# Apache Kafka®

BY **JUN RAO**

CONTENTS



## WHY APACHE KAFKA

Two trends have emerged in the information technology space. First, the diversity and velocity of the data that an enterprise wants to collect for decision-making continues to grow. Such data includes not only transactional records, but also business metrics, IoT data, operational metrics, application logs, etc.

Second, there is a growing need for an enterprise to make decisions in real time based on that collected data. Finance institutions want to not only detect fraud immediately, but also offer a better banking experience through features like real-time alerting, real-time recommendation, more effective customer service, and so on. Similarly, it's critical for retailers to make changes in catalog, inventory, and pricing as quickly as possible. It is truly a real-time world.

Before Apache Kafka, there wasn't a system that perfectly met all of the above business needs. Traditional messaging systems are real-time, but weren't designed to handle data at scale. Newer systems such as Hadoop are much more scalable, but were mostly designed for batch processing.

Apache Kafka is a streaming platform for collecting, storing, and processing high volumes of data in real time. As illustrated in Figure 1, Kafka typically serves as a central data hub in which all data within an enterprise is collected. The data can then be used for continuous processing or fed into other systems and applications in real time. Kafka is in use by more than 30% of Fortune 500 companies across all industries.

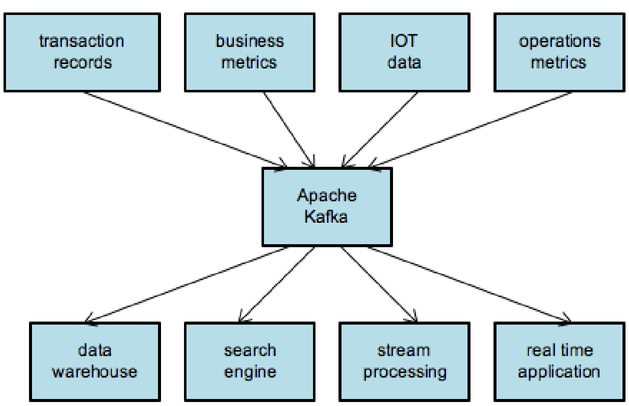



***Figure 1***: *Kafka as a central real-time hub*

## ABOUT APACHE KAFKA

Kafka was originally developed at LinkedIn in 2010, and became a top level Apache project in 2012. It has three main components: Pub/Sub, Kafka Connect API, and Kafka Streams API. The role of each component is summarized in the table below.

| | |
|---|---|
| **Pub/Sub** | Storing and delivering data effciently and reliably at scale. |
| **Kafka Connect** | Intgrating Kafka with external data sources and data sinks |
| **Kafka Streams** | Processing data in Kafka in real time |

The main benefits of Kafka are:

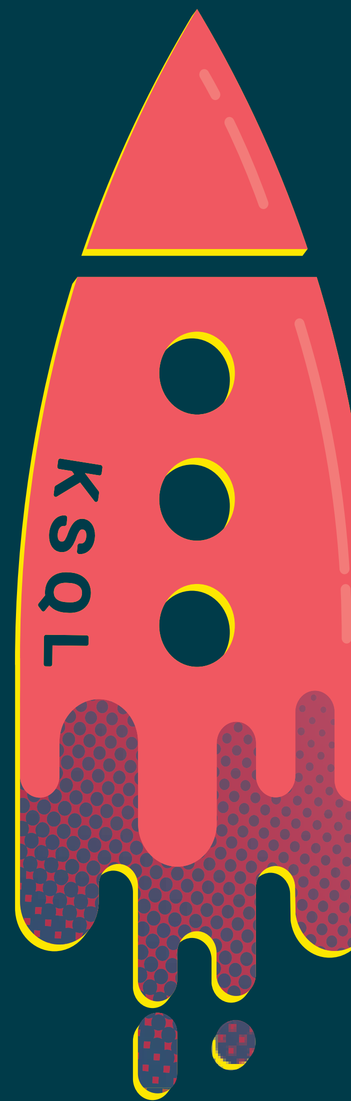
1. **High throughput**: Each server is capable of handling hundreds of MB/sec of data.
2. **High availability**: Data can be stored redundantly in multiple servers and can survive individual server failure.
3. **High scalability**: New servers can be added over time to scale out the system.
4. Easy integration with external data sources or data sinks.
5. Built-in real-time processing layer.

