Write streaming queries against Apache Kafka using ksqlDB

This tutorial demonstrates a simple workflow using ksqlDB to write streaming queries against messages in Kafka.

Create Topics and Produce Data

Create and produce data to the Kafka topics `pageviews` and `users`. These steps use the ksqlDB datagen tool that's included with Confluent Platform.

1. Open a new terminal window and run the folowing command to create the `pageviews` topic and produce data using the data generator. The following example continuously generates data in DELIMITED format.

```
2. $CONFLUENT_HOME/bin/ksql-datagen quickstart=pageviews format=delimited topic=pa
   geviews msgRate=5
```

3. Open another terminal window and run the following command to produce Kafka data to the `users` topic using the data generator. The following example continuously generates data with in DELIMITED format.

```
4. $CONFLUENT_HOME/bin/ksql-datagen quickstart=users format=delimited topic=users
   msgRate=1
```

You can also produce Kafka data using the `kafka-console-producer` CLI provided with Confluent Platform.

Launch the ksqlDB CLI

Open a new terminal window and run the following command to set the `LOG_DIR` environment variable and launch the ksqlDB CLI.

```
LOG_DIR=./ksql_logs $CONFLUENT_HOME/bin/ksql
```

This command routes the CLI logs to the `./ksql_logs` directory, relative to your current directory. By default, the CLI looks for a ksqlDB Server running at `http://localhost:8088`.

By default ksqlDB attempts to store its logs in a directory called `logs` that is relative to the location of the `ksql` executable. For example, if `ksql` is installed at `/usr/local/bin/ksql`, then it would attempt to store its logs in `/usr/local/logs`. If you are running `ksql` from the default Confluent Platform location, `$CONFLUENT_HOME/bin`, you must override this default behavior by using the `LOG_DIR` variable.

After ksqlDB is started, your terminal should resemble this.

```
                  ===============================================
                  =                  _              _ ____  ____       =
                  =                 | | _____  __ _| |  _ \| __ )      =
                  =                 | |/ / __|/ _` | | | | |  _ \      =
                  =                 |   <\__ \ (_| | | |_| | |_) |     =
                  =                 |_|\_\___/\__, |_|____/|____/      =
                  =                            |_|                    =
                  =  Event Streaming Database purpose-built           =
                  =        for stream processing apps                 =
                  ===============================================

Copyright 2017-2020 Confluent Inc.

CLI v6.0.1, Server v6.0.1 located at http://localhost:8088

Having trouble? Type 'help' (case-insensitive) for a rundown of how things work!

ksql>
```

Inspect Kafka Topics By Using SHOW and PRINT Statements

ksqlDB enables inspecting Kafka topics and messages in real time.

- Use the SHOW TOPICS statement to list the available topics in the Kafka cluster.
- Use the PRINT statement to see a topic's messages as they arrive.

In the ksqlDB CLI, run the following statement:

```
SHOW TOPICS;
```

Your output should resemble:

```
 Kafka Topic                 | Partitions | Partition Replicas
------------------------------------------------------------------
 default_ksql_processing_log | 1          | 1
```

```
 pageviews                    | 1          | 1
 users                        | 1          | 1
---------------------------------------------------------------
```

By default, ksqlDB hides internal and system topics. Use the SHOW ALL TOPICS statement to see the full list of topics in the Kafka cluster:

```
SHOW ALL TOPICS;
```

Your output should resemble:

```
 Kafka Topic                             | Partitions | Partition Replicas
-------------------------------------------------------------------------
 __confluent.support.metrics             | 1          | 1
 _confluent-ksql-default__command_topic  | 1          | 1
 _confluent-license                      | 1          | 1
 _confluent-metrics                      | 12         | 1
 default_ksql_processing_log             | 1          | 1
 pageviews                               | 1          | 1
 users                                   | 1          | 1
-------------------------------------------------------------------------
```

Inspect the `users` topic by using the PRINT statement:

```
PRINT users;
```

The PRINT statement is one of the few case-sensitive commands in ksqlDB, even when the topic name is not quoted.

Your output should resemble:

```
Key format: KAFKA_STRING
Value format: AVRO
rowtime: 10/30/18 10:15:51 PM GMT, key: User_1, value: {"registertime":1516754966866,"userid":"User_1","regionid":"Region_9","gender":"MALE"}
rowtime: 10/30/18 10:15:51 PM GMT, key: User_3, value: {"registertime":1491558386780,"userid":"User_3","regionid":"Region_2","gender":"MALE"}
rowtime: 10/30/18 10:15:53 PM GMT, key: User_7, value: {"registertime":1514374073235,"userid":"User_7","regionid":"Region_2","gender":"OTHER"}
^Crowtime: 10/30/18 10:15:59 PM GMT, key: User_4, value: {"registertime":1510034151376,"userid":"User_4","regionid":"Region_8","gender":"FEMALE"}
Topic printing ceased
```

Press CTRL+C to stop printing messages.

Inspect the `pageviews` topic by using the PRINT statement:

```
PRINT pageviews;
```

