Unifying Metadata for Stream and Batch Queries in a Cloud Service

Florian Fröse
Adobe Research Schweiz AG
Basel, Switzerland

Daniel Bauer
IBM Research Europe
Rüschlikon, Switzerland

Daniel Pittner
IBM Deutschland Research und Development GmbH
Böblingen, Germany

Sean Rooney
IBM Research Europe
Rüschlikon, Switzerland

ABSTRACT

SQL interfaces have been independently built both for big data stream and batch processing systems. However, semantically there is an impedance between these two worlds. Streaming systems favor evolution and change while batch systems have historically been designed for *write once read many*. It is common for schemas to evolve in a streaming system overtime meaning that they have developed different approaches for storing metadata. We describe how we resolved one aspect of this semantic gap by storing and updating metadata in a single system and enabling the use of this unified metadata structure to run streaming and batch queries within the same commercial cloud SQL service. The *transparent* unification of batch and stream metadata catalogs is, as far as we are aware, novel.

1 INTRODUCTION

The scope of this work is a commercial cloud-based data processing system designed for data analytics on structured data. Typical use-cases include long-running queries to compute aggregated or statistical values across groups of records. These queries operate on data sets that are divided into records with a common schema. The service is a "BigData system" where data is stored on storage systems that are separate from the processing systems. This decoupling enables the processing of data sets from various sources stored in different formats.

For static data, a common pattern is a batch query where data sets are read and processed into a single result data set that is stored back onto the storage system. A classic example is a transfer job that reads data from operational data sources ("system-of-records"), denormalizes the data, and stores the result in a data analytic systems ("system-of-insights").

Often, data is available as a continuous stream of records. For example, by continuously streaming records from operational data sources into analytic systems, insights on the data becomes available much more quickly than when having to wait for the next periodic batch query. In other cases, data is produced as a stream at the source, for example by sensors that periodical transmit a stream of measurements. In this context, stream processing is therefore often also called real-time processing.

SQL is firmly established as a generic data query language for structured data. This paper describes an approach that supports both batch queries as well as stream queries on top of an SQL-based data processing system. Our contribution covers the control- and management layer of an existing cloud-based data processing system that was initially designed for batch queries. Our work seamlessly integrates stream data sources into the system such that the same SQL interface can be used to formulate stream queries as well as batch queries. In addition, data from streams and batch sources can be joined together in the same query.

A stream is a continuous sequence of records. Stream queries differ from batch queries in that:

- A stream query nominally runs for an indefinite amount of time and continuously produces a stream of results.
- The processing system must allow for schema evolution if the structure of the records in a stream changes while the query is running.
- Aggregating, grouping and joining of data have a different semantic in streams queries than in batch queries.

The following sections describe how we integrate stream queries into an existing batch query processing system. We also address the issue of schema changes and the effect on running queries. The final section outlines future extensions on how to address the semantic differences for aggregation, grouping and join functions in stream and batch queries, respectively.

2 RELATED WORK

Initially, Hadoop proposed Map/Reduce as the preferred way of processing big data, but because of the ubiquity of SQL the community developed the Apache Hive system for executing SQL on Hadoop [15] via the dynamic creation of Map/Reduce jobs. The Hive processing engine was made redundant by other SQL/Hadoop systems such as Impala [10], BigSQL [7], and Presto [8]. However, the component of Hive for storing metadata — the Hive Meta Store (HMS) — is still used by all of those systems. HMS stores the technical metadata, e.g. the table schemas, as well as other metadata such as the format of the data, the location on the storage system, the partitioning strategy etc. The SQL engine obtains all necessary metadata from the HMS and then generates and executes the query plan with its own specific technology.

Apache Kafka [12] has become one of the basic substrates for stream processing, playing a role analogous to the Hadoop file system in batch processing. Apache Kafka uses a publish/subscribe semantic for distributing messages but a log capability for storing them. Plain Kafka supports at-least-once delivery semantics. Messages in logs may be retained for a specific amount of time or until they are replaced by a more recent message with the same key.

