*Industry and Applications Paper*

# KSQL: Streaming SQL Engine for Apache Kafka

Hojjat Jafarpour
Confluent Inc.
Palo Alto, CA
hojjat@confluent.io

Rohan Desai
Confluent Inc.
Palo Alto, CA
rohan@confluent.io

Damian Guy
Confluent Inc.
London, UK
damian@confluent.io

## ABSTRACT

Demand for real-time stream processing has been increasing and Apache Kafka has become the de-facto streaming data platform in many organizations. Kafka Streams API along with several other open source stream processing systems can be used to process the streaming data in Kafka, however, these systems have very high barrier of entry and require programming in languages such as Java or Scala.

In this paper, we present KSQL, a streaming SQL engine for Apache Kafka. KSQL provides a simple and completely interactive SQL interface for stream processing on Apache Kafka; no need to write code in a programming language such as Java or Python. KSQL is open-source, distributed, scalable, reliable, and real-time. It supports a wide range of powerful stream processing operations including aggregations, joins, windowing, sessionization, and much more. It is extensible using User Defined Functions (UDFs) and User Defined Aggregate Functions (UDAFs). KSQL is implemented on Kafka Streams API which means it provides exactly once delivery guarantee, linear scalability, fault tolerance and can run as a library without requiring a separate cluster.

## 1 INTRODUCTION

In recent years, the volume of data that is generated in organizations has been growing rapidly. From transaction log data in e-commerce platforms to sensor generated events in IoT systems to network monitoring events in IT infrastructures, capturing large volumes of data reliably and processing them in a timely fashion has become an essential part of every organization. This has resulted in an emerging paradigm where organizations have been moving from batch oriented data processing platforms towards realtime stream processing platforms.

Initially developed at LinkedIn, Apache Kafka is a battle hardened streaming platform that has been used to capture trillions of events per day [2] [16]. Apache Kafka has become the de-facto streaming platform in many organizations where it provides a scalable and reliable platform to capture and store all the produced data from different systems. It also efficiently provides the captured data to all the systems that want to consume it. While capturing and storing streams of generated data is essential, processing and extracting insight from this data in timely fashion has become even more valuable. Kafka Streams API along with other open source stream processing systems have been used to perform such real time stream processing. Such real time stream processing systems have been used to develop applications such as Streaming ETL, anomaly detection, real time monitoring and many more. Many of these stream processing systems require users to write code in complex languages such as Java or Scala and can only be used by users who are fluent in such languages. This is a high barrier of entry that limits the usability of such systems.

Motivated by this challenge, in this paper we present KSQL, a streaming SQL engine for Apache Kafka that offers an easy way to express stream processing transformations[8]. While the existing open source stream processing systems require expression of stream processing in programming languages such as Java, Scala or Python or offer limited SQL support where SQL statements should be embedded in the Java or Scala code, KSQL offers an interactive environment where SQL is the only language that is needed. KSQL also provides powerful stream processing capabilities such as joins, aggregations, event-time windowing, and many more.

KSQL is implemented on top of the Kafka Streams API which means you can run continuous queries with no additional cluster; streams and tables are first-class constructs; and you have access to the rich Kafka ecosystem. Similar to addition of SQL to other systems such as Apache Hive[17] and Apache Phoenix[4], we believe that introduction of SQL for stream processing in Kafka will significantly broaden the users base for stream processing and bring the stream processing to the masses.

The rest of the paper is organized as the following. In the next section we provide a brief overview on Apache Kafka and the Kafka Streams API. Section 3 presents our contribution in design and development of KSQL. We describe data model, basic concepts, query language and the internals of our SQL engine. In Section 4, we present how KSQL can be extended using UDFs and UDAFs. We describe different execution modes for KSQL in Section 5. We present our experimental evaluation results for KSQL in Section 6. Section 7 describes the related work. We present the future work directions and conclude the paper in Section 8.

## 2 BACKGROUND

KSQL is implemented on top of the Kafka Streams API. In this section we will provide a brief overview on Apache Kafka and the Kafka Streams API.

### 2.1 Apache Kafka

Apache Kafka is a large-scale distributed publish/subscribe messaging system where data is produced to and consumed from topics [2] [15] [16]. Messages in Kafka include a key and a value. Figure 1 depicts the anatomy of a topic in Kafka. Each topic consists of several partitions where messages are assigned to based on their key. Each partition is an ordered, immutable sequence of records that is continually appended to a structured commit log. To achieve fault tolerance, partitions are replicated across a configurable number of servers, called brokers, in a Kafka cluster. One broker for each partition acts as the leader and zero or more brokers act as followers.

Producers publish data to their desired topics by assigning the messages to the specific partition in the topic based on the message key. To consume the published data to a topic, consumers

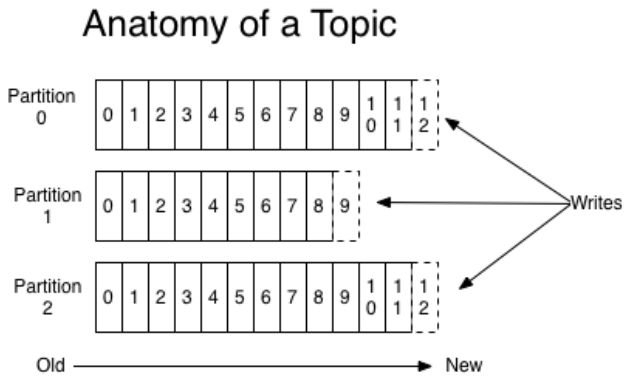


**Figure 1: Anatomy of a Kafka topic**

form consumer groups where each published message will be delivered to one instance in the consumer group. Figure 2 shows two consumer groups that consume messages from a topic with four partitions in a Kafka cluster with two brokers.