Your output should resemble:

```
Key format: KAFKA_INTEGER
```

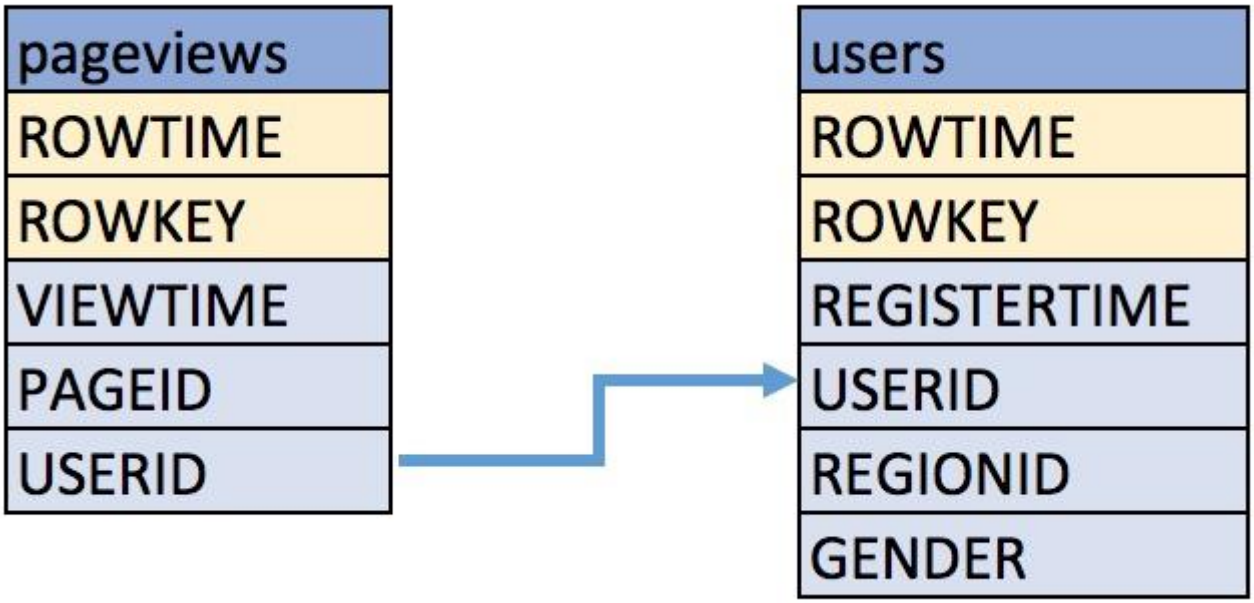
```
Format: KAFKA_STRING
rowtime: 10/23/18 12:24:03 AM PSD, key: 1540254243183, value: 1540254243183,User_9
,Page_20
rowtime: 10/23/18 12:24:03 AM PSD, key: 1540254243617, value: 1540254243617,User_7
,Page_47
rowtime: 10/23/18 12:24:03 AM PSD, key: 1540254243888, value: 1540254243888,User_4
,Page_27
^Crowtime: 10/23/18 12:24:05 AM PSD, key: 1540254245161, value: 1540254245161,User
_9,Page_62
Topic printing ceased
```

Press CTRL+C to stop printing messages.

For more information, see ksqlDB Syntax Reference.

Create a Stream and Table

These examples query messages from Kafka topics called `pageviews` and `users` using the following schemas:



1. Create a stream, named `pageviews_original`, from the `pageviews` Kafka topic, specifying the `value_format` of `DELIMITED`.

```
2. CREATE STREAM pageviews_original (rowkey bigint key, userid varchar, pageid var
   char)
3. WITH (kafka_topic='pageviews', value_format='DELIMITED');
```

   Your output should resemble:

```
 Message
---------------
 Stream created
```

```
--------------
```

You can run `DESCRIBE pageviews_original;` to see the schema for the stream. Notice that ksqlDB created an additional column, named `ROWTIME`, which corresponds with the Kafka message timestamp.

4. Create a table, named `users_original`, from the `users` Kafka topic, specifying the `value_format` of `AVRO`.

```
5. CREATE TABLE users_original WITH (kafka_topic='users', value_format='AVRO', key
   = 'userid');
```

Your output should resemble:

```
 Message
--------------
 Table created
--------------
```

You can run `DESCRIBE users_original;` to see the schema for the table.

You may have noticed the CREATE TABLE did not define the set of columns like the CREATE STREAM statement did. This is because the value format is Avro, and the DataGen tool publishes the Avro schema to Schema Registry. ksqlDB retrieves the schema from Schema Registry and uses this to build the SQL schema for the table. You may still provide the schema if you wish. Until Github issue #4462 is complete, schema inference is available only where the key of the data is a STRING, as is the case here. The data generated has the same value in the Kafka record's key as the `userId` field in the value. Specifying `key='userId'` in the WITH clause above lets ksqlDB know this. ksqlDB uses this information to allow joins against the table to use the more descriptive `userId` column name, rather than `ROWKEY`. Joining on either yields the same results. If your data doesn't contain a copy of the key in the value, you can join on `ROWKEY`.

6. Optional: Show all streams and tables.

```
7. ksql> SHOW STREAMS;
8.
9.  Stream Name         | Kafka Topic                 | Format
10.-----------------------------------------------------------------
11. KSQL_PROCESSING_LOG | default_ksql_processing_log | JSON
12. PAGEVIEWS_ORIGINAL  | pageviews                   | DELIMITED
13.-----------------------------------------------------------------
14.
15.ksql> SHOW TABLES;
16.
17. Table Name     | Kafka Topic | Format | Windowed
18.--------------------------------------------------
19. USERS_ORIGINAL | users       | AVRO   | false
20.--------------------------------------------------
```

Notice the `KSQL_PROCESSING_LOG` stream listed in the SHOW STREAMS output? ksqlDB appends messages that describe any issues it encountered while processing your data. If

things aren't working as you expect, check the contents of this stream to see if ksqlDB is encountering data errors.

View your data

1. Use SELECT to create a query that returns data from a TABLE. This query includes the LIMIT keyword to limit the number of rows returned in the query result, and the EMIT CHANGES keywords to indicate we wish to stream results back. This is known as a pull query. for an explanation of the different query types, see Queries. Note that exact data output may vary because of the randomness of the data generation.

```
2. SELECT * from users_original emit changes limit 5;
```

Your output should resemble:

```
+--------------------+-------------+-------------+---------+----------+------
-------+
|ROWTIME             |ROWKEY       |REGISTERTIME  |GENDER   |REGIONID  |USERID
|
+--------------------+-------------+-------------+---------+----------+------
-------+
|1581077558655       |User_9       |1513529638461 |OTHER    |Region_1  |User_9
|
|1581077561454       |User_7       |1489408314958 |OTHER    |Region_2  |User_7
|
|1581077561654       |User_3       |1511291005264 |MALE     |Region_2  |User_3
|
|1581077561857       |User_4       |1496797956753 |OTHER    |Region_1  |User_4
|
|1581077562858       |User_8       |1489169082491 |FEMALE   |Region_8  |User_8
|
Limit Reached
Query terminated
```

Push queries on tables output the full history of the table that is stored in the Kafka changelog topic, which mean that it outputs historic data, followed by the stream of updates to the table. It is therefore likely that rows with matching `ROWKEY` are output as existing rows in the table are updated.