The lambda architecture [13] is an attempt to unify the two worlds of stream and batch processing where the same data is processed by a batch processor and a stream processor. The batch process computes highly accurate results with some delay while the stream processor is producing time-critical output. A serving layer combines the two outputs again.

The kappa architecture [11] simplifies lambda by moving all processing on top of streams. *Kafka Streams* [2] follows the kappa design. It is a library and API for stream processing. Applications define a topology of stream processors that operate on data streams. Source processors continuously read input data streams from Kafka topics and sink processors write the end result into Kafka topics. Kafka Streams standardizes a specific representation of a relational table called a KTable. Each message corresponds to an operation on a relational table row. KTables assume that the table has a primary key, i.e. that every row can be uniquely identified.

ksqlDB [1] builds on Kafka Streams to offer an SQL interface for interactive querying of data stored in topics. *ksqlDB* extends SQL with support for *windows* — bounded segments of the stream that change as the stream advances. Windows are often time based, e.g. the data received during the last hour. *ksqlDB* uses windows both for aggregation and join operations across multiple streams and KTables. Data from batch sources such as relational database tables first need to be converted into a stream before *ksqlDB* can process them.

Our work differs from *ksqlDB* in that we do not require the conversion of batch sources into streams to allow a static data set and a stream to be combined within a query.

3 APPROACH OVERVIEW

The SQL Service that we extended is IBM's Cloud Data Engine [4]. This is offered by IBM as a support for cloud based data lakes. The systems offers both functionality for ingestion and querying. The query system is built around Apache Spark. It allows ANSI SQL query and is stateless. It supports multiple different formats including CSV, JSON and Parquet. The charging model is pay-per-query. IBM Identity and Access Management and IBM Key Protect is used to control access to data.

The main components of the system are schematically shown in Figure 1. Queries are formulated in the SQL console and handed over to the service API and job handler. The service API parses the query and checks the syntax using the table's schema definition from the metastore. The job handler then generates code that it passes to the SQL execution engine, which then processes the input data sets and produces a result set.

In the batch-only solution, the table definition in the metastore describes static data residing in a storage system such as a distributed file system or an object store. We extend the description to tables backed by structured streams, made available from streaming platforms. We also extend the SQL syntax for stream queries to use the `EMIT` keyword to define where the resulting stream data

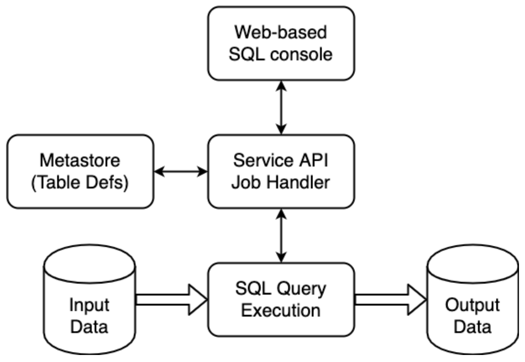

Figure 1: System Schematics

is stored. Finally, the job handler is extended to generate the corresponding stream processing code which is then executed by the processing engine.

For stream-backed tables, the metadata consists of connection information to the stream platform and a description of the stream's structure. The schema of stream records may change over time. The metastore must therefore keep track of schema evolution as change happens. Depending on the data serialization format used in the stream, schema information is either encoded in every record, e.g. when using `Json`, or externally in a schema registry for formats such as `Avro` or `Protobuf`. In our approach, we assume a schema registry and place a proxy between stream producers and the registry. This *registry proxy* observes the control flow from the stream producers to the schema registry and uses this information to create and update table definitions in the metastore.

