL statement using the execution plan
- **Fetch**—Open a cursor and return the data to the user, closing the cursor when the data is done being returned



"Data being emitted via Streaming SQL are continuous results—there is a beginning, but no end."

In contrast, Streaming SQL queries continuously process results to a sink of some type. The SQL statement is interpreted and validated against a schema (the set of tuples). The statement is then executed and the results matching the criteria are continuously returned. Jobs defined in SQL look a lot like regular stream processing jobs—the difference being they were created using SQL vs something like Java, Scala or python. Data being emitted via Streaming SQL are continuous results—there is a beginning, but no end. A boundless stream of tuples.

- **Parse**—Validate the SQL statement, give feedback to the user
- **Execute**—Run the SQL statement
- **Continuously Process**—Push the results of the query to a sink



Streaming SQL looks a lot like standard SQL:

```
-- detect fraudulent auths
SELECT
COUNT(*) AS auth_count,
MAX(amount) AS max_amount,
TUMBLE_END(eventTimestamp, interval '1' second) AS ts_end
FROM paymentauths
WHERE amount > 10
GROUP BY card, TUMBLE(eventTimestamp, interval '1' second)
HAVING COUNT(*) > 2;
```

Time

One critical aspect of streaming SQL that differentiates itself from traditional SQL is the aspect of time. Stream processing systems must support grammar that allows declaring the time boundary or window that the data should be returned over. For instance, it's common to aggregate data over a time window that tumbles second to second.

"Stream processing systems must support grammar that allows declaring the time boundary or window that the data should be returned over. For instance, it's common to aggregate data over a time window that tumbles second to second."

```
-- group by key, and tumbling window interval
GROUP BY card, TUMBLE(eventTimestamp, interval '1' second)
```

Time can be expressed as "ingestion time", "event time" or "processing time". In many stream processing systems like Flink, the distributed log system will automatically capture ingestion time, and it's up to the user to create a field with event time. Processing time can be generated by the processing system logic itself. Performing calculations over time periods requires state management in the underlying processing system. Time is also important in identifying and processing late arriving data—these systems are distributed event processing systems. No serious streaming architecture could ignore highly asynchronous and late arriving data.

Schema

Messaging systems like Kafka don't inherently enforce a schema on the data flowing through it. They ingest and store messages over time, and data can be in any format. Clearly, without some known schema, data would be a mess and near impossible to query or filter. Thus, there must be some schema assigned to the data. Common formats are JSON and AVRO. In the case of AVRO, schemas are defined and versioned using a separate storage system. With Apache Kafka, drivers are schema-aware and can enforce compliance at produce-time validating against a central repository (like Schema Registry). In the case of JSON, a schema must be defined in order to query by values nested in the data structure, as well as assign types. In order to run Streaming SQL, a schema must exist. This schema is the tuple and types that are part of the query (the columns and data types). This schema also provides the definition the parser will validate the statement against for validity (naming, operators, types, etc).

```
-- example schema for paymentauths
card                  varchar
amount                integer
eventTimestamp        timestamp
```

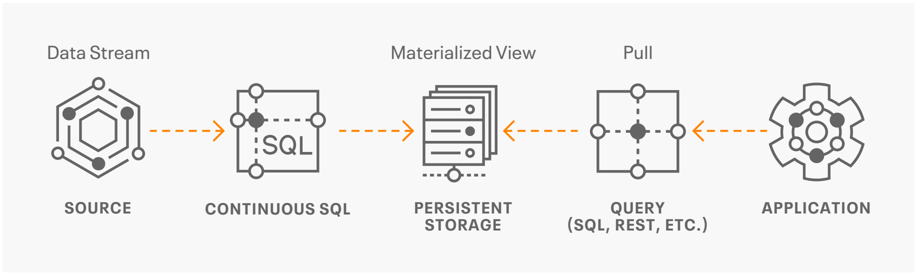
Running SQL against boundless streams of data requires a bit of a mindset change. While much of the SQL grammar remains the same, how the queries work, the results that are shown, and the overall processing paradigm is different than traditional RDBMS systems. Filters, subqueries, joins, functions, literals and all the myriad of useful language features generally exist but may have different behaviors. New constructs around time, specifically windowing, are introduced.