3. View the data in your pageviews_original stream by issuing the following push query:

```
4. SELECT viewtime, userid, pageid FROM pageviews_original emit changes LIMIT 3;
```

Your output should resemble:

```
+--------------+--------------+--------------+
|VIEWTIME      |USERID        |PAGEID        |
+--------------+--------------+--------------+
|1581078296791 |User_1        |Page_54       |
|1581078297792 |User_8        |Page_93       |
```

```
|1581078298792 |User_6         |Page_26        |
Limit Reached
Query terminated
```

By default, push queries on streams output only changes that occur after the query is started, which means that historic data is not included.
Run `set 'auto.offset.reset'='earliest';` to update your session properties if you want to see the historic data.

Write Queries

These examples write queries using ksqlDB.

Note: By default ksqlDB reads the topics for streams and tables from the latest offset.

1. Create a query that enriches the `pageviews` data with the user's `gender` and `regionid` from the `users` table. The following query enriches the `pageviews_original` stream by doing a LEFT JOIN with the `users_original` table on the `userid` column.

```
2. SELECT users_original.userid AS userid, pageid, regionid, gender
3.  FROM pageviews_original
4.  LEFT JOIN users_original
5.    ON pageviews_original.userid = users_original.userid
6.  EMIT CHANGES
7.  LIMIT 5;
```

Your output should resemble:

```
+------------------+------------------+------------------+-----------------
-+
|USERID            |PAGEID            |REGIONID          |GENDER
|
+------------------+------------------+------------------+-----------------
-+
|User_7            |Page_23           |Region_2          |OTHER
|
|User_3            |Page_42           |Region_2          |MALE
|
|User_7            |Page_87           |Region_2          |OTHER
|
|User_2            |Page_57           |Region_5          |FEMALE
|
|User_9            |Page_59           |Region_1          |OTHER
|
Limit Reached
Query terminated
```

The join to the `users` table is on the `userid` column, which was identified as an alias for the table's primary key, `ROWKEY`, in the CREATE TABLE statement. `userId` and `ROWKEY` can

be used interchangeably as the join criteria for the table. But the data in `userid` on the stream side doesn't match the stream's key, so ksqlDB internally repartitions the stream by the `userId` column before the join.

8. Create a persistent query by using the `CREATE STREAM` keywords to precede the `SELECT` statement. The results from this query are written to the `PAGEVIEWS_ENRICHED` Kafka topic. The following query enriches the `pageviews_original` STREAM by doing a `LEFT JOIN` with the `users_original` TABLE on the user ID.

```
9. CREATE STREAM pageviews_enriched AS
10.SELECT users_original.userid AS userid, pageid, regionid, gender
11.FROM pageviews_original
12.LEFT JOIN users_original
13.  ON pageviews_original.userid = users_original.userid
14.EMIT CHANGES;
```

Your output should resemble:

```
 Message
----------------------------------------------------------------------------
---------------------------
 Stream PAGEVIEWS_ENRICHED created and running. Created by query with query ID:
CSAS_PAGEVIEWS_ENRICHED_0
----------------------------------------------------------------------------
---------------------------
```

You can run `DESCRIBE pageviews_enriched;` to describe the stream.

15. Use `SELECT` to view query results as they come in. To stop viewing the query results, press Ctrl+C. This stops printing to the console but it does not terminate the actual query. The query continues to run in the underlying ksqlDB application.

```
16.SELECT * FROM pageviews_enriched emit changes;
```

Your output should resemble:

```
    +-------------+------------+------------+------------+------------+--------
----+
    |ROWTIME      |ROWKEY      |USERID      |PAGEID      |REGIONID    |GENDER
|
    +-------------+------------+------------+------------+------------+--------
----+
    |1581079706741|User_5      |User_5      |Page_53     |Region_3    |FEMALE
|
    |1581079707742|User_2      |User_2      |Page_86     |Region_5    |OTHER
|
    |1581079708745|User_9      |User_9      |Page_75     |Region_1    |OTHER
|

    ^CQuery terminated

Use Ctrl+C to terminate the query.
```

17. Create a new persistent query where a condition limits the streams content, using `WHERE`. Results from this query are written to a Kafka topic called `PAGEVIEWS_FEMALE`.

```
CREATE STREAM pageviews_female AS
  SELECT * FROM pageviews_enriched
  WHERE gender = 'FEMALE'
  EMIT CHANGES;
```

Your output should resemble:

```
 Message
----------------------------------------------------------------------------
------------------------
 Stream PAGEVIEWS_FEMALE created and running. Created by query with query ID: C
SAS_PAGEVIEWS_FEMALE_11
----------------------------------------------------------------------------
------------------------
```

You can run `DESCRIBE pageviews_female;` to describe the stream.

22. Create a new persistent query where another condition is met, using LIKE. Results from this query are written to the `pageviews_enriched_r8_r9` Kafka topic.

```
CREATE STREAM pageviews_female_like_89
  WITH (kafka_topic='pageviews_enriched_r8_r9') AS
  SELECT * FROM pageviews_female
  WHERE regionid LIKE '%_8' OR regionid LIKE '%_9'
  EMIT CHANGES;
```

Your output should resemble:

```
 Message
----------------------------------------------------------------------------
---------------------------------------
 Stream PAGEVIEWS_FEMALE_LIKE_89 created and running. Created by query with que
ry ID: CSAS_PAGEVIEWS_FEMALE_LIKE_89_13
----------------------------------------------------------------------------
---------------------------------------
```

28. Create a new persistent query that counts the pageviews for each region and gender combination in a tumbling window of 30 seconds when the count is greater than one. Results from this query are written to the `PAGEVIEWS_REGIONS` Kafka topic in the Avro format. ksqlDB registers the Avro schema with the configured Schema Registry when it writes the first message to the `PAGEVIEWS_REGIONS` topic.

```
CREATE TABLE pageviews_regions
  WITH (VALUE_FORMAT='avro') AS
SELECT gender, regionid , COUNT(*) AS numusers
FROM pageviews_enriched
  WINDOW TUMBLING (size 30 second)
GROUP BY gender, regionid
```

```
EMIT CHANGES;
```

Your output should resemble:

```
 Message
----------------------------------------------------------------------------
-------------------------
 Table PAGEVIEWS_REGIONS created and running. Created by query with query ID: C
TAS_PAGEVIEWS_REGIONS_15
----------------------------------------------------------------------------
-------------------------
```

You can run `DESCRIBE pageviews_regions;` to describe the table.