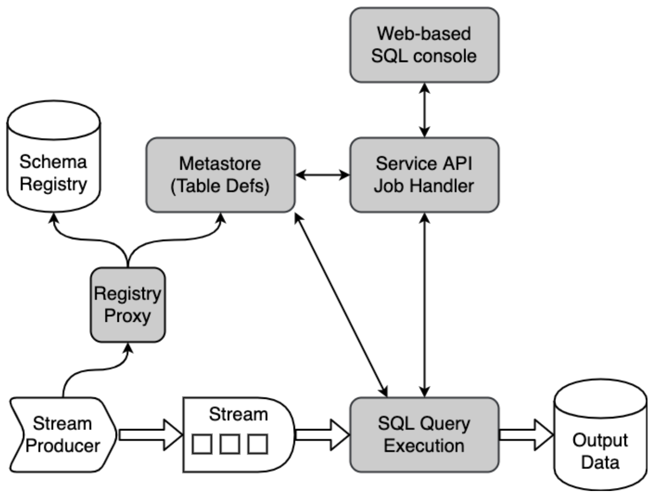

Figure 2: Stream Query

Figure 2 gives a high-level overview of the components that are involved in processing a stream query. The components in grey depict part of the core SQL service whereas the white components belong to different services or custom components. In particular, the schema registry that manages stream metadata is not part of the core SQL service. This leads to a design where the metastore is the only source of metadata in the SQL service and the registry proxy guarantees that metadata from all schema registries is made available in the metastore.

4 IMPLEMENTATION DESCRIPTION

Our work extends an existing cloud-based SQL-as-a-Service system. Metadata, i.e. table descriptions, is stored in Apache's Hive metastore. Apache Spark [5] is used as the processing engine for both batch- and stream-processing. Static data sets are stored mainly in cloud object stores and relational databases, whereas data streams are managed by Kafka [12].

4.1 Metastore and Catalog Management

The Hive metastore is used for catalog management. It stores:

- table definitions containing schema information;
- the storage location of each table, e.g. the S3 bucket or the topic name and Kafka cluster;
- the storage format, e.g. parquet;
- how the data is partitioned.

For static data sets, catalog management is not automatic. Entries in the Hive metastore are defined manually using Hive's data definition language. The schema information is extracted from the underlying data. The following definition creates a table `employees` in the metastore that is backed by a parquet file stored in a cloud object store.

```
CREATE TABLE employees USING parquet LOCATION
s3://us-south/sql-query/employees.parquet;
```

Stream-backed tables are created *automatically* by the registry proxy whenever the proxy intercepts a registration for a new stream. In order to make a stream visible to the SQL query service, the stream producer connects to the registry proxy instead of the original schema registry. The registry proxy requires a few additional configuration properties that are not part of the standard exchange between the stream producer and the schema registry. Since schema registries support custom fields, these properties are easily added to the configuration of the stream producer. They include:

- the URL of the original schema registry, required by the proxy;
- connection information for the Kafka cluster and Kafka topic name, part of the stream's metadata;
- optional table name if different from the topic name;
- encoding format of the stream, e.g. Avro;
- optional parameter that indicates the stream category and controls the behaviour during query execution, see Section 4.5.

Together with the schema information from the standard registry exchange, the proxy has all the information that it needs to create a table definition in the Hive metastore. The proxy guarantees consistency between the schema registry and the Hive metastore in a fully transparent way. Stream producers only require the aforementioned configuration settings.

4.2 Stream Jobs and Query Execution

The service API parses and analyzes the SQL query that is entered by a user on the web console or that arrives as a request from other services. It then generates a code fragment for Spark's Python API [6]. Stream queries are recognized by the presence of the `EMIT` keyword in the query and by stream specific property fields in the table's metadata. Multiple different categories of stream queries are supported, as described in the following sections. For each stream category, an implementation in Scala is made available as library code in the Spark execution engine. The service API accepts the query if it follows the pattern of a known category, otherwise the query is rejected. For accepted queries, PySpark code is generated that invokes the corresponding library code with the parameters extracted from the query and the metadata store. The query is then handed over to Spark for execution. The job handler observes the execution and reports any errors back to the console or other sources of the query. Stream queries execute as long as the stream exists and they aren't explicitly stopped, or until an unrecoverable error occurs, such as an incompatible schema change.