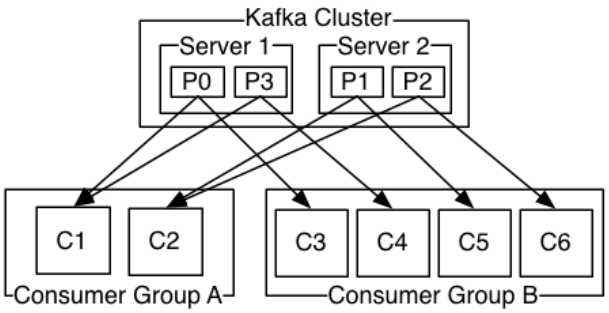


**Figure 2: Two consumer groups reading from a topic with four partitions.**

A consumer groups can expand by adding more members to it. It can also shrink when group members fail or are removed explicitly. Whenever, a consumer group changes, Kafka cluster will go through a rebalancing process for the consumer group to guarantee every partition in the topic will be consumed by one instance in the consumer group. This is done by Kafka group management protocol which is one of the fundamental building blocks of the Kafka streams API as we describe below.

## 2.2 Kafka Streams API

Kafka Streams API is a Java library that enables users to write highly scalable, elastic, distributed and fault-tolerant stream processing applications on top of Apache Kafka [2]. Unlike other stream processing frameworks that need a separate compute cluster to run stream processing jobs, Kafka streams runs as an application. You can write your stream processing application and package it in your desired way, such as an executable jar file, and run instances of it independently. If you need to scale out your application, you just need to bring up more instances of the app and Kafka streams along with Kafka cluster will take care of distributing the load among the instances. The distribution of load in Kafka streams is done with the help of Kafka group management protocol. Figure 3 depicts the architecture of a Kafka streams app.

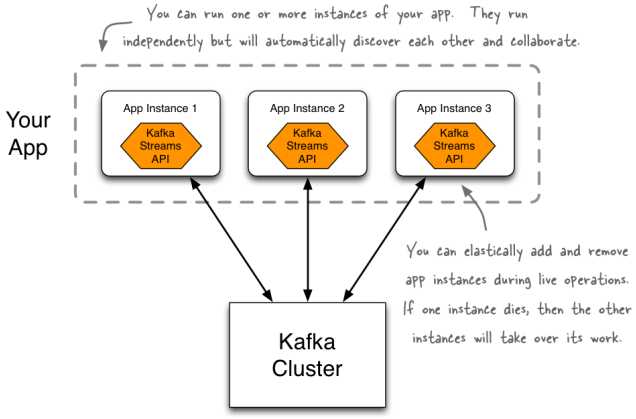


**Figure 3: Two consumer groups reading from a topic with four partitions.**

A typical Kafka streams application will read from one or more Kafka topic and process the data and writes the results into one or more Kafka topics. Kafka streams app uses the same data model as Kafka where messages include a key and a value along with a timestamp and offset of the message in its corresponding partition. The processing logic in a Kafka streams app is defined as a processing topology that include source, stream processor and sink nodes. The processing model is one record at a time where an input record from the source is processed by passing through the whole topology before the next recored is processed. Kafka streams provides powerful stream processing capabilities such as joins, aggregations, event-time windowing, sessionization and more. Operations such as join and aggregation are done based on the message key. Kafka streams uses intermediate Kafka topics to perform shuffle for operations such as aggregation and join that need to colocate data based on a key. For instance, if two streams are being joined and the join key is not the same as the message key for both streams, Kafka streams repartitions both stream topics based on the join key and produces new intermediate topics where message key is the same as the join. This will ensure the colocation of the records with the same key that can be joined at the same node.

Kafka streams provides *stateful* stream processing through the so-called *state stores*. The state stores exist in every instance of the streaming application and are used to store the state in operations such as join and aggregation in a distributed fashion. By default Kafka streams uses RocksDB[9] to store application state, however, any in-memory hash map or other data structures can be plugged in.

# 3 KSQL

In this section we present KSQL, streaming SQL engine for Kafka. As mentioned KSQL uses Kafka streams to run the user queries, therefore, it inherits many properties of Kafka streams.

## 3.1 Data Model

As mentioned a message in a Kafka topic consists of a key and a value. To keep the messages generic, Kafka does not assume any specific format for the messages and both key and value are treated as array of bytes. In addition to the key and value, a message also includes a timestamp, a partition number that it belongs to and the offset value in the corresponding partition. In

order to use SQL on top of Kafka topics we need to impose the relational data model, *schema*, on the value part of messages in Kafka topics. All of the message values in a topic should conform to the associated schema to the topic. The schema defines a message value as a set of columns where each column conforms to the defined data type. Currently, we support the primitive types of BOOLEAN, INTEGER, BIGINT, DOUBLE and VARCHAR along with the complex types of ARRAY, MAP and STRUCT. We plan to add DECIMAL, DATE and TIME types in future. KSQL supports nested column type using the STRUCT type. The fields in a ARRAY, MAP and STRUCT types themselves can be any of the supported types including complex types. As you can see, there is no limit in the level of nesting and users can have as many levels of nesting as they desire. The schema of message values for Kafka topic is used for serialization and deserialization of message values.

## 3.2 Basic Concepts

KSQL provides streaming SQL for Kafka topics meaning that you can write continuous queries that run indefinitely querying future data. There are two basic concepts in KSQL that users can use in their queries, stream and table. Depending on how we interpret the messages in a Kafka topic we can define streams or tables over Kafka topics in KSQL.

If we consider the messages arriving into a topic as independent and unbounded sequence of structured values, we interpret the topic as a ***stream***. Messages in a stream do not have any relation with each other and will be processed independently.

On the other hand, if we consider the messages arriving into a topic as an evolving set of messages where a new message either updates the previous message in the set with the same key, or adds a new message when there is no message with the same key, then we interpret the topic as a ***table***. Note that a table is an state-full entity since we need to keep track of the latest values for each key. In other words, if we interpret the messages in a topic as a **change log** with a state store that represent the latest state, then we interpret the topic as a table.

As an example consider we store the page view events for a website in a Kafka topic. In this case, we should interpret the topic as a stream since each view is an independent message. On the other hand, consider we are storing user information is a Kafka topic where each message either adds a new user if it is not stored already or updates the user information if we already have stored the user. In this case, we should interpret the topic as a table. Note that at any moment, the table should have the most up to date information for every user.