36. Optional: View results from the above queries by using a push query.

```
SELECT * FROM pageviews_regions EMIT CHANGES LIMIT 5;
```

Your output should resemble:

```
+---------------+----------------+---------------+---------------+-----------
---+---------------+---------------+
|ROWTIME        |ROWKEY          |WINDOWSTART    |WINDOWEND      |GENDER
|REGIONID       |NUMUSERS       |
+---------------+----------------+---------------+---------------+-----------
---+---------------+---------------+
|1581080500530  |OTHER|+|Region_9 |1581080490000  |1581080520000  |OTHER
|Region_9       |1              |
|1581080501530  |OTHER|+|Region_5 |1581080490000  |1581080520000  |OTHER
|Region_5       |2              |
|1581080510532  |MALE|+|Region_7  |1581080490000  |1581080520000  |MALE
|Region_7       |4              |
|1581080513532  |FEMALE|+|Region_1|1581080490000  |1581080520000  |FEMALE
|Region_1       |2              |
|1581080516533  |MALE|+|Region_2  |1581080490000  |1581080520000  |MALE
|Region_2       |3              |
Limit Reached
Query terminated
```

Notice the addition of the WINDOWSTART and WINDOWEND columns. These are available because `pageviews_regions` is aggregating data per 30 second *window*. ksqlDB automatically adds these system columns for windowed results.

38. Optional: View results from the previous queries by using pull query.

When a CREATE TABLE statement contains a GROUP BY clause, ksqlDB is internally building an table containing the results of the aggregation. ksqlDB supports pull queries against such aggregation results.

Unlike the push query used in the previous step, which *pushes* a stream of results to you, pull queries *pull* a result set and automatically terminate.

Pull queries do not have the EMIT CHANGES clause.

View all the windows and user counts available for a specific gender and region using a pull query:

```
SELECT * FROM pageviews_regions WHERE ROWKEY='OTHER|+|Region_9';
```

Your output should resemble:

```
+-----------------+-----------------+-----------------+-----------------+--
---------------+-----------------+-----------------+
|ROWKEY           |WINDOWSTART      |WINDOWEND        |ROWTIME          |GE
NDER           |REGIONID         |NUMUSERS         |
+-----------------+-----------------+-----------------+-----------------+--
---------------+-----------------+-----------------+
|OTHER|+|Region_9  |1581080490000    |1581080520000    |1581080500530    |OT
HER            |Region_9         |1                |
|OTHER|+|Region_9  |1581080550000    |1581080580000    |1581080576526    |OT
HER            |Region_9         |4                |
|OTHER|+|Region_9  |1581080580000    |1581080610000    |1581080606525    |OT
HER            |Region_9         |4                |
|OTHER|+|Region_9  |1581080610000    |1581080640000    |1581080622524    |OT
HER            |Region_9         |3                |
|OTHER|+|Region_9  |1581080640000    |1581080670000    |1581080667528    |OT
HER            |Region_9         |6                |
```

Pull queries on windowed tables such as pageviews_regions also supports querying a single window's result:

```
SELECT NUMUSERS FROM pageviews_regions WHERE ROWKEY='OTHER|+|Region_9' AND WIND
OWSTART=1581080550000;
```

You must change the value of `WINDOWSTART` in the previous SQL to match one of the window boundaries in your data. Otherwise, no results are returned.

Your output should resemble:

```
+----------+
|NUMUSERS  |
+----------+
|4         |
Query terminated
```

To query a range of windows:

```
SELECT WINDOWSTART, WINDOWEND, NUMUSERS FROM pageviews_regions WHERE ROWKEY='OT
HER|+|Region_9' AND 1581080550000 <= WINDOWSTART AND WINDOWSTART <= 15810806100
00;
```

You must change the value of `WINDOWSTART` in the previous SQL to match one of the window boundaries in your data. Otherwise, no results are returned.

Your output should resemble:

```
+-------------------------+-------------------------+-------------------
--------+
|WINDOWSTART              |WINDOWEND                |NUMUSERS
|
+-------------------------+-------------------------+-------------------
--------+
```

```
|1581080550000               |1581080580000               |4
|
|1581080580000               |1581080610000               |4
|
|1581080610000               |1581080640000               |3
|
Query terminated
```

39. Optional: Show all persistent queries.

```
40. SHOW QUERIES;
```

Your output should resemble:

```
Query ID                          | Status  | Sink Name                | Sink Ka
fka Topic         | Query String
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------
CTAS_PAGEVIEWS_REGIONS_15         | RUNNING | PAGEVIEWS_REGIONS        | PAGEVIE
WS_REGIONS        | CREATE TABLE PAGEVIEWS_REGIONS WITH (KAFKA_TOPIC='PAGEVIEWS
_REGIONS', PARTITIONS=1, REPLICAS=1, VALUE_FORMAT='avro') AS SELECT  PAGEVIEWS_
ENRICHED.GENDER GENDER,  PAGEVIEWS_ENRICHED.REGIONID REGIONID,  COUNT(*) NUMUSE
RSFROM PAGEVIEWS_ENRICHED PAGEVIEWS_ENRICHEDWINDOW TUMBLING ( SIZE 30 SECONDS )
GROUP BY PAGEVIEWS_ENRICHED.GENDER, PAGEVIEWS_ENRICHED.REGIONIDEMIT CHANGES;
CSAS_PAGEVIEWS_FEMALE_LIKE_89_13 | RUNNING | PAGEVIEWS_FEMALE_LIKE_89 | pagevie
ws_enriched_r8_r9 | CREATE STREAM PAGEVIEWS_FEMALE_LIKE_89 WITH (KAFKA_TOPIC='p
ageviews_enriched_r8_r9', PARTITIONS=1, REPLICAS=1) AS SELECT *FROM PAGEVIEWS_F
EMALE PAGEVIEWS_FEMALEWHERE ((PAGEVIEWS_FEMALE.REGIONID LIKE '%_8') OR (PAGEVIE
WS_FEMALE.REGIONID LIKE '%_9'))EMIT CHANGES;
CSAS_PAGEVIEWS_ENRICHED_0         | RUNNING | PAGEVIEWS_ENRICHED       | PAGEVIE
WS_ENRICHED       | CREATE STREAM PAGEVIEWS_ENRICHED WITH (KAFKA_TOPIC='PAGEVIE
WS_ENRICHED', PARTITIONS=1, REPLICAS=1) AS SELECT  USERS_ORIGINAL.USERID USERID
,  PAGEVIEWS_ORIGINAL.PAGEID PAGEID,  USERS_ORIGINAL.REGIONID REGIONID,  USERS_
ORIGINAL.GENDER GENDERFROM PAGEVIEWS_ORIGINAL PAGEVIEWS_ORIGINALLEFT OUTER JOIN
USERS_ORIGINAL USERS_ORIGINAL ON ((PAGEVIEWS_ORIGINAL.USERID = USERS_ORIGINAL.U
SERID))EMIT CHANGES;
CSAS_PAGEVIEWS_FEMALE_11          | RUNNING | PAGEVIEWS_FEMALE         | PAGEVIE
WS_FEMALE         | CREATE STREAM PAGEVIEWS_FEMALE WITH (KAFKA_TOPIC='PAGEVIEWS
_FEMALE', PARTITIONS=1, REPLICAS=1) AS SELECT *FROM PAGEVIEWS_ENRICHED PAGEVIEW
S_ENRICHEDWHERE (PAGEVIEWS_ENRICHED.GENDER = 'FEMALE')EMIT CHANGES;
```