4.3 Ingesting Stream Data

The first stream category supported by our implementation are stream queries that use a *Time Series* model where each record in the stream is independently processed. A frequent use case is a stream of data that is ingested into a cloud storage system. These queries are sometimes also called *landing queries*. The motivation is to stream data from an operational source into an analytics system for further batch processing. The landing query is a form of an extract, transfer, and load process that filters, transforms, and stores the records into a cloud-based data lake. An example is a sensor or edge device that publishes Avro-encoded measurements into a Kafka topic. The stream query consumes these messages from Kafka, applies filters and transformations, and continuously stores the resulting stream into, for example, a cloud object store. A landing query for storing a stream of temperature readings into an object store looks as follows:

```
SELECT temp, sensor_id FROM temperatures
EMIT INTO s3://us-south/sql-query/temps.parquet;
```

The `temperatures` identifier refers to a Hive table that has been *automatically* created by the registry proxy when the temperature sensor started publishing into the Kafka topic. The Hive metadata contains the Avro schema definitions, and the Kafka connection information which are necessary for the SQL query engine to read the data stream.

4.4 Lookup Joins

The cloud SQL service offers a specific join between a stream and one or more tables, called *lookup* joins. More general joins are the topic of future work and discussed in Section 5. In lookup joins, records are augmented with information from one or more lookup tables. In the example query shown below, a stream of sensor readings is augmented with additional information about the sensors. Each record contains the id of the sensor that provided the readings. Additional information about the sensor such as a name, the geographic location etc. are looked up from a table that is available in an S3 bucket. The join then links the two sources of information as the stream arrives using the sensor identifier as the join key.

```
SELECT t.temp, t.created, s.name, s.geography
FROM temperatures AS t
JOIN s3://us-south/sql-query/sensors.parquet AS s
ON t.sensor_id = s.sensor_id
EMIT INTO s3://us-south/sql-query/temps.parquet;
```

4.5 Change-Log Streams

A Change Data Capture (CDC) [3] system such as Debezium [14] observes the transaction log of a relational database and creates a change-log stream. In a change-log stream, items are not simple values in themselves but descriptions of how the source data has changed. Also referred to as *KTable*, every stream message represents an insert, update or delete of a record. Records are uniquely identified by a key.

From an algorithmic point of view the difference between the time series category and change-log streams is that in the time series model each item in the stream can be considered in isolation and the result is written in append mode into the storage system. A change-log model is stateful and messages with the same key need to update the same record. Change-log streams can therefore only be landed into cloud data-lakes if the target storage system supports updates of previously written records, e.g. into a relational database or a lake-house system such as Iceberg.

Figure 3 shows the schematics of a CDC pipeline. The CDC system observes the transactions of an operational database and translates the modifications that are executed on the database tables into streams of change log events, one stream per table. Insert and update operations on a record generate so called *Upsert* events where a key identifies the record and a value part contains all of the record's data. For delete events, the key is sufficient to identify the record that has been deleted and the value part is empty. The query engine interprets these events and executes the appropriate insert, update, or delete operations on the mutable analytics store where the stream is landed.

When querying a stream, the query engine needs to know both the structure of the items and the type of streaming model being used. By default, a time-series model is assumed where items in the stream aren't consolidated. This can also make sense for streams containing change events if it is important to keep a record of previous changes. The category of a stream, and thus its interpretation, can be set as a property in the table description, either as a custom attribute in the stream producer or via an `ALTER TABLE` statement that modifies the content of the Hive metastore.