KSQL also provides *windowed* stream processing where you can group records with the same key to perform stateful processing operations such as aggregations and joins. Currently, KSQL supports three types of windows:

- **Tumbling window** which are time-based, fixed-sized, non-overlapping and gap-less windows
- **Hopping window** which are time-based, fixed-sized and overlapping windows
- **Session window** which are session-based, dynamically-sized, non-overlapping and data-driven windows

Note that the results of windowed aggregations are tables in KSQL where we need to keep the state for each window and aggregation group and update them upon receiving new values.

KSQL also support join operation between two streams, a stream and a table or two tables. The stream-stream join always requires a sliding window since we should prevent the size of the state store growing indefinitely. Every time a new message arrives to either of the streams, the join operation will be triggered and the new message for each matching message from the other stream within the join window will be produced. KSQL supports INNER, LEFT OUTER and FULL OUTER join operations for two streams. The RIGHT OUTER join can be implemented via the LEFT OUTER join by simply change the left and right sides of the join. Joining a stream with a table is a stateless operation where each new message in the stream will be matches with the table resulting in emission of zero or one message. Finally, joining two tables in KSQL is consistent with joining them in relational databases if we materialize both of them. KSQL supports INNER, LEFT OUTER and FULL OUTER joins for two tables.

## 3.3 Query Language

KSQL query language is a SQL-like language with extensions to support stream processing concepts. Similar to the standard SQL language, we have DDL and DML statements. DDL statements are used to create or drop streams or tables on top of existing Kafka topic. The followings are two DDL statements to create a pageviews stream and a users table:

```
CREATE STREAM pageviews (viewtime BIGINT,
 userid VARCHAR, pageid VARCHAR) WITH
 (KAFKA_TOPIC='pageviews_topic',
 VALUE_FORMAT='JSON');

CREATE TABLE users (registertime BIGINT,
  gender VARCHAR,
  regionid VARCHAR,
  userid VARCHAR,
address STRUCT<
  street VARCHAR, zip INTEGER
>
 ) WITH (
  KAFKA_TOPIC='user_topic',
  VALUE_FORMAT='JSON',
KEY='userid'
 );
```

Note that in addition to defining the schema for the stream or table we need to provide information on the Kafka topic and the data format in the WITH clause. After declaring streams and tables, we can write continuous queries on them.

Unlike standard SQL statements where the queries return finite set of records as the result, in streaming systems we have continuous queries and therefore the results also will be continuous while the query runs. To address this, KSQL provides two types of query statements. If the results of the query are stored as a new stream or table into a new Kafka topic we use **CSAS(CREATE STREAM AS SELECT)** , **CTAS(CREATE TABLE AS SELECT)** or **INSERT INTO** statements depending on the type of the query results. For instance, the following statement enriches the pageviews stream with extra user information by joining it with the users table and only passes the records with regionid = 'region 10'. The result is a new stream that we call enrichedpageviews:

```
CREATE STREAM enrichedpageviews AS
 SELECT * FROM pageviews LEFT JOIN
 users ON pageviews.userid = users.userid
 WHERE regionid = 'region 10';
```

On the other hand, the following CTAS statement creates a table that contains the pageviews for each user in every 1 hour. Here we use an aggregate query with tumbling window with size of 1 hour.

```
CREATE TABLE userviewcount AS
 SELECT userid, count(*)
 FROM pageviews
 WINDOW TUMBLING (SIZE 1 HOUR)
 GROUP BY useid;
```

Note that the results of the above query will be continuous count values for each user and window that will be stored in a Kafka topic. Every time we receive a new record the current count value for the corresponding userid and window will be updated and the new updated value will be written to the topic. As it can be seen the result will be a change log topic.

Depending on the execution model that we will discuss in the later section, KSQL also provides query manipulation statements where user can submit continuous queries, list the currently running continuous queries and terminate the desired ones.

### 3.4 KSLQ Engine

As mentioned, KSQL uses Kafka streams to run the streaming queries. The main responsibility of the KSQL engine is to compile the KSQL statements into Kafka streams apps that can continuously run and process data streams in Kafka topics. To achieve this KSQL has a metastore component that acts as a system catalog storing information about all the available streams and tables in the system. Currently metastore is an internal component in the KSQL engine. Depending on the execution mode the metastore can be backed by a Kafka topic to provide fault tolerance.

Figure 4 depicts the steps that are taken in the engine to compile KSQL statements into Kafka streams applications to run. As it can be seen, the first step is to parse the statements where the KSQL parser generates an Abstract Syntax Tree (AST). Using the metastore, the generated AST will be analyzed and the unresolved columns references will be resolved. This include detecting the column types along with resolving expression types in the queries along with extracting different components of a query including source, output, projection, filters, join and aggregation. After analyzing each query, we will build a logical plan for it. Logical plan is a tree structure where the nodes are instances of PlanNode class. Currently, we can have the following node types in KSQL logical plan: SourceNode, JoinNode, FilterNode, ProjectNode, AggregateNode and OutputNode. The leaf node(s) are of SourceNode type and the root node is OutputNode type. As it is indicated in Figure 4. rule-based optimization techniques can be applied to the generated logical plan, however, at the moment we do not apply any rule to the logical plan other than pushing down the filters.

The final step is to generate a physical execution plan from the logical plan. The physical plan in KSQL is a Kafka streams topology that runs the stream processing logic. We use the higher level topology structure that is called Kafka streams DSL[2]. Kafka streams defines two fundamental building blocks, **KStream** and **KTable** which are synonymous to stream and table in KSQL. Indeed, A KSQL stream represents a KStream along with a schema. Similarly, a KSQL table represents a KTable along with the associated schema. Kafka streams DSL also provides operations such as map, join, filter, aggregate, etc that convert KStreams/KTables into new KStream/KTables. These operations work on key and value of Kafka messages where there is no assumption on the schema of message value. KSQL defines similar operations on streams and tables, however, in KSQL we impose the proper schema on the message value.