For detailed information on a Query run: `EXPLAIN <Query ID>;`

41. Optional: Examine query run-time metrics and details. Observe that information including the target Kafka topic is available, as well as throughput figures for the messages being processed.

```
42. DESCRIBE EXTENDED PAGEVIEWS_REGIONS;
```

Your output should resemble:

```
Name                 : PAGEVIEWS_REGIONS
Type                 : TABLE
Key field            :
Timestamp field      : Not set - using <ROWTIME>
Key format           : KAFKA
```

```
Value format         : AVRO
Kafka topic          : PAGEVIEWS_REGIONS (partitions: 1, replication: 1)
Statement            : CREATE TABLE PAGEVIEWS_REGIONS WITH (KAFKA_TOPIC='PAGEVI
EWS_REGIONS', PARTITIONS=1, REPLICAS=1, VALUE_FORMAT='json') AS SELECT
  PAGEVIEWS_ENRICHED.GENDER GENDER,
  PAGEVIEWS_ENRICHED.REGIONID REGIONID,
  COUNT(*) NUMUSERS
FROM PAGEVIEWS_ENRICHED PAGEVIEWS_ENRICHED
WINDOW TUMBLING ( SIZE 30 SECONDS )
GROUP BY PAGEVIEWS_ENRICHED.GENDER, PAGEVIEWS_ENRICHED.REGIONID
EMIT CHANGES;

 Field    | Type
-----------------------------------------------------------------
 ROWTIME  | BIGINT           (system)
 ROWKEY   | VARCHAR(STRING)  (system) (Window type: TUMBLING)
 GENDER   | VARCHAR(STRING)
 REGIONID | VARCHAR(STRING)
 NUMUSERS | BIGINT
-----------------------------------------------------------------

Queries that write from this TABLE
-----------------------------------
CTAS_PAGEVIEWS_REGIONS_15 (RUNNING) : CREATE TABLE PAGEVIEWS_REGIONS WITH (KAFK
A_TOPIC='PAGEVIEWS_REGIONS', PARTITIONS=1, REPLICAS=1, VALUE_FORMAT='json') AS
SELECT  PAGEVIEWS_ENRICHED.GENDER GENDER,  PAGEVIEWS_ENRICHED.REGIONID REGIONID
,  COUNT(*) NUMUSERSFROM PAGEVIEWS_ENRICHED PAGEVIEWS_ENRICHEDWINDOW TUMBLING (
SIZE 30 SECONDS ) GROUP BY PAGEVIEWS_ENRICHED.GENDER, PAGEVIEWS_ENRICHED.REGION
IDEMIT CHANGES;

For query topology and execution plan please run: EXPLAIN <QueryId>

Local runtime statistics
------------------------
messages-per-sec:      0.90   total-messages:       498     last-message: 2020-
02-07T13:10:32.033Z
```

Using Nested Schemas (STRUCT) in ksqlDB

Struct support enables the modeling and access of nested data in Kafka topics, from both JSON and Avro.

Here we'll use the `ksql-datagen` tool to create some sample data which includes a nested `address` field. Run this in a new window, and leave it running.

```
<path-to-confluent>/bin/ksql-datagen  \
     quickstart=orders \
     format=avro \
     topic=orders
```

From the ksqlDB command prompt, register the topic in ksqlDB:

```
CREATE STREAM ORDERS
```

```
  (
    ROWKEY INT KEY,
    ORDERTIME BIGINT,
    ORDERID INT,
    ITEMID STRING,
    ORDERUNITS DOUBLE,
    ADDRESS STRUCT<CITY STRING, STATE STRING, ZIPCODE BIGINT>
  )
   WITH (KAFKA_TOPIC='orders', VALUE_FORMAT='json', key='orderid');
```

Your output should resemble:

```
 Message
----------------
 Stream created
----------------
```

Use the `DESCRIBE` function to observe the schema, which includes a `STRUCT`:

```
DESCRIBE ORDERS;
```

Your output should resemble:

```
Name                 : ORDERS
 Field      | Type
-----------------------------------------------------------------------------------------
 ROWTIME    | BIGINT           (system)
 ROWKEY     | INT              (system)
 ORDERTIME  | BIGINT
 ORDERID    | INTEGER
 ITEMID     | VARCHAR(STRING)
 ORDERUNITS | DOUBLE
 ADDRESS    | STRUCT<CITY VARCHAR(STRING), STATE VARCHAR(STRING), ZIPCODE BIGINT>
-----------------------------------------------------------------------------------------
For runtime statistics and query details run: DESCRIBE EXTENDED <Stream,Table>;
ksql>
```

Query the data, using `->` notation to access the Struct contents:

```
SELECT ORDERID, ADDRESS->CITY FROM ORDERS EMIT CHANGES LIMIT 5;
```

Your output should resemble:

```
+----------------------------------+----------------------------------+
|ORDERID                           |ADDRESS__CITY                     |
+----------------------------------+----------------------------------+
|1188                              |City_95                           |
|1189                              |City_24                           |
|1190                              |City_57                           |
|1191                              |City_37                           |
|1192                              |City_82                           |
Limit Reached
```

```
Query terminated
```

Stream-Stream join

Using a stream-stream join, it is possible to join two *streams* of events on a common key. An example of this could be a stream of order events, and a stream of shipment events. By joining these on the order key, it is possible to see shipment information alongside the order.