# KSQL

## An Open Source Streaming SQL Engine for Apache Kafka®

**Simple real-time streaming ETL, anomaly detection, and monitoring.**

```
CREATE TABLE possible_fraud AS
SELECT card_number, count(*)
FROM authorization_attempts
WINDOW TUMBLING (SIZE 5 SECONDS)
GROUP BY card_number
HAVING count (*)>3;
```




**NO CODING REQUIRED**

**Learn more about KSQL**
confluent.io/ksql

confluent

## QUICK START APACHE KAFKA

It's easy to get started with Kafka. The following are the steps to run the quickstart script.

1. Download the Kafka binary distribution from kafka.apache.org/downloads and untar it:

```
> tar -xzf kafka_2.11-1.0.0.tgz
     > cd kafka_2.11-1.0.0
```

2. Start the Zookeeper server:

```
> bin/zookeeper-server-start.sh
     config/zookeeper.properties
```

3. Start the Kafka broker:

```
> bin/kafka-server-start.sh
     config/server.properties
```

4. Create a topic:

```
> bin/kafka-topics.sh --create
      --zookeeper localhost:2181
      --topic test --partitions 1
      --replication-factor 1
```

5. Produce data:

```
> bin/kafka-console-producer.sh
    --broker-list localhost:9092
    --topic test

      hello
      world
```

6. Consume data:

```
> bin/kafka-console-consumer.sh
    --bootstrap-server localhost:9092
    --topic test --from-beginning

     hello
      world
```

After step 5, one can type in some text in the console. Then, in step 6, the same text will be displayed after running the consumer.

## PUB/SUB APACHE KAFKA

The first component in Kafka deals with the production and the consumption of the data. The following table describes a few key concepts in Kafka:

| | |
|---|---|
| topic | Defines a logical name for producing and consuming records. |
| partition | Defines a non-overlapping subset of records within a topic. |
| offset | A unique sequential number assigned to each record within a topic partion. |
| record | A record contains a key, a value, a timestamp, and a list of headers. |
| broker | Servers where records are stored. Multiple brokers can be used to form a cluster. |

Figure 2 depicts a topic with two partitions. Partition 0 has 5 records, with offsets from 0 to 4, and partition 1 has 4 records, with offsets from 0 to 3.

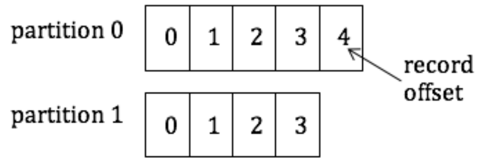



***Figure 2:*** *Partions in a topic*

The following code snippet shows how to produce records to a topic "test" using the Java API:

```
Properties props = new Properties();
props.put("bootstrap.servers",
  "localhost:9092");
props.put("key.serializer",
  "org.apache.kafka.common.serialization.
StringSerializer");
props.put("value.serializer",
  "org.apache.kafka.common.serialization.
  StringSerializer");
Producer<String, String> producer = new
  KafkaParoducer<>(props);
producer.send(
  new ProducerRecord<String, String>("test", "key",
"value"));
```

In the above example, both the key and value are strings, so we are using a `StringSerializer`. It's possible to customize the serializer when types become more complex. For example, the `KafkaAvroSerializer` from docs.confluent.io/current/schema-registry/docs/serializer-formatter.html allows the user to produce Avro records.