4.6 Schema Evolution

Queries on data streams are long-running operations where the structure of the data may change over time. For example, a column might be added to a table in the relational database from which changes are being streamed. The SQL query system keeps track of schema evolution and updates the table definition in the metastore accordingly. This enforces that new queries have all the schema

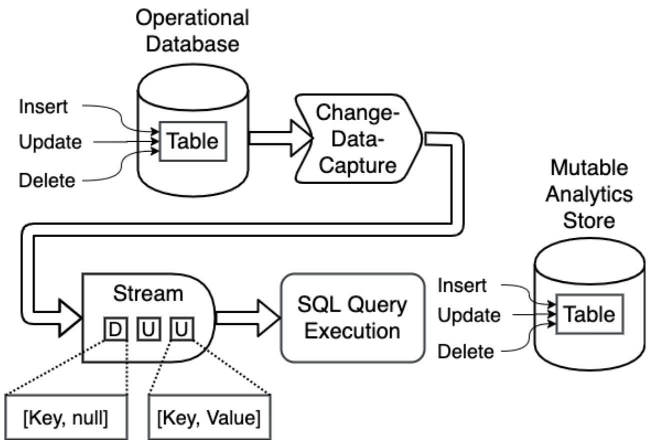

Figure 3: Change Log

information available that they need and, more importantly, that running queries continue to run on compatible schema changes. In Avro, it is a technical necessity that data from the stream can only be correctly interpreted when the *writer's schema*, i.e. the schema with which data is written, is available for de-serialization. The *reader's schema* is the structure used by the query.

When a stream is initially defined, the writer's and reader's schema are the same. They diverge if the structure of the source changes and the stream producer registers a new writer's schema in the schema registry. The registry proxy adds this new schema to the Hive metastore before the producer starts publishing data in the new format. As soon as the query execution engine encounters an event with an unknown schema, it refreshes its schema cache from the Hive metastore as shown in Figure 4. If the schema change is forward compatible with the existing reader's schema, then the new data is mapped into the existing reader's schema. For example, the addition of a new column is compatible as it can simply be ignored when copying data into the reader's schema. Nothing changes from the query's points of view and the query continues to execute as before.

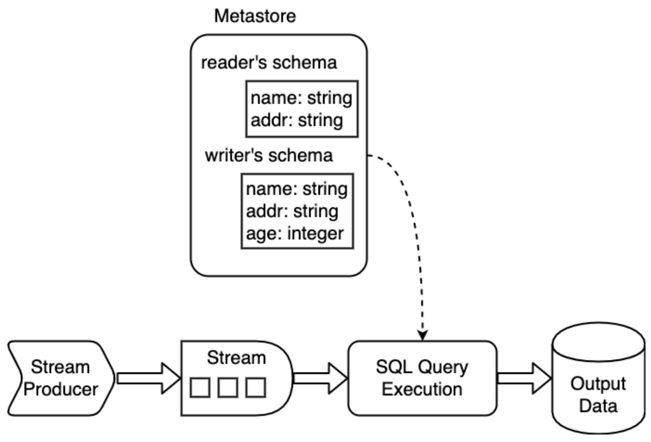

Figure 4: Schema Evolution

If the new writer's schema is incompatible with the existing reader's schema then the query cannot continue without change. Examples of incompatible schema changes include:

- Changing the type of an existing field in an incompatible way, for example changing a numerical field to a text field.
- Allowing `null` values for a field that previously didn't allow nulls.
- Removing a field for which no default value is defined.

The user can specify the preferred behavior when an incompatible schema change occurs. By default the query stops and the query owner is notified. The user can then reformulate the query using the new schema and restart it.

As an alternative, stream producers can be prevented from incompatible schema changes. In this setting, the registration of incompatible schemas fails whenever a producer attempts to publish records with an incompatible schema.

5 FUTURE WORK

Future works includes support for more generic join operations. These joins are only feasible if they operate on well-defined subsets of the involved streams. A stream subset is traditionally called a window [9]. The lookup join described in Section 4.4 can be seen as a special case of generic stream joins where an individual record of the stream is matched against the finite number of records of the lookup table, i.e. a join with an implicit window of size 1.