KSQL engine also is responsible for keeping the metastore and queries in the correct state. This includes rejecting statements when they result in incorrect state in the engine. Dropping streams or tables while there are queries that are reading from or writing into them is one of the cases that would result the system to go into an incorrect state. A stream or table can only be dropped if there is no query reading from or writing into it. KSQL engine keeps track of queries that read from or write into a stream or table in the metastore and if it receives a **DROP** statement for s stream or table that is still being used, it will reject the **DROP** statement. Users should make sure that all of the queries that use a stream or table are terminated before they can drop the stream or table.

## 4 UDFS AND UDAFS

Although standard SQL statements provide a good set of capabilities for data processing, many use cases need to perform more complex and custom operation on data. By using functions in queries SQL systems enhance their data processing capabilities. KSQL also provide an extensive set of built in scalar and aggregate functions that can be use in queries. However, to even make KSQL more extensible, we have added capability of adding custom User Defined Functions (UDFs) and User Defined Aggregate Functions (UDTFs).

### 4.1 User Defined Functions

UDF functions are scalar functions that take one input row and return one output value. These functions are stateless, meaning there is no state is maintained between different function calls. Currently, KSQL supports UDFs written in Java. Implementing a new UDF is very straightforward using only two annotations. A Java class annotated by *@UdfDescription* annotation will be considered as a UDF containing class. User provide the name of the UDF by setting the *name* parameter of the *@UdfDescription* annotation. Any function in this class that is annotated by *@Udf* annotation will be considered as a UDF that can be used in any query similar to any other built in function. The following is an example UDF that implements multiplication.

```
@UdfDescription(name = "multiply", description =
    "multiplies 2 numbers")
public class Multiply {

  @Udf(description = "multiply two non-nullable INTs.")
  public long multiply(final int v1, final int v2) {
    return v1 * v2;
  }
}
```

After implementing the UDFs, users can package them in a JAR file and upload it to the designated directory in the KSQL engine where the functions will be loaded from when the KSQL engine starts up. The following query shows how the above UDF can be used in a query:

```
CREATE STREAM test AS
 SELECT multiply(col1, 25)
```

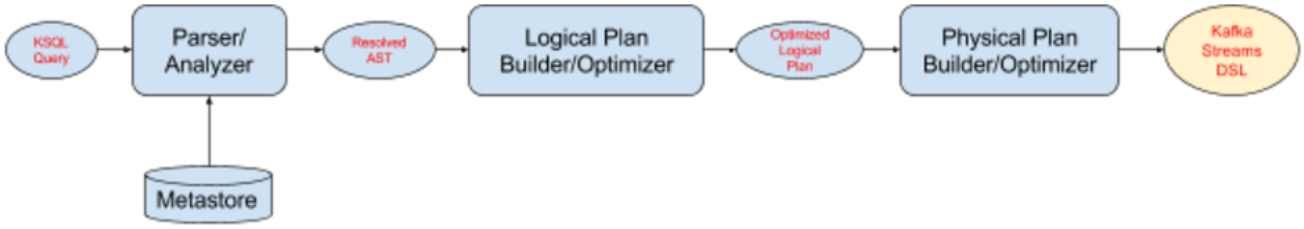


**Figure 4: Steps to convert KSQL statements into Kafka streams apps.**

```
FROM inputStream;
```

## 4.2 User Defined Aggregate Functions

Aggregate functions are applied to a set of rows and compute a single value for them. Similar to the UDFs, KSQL UDAFs are implemented in Java using annotations. To create a new UDAF, users need to create a class that is annotated with *@UdafDescription*. Methods in the class that are used as a factory for creating an aggregation must be *public* and *static*, be annotated with *@UdafFactory*, and must return an instance of *Udaf* class. The instances of a textitUdaf class should implement *initialize*, *aggregate* and *merge* methods. The following is an example UDAF that will perform *sum* operation over double values:

```
@UdafDescription(name = "my_sum", description = "sums")
public class SumUdaf {
@UdafFactory(description = "sums double")
  public static Udaf<Double, Double> createSumDouble() {
    return new Udaf<Double, Double>() {
      @Override
      public Double initialize() {
        return 0.0;
      }

      @Override
      public Double aggregate(final Double aggregate,
           final Double val) {
        return aggregate + val;
      }

      @Override
      public Double merge(final Double aggOne, final
           Double aggTwo) {
        return aggOne + aggTwo;
      }
    };
  }
}
```

Similar to UDFs, UDAFs should be packaged in a JAR file and uploaded to the designated folder in the KSQL engine so the functions can be loaded at the engine start up.

We plan to add support for User Defined Table Functions (UDTFs) in near future.

# 5 EXECUTION MODES

As discussed above, KSQL engine creates Kafka streams topologies that execute the desired processing logic for Kafka topics. Therefore, running KSQL queries is the same as running Kafka streams topologies. Currently, KSQL provides three different execution modes that we describe here.

## 5.1 Application Mode

The application mode is very similar to running a Kafka streams app as described earlier. To deploy and run your queries, you need to put them in a query file and pass it as an input parameter to the KSQL executable jar. Depending on the required resources, you determine the number of instances that your application needs and similar to a Kafka streams execution model you will instantiate the instances by running the KSQL jar with the query file as input parameter. The deployment process can be done manually or through third party resource managers such as Mesos[3] or Kubernetes[10]. Note that you don't need any extra processing cluster and the only thing you need is to run your KSQL app by bringing up desired number of instances independently. Everything else will be handled by KSQL and Kafka streams.

## 5.2 Interactive Mode

KSQL also provides an interactive execution mode where users can interact with a distributed service through a REST API. One way of using the provided REST API is to use KSQL CLI which includes a REST client that sends user requests to the service and receives the response. The building block of the interactive mode is KSQL server that provides a REST end point for users to interact with the service and also KSQL engine to execute the user queries. Figure 5 depicts the architecture of the KSQL service with three servers in the interactive client-server execution mode.