The following code snippet shows how to consume records with string key and value in Java.

```
Properties props = new Properties();  props.
put("bootstrap.servers", "localhost:9092");
props.put("key.deserializer",
   "org.apache.kafka.common.serialization.
StringDeserializer");
props.put("value.deserializer",
   "org.apache.kafka.common.serialization.
StringDeserializer");
KafkaConsumer<String, String> consumer =
   new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("test"));

while (true) {
  ConsumerRecords<String, String> records =
     consumer.poll(100);
  for (ConsumerRecord<String, String> record :
records)
    System.out.printf("offset=%d, key=%s, value=%s",
      record.offset(), record.key(), record.value());
  consumer.commitSync();
}
```

Records within a partition are always delivered to the consumer in offset order. By saving the offset of the last consumed record from each partition, the consumer can resume from where it left off after a restart. In the example above, we use the `commitSync()` API to save the offsets explicitly after consuming a batch of records. One can also save the offsets automatically by setting the property `enable.auto.commit` to true.

Unlike other messaging systems, a record in Kafka is not removed from the broker immediately after it is consumed. Instead, it is retained according to a configured retention policy. The following table summarizes the two common policies:

| RETENTION POLICY | MEANING |
|---|---|
| `log.retention.hours` | The number of hours to keep a record on the broker. |
| `log.retention.bytes` | The maximum size of records retained in a partition. |

## KAFKA CONNECT

The second component in Kafka is Kafka Connect, which is a framework for making it easy to stream data between Kafka and other systems. As shown in Figure 3, one can deploy a Connect cluster and run various connectors to import data from sources like MySQL, MQ, or Splunk into Kafka and export data in Kafka to sinks such as HDFS, S3, and Elastic Search. A connector can be either of source or sink type:

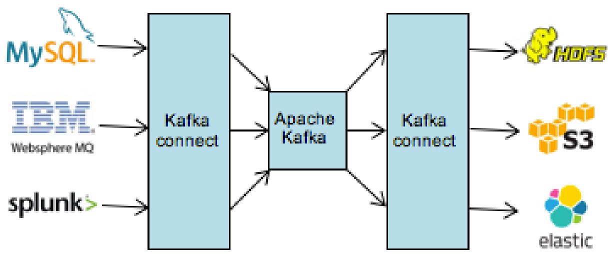



***Figure 3:*** *Using Kafka Connect*

The benefits of using Kafka Connect are:

- Parallelism and fault tolerance
- Avoiding ad-hoc code by reusing existing connectors
- Built-in offset and configuration management

## QUICK STARTS FOR KAFKA CONNECT

The following steps show how to run the existing file connector in standalone mode to copy the content from a source file to a destination file via Kafka:

1. Prepare some data in a source file:

```
> echo -e "hello\nworld" > test.txt
```

2. Start a file source and a file sink connector:

```
> bin/connect-standalone.sh
config/connect-file-source.properties
config/connect-file-sink.properties
```

3. Verify the data in the destination file:

```
> more test.sink.txt

  hello
  world
```

4. Verify the data in Kafka:

```
> bin/kafka-console-consumer.sh
        --bootstrap-server localhost:9092
        --topic connect-test
        --from-beginning

       {"schema":{"type":"string",
                  "optional":false},
        "payload":"hello"}
       {"schema":{"type":"string",
                  "optional":false},
        "payload":"world"}
```

In the above example, the data in the source file test.txt is first streamed into a Kafka topic `connect-test` through a file source connector. The records in `connect-test` are then streamed into the destination `file test.sink.txt`. If a new line is added to `test.txt`, it will show up immediately in `test.sink.txt`. Note that we achieve the above by running two connectors without writing any custom code.