Continuous results

Another key difference between Streaming SQL and traditional SQL (and stream processing in general) is how the results of the query are handled. In traditional SQL, a result set is returned to the calling application. Using Streaming SQL, the results are continuously returned to a sink. Sinks can be streams of data like Kafka topics, or sinks can be more traditional systems like a RDBMS. More than one stream can utilize the same sink, sometimes joined with other streams.

But careful consideration must be given to how the results of a stream of data are ultimately persisted. An impedance mismatch exists between streams and traditional storage systems like databases. Streams are a boundless set of tuples, and databases (generally) store the latest state of a tuple. This impedance mismatch must be handled in some manner. Systems like Apache Flink provide a number of handlers for sinks that describe the behavior of the transition. Typical constructs are:

- **Append**—Every message is a new insert
- **Upsert**—Update by key, insert if key missing (idempotent)
- **Retract**—Upsert with delete if key no longer present in window



Which option to choose is highly dependent on the type of the sink, its native capabilities, and the use case. For instance, a RDBMS could work with a retract stream, but a time-series DB would only support append only.

Materialized views

"Materialized views are continuously updated and always represent the latest state by a given key—except they are declared via streaming SQL."

Recently, the ability to create materialized views on streams (streaming materialized views) has been an important development in stream processing systems. Materialized views blur the line between streaming systems and traditional databases. Similar to how traditional databases have worked for decades, materialized views are continuously updated and always represent the latest state by a given key—except they are declared via streaming SQL. When you create a view you define the creation query in SQL, define a primary key, and some basic rules about cleanup, null handling, etc. The resulting view is a picture of events at a given point in time—something typically missing or resource expensive to create. This capability is important to modern developers who need to query (typically called a pull-query) this view for their applications. Depending on the particular platform, materialized views may not require a database at al—something that drastically reduces complexity, costs, and latency.

Materializing data requires a persistent storage mechanism. Some systems use simple key/value stores, others more complex strategies. This choice is critical because it defines the capabilities the developer can expect from the view engine. In either case—materialized views should be thought of as more of a data cache than a full fledged database. If a developer needs full historical data or the entire event flow—then a traditional database might be the right choice. However, if they need to aggregate down a massive data flow to present a low latency data set for an application—that is bread and butter for streaming materialized views.

For example, perhaps the developer is building a javascript application that is plotting aircraft locations on a map using streamed ADSB data. The source data comes in many times a second in JSON format:

```
{"icao": "AB2404", "altitude": "37025"},
{"icao": "AB2404", "lat": 37.529572, "lon": -122.265806},
{"icao": "AB2404", "altitude": "37095"},
{"icao": "AB2404", "altitude": "37125"},
{"icao": "AB2404", "lat": 37.530032, "lon": -122.265130},
...
```

You will notice each message is incomplete—only a portion of the full schema is presented each message. We need to aggregate the data by aircraft ID, and use the latest version of location pairs and altitude by timestamp. Something like this:

```
-- aggregate over window
CREATE MATERIALIZED VIEW aircraft_locations
AS
SELECT icao, -- unique key
FIRST_VALUE(lat) OVER w AS lat,
FIRST_VALUE(lon) OVER w AS lon,
FIRST_VALUE(altitude) OVER w AS lon,
eventTimestamp as TS -- use embedded kafka timestamp
FROM airplanes
WINDOW w AS (
   PARTITION BY icao -- unique key
   ORDER BY eventTimestamp
   ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
);
```

This statement materializes the data into a view that can be queried (sometimes referred to as a "pull query") using traditional and widely available tooling, communications protocols, and APIs. This can be pulled into a notebook/pandas for building a ML model, or maybe used to plot a map using javascript.

Wrapping it up

Adding streaming data systems to an enterprise architecture can massively boost the ability to provide highly differentiated, competitive, and compelling user experiences. Messaging systems and stream processors are the building blocks that allow streaming SQL to open up new opportunities—in a simple yet powerful way. Lastly, materializing data into views for tooling and applications provides highly usable datasets for a variety of streaming data powered applications making end-to-end applications much more simplistic to create. The net effect is a processing architecture that can handle massive amounts of data—yet is easily democratized within the organization—maximizing value and delighting customers.