Each KSQL server instance includes two components, the KSQL engine and the REST server. The KSQL service uses a special Kafka topic, **KSQL command topic**, to coordinate among the service instances. When a service instance is started, it first checks the Kafka cluster for the command topic. If the topic does not exists it creates a new command topic with a single partition and a configurable number of replications. All of the service instances then subscribe to the command topic. User interacts with the service by connecting to REST endpoint on one of the instances. The **KSQL command topic** has only one partition to ensure the order of KSQL statements for all server is exactly the same. The **KSQL command topic** can have more than one replicas to prevent loss of KSQL statements in presence of failure. The figure shows the KSQL CLI that connects to either of the instances.

When user submits a new KSQL statement through the REST endpoint, the instance that receives the request will append it to the KSQL command topic. The KSQL engine components in all of the instances will pull the new statement from the command topic and execute the statement concurrently. For instance, consider the following statement is submitted to the service through one of the instances and appended to the command topic.

```
CREATE TABLE userviewcount AS
 SELECT userid, count(*)
 FROM pageviews
```

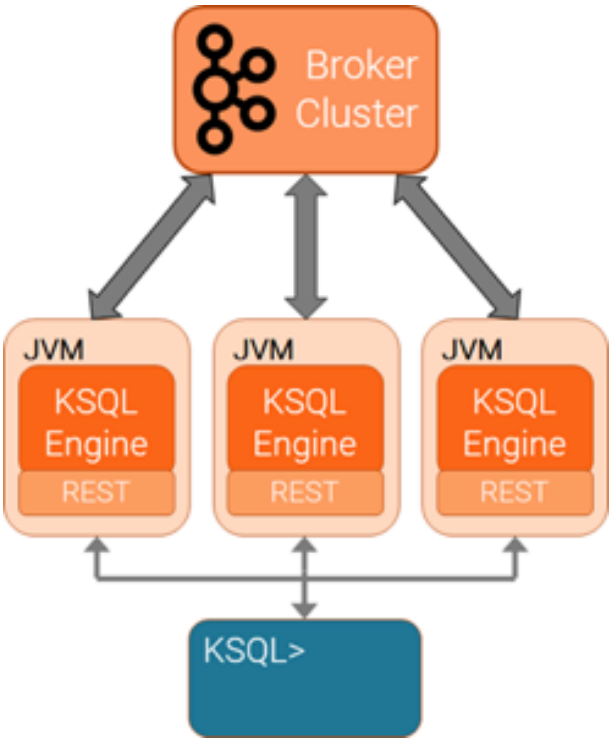


**Figure 5: A KSQL interactive service deployment with three server instances.**

```
WINDOW TUMBLING (SIZE 1 HOUR)
GROUP BY useid;
```

All of the instances will use the KSQL engine to start the processing of the query and as mentioned above each instance will process portion of the input data from the *pageviews* topic. The service instances continue running the queries until they are explicitly terminated using *TERMINATE* statement.

The KSQL service provides both elasticity and fault tolerance. Service instances can be added or removed independently, depending on the load and performance requirements. When a new instance is started it subscribes to the command topic and fetches all the existing KSQL statements from the command topic and starts executing them. When the new instance starts executing an existing query a rebalance process is triggered and the execution load will be redistributed among the existing instances. Note that the rebalance protocol is handled by the underlying Kafka consumers in the Kafka streams and is transparent for the KSQL service instance. After all of the existing continuous queries start running on the new instance too, it starts listening to the command topic for new queries. Similarly when an instance fails or terminated a rebalance process among the remaining instances of the service happens and the load of the removed instance will be distributed among the remaining instances. Even if all the instances fail and the whole system is restarted, the instances will pick up all the existing KSQL queries from the command topic when they come back online and the whole system will continue processing the queries from the point they were left.

### 5.3 Embedded Mode

The third execution mode available in KSQL is the embedded mode where we can embed KSQL statements in a Kafka streams application. As mentioned Kafka streams apps are Java programs that use the Kafka streams as library. KSQL provides a **KSQLContext** class with a **sql()** method where KSQL statements can be passed to run in the embedded engine. The following code snippet show a very simple example of using the embedded mode.

```
KSQLContext ksqlContext = new KSQLContext();
ksqlContext.sql("CREATE STREAM pageviews
 (viewtime BIGINT, userid VARCHAR,
  pageid VARCHAR) WITH
  (KAFKA_TOPIC='pageviews_topic',
  VALUE_FORMAT='JSON');");
ksqlContext.sql("CREATE STREAM pageviewfilter
 AS SELECT * FROM pageviews
  WHERE userid LIKE '\%10';");
```

Similar to the Kafka streams apps, in order to execute the queries in the embedded mode, you need to package the application and run instances of it. The deployment can be done through a range of available options such as manually bringing up instances or using more sophisticated tools such as Mesos or Kubernetes.

## 6 EXPERIMENTAL EVALUATION

### 6.1 Methodology

We want to evaluate KSQL's ability to handle different types of workloads. To do this, we ran a series of tests with different query types and measured the throughput that a single KSQL server can process. Each test case runs for ten minutes and periodically measures throughput in messages / second and bytes / second.

In practice, users will likely run multiple queries that feed into each other to form a streaming pipeline. We've included a multi-query test to measure performance for this scenario.

Finally, we run multiple queries on a pool of KSQL nodes to see how KSQL scales as servers are added.

*6.1.1 Load Generation.* To generate load against KSQL we ran a modified version of the "'ksql-datagen"' tool that has had some changes made to improve its performance and to allow it to produce different types of workloads. "'ksql-datagen"' is a tool for generating data into a kafka cluster. The tool generates Kafka records conforming to a specified schema. The schema can be one of a set of pre-defined schemas, or the user can specify their own avro schema for the tool to use. Records are written to Kafka using the Java Kafka producer client. For our tests, we extended "'ksql-datagen"' to support multiple threads, and to support a rate-limit parameter (specified in messages produced per second) implemented using a token-bucket algorithm.

*6.1.2 Measuring Throughput.* We measured throughput using a counter exported by KSQL that provides the number of messages consumed by a given topic. Throughput for a given time period is computed by sampling the counter at two points in time and dividing the difference in consumed messages by the difference in time. For tests that run multiple queries in a pipeline, consumption is measured against the topic that is the source for the final query in the pipeline. This gives us the throughput of the pipeline as a whole.