The following is a partial list of existing connectors. A more complete list can be found at [confluent.io/product/connectors](confluent.io/product/connectors).

| CONNECTOR | TYPE | CONNECTOR | TYPE |
|---|---|---|---|
| elastic search | sink | HDFS | sink |
| Amazon S3 | sink | Cassandra | sink |
| Oracle | source | Mongo DB | source |
| MQTT | source | JMS | sink |
| Couchbase | sink<br>source | JDBC | sink<br>source |
| IBM MQ | sink<br>source | Dynamo DB | sink<br>source |

## TRANSFORMATIONS IN CONNECT

Connect is primarily designed to stream data between systems as-is, whereas Streams is designed to perform complex transformations once the data is in Kafka. That said, Connect does provide a mechanism to perform some simple transformations per record. The following example shows how to enable a couple of transformations in the file source connector.

1. Add the following lines to connect-file-source.properties:

```
transforms=MakeMap, InsertSource
transforms.MakeMap.type=org.apache.kafka
  .connect.transforms.HoistField$Value
transforms.MakeMap.field=line
transforms.InsertSource.type=org.apache
  .kafka.connect.transforms
  .InsertField$Value
transforms.InsertSource.static.field=
  data_source
transforms.InsertSource.static.value=
  test-file-source
```

2. Start a file source connector:

```
> bin/connect-standalone.sh
config/connect-file-source.properties
```

3. Verify the data in Kafka:

```
> bin/kafka-console-consumer.sh
  --bootstrap-server localhost:9092
  --topic connect-test
{"line":"hello","data_source":"test
  -file-source"}
{"line":"world","data_source":"test
  -file-source"}
```

In step 1 above, we add two transformations MakeMap and InsertSource, which are implemented by class HoistField$Value and InsertField$Value, respectively. The first one adds a field name "line" to each input string. The second one adds an additional field "data_source" that indicates the name of the source file. After applying the transformation logic, the data in the input file is now transformed to the output in step 3.

## CONNECT REST API

In production, Connect typically runs in distributed mode and can be managed through REST APIs. The following table shows the common APIs.

| CONNECT REST API | MEANING |
|---|---|
| GET /connectors | Return a list of active connectors |
| POST /connectors | Create a new connector |
| GET /connectors/{name} | Get the information of a specific connector |
| GET /connectors/{name}/config | Get configuration parameters for a specific connector |
| PUT /connectors/{name}/config | Update configuration parameters for a specific connector |
| GET /connectors/{name}/status | Get the current status of the connector |

## KAFKA STREAMS

Kafka Streams is a client library for building real-time applications and microservices, where the input and/or output data is stored in Kafka. The benefits of using Kafka Streams are:

- Less code in the application
- Built-in state management
- Lightweight
- Parallelism and fault tolerance

The most common way of using Kafka Streams is through the Streams DSL, which includes operations such as filtering, joining, grouping, and aggregation. The following code snippet shows the main logic of a Streams example called WordCountDemo.

```
final Serde<String> stringSerde = Serdes.String();
final Serde<Long> longSerde = Serdes.Long();


StreamsBuilder builder = new StreamsBuilder();


// build a stream from an input topic
KStream<String, String> source = builder.stream(

     "streams-plaintext-input",
     Consumed.with(stringSerde, stringSerde));


KTable<String, Long> counts = source
    .flatMapValues(value -> Arrays.asList(value.
toLowerCase().split(" ")))
     .groupBy((key, value) -> value)

    .count();

// convert the output to another topic
counts.toStream().to("streams-wordcount-output",

    Produced.with(stringSerde, longSerde));
```

The above code first creates a stream from an input topic streams-plaintext-input. It then applies a transformation to split each input line into words. After that, it groups by the words and count the number of occurrences in each word. Finally, the results are written to an output topic streams-wordcount-output.

The following are the steps to run the example code.