In a more generic approach, a window is explicitly defined for each stream involved in a join. The window slides across the stream as the stream progresses. Windows have a width and a sliding property. These can be defined based either on time or on volume. For example, a time-based sliding window may cover all the messages received in the last hour and it is advanced every 10 minutes. A volume-based definition defines both window-width and sliding property through the number of messages, e.g. a window containing the last 1000 messages received that is advanced every 100 messages. Windows are also required as the basis for aggregation- and grouping functions on streams.

A join on streams is then decomposed into a series of "sub-joins" on the data contained in the corresponding windows. Each time a window slides ahead, the sub-join is executed again and the result is emitted and appended to the result set. As a window slides along a stream, it may overlap with its predecessor window, causing duplicates.

The design of the SQL service allows a straight forward extension to more generic join operations on stream. The SQL dialect needs to be equipped with additional keywords that allow for the definition of time- and volume-based sliding windows. This requires extensions in the SQL parser that is part of the service API and job handler. The implementation of generic joins is simplified by the fact that Spark already supports stream-to-stream joins.

6 CONCLUSION

We have shown how we have extended an existing commercial cloud SQL service to allow queries to transparently include both static data stored in the cloud and streams of data arriving from external systems. Our implementation uses the Hive metastore as a single location for storing all the metadata required to support these hybrid queries. Streaming sources uses standard schema registries to store their metadata and these are dynamically and transparently integrated into Hive without the sources having to be altered in any way. Queries can then combine tables backed by streaming and batch sources within the SQL service. We have described how we handle the schema evolution of streaming sources and have discussed the different semantics that one can give to streaming queries showing how we currently support various types of joins over streams and batch sources.

REFERENCES

[1] [n.d.]. *ksqlDB Overview.* https://docs.confluent.io/platform/current/ksqldb/index.html

[2] Apache Software Foundation. [n.d.]. *Kafka Streams.* https://kafka.apache.org/documentation/streams/

[3] Igor Bralgin. 2020. Change Data Capture (CDC) Methods. https://www.dwh-club.com/dwh-bi-articles/change-data-capture-methods.html

[4] IBM Cloud. [n.d.]. *Get Started with IBM Cloud Data Engine.* Retrieved Jan 2022 from https://cloud.ibm.com/docs/sql-query?topic=sql-query-getting-started

[5] Apache Foundation. [n.d.]. *Apache Spark.* Retrieved May 2022 from https://spark.apache.org

[6] Apache Foundation. [n.d.]. *PySpark Documentation.* Retrieved May 2022 from https://spark.apache.org/docs/latest/api/python/

[7] Simon Harris and Nailah Bissoon. [n.d.]. Big SQL vs Spark SQL at 100TB: How Do They Stack Up? https://developer.ibm.com/hadoop/2017/02/07/experiences-comparing-big-sql-and-spark-sql-at-100tb/

[8] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. 2015. Presto: Edge-Based Load Balancing for Fast Datacenter Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 465–478. https://doi.org/10.1145/2785956.2787507

[9] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. 2003. Evaluating Window Joins over Unbounded Streams. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman (Eds.). IEEE Computer Society, 341–352.

[10] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-milne, and Michael Yoder. [n.d.]. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *In Proc. CIDR 2015* (2015).

[11] Jay Kreps. 2014. *Kappa Architecture.* Retrieved March 2022 from https://www.oreilly.com/ideas/questioning-the-lambda-architecture

[12] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A Distributed Messaging System for Log Processing. NetDB'11. In *NetDB'11*.

[13] Nathan Marz. 2013. *Big data: principles and best practices of scalable realtime data systems.* O'Reilly Media, [S.l.]. http://www.amazon.de/Big-Data-Principles-Practices-Scalable/dp/1617290343

[14] Red Hat. [n.d.]. *Debezium.* https://debezium.io/

[15] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. [n.d.]. Hive: A Warehousing Solution over a Map-Reduce Framework. 2, 2 ([n. d.]), 1626–1629.