*6.1.3 Environment.* We ran our test in AWS EC2. The test environment consists of a Kafka cluster, a KSQL cluster, and a set of nodes that run the load-generator.

The Kafka cluster consists of 5 Kafka nodes, and 1 node for running ZooKeeper. To run Kafka we chose the i3.xlarge instance type, which has 4 2.3GHz VCPUs, 30.5GB of memory, "up-to-10Gbit" of network, and a 950GB NVME local disk which we'll use to store the Kafka topics. We used fio to benchmark the instance storage, and observed 380MBps write throughput, 950MBps read throughput, 2000016K random write IOPS, and 61000 16K random read IOPS. Finally, we ran some network benchmarks using iperf to get an idea of what network performance to expect, and observed 1.24Gbit of network throughput.

Zookeeper runs on an m3.medium.

The KSQL cluster consists of 1-4 nodes, depending on the test case. We chose i3.xlarge instance for KSQL as well, and use the NVME local disk to store the local state stores.

## 6.2 Test Data

The test data and workload are derived from a use case for analyzing metrics. Metrics are published by services that comprise a distributed application. The queries consume, transform, aggregate and enrich this metric data. The source stream for the metrics has the following schema:

```
ID VARCHAR,
SOURCE VARCHAR,
INSTANCE LONG,
CLUSTERID VARCHAR,
METRIC STRUCT<
    NAME STRING,
    VALUE INTEGER,
    METADATA1 STRING,
    METADATA4 STRING,
    METADATA5 STRING,
    METADATA6 STRING,
    METADATA7 STRING,
    METADATA8 STRING,
    METADATA9 STRING,
    METADATA10 STRING
>
```

For our tests, we produced test data serialized using AVRO. The average size for test records was 220 bytes. The source topic containing the metrics data has 32 partitions.

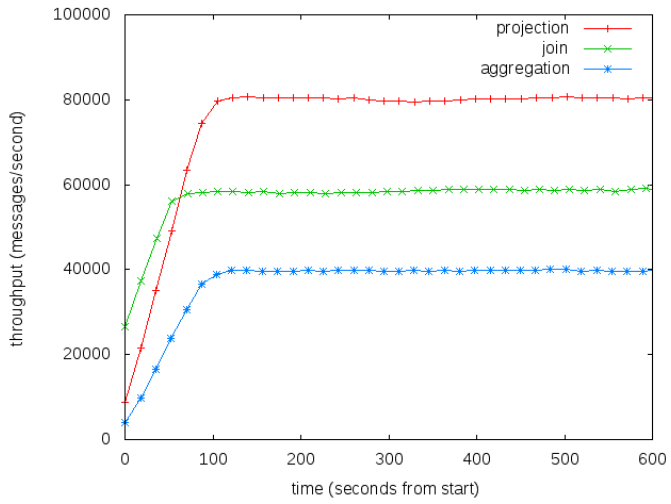


**Figure 6: Throughput for Basic Query Types**

## 6.3 Projection

The first type of query we'll evaluate is a basic projection. Users can use similar queries to apply basic stateless transformations and filters to their data. For the test, we'll run the following query:

```
CREATE STREAM SINK
 AS SELECT *
 FROM METRICS_STREAM;
```

Figure 6 depicts the measured throughput over the test run. Throughput starts low, increases as the JVM warms up, and stabilizes just under 90000 messages per second. The main bottleneck for this query type is CPU utilization, which is nearly 100% on the KSQL server during the test run. To discover why, we re-ran the test case while profiling using the YourKit JVM profiler and found the main hot-spots to be deserialization and serialization of the records after reading from, and before writing back to Kafka.

## 6.4 Aggregation

Next lets look at an aggregation. Aggregations allow the user to group records in a stream or table, and then apply an aggregation function on the grouped records. For this test, we'll run an aggregation to compute the average value of a metric:

```
CREATE TABLE SINK
 AS SELECT
  INSTANCE,
  SUM(METRIC->VALUE) / COUNT(*)
 FROM METRICS_STREAM
 GROUP BY INSTANCE
```

Figure 6 depicts the measured throughput over the test run. Again, the bottleneck is CPU utilization on the KSQL server. In this case, profiling revealed additional overhead from reading from / writing to RocksDB. Each read from RocksDB has to potentially traverse multiple SSTs within the database to find the latest update for the grouping key. The state for aggregations is quite small and fits comfortably within the block cache, so there is no added cost for reading from the kernel cache or decompressing. Writing also adds some cost to write to the memtable and append to the log. The source data for this aggregation is keyed on the grouping column, so there is no additional cost for serializing/deserializing to/from the repartition topic. Aggregations that require a repartition would incur this cost as well.

Space usage on the local disk is quite small, just a few hundred MBs. This makes sense since the state required to compute each aggregation result is small - just the key, sum, and current count, which are all 32-bit or 64-bit integers. Utilization is also quite low - around 50 IOPS and 1MBps. Interestingly, a good portion of the disk usage comes from storing checkpoints for per-task topic offsets rather than from RocksDB itself.

## 6.5 Windowing

Usually, users want to compute aggregations that correspond to a specific time interval - for example, the average value of a metric over a day or hour. KSQL provides tumbling and hopping windows to allow you to window your aggregation computation using a fixed-size window. A tumbling window computes results for one time interval without any overlap with the previous interval. A hopping window has two arguments: the time interval, and the hop size. Results are returned for windows that are as wide as the interval and advance by the hop size. This lets you

approximate a sliding window. In this experiment, we'll compute tumbling and sliding windows and observe the effect on performance:

```
CREATE TABLE SINK
 AS SELECT
  INSTANCE,
  SUM(METRIC->VALUE) / COUNT(*)
 FROM METRICS_STREAM
 WINDOW TUMBLING (
  SIZE 60 SECONDS
 )
 GROUP BY INSTANCE

CREATE TABLE SINK AS SELECT
 INSTANCE,
 SUM(METRIC->VALUE) / COUNT(*)
 FROM METRICS_STREAM
 WINDOW HOPPING(
  SIZE 60 SECONDS,
  ADVANCE BY ? SECONDS
 )
 GROUP BY INSTANCE
```