1. Create the input topic:

```
> bin/kafka-topics.sh --create
    --zookeeper localhost:2181
    --replication-factor 1
    --partitions 1
    --topic streams-plaintext-input
```

2. Run the stream application:

```
> bin/kafka-run-class.sh org.apache.
   kafka.streams.examples.wordcount.
   WordCountDemo
```

3. Produce some data in the input topic:

```
> bin/kafka-console-producer.sh
    --broker-list localhost:9092
    --topic streams-plaintext-input

  hello world
```

4. Verify the data in the output topic:

```
> bin/kafka-console-consumer.sh
    --bootstrap-server localhost:9092
    --topic streams-wordcount-output
    --from-beginning
    --formatter kafka.tools.
        DefaultMessageFormatter
    --property print.key=true
    --property print.value=true
    --property key.deserializer=
     org.apache.kafka.common.
     serialization.StringDeserializer
    --property value.deserializer=
     org.apache.kafka.common.
     serialization.LongDeserializer

   hello 1
   world 1
```

## KSTREAM VS. KTABLE

There are two key concepts in Kafka Streams: KStream and KTable. A topic can be viewed as either of the two. Their differences are summarized in the table below.

| | KSTREAM | KTABLE |
|---|---|---|
| CONCEPT | Each record is treated as an append to the stream. | Each record is treated as an update to an existing key |
| USAGE | Model append-only data such as click streams. | Model updatable reference data such as user profiles. |

The following example illustrates the difference between the two:

| (key , value) records | Sum of values As KStream | Sum of values As KTable |
|---|---|---|
| (“k1”, 2)<br>(“k1”, 5) | 7 | 5 |

When a topic is viewed as a KStream, there are two independent records and thus the sum of the values is 7. On the other hand, if the topic is viewed as a KTable, the second record is treated as an update to the first record since they have the same key “k1”. Therefore, only the second record is retained in the stream and the sum is 5 instead.

## KSTREAMS DSL

The following tables show a list of common operations available in Kafka Streams:

### COMMONLY USED OPERATIONS IN KSTREAM

| OPERATIONS | EXAMPLE |
|---|---|
| `filter(Predicate)`<br><br>Create a new KStream that consists of all records of this stream that satisfy the given predicate. | `ks_out = ks_in.filter(`<br>`    (key, value) -> value > 5`<br>`);`<br><br>`ks_in:          ks_out:`<br>`("k1", 2)       ("k2", 7)`<br>`("k2", 7)` |
| `map(KeyValueMapper)`<br><br>Transform each record of the input stream into a new record in the output stream (both key and value type can be altered arbitrarily). | `ks_out = ks_in..map(`<br>`  (key, value) ->`<br>`     new KeyValue<>(key, key)`<br>`)`<br><br>`ks_in:          ks_out:`<br>`("k1", 2)       ("k1", "k1")`<br>`("k2", 7)       ("k2", "k2")` |
| `groupBy()`<br><br>Group the records by their current key into a KGroupedStream while preserving the original values. | `ks_out = ks.groupBy()`<br><br>`ks_in:          ks_out:`<br>`("k1", 1)       ("k1", (("k1", 1),`<br>`("k2", 2)               ("k1", 3)))`<br>`("k1", 3)       ("k2", (("k2", 2)))` |
| `join(KTable,`<br>`      ValueJoiner)`<br><br>Join records of the input stream with records from the KTable if the keys from the records match. Return a stream of the key and the combined value using ValueJoiner. | `ks_out = ks_in.join(`<br>`  kt,`<br>`  (value1, value2) ->`<br>`    value1 + value2`<br>`);`<br><br>`ks_in:      kt:`<br>`("k1", 1)   ("k1", 11)`<br>`("k2", 2)   ("k2", 12)`<br>`("k3", 3)   ("k4", 13)`<br><br>`ks_out:`<br>`("k1",  12)`<br>`("k2",  14)` |
| `join(KStream,`<br>`      ValueJoiner,`<br>`      JoinWindows)`<br><br>Join records of the two streams if the keys match and the timestamp from the records satisfy the time constraint specified by JoinWindows. Return a stream of the key and the combined value using ValueJoiner. | `ks_out = ks1.join(`<br>`  ks2,`<br>`  (value1, value2) ->`<br>`     value1 + value2,  JoinWindows.`<br>`     of(100)`<br>`);`<br><br>`ks1:              ks2:`<br>`("k1", 1, 100t)   ("k1", 11, 150t)`<br>`("k2", 2, 200t)   ("k2", 12, 350t)`<br>`("k3", 3, 300t)   ("k4", 13, 380t)`<br>`* t indicates a timestamp.`<br><br>`ks_out:`<br>`("k1",  12)` |