In the ksqlDB CLI create two new streams. Both streams store their order id in ROWKEY:

```
<path-to-confluent>/bin/kafka-console-producer \
      --broker-list localhost:9092 \
      --topic new_orders \
      --property "parse.key=true" \
      --property "key.separator=:"<<EOF
1:{"order_id":1,"total_amount":10.50,"customer_name":"Bob Smith"}
2:{"order_id":2,"total_amount":3.32,"customer_name":"Sarah Black"}
3:{"order_id":3,"total_amount":21.00,"customer_name":"Emma Turner"}
EOF

<path-to-confluent>/bin/kafka-console-producer \
      --broker-list localhost:9092 \
      --topic shipments \
      --property "parse.key=true" \
      --property "key.separator=:"<<EOF
1:{"order_id":1,"shipment_id":42,"warehouse":"Nashville"}
3:{"order_id":3,"shipment_id":43,"warehouse":"Palo Alto"}
EOF
```

Note that you may see the following warning message when running the above statements—it can be safely ignored:

```
Error while fetching metadata with correlation id 1 : {new_orders=LEADER_NOT_AVAIL
ABLE} (org.apache.kafka.clients.NetworkClient)
Error while fetching metadata with correlation id 1 : {shipments=LEADER_NOT_AVAILA
BLE} (org.apache.kafka.clients.NetworkClient)
CREATE STREAM NEW_ORDERS (ROWKEY INT KEY, TOTAL_AMOUNT DOUBLE, CUSTOMER_NAME VARCH
AR)
WITH (KAFKA_TOPIC='new_orders', VALUE_FORMAT='JSON', PARTITIONS=2);

CREATE STREAM SHIPMENTS (ROWKEY INT KEY, SHIPMENT_ID INT, WAREHOUSE VARCHAR)
WITH (KAFKA_TOPIC='shipments', VALUE_FORMAT='JSON', PARTITIONS=2);
```

ksqlDB creates the underlying topics in Kafka when these statements are executed. Also, you can specify the `REPLICAS` count.

After both `CREATE STREAM` statements, your output should resemble:

```
 Message
```

```
---------------
 Stream created
---------------
```

Populate the streams with some sample data using the INSERT VALUES statement:

```
-- Insert values in NEW_ORDERS:
-- insert supplying the list of columns to insert:
INSERT INTO NEW_ORDERS (ROWKEY, CUSTOMER_NAME, TOTAL_AMOUNT)
VALUES (1, 'Bob Smith', 10.50);

-- shorthand syntax can be used when inserting values for all columns (except ROWT
IME), in column order:
INSERT INTO NEW_ORDERS  VALUES (2, 3.32, 'Sarah Black');
INSERT INTO NEW_ORDERS  VALUES (3, 21.00, 'Emma Turner');

-- Insert values in SHIPMENTS:
INSERT INTO SHIPMENTS VALUES (1, 42, 'Nashville');
INSERT INTO SHIPMENTS VALUES (3, 43, 'Palo Alto');
```

Query the data to confirm that it's present in the topics.

Run the following to tell ksqlDB to read from the `beginning` of each stream:

```
SET 'auto.offset.reset' = 'earliest';
```

You can skip this if you have already run it within your current ksqlDB CLI session.`

For the `NEW_ORDERS` topic, run:

```
SELECT * FROM NEW_ORDERS EMIT CHANGES LIMIT 3;
```

Your output should resemble:

```
+------------------------+-----------------------+------------------------+---
----------------------+
|ROWTIME                 |ROWKEY                 |TOTAL_AMOUNT            |CUS
TOMER_NAME           |
+------------------------+-----------------------+------------------------+---
----------------------+
|1581083057609           |1                      |10.5                    |Bob
Smith                |
|1581083178418           |2                      |3.32                    |Sar
ah Black             |
|1581083210494           |3                      |21.0                    |Emm
a Turner             |
Limit Reached
Query terminated
```

For the `SHIPMENTS` topic, run:

```
SELECT * FROM SHIPMENTS EMIT CHANGES LIMIT 2;
```

Your output should resemble:

```
+------------------------+------------------------+------------------------+---
---------------------+
|ROWTIME                 |ROWKEY                  |SHIPMENT_ID             |WAR
EHOUSE               |
+------------------------+------------------------+------------------------+---
---------------------+
|1581083340711           |1                       |42                      |Nas
hville               |
|1581083384229           |3                       |43                      |Pal
o Alto               |
Limit Reached
Query terminated
```

Run the following query, which will show orders with associated shipments, based on a join window of 1 hour.

```
SELECT O.ROWKEY AS ORDER_ID, O.TOTAL_AMOUNT, O.CUSTOMER_NAME, S.SHIPMENT_ID, S.WAR
EHOUSE
FROM NEW_ORDERS O
INNER JOIN SHIPMENTS S
  WITHIN 1 HOURS
  ON O.ROWKEY = S.ROWKEY
EMIT CHANGES;
```

Your output should resemble:

```
+------------------------+------------------------+------------------------+
------------------------+------------------------+
|ORDER_ID                |TOTAL_AMOUNT            |CUSTOMER_NAME           |
SHIPMENT_ID            |WAREHOUSE               |
+------------------------+------------------------+------------------------+
------------------------+------------------------+
|1                       |10.5                    |Bob Smith               |
42                     |Nashville               |
|3                       |21.0                    |Emma Turner             |
43                     |Palo Alto               |
```

Note that message with `ORDER_ID=2` has no corresponding `SHIPMENT_ID` or `WAREHOUSE`. This is because there is no corresponding row on the `shipments` stream within the specified time window.