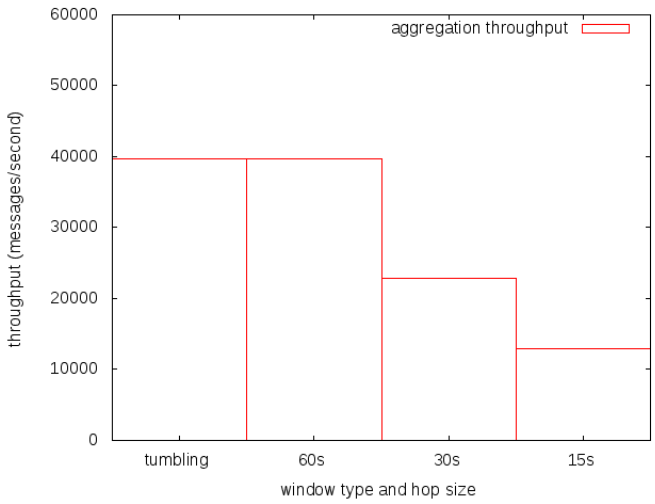


**Figure 7: Throughput for Varying Window Sizes**

Figure 7 shows that the realized throughput decreases as the number of windows that must be maintained increases. This makes sense - under the hood the aggregation result for each window that a given record maps to is maintained independently. If record arriving in the stream must update 4 windows (as in the ADVANCE BY 15 SECONDS case), then KSQL will do 4 separate reads and write from/to RocksDB. This adds to the CPU cost to process each record.

## 6.6 Stream Enrichment

Another common use case for KSQL is to enrich a stream of data with a dimension table. In this test, we'll enrich our metric stream by adding information from an instance table to each metric record:

```
CREATE STREAM SINK
 AS SELECT S.*, T.*
 FROM
  METRIC_STREAM S
  LEFT JOIN
  INSTANCE_TABLE S
 ON S.INSTANCE = T.INSTANCE;
```

Figure 6 shows the throughput over time. Again, the primary bottleneck is CPU utilization, which is near 100%. The realized throughput is around 55000 records per second - greater than the aggregation but lower than the projection. This is because to process each record in the stream we must read from RocksDB, but nothing is written back to the state store. Therefore, the CPU utilization per record is lower than that for aggregations, and the realized throughput is higher.

## 6.7 Multi-Query Application

Usually, KSQL users run multiple queries in a streaming pipeline to form a KSQL "application". To evaluate KSQL in this scenario, we'll build the following test app:

```
CREATE STREAM METRICS_WITH_INSTANCE_INFO
 AS SELECT S.*, T.*
 FROM
  METRIC_STREAM S
  LEFT JOIN
  INSTANCE_TABLE S
 ON S.INSTANCE = T.INSTANCE;

CREATE TABLE AVERAGE_VALUE_BY_PHYSICAL_INSTANCE
 AS SELECT
  T_PHYSICAL_INSTANCE,
  SUM(METRIC->VALUE) / COUNT(*)
 FROM METRICS_WITH_INSTANCE_INFO
 GROUP BY PHYSICAL_INSTANCE;
```

In the app we first enrich the metrics stream with the instance table (as in the previous section), and then compute average metrics grouped by the "physical instance" each instance resides on (which is another field of the instance table schema).

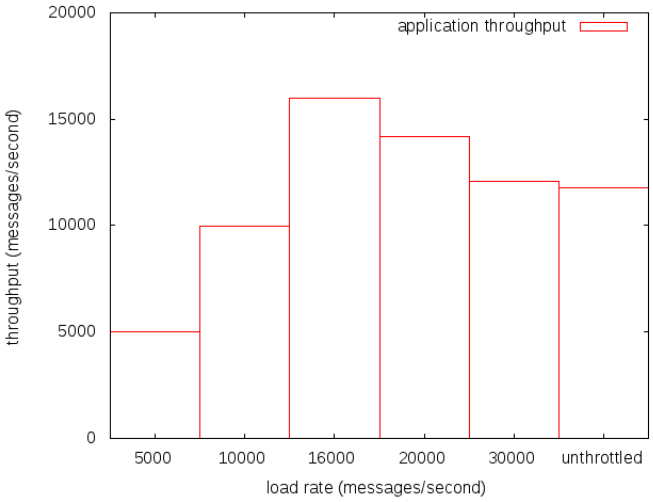


**Figure 8: Throughput for Varying Load Rates**

To evaluate the possible throughput a KSQL server can process against the app, we generate load at varying rates and measure the consumption rate from METRICS_WITH_INSTANCE_INFO. To understand why its important to rate limit the load generator

to determine the maximum possible throughput, lets consider the behavior if load generation was run at a rate that would produce records faster than KSQL could consume them. The "application" presented in Figure 8 consists of 2 queries. The 2 queries have different performance characteristics. Specifically, the join consumes significantly fewer CPU cycles than the aggregation to process the same set of input records. Currently, KSQL does not coordinate execution between separate queries. So, each query would be afforded around half the available CPU time. Ideally, the aggregation should be afforded more CPU than the join. Throttling the input rate gives us a knob to do this indirectly. In practice, a user could accomplish the same thing by scaling out the capacity of their KSQL cluster, or allocating varying numbers of threads to different queries. Figure 8 shows that the maximum throughput a single KSQL server can process running this application is around 16000 records/second.

### 6.8 Scale-Out

KSQL is designed to scale record throughput as KSQL servers are added. In this section, we'll show that KSQL can process the application we evaluated in the previous section at higher volumes as the cluster grows. We'll evaluate a 2-node and 4-node KSQL cluster. For each configuration, we'll run at 2 times and 4 times the ideal load observed in the previous section, and measure the realized throughput.