### COMMONLY USED OPERATIONS IN KGROUPEDSTREAM

| OPERATION | EXAMPLE |
|---|---|
| `count()`<br><br>Count the number of records in this stream by the grouped key and return it as a KTable. | `kt = kgs.count();`<br>`kgs:`<br>`("k1", (("k1", 1),`<br>`        ("k1", 3)))`<br>`("k2", (("k2", 2)))`<br><br>`kt:`<br>`("k1", 2)`<br>`("k2", 1)` |

| OPERATION | EXAMPLE |
|---|---|
| `reduce(Reducer)`<br><br>Combine the values of records in this stream by the grouped key and return it as a KTable. | `kt = kgs.reduce(`<br>`  (aggValue, newValue) ->`<br>`    aggValue + newValue`<br>`);`<br><br>`kgs:`<br>`(“k1”, ((“k1”, 1),`<br>`         (“k1”, 3)))`<br>`(“k2”, ((“k2”, 2)))`<br><br>`kt:`<br>`(“k1”, 4)`<br>`(“k2”, 2)` |
| `windowedBy(Windows)`<br><br>Further group the records by the timestamp and return it as a `TimeWindowedKStream`. | `twks = kgs.windowedBy(`<br>`  TimeWindows.of(100)`<br>`);`<br><br>`kgs:`<br>`(“k1”, ((“k1”, 1, 100t),`<br>`         (“k1”, 3, 150t)))`<br>`(“k2”, ((“k2”, 2, 100t),`<br>`         (“k2”, 4, 250t)))`<br>`* t indicates a timestamp.`<br><br>`twks:`<br>`(“k1”, 100t - 200t,`<br>`       ((“k1”, 1, 100t),`<br>`       (“k1”, 3, 150t)))`<br>`(“k2”, 100t - 200t,`<br>`       ((“k2”, 2, 100t)))`<br>`(“k2”, 200t - 300t,`<br>`       ((“k2”, 4, 250t)))` |

A similar set of operations is available on KTable and KGroupedTable. See details at kafka.apache.org/documentation/streams.

## QUERYING THE STATES IN KSTREAMS

While processing the data in real time, a KStreams application locally maintains the states such as the word counts in the previous example. Those states can be queried interactively through an API described in docs.confluent.io/current/streams/developer-guide/index.html#interactive-queries. This avoids the need of an external data store for exporting and serving those states.

## EXACTLY-ONCE PROCESSING IN KSTREAMS

Failures in the brokers or the clients may introduce duplicates during the processing of records. KStreams provides the capability of processing records exactly once even under failures. This can be achieved by simply setting the property `processing.guarantee` to `exactly_once` at KStreams. More details on exactly-once processing can be found at confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it.

## KSQL

KSQL is an open source streaming SQL engine that implements continuous, interactive queries against Apache Kafka. It's built using the Kafka Streams API and further simplifies the job of a developer. Currently, KSQL is not part of the Apache Kafka project, but is available under the Apache 2.0 license.

To see how KSQL works, let's first download it and prepare some data sets.

1. Clone the KSQL repository and compile the code:

```
> git clone git@github.com:
    confluentinc/ksql.git
> cd ksql
> mvn clean compile install
    -DskipTests
```

2. Produce some data in two topics:

```
> java -jar ksql-examples/target/
    ksql-examples-0.1-SNAPSHOT-
      standalone.jar
    quickstart=pageviews
     format=delimited
     topic=pageviews
     maxInterval=10000


> java -jar ksql-examples/target/
    ksql-examples-0.1-SNAPSHOT-
      standalone.jar
    quickstart=users
    format=json
    topic=users
    maxInterval=10000
```

Next, let's define the schema of the input topics. Similar to Kafka Streams, one can define a schema as either a stream or a table.