Start the ksqlDB CLI in a second window by running:

```
LOG_DIR=./ksql_logs $CONFLUENT_HOME/bin/ksql
```

Enter the following INSERT VALUES statement to insert the shipment for order id 2:

Switch to your first ksqlDB CLI window. A third row has now been output:

```
+-------------------------+-------------------------+-------------------------+
-------------------------+-------------------------+
|ORDER_ID                 |TOTAL_AMOUNT             |CUSTOMER_NAME            |
SHIPMENT_ID               |WAREHOUSE                |
+-------------------------+-------------------------+-------------------------+
-------------------------+-------------------------+
|1                        |10.5                     |Bob Smith                |
42                        |Nashville                |
|3                        |21.0                     |Emma Turner              |
43                        |Palo Alto                |
|2                        |3.32                     |Sarah Black              |
49                        |London                   |
```

Press Ctrl+C to cancel the `SELECT` query and return to the ksqlDB prompt.

Table-Table join

Using a table-table join, it is possible to join two *tables* of on a common key. ksqlDB tables provide the latest *value* for a given *key*. They can only be joined on the *key*, and one-to-many (1:N) joins are not supported in the current semantic model.

In this example we have location data about a warehouse from one system, being enriched with data about the size of the warehouse from another.

In the ksqlDB CLI, register both topics as ksqlDB tables. Note, in this example the warehouse id is stored both in the key and in the WAREHOUSE_ID field in the value:

```
<path-to-confluent>/bin/kafka-console-producer \
      --broker-list localhost:9092 \
      --topic warehouse_location \
      --property "parse.key=true" \
      --property "key.separator=:"<<EOF
```

Your output should resemble:

```
1:{"warehouse_id":1,"city":"Leeds","country":"UK"}
2:{"warehouse_id":2,"city":"Sheffield","country":"UK"}
3:{"warehouse_id":3,"city":"Berlin","country":"Germany"}
EOF
<path-to-confluent>/bin/kafka-console-producer \
      --broker-list localhost:9092 \
      --topic warehouse_size \
      --property "parse.key=true" \
      --property "key.separator=:"<<EOF
```

Your output should resemble:

```
1:{"warehouse_id":1,"square_footage":16000}
2:{"warehouse_id":2,"square_footage":42000}
3:{"warehouse_id":3,"square_footage":94000}
EOF
```

```
CREATE TABLE WAREHOUSE_LOCATION
   (ROWKEY INT KEY, WAREHOUSE_ID INT, CITY VARCHAR, COUNTRY VARCHAR)
   WITH (KAFKA_TOPIC='warehouse_location',
      VALUE_FORMAT='JSON',
      KEY='WAREHOUSE_ID',
      PARTITIONS=2);

CREATE TABLE WAREHOUSE_SIZE
   (ROWKEY INT KEY, WAREHOUSE_ID INT, SQUARE_FOOTAGE DOUBLE)
   WITH (KAFKA_TOPIC='warehouse_size',
      VALUE_FORMAT='JSON',
      KEY='WAREHOUSE_ID',
      PARTITIONS=2);
```

After both `CREATE TABLE` statements, your output should resemble:

```
 Message
---------------
 Table created
---------------
-- note: ksqlDB will automatically populate ROWKEY with the same value as WAREHOUS
E_ID:
INSERT INTO WAREHOUSE_LOCATION (WAREHOUSE_ID, CITY, COUNTRY) VALUES (1, 'Leeds', '
UK');
INSERT INTO WAREHOUSE_LOCATION (WAREHOUSE_ID, CITY, COUNTRY) VALUES (2, 'Sheffield
', 'UK');
INSERT INTO WAREHOUSE_LOCATION (WAREHOUSE_ID, CITY, COUNTRY) VALUES (3, 'Berlin',
'Germany');

INSERT INTO WAREHOUSE_SIZE (WAREHOUSE_ID, SQUARE_FOOTAGE) VALUES (1, 16000);
INSERT INTO WAREHOUSE_SIZE (WAREHOUSE_ID, SQUARE_FOOTAGE) VALUES (2, 42000);
INSERT INTO WAREHOUSE_SIZE (WAREHOUSE_ID, SQUARE_FOOTAGE) VALUES (3, 94000);
```

Inspect the WAREHOUSE_LOCATION table:

```
SELECT ROWKEY, WAREHOUSE_ID FROM WAREHOUSE_LOCATION EMIT CHANGES LIMIT 3;
```

Your output should resemble:

```
+---------------------------------------+---------------------------------------+
|ROWKEY                                 |WAREHOUSE_ID                           |
+---------------------------------------+---------------------------------------+
|2                                      |2                                      |
|1                                      |1                                      |
|3                                      |3                                      |
Limit Reached
Query terminated
```

Inspect the WAREHOUSE_SIZE table:

```
SELECT ROWKEY, WAREHOUSE_ID FROM WAREHOUSE_SIZE EMIT CHANGES LIMIT 3;
```

Your output should resemble:

```
+---------------------------------------+---------------------------------------+
|ROWKEY                                 |WAREHOUSE_ID                           |
+---------------------------------------+---------------------------------------+
|2                                      |2                                      |
|1                                      |1                                      |
|3                                      |3                                      |
Limit Reached
Query terminated
```

Now join the two tables:

```
SELECT WL.WAREHOUSE_ID, WL.CITY, WL.COUNTRY, WS.SQUARE_FOOTAGE
FROM WAREHOUSE_LOCATION WL
  LEFT JOIN WAREHOUSE_SIZE WS
    ON WL.WAREHOUSE_ID=WS.WAREHOUSE_ID
EMIT CHANGES
LIMIT 3;
```

Your output should resemble:

```
+------------------+------------------+------------------+------------------+
|WL_WAREHOUSE_ID   |CITY              |COUNTRY           |SQUARE_FOOTAGE    |
+------------------+------------------+------------------+------------------+
|1                 |Leeds             |UK                |16000.0           |
|1                 |Leeds             |UK                |16000.0           |
|2                 |Sheffield         |UK                |42000.0           |
Limit Reached
Query terminated
```

INSERT INTO

The `INSERT INTO` syntax can be used to merge the contents of multiple streams. An example of this could be where the same event type is coming from different sources.