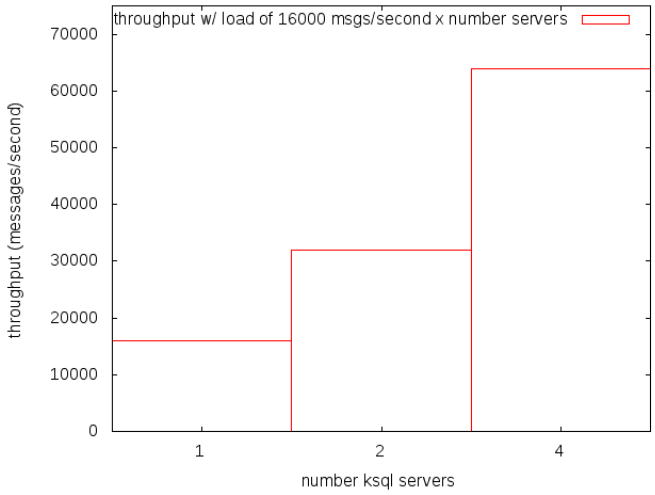


**Figure 9: Throughput For Varying Cluster Sizes**

Figure 9 shows the results of this experiment. Each bar represents the throughput processed by a KSQL cluster of a given size. Load was generated proportional to the cluster size - 16000 messages per second for the single node cluster, and 32000 messages per second and 64000 messages per second for the 2 and 4 node clusters, respectively. We can see that KSQL is able to process proportionally higher volumes of data as the cluster is grown.

### 6.9 Future Experiments

The previous sections detailed some initial results of our evaluation of KSQL performance. In the future, we plan to do additional analysis into performance as well as fix some of the issues identified. Latency is a very important metric when evaluating streaming systems that was not measured by our study. After all, one of the primary advantages of stream processing is to be able react to events in real-time. Therefore we plan to measure percentile latency as we invest in evaluating and improving KSQL performance. We also plan to evaluate additional query types. KSQL is a very powerful tool that can perform a wide variety of computations not covered by the evaluation presented here. Some candidate query types for the next round of testing include stream-stream joins and aggregations over session windows.

## 7 RELATED WORK

Over the past couple of decades stream processing has been one of the main subjects of interest in the data management community. Systems like STREAM [12], Aurora [11] and TelegraphCQ [14] are some of the pioneers in this area.

Increasing demand for scalable streaming engine resulted in development of many systems including open source systems such as Storm [7], Spark [6], Flink [1] and Samza [5]. Although these systems provide a scalable stream processing engine, however, all of them need deployment and management of a complex processing cluster in order to process the streaming jobs. On the other hand, KSQL uses Kafka streams and KSQL queries can be deployed as an application without requiring another complex processing cluster. This significantly simplifies deployment and management of long running streaming jobs in KSQL and provides a wide range of possible deployment options.

SQL language has been the de-facto data management language and many systems support SQL or SQL-like interface. Apache Hive [17] and Apache Spark SQL [13] have shown the effectiveness of using SQL to express computations in batch data processing. They are also great examples of how providing SQL interface can expand the access to scalable data processing systems by eliminating the need to write code in complex programming languages. Some of the open source stream processing frameworks including Apache Spark [6], Apache Flink [1] take this idea to scalable stream processing by providing SQL support. However, the level of SQL support in these system is not at the same level as in KSQL and in order to use SQL for stream processing in these systems users still need to write code in languages such as Java or Scala and embed their SQL statements in their code. On the other hand, in KSQL users describe their processing in KSQL statements and there is no need to write any code in Java or Scala or any other language. This approach significantly simplifies scalable stream processing and makes it available for greater audience.

## 8 CONCLUSIONS AND FUTURE WORK

In this paper we introduced KSQL, a streaming SQL engine for Apache Kafka. KSQL uses Kafka streams API to run continuous queries and is tightly integrated with Apache Kafka. We showed how a KSQL query is translated into a Kafka streams app in KSQL engine and how we can run KSQL queries in different execution modes. One of the main advantages of KSQL compared to other open source stream processing systems is the elimination of need to code in any other language. KSQL users can use the KSQL CLI and run their streaming queries by expressing them in only KSQL statements. The other main differentiator of KSQL compared to other open source streaming platforms is the query execution model. Unlike other systems that need deployment of a separate processing cluster to handle streaming queries, KSQL queries can run as applications independently without requiring deployment of additional complex cluster. This significantly simplifies deployment and management of streaming queries in KSQL.

We just announced availability of KSQL as an open source streaming SQL engine and released it under Apache license in 2017 Kafka Summit in San Francisco and since then we have witnessed significant interest from the community[8]. We plan to invest heavily in development and expansion of KSQL. Currently we are looking into expanding supported data formats in KSQL along with providing custom functionality through UDF and UDAF in addition to the ones we already have in KSQL. Improving the performance through optimized query planning along with even more simplified deployment and maintenance of KSQL service in the production environment are part of our near future work on KSQL. We believe KSQL along with Apache Kafka provide a full stack stream processing environment that can satisfy the all the real-time stream processing needs in enterprises and will be working towards this goal.

## REFERENCES


[1] 2018. Apache Flink. (2018). http://flink.apache.org
[2] 2018. Apache Kafka. (2018). http://kafka.apache.org
[3] 2018. Apache Mesos. (2018). http://mesos.apache.org
[4] 2018. Apache Phoenix. (2018). http://phoenix.apache.org
[5] 2018. Apache Samza. (2018). http://samza.apache.org
[6] 2018. Apache Spark. (2018). http://spark.apache.org
[7] 2018. Apache Storm. (2018). http://storm.apache.org
[8] 2018. KSQL. (2018). http://github.com/confluentinc/ksql
[9] 2018. RocksDB. (2018). http://rocksdb.org
[10] 2018. RocksDB. (2018). http://kubernetes.io
[11] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB Journal* 12 (2003), 120–139.
[12] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *VLDB Journal* 15 (2006), 121–142.
[13] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*.
[14] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Frederick Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*.
[15] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka a Distributed Messaging System for Log Processing. In *Proc. NetDB 11*.
[16] M.s Sax, G. Wang, M. Weidlich, and J. Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proc. BRITE 18*.
[17] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering*.