

1.	Prerequisite	3
2.	Install KSQL DB – 60 Minutes(D).....	4
4.	Workflow using KSQL - CLI – 90 Minutes(D)	10
5.	KSQL - Kafka Aggregation- 45minutes(D).....	32
6.	Kafka – UDAF – 60 Minutes	48
7.	Testing and Troubleshooting – 60 Minutes	63
8.	Kafkatools	89
9.	Errors	90
I.	LEADER_NOT_AVAILABLE	90
	java.util.concurrent.ExecutionException:.....	90
10.	Annexure Code:	94
II.	DumplogSegment	94
III.	Data Generator – JSON.....	96
IV.	Configuration	106
V.	Resources	106

2 Kafka – Dev Ops

First Round Done – Need some find tuning

Date : 17th Mar 2024.

<https://docs.confluent.io/current/ksql/docs/tutorials/examples.html#ksql-examples>

3 Kafka – Dev Ops

1. Prerequisite

JDK : 1.11

Confluent : 7.3.3

Kafka : 3.11

Registry & KSQL DB

Since ksqlDB runs natively on Apache Kafka®, you'll need to have a Kafka cluster that ksqlDB is configured to use. Use the steps to the right to install the latest release of ksqlDB.

Use 0.23.1 only

```
#curl http://ksqldb-packages.s3.amazonaws.com/archive/0.29/confluent-ksqldb-0.29.0.tar.gz --output  
confluent-ksqldb-0.23.1.tar.gz
```

```
# Download the archive and its signature
```

```
#curl http://ksqldb-packages.s3.amazonaws.com/archive/0.29/confluent-ksqldb-0.29.0.tar.gz --output  
confluent-ksqldb-0.29.0.tar.gz
```

<https://ksqldb.io/quickstart-standalone-tarball.html#quickstart-content>

Configure Kafka confluent and kafka cluster

2. Install KSQL DB – 60 Minutes(D)

Prerequisite: Kafka Node installation.

And kafka registry, required for Avro integration.

Update kafka server.properties with the following entries.

```
#vi /opt/kafka/config/server.properties
```

```
transaction.state.log.replication.factor=1  
transaction.state.log.min_isr=1  
offsets.topic.replication.factor=1
```

Restart the kafka broker.

Get standalone ksqlDB

Since ksqlDB runs natively on Apache Kafka®, you'll need to have a Kafka cluster that ksqlDB is configured to use. Use the steps to the right to install the latest release of ksqlDB.

```
# Extract the tarball to the directory of your choice
```

```
#tar -xf confluent-ksqldb-0.29.0.tar -C /opt/
```

5 Kafka – Dev Ops

Rename the folder to ksqlDB for brevity

```
#mv confluent-ksq* ksqlDB
```

Configure ksqlDB server

Ensure your ksqlDB server has network connectivity to Kafka.

Edit the highlighted line in `/opt/ksqlDB/etc/ksqlDB/ksql-server.properties` to match your Kafka hostname and port.

```
#----- Kafka -----
```

```
# The set of Kafka brokers to bootstrap Kafka cluster information from:  
bootstrap.servers=kafka0:9092
```

```
# Enable snappy compression for the Kafka producers  
compression.type=snappy
```

To enable Schema Registry Add the following line at the end of the configuration file.

6 Kafka – Dev Ops

#----- Schema Registry -----

Uncomment and complete the following to enable KSQL's integration to the Confluent Schema Registry:

```
ksql.schema.registry.url=http://kafka:8081
```

Start ksqlDB's server

ksqlDB is packaged with a startup script for development use. We'll use that here.

When you're ready to run it as a service, you'll want to manage ksqlDB with something like `systemd`.

```
#/opt/ksqldb/bin/ksql-server-start /opt/ksqldb/etc/ksqldb/ksql-server.properties
```

If any issue in start up because of jar.