1. Start KSQL CLI:

```
> ./bin/ksql-cli local
```

2. Create a KStream from a topic:

```
ksql> CREATE STREAM pageviews_stream
      (viewtime bigint,
       userid varchar,
       pageid varchar)
      WITH (kafka_topic='pageviews',
        value_format='DELIMITED');

ksql> DESCRIBE pageviews_stream;

      Field    | Type
     ----------------------------
      ROWTIME  | BIGINT
      ROWKEY   | VARCHAR(STRING)
      VIEWTIME | BIGINT
      USERID   | VARCHAR(STRING)
      PAGEID   | VARCHAR(STRING)
```

3. Create a KTable from a topic:

```
ksql> CREATE TABLE users_table
      (registertime bigint,
       gender varchar,
       regionid varchar,
       userid varchar)
      WITH (kafka_topic='users',
        value_format='JSON');

ksql> DESCRIBE users_ktable;

      Field        | Type
      ----------------------------
      ROWTIME      | BIGINT
      ROWKEY       | VARCHAR(STRING)
      REGISTERTIME | BIGINT
      GENDER       | VARCHAR(STRING)
      REGIONID     | VARCHAR(STRING)
      USERID       | VARCHAR(STRING)
```

Note that in the above, each schema always contains two built-in fields, `ROWTIME` and `ROWKEY`. They correspond to the timestamp and the key of the record, respectively. Finally, let's run some KSQL queries using the data and the schema that we prepared.

1. Select a field from a stream:

```
ksql> SELECT pageid
      FROM pageviews_stream
      LIMIT 3;

Page_24
Page_73
Page_78
```

2. Join a stream and a table:

```
ksql> CREATE STREAM pageviews_female
        AS SELECT
        users_table.userid AS userid,
           pageid, regionid, gender
           FROM pageviews_stream
        LEFT JOIN users_table ON
           pageviews_stream.userid =
            users_table.userid
        WHERE gender = 'FEMALE';

 ksql> SELECT userid, pageid,
            regionid, gender
        FROM pageviews_female;


User_2 | Page_55 | Region_9 | FEMALE
User_5 | Page_14 | Region_2 | FEMALE
User_3 | Page_60 | Region_3 | FEMALE
```

Note that in step 2 above, the query `pageviews_female` runs continuously in the background once it's issued. The results from this query are written to a Kafka topic named `pageviews_female`.

More examples of KSQL can be found at github.com/confluentinc/ksql.

## ADDITIONAL RESOURCES

- Documentation of Apache Kafka
  kafka.apache.org/documentation
- Kafka connectors
  confluent.io/product/connectors
- KSQL
  confluent.io/product/ksql
- Apache Kafka Summit
  kafka-summit.org

## ABOUT THE AUTHOR

**JUN RAO** is a co-creator of Apache Kafka and a co-founder of Confluent, a company that offers an enterprise stream data platform on top of Apache Kafka. Before Confluent, Jun Rao was a senior staff engineer at LinkedIn where he led the development of Kafka. Before LinkedIn, Jun Rao was a researcher at IBM's Almaden research data center, where he conducted research on databases and distributed systems. Jun Rao is the PMC chair of Apache Kafka and a committer of Apache Cassandra. He is the co-author of more than 20 referenced research papers, and the co-inventor of more than a dozen U.S. software patents.



**Refcardz Feedback Welcome:** refcardz@dzone.com

**Sponsorship Opportunities:** sales@dzone.com

DZone, Inc. 150 Preston Executive Dr. Cary, NC 27513

888.678.0399 - 919.678.0300