Run two datagen processes, each writing to a different topic, simulating order data arriving from a local installation vs from a third-party:

Each of these commands should be run in a separate window. When the exercise is finished, exit them by pressing Ctrl-C.

```
<path-to-confluent>/bin/ksql-datagen \
     quickstart=orders \
     format=json \
     topic=orders_local \
     msgRate=2

<path-to-confluent>/bin/ksql-datagen \
     quickstart=orders \
     format=json \
     topic=orders_3rdparty \
```

```
     msgRate=2
```

In the ksqlDB CLI, register the source topic for each:

```
CREATE STREAM ORDERS_SRC_LOCAL
 (
   ROWKEY INT KEY,
   ORDERTIME BIGINT,
   ORDERID INT,
   ITEMID STRING,
   ORDERUNITS DOUBLE,
   ADDRESS STRUCT<CITY STRING, STATE STRING, ZIPCODE BIGINT>
 )
  WITH (KAFKA_TOPIC='orders_local', VALUE_FORMAT='JSON');

CREATE STREAM ORDERS_SRC_3RDPARTY
 (
   ROWKEY INT KEY,
   ORDERTIME BIGINT,
   ORDERID INT,
   ITEMID STRING,
   ORDERUNITS DOUBLE,
   ADDRESS STRUCT<CITY STRING, STATE STRING, ZIPCODE BIGINT>
 )
  WITH (KAFKA_TOPIC='orders_3rdparty', VALUE_FORMAT='JSON');
```

After each `CREATE STREAM` statement you should get the message:

```
 Message
----------------
 Stream created
----------------
```

Create the output stream, using the standard `CREATE STREAM … AS` syntax. Because multiple sources of data are being joined into a common target, it is useful to add in lineage information. This can be done by simply including it as part of the `SELECT`:

```
CREATE STREAM ALL_ORDERS AS SELECT 'LOCAL' AS SRC, * FROM ORDERS_SRC_LOCAL EMIT CH
ANGES;
```

Your output should resemble:

```
 Message
----------------------------------------------------------------------------
---------
 Stream ALL_ORDERS created and running. Created by query with query ID: CSAS_ALL_O
RDERS_17
----------------------------------------------------------------------------
---------
```

Use the `DESCRIBE` command to observe the schema of the target stream.

```
DESCRIBE ALL_ORDERS;
```

Your output should resemble:

```
Name                 : ALL_ORDERS
 Field      | Type
----------------------------------------------------------------------------------------
 ROWTIME    | BIGINT           (system)
 ROWKEY     | INTEGER          (system)
 SRC        | VARCHAR(STRING)
 ORDERTIME  | BIGINT
 ORDERID    | INTEGER
 ITEMID     | VARCHAR(STRING)
 ORDERUNITS | DOUBLE
 ADDRESS    | STRUCT<CITY VARCHAR(STRING), STATE VARCHAR(STRING), ZIPCODE BIGINT>
----------------------------------------------------------------------------------------
For runtime statistics and query details run: DESCRIBE EXTENDED <Stream,Table>;
```

Add stream of 3rd party orders into the existing output stream:

```
INSERT INTO ALL_ORDERS SELECT '3RD PARTY' AS SRC, * FROM ORDERS_SRC_3RDPARTY EMIT
CHANGES;
```

Your output should resemble:

```
 Message
------------------------------------------------------------
 Insert Into query is running with query ID: INSERTQUERY_43
------------------------------------------------------------
```

Query the output stream to verify that data from each source is being written to it:

```
SELECT * FROM ALL_ORDERS EMIT CHANGES;
```

Your output should resemble the following. Note that there are messages from both source topics (denoted by `LOCAL` and `3RD PARTY` respectively).

```
+--------------+----------+-----------+--------------+----------+-------------+---
------------------+-----------------------------------------------+
|ROWTIME       |ROWKEY    |SRC        |ORDERTIME     |ORDERID   |ITEMID       |ORD
ERUNITS            |ADDRESS                                        |
+--------------+----------+-----------+--------------+----------+-------------+---
------------------+-----------------------------------------------+
|1581085344272 |510       |3RD PARTY  |1503198352036 |510       |Item_643     |1.6
53210222047296     |{CITY=City_94, STATE=State_72, ZIPCODE=61274}|
|1581085344293 |546       |LOCAL      |1498476865306 |546       |Item_234     |9.2
84691223615178     |{CITY=City_44, STATE=State_29, ZIPCODE=84678}|
|1581085344776 |511       |3RD PARTY  |1489945722538 |511       |Item_264     |8.2
13163488516212     |{CITY=City_36, STATE=State_13, ZIPCODE=44821}|
…
```

Press Ctrl+C to cancel the `SELECT` query and return to the ksqlDB prompt.

You can view the two queries that are running using `SHOW QUERIES`:

```
SHOW QUERIES;
```

Your output should resemble:

```
 Query ID                          | Status  | Sink Name               | Sink Kafk
a Topic         | Query String
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
---------------------------------
 INSERTQUERY_43                    | RUNNING | ALL_ORDERS              | ALL_ORDER
S               | INSERT INTO ALL_ORDERS SELECT '3RD PARTY' AS SRC, * FROM ORDERS_
SRC_3RDPARTY EMIT CHANGES;
 CSAS_ALL_ORDERS_17                | RUNNING | ALL_ORDERS              | ALL_ORDER
S               | CREATE STREAM ALL_ORDERS WITH (KAFKA_TOPIC='ALL_ORDERS', PARTITI
ONS=1, REPLICAS=1) AS SELECT  'LOCAL' SRC,  *FROM ORDERS_SRC_LOCAL ORDERS_SRC_LOCA
LEMIT CHANGES;
…
```

Terminate and Exit

ksqlDB

Persisted queries will continuously run as ksqlDB applications until they are manually terminated. Exiting ksqlDB CLI does not terminate persistent queries.

1. From the output of `SHOW QUERIES;` identify a query ID you would like to terminate. For example, if you wish to terminate query ID `CTAS_PAGEVIEWS_REGIONS_15`:

```
2. TERMINATE CTAS_PAGEVIEWS_REGIONS_15;
```

   The actual name of the query running may vary; refer to the output of `SHOW QUERIES;`.

3. Run the `exit` command to leave the ksqlDB CLI.

```
4. ksql> exit
5. Exiting ksqlDB.
```

Confluent CLI

If you are running Confluent Platform using the CLI, you can stop it with this command.

```
<path-to-confluent>/bin/confluent local stop
```