Download and store in the following folder.

```
#cd /opt/ksqldb/share/java/ksqldb  
#wget https://repo1.maven.org/maven2/io/netty/netty-all/4.1.30.Final/netty-all-4.1.30.Final.jar
```

7 Kafka – Dev Ops

If you are able to successfully start the KSQL DB server, you should get the following output.

Start ksqlDB's interactive CLI

ksqldb runs as a server which clients connect to in order to issue queries.

8 Kafka – Dev Ops

Run this command to connect to the ksqlDB server and enter an interactive command-line interface (CLI) session.

```
#/opt/ksqldb/bin/ksql http://0.0.0.0:8088
```

```
[root@kafka0 ksqlDB]# /opt/ksqldb/bin/ksql http://0.0.0.0:8088

=====
=   _   _   _   _   _   _   _   _   _   _   =
=   | | ____ _ - _ | | _ \| _ )   =
=   | | / \_ \| _ \ | | | | | _ \   =
=   | | <\_ \(_| | | | | | | | | | ) |   =
=   | | \_ \_ \_ \_ , | | _ / | _ /   =
=   | |   _ | |   =
=   The Database purpose-built   =
=   for stream processing apps   =
=====

Copyright 2017-2021 Confluent Inc.

CLI v0.23.1, Server v0.23.1 located at http://0.0.0.0:8088
Server Status: RUNNING

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

ksql> 
```

9 Kafka – Dev Ops

```
#show topics;
```

```
ksql> show topics;

Kafka Topic           | Partitions | Partition Replicas
-----
default_ksql_processing_log | 1          | 1
test                      | 1          | 1
topic1                   | 2          | 1
-----
```

The topics listed above depends on the number of topic in your kafka system.

-----Lab Ends Here -----

4. Workflow using KSQL - CLI – 90 Minutes(D)

Following features will be demonstrated.

- Create Topics and Produce Data
- Create and produce data to the Kafka topics pageviews and users.
- Inspect Kafka Topics by Using SHOW and PRINT Statements
- Create a Stream and Table
- Write Queries

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

To get started, you must start a Kafka cluster, including ZooKeeper and a Kafka broker.

Start Schema Registry

KSQL will then query messages from this Kafka cluster.

KSQL is installed in the Confluent Platform by default.

Confluent Kafka needs to be installed for data generation.

Create Topics and Produce Data

Create and produce data to the Kafka topics `pageviews` and `users`. These steps use the KSQL datagen that is included with Confluent Platform.

1. Create the `pageviews` topic and produce data using the data generator. The following example continuously generates data with a value in DELIMITED format.

Open a new terminal and execute the following commands:

```
kafka-topics --bootstrap-server kafka:9092 --create --topic pageviews
```

```
ksql-datagen bootstrap-server=kafka:9092 quickstart=pageviews format=json topic=pageviews maxInterval=2500
```

```
(base) [root@tos ~]#
(base) [root@tos ~]#
(base) [root@tos ~]# ksql-datagen quickstart=pageviews format=delimited topic=pageviews maxInterval=500
[2019-07-31 21:35:34,823] INFO AvroDataConfig values:
    schemas.cache.config = 1
    enhanced.avro.schema.support = false
    connect.meta.data = true
  (io.confluent.connect.avro.AvroDataConfig:179)
1 --> ([[ 1564589135082 | 'User_3' | 'Page_97' ]] ts:1564589135333
11 --> ([[ 1564589135590 | 'User_7' | 'Page_66' ]] ts:1564589135591
21 --> ([[ 1564589135857 | 'User_1' | 'Page_34' ]] ts:1564589135861
31 --> ([[ 1564589135959 | 'User_6' | 'Page_37' ]] ts:1564589135959
41 --> ([[ 1564589136036 | 'User_6' | 'Page_66' ]] ts:1564589136036
51 --> ([[ 1564589136428 | 'User_2' | 'Page_98' ]] ts:1564589136428
61 --> ([[ 1564589136761 | 'User_9' | 'Page_26' ]] ts:1564589136761
```

Keep the terminal open.

2. Open a new terminal
3. Produce Kafka data to the **users** topic using the data generator. The following example continuously generates data with a value in JSON format.

kafka-topics --bootstrap-server kafka:9092 --create --topic users

```
$ ksql-datagen bootstrap-server=kafka:9092 quickstart=users format=json topic=users
maxInterval=2500
```

Keep open the terminal.

```
(base) [root@tos ~]#  
(base) [root@tos ~]# ksql-datagen quickstart=pageviews format=delimited topic=pa  
geviews maxInterval=500  
[2019-07-31 21:35:34,823] INFO AvroDataConfig values:  
    schemas.cache.config = 1  
    enhanced.avro.schema.support = false  
    connect.meta.data = true  
(io.confluent.connect.avro.AvroDataConfig:179)  
1 --> ([[ 1564589135082 | 'User_3' | 'Page_97' ]) ts:1564589135333  
11 --> ([[ 1564589135590 | 'User_7' | 'Page_66' ]) ts:1564589135591  
21 --> ([[ 1564589135857 | 'User_1' | 'Page_34' ]) ts:1564589135861  
31 --> ([[ 1564589135959 | 'User_6' | 'Page_37' ]) ts:1564589135959  
41 --> ([[ 1564589136036 | 'User_6' | 'Page_66' ]) ts:1564589136036  
51 --> ([[ 1564589136428 | 'User_2' | 'Page_98' ]) ts:1564589136428  
61 --> ([[ 1564589136761 | 'User_9' | 'Page_26' ]) ts:1564589136761
```

Launch the KSQL CLI

To launch the CLI, run the following command. It will route the CLI logs to the `./ksql_logs` directory, relative to your current directory. By default, the CLI will look for a KSQL Server running at `http://localhost:8088`.

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

Important

By default KSQL 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, `<path-to-confluent>/bin`, you must override this default behavior by using the `LOG_DIR` variable.

After KSQL is started, your terminal should resemble this.

Inspect Kafka Topics By Using SHOW and PRINT Statements

KSQL 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 KSQL CLI, run the following statement:

SHOW TOPICS;

Your output should resemble as shown below, it depends on the context. You should be able to see the earlier two topics, you have created earlier i.e **pageviews** and **users**:

Kafka Topic	Registered	Partitions	Partition Replicas	Consumers	ConsumerGroups
_confluent-metrics	false	12	1	0	0
_schemas	false	1	1	0	0
pageviews	false	1	1	0	0
users	false	1	1	0	0

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

PRINT 'users';

Your output should resemble:

Format:JSON

```
{"ROWTIME":1540254230041,"ROWKEY":"User_1","registertime":1516754966866,"useri  
d":"User_1","regionid":"Region_9","gender":"MALE"}  
{"ROWTIME":1540254230081,"ROWKEY":"User_3","registertime":1491558386780,"useri  
d":"User_3","regionid":"Region_2","gender":"MALE"}  
{"ROWTIME":1540254230091,"ROWKEY":"User_7","registertime":1514374073235,"useri  
d":"User_7","regionid":"Region_2","gender":"OTHER"}  
^C{"ROWTIME":1540254232442,"ROWKEY":"User_4","registertime":1510034151376,"us  
erid":"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:

Format:STRING

```
10/23/18 12:24:03 AM UTC , 9461 , 1540254243183,User_9,Page_20  
10/23/18 12:24:03 AM UTC , 9471 , 1540254243617,User_7,Page_47  
10/23/18 12:24:03 AM UTC , 9481 , 1540254243888,User_4,Page_27  
^C10/23/18 12:24:05 AM UTC , 9521 , 1540254245161,User_9,Page_62
```

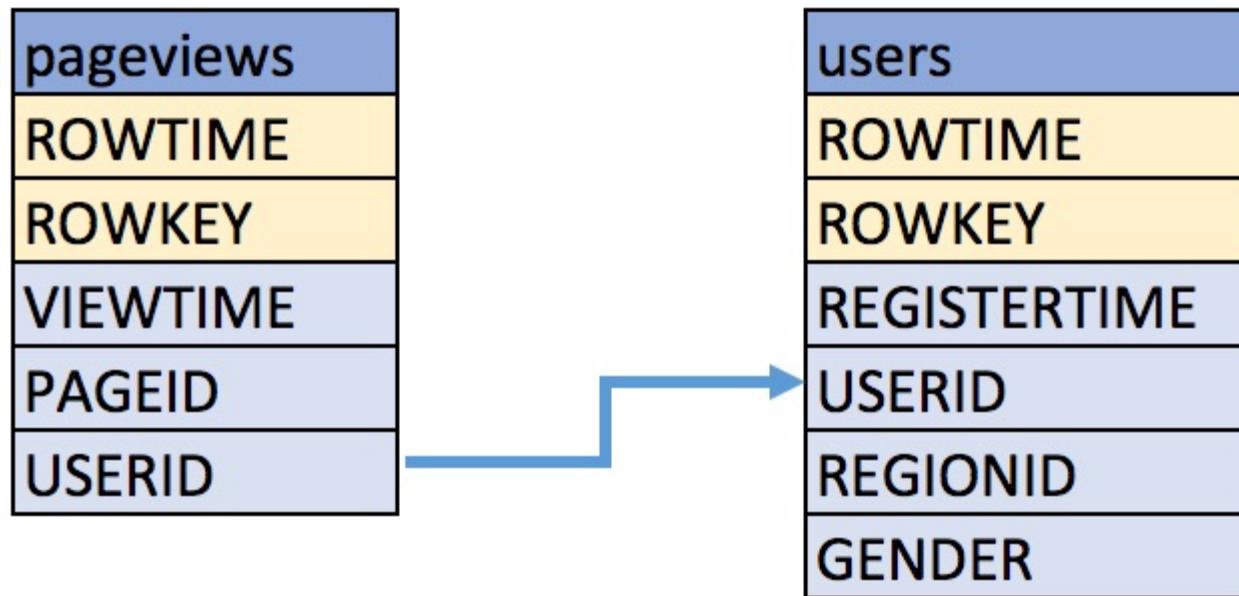
Topic printing ceased

```
ksql>
```

Press CTRL+C to stop printing messages.

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`.

```
CREATE STREAM pageviews_original (viewtime bigint, userid varchar, pageid varchar) WITH  
(kafka_topic='pageviews', value_format='JSON');
```

Your output should resemble:

```
-----  
ksql> CREATE STREAM pageviews_original (viewtime bigint, userid varchar, pageid varchar) WITH  
>(kafka_topic='pageviews', value_format='DELIMITED');  
>  
  
Message  
-----  
Stream created  
-----  
ksql> [green square]
```

Tip

You can run `DESCRIBE pageviews_original;` to see the schema for the stream. Notice that KSQL created two additional columns, named `ROWTIME`, which corresponds with the Kafka message timestamp, and `ROWKEY`, which corresponds with the Kafka message key. – Removed in the latest version.

```
DESCRIBE pageviews_original;
```

```
ksql> DESCRIBE pageviews_original;  
  
Name : PAGEVIEWS_ORIGINAL  
Field | Type  
-----  
ROWTIME | BIGINT      (system)  
ROWKEY  | VARCHAR(STRING) (system)  
VIEWTIME | BIGINT  
USERID   | VARCHAR(STRING)  
PAGEID   | VARCHAR(STRING)  
-----  
For runtime statistics and query details run: DESCRIBE EXTENDED <Stream,Table>;  
ksql>
```

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

`CREATE TABLE users_original (registertime BIGINT, gender VARCHAR, regionid VARCHAR, userid VARCHAR PRIMARY KEY) WITH
(kafka_topic='users', value_format='JSON');`

Your output should resemble:

Message

Table created

Tip

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

3. Optional: Show all streams and tables.

```
ksql> SHOW STREAMS;
```

Stream Name	Kafka Topic	Format
PAGEVIEWS_ORIGINAL	pageviews	DELIMITED

```
ksql> SHOW TABLES;
```

Table Name	Kafka Topic	Format	Windowed
USERS_ORIGINAL	users	JSON	false

Write Queries

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

These examples write queries using KSQL.

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

1. Use **SELECT** to create a query that returns data from a STREAM. This query includes the **LIMIT** keyword to limit the number of rows returned in the query result. Note that exact data output may vary because of the randomness of the data generation.

```
SELECT pageid FROM pageviews_original EMIT changes LIMIT 3;
```

Your output should resemble:

```
Page_24
Page_73
Page_78
LIMIT reached
Query terminated
```

2. 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.

```
CREATE STREAM pageviews_enriched AS
SELECT users_original.userid AS userid, pageid, regionid, gender
FROM pageviews_original
JOIN users_original
ON pageviews_original.userid = users_original.userid
Emit changes;
```

Your output should resemble:

Message

Stream created **and** running

Tip

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

3. 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 KSQL application.

```
SELECT * FROM pageviews_enriched Emit Changes;
```

Your output should resemble:

User_9	Page_92	Region_2	MALE
User_2	Page_66	Region_6	MALE
User_3	Page_10	Region_7	MALE
User_5	Page_30	Region_3	OTHER
User_2	Page_85	Region_6	MALE
User_1	Page_46	Region_7	OTHER
User_6	Page_56	Region_3	FEMALE
User_8	Page_13	Region_2	MALE
User_4	Page_19	Region_4	FEMALE
User_3	Page_44	Region_7	MALE
User_8	Page_57	Region_2	MALE
User_8	Page_39	Region_2	MALE
User_9	Page_15	Region_2	MALE
User_9	Page_71	Region_2	MALE
User_7	Page_69	Region_8	MALE

4. 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';
```

Your output should resemble:

Message

Stream created **and** running

Tip

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

5. 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';
```

Your output should resemble:

Message

Stream created **and** running

6. Verify the above 2 streams:

```
select * from PAGEVIEWS_FEMALE_LIKE_89 emit changes limit 6;  
select * from PAGEVIEWS_FEMALE emit changes limit 3;
```

```
ksql> select * from PAGEVIEWS_FEMALE_LIKE_89 emit changes limit 6;  
+-----+-----+-----+-----+  
| IUSERID | IPAGEID |IREGIONID |IGENDER |  
+-----+-----+-----+-----+  
| User_9 | Page_15 | Region_9 | FEMALE |  
| User_9 | Page_17 | Region_8 | FEMALE |  
| User_9 | Page_66 | Region_8 | FEMALE |  
| User_9 | Page_62 | Region_8 | FEMALE |  
| User_9 | Page_71 | Region_8 | FEMALE |  
| User_6 | Page_31 | Region_8 | FEMALE |  
  
Limit Reached  
Query terminated  
ksql> select * from PAGEVIEWS_FEMALE emit changes limit 3;  
+-----+-----+-----+-----+  
| IUSERID | IPAGEID |IREGIONID |IGENDER |  
+-----+-----+-----+-----+  
| User_1 | Page_30 | Region_8 | FEMALE |  
| User_3 | Page_23 | Region_6 | FEMALE |  
| User_1 | Page_81 | Region_8 | FEMALE |  
  
Limit Reached  
Query terminated  
ksql>
```

7. Create a new persistent query that counts the pageviews for each region 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. **KSQL** will register the Avro schema with the configured Schema Registry when it writes the first message to the **PAGEVIEWS_REGIONS** topic.

```
CREATE TABLE pageviews_regions
  WITH (
    KAFKA_TOPIC = 'pageviews_regions', VALUE_FORMAT='AVRO'
  ) AS
SELECT regionid , COUNT(*) AS numusers
FROM pageviews_enriched
  WINDOW TUMBLING (size 30 second)
GROUP BY regionid
HAVING COUNT(*) > 1 emit changes;
```

Your output should resemble:

Message

Table created **and** running

Tip

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

8. Optional: View results from the above queries using `SELECT`.

`SELECT regionid, numusers FROM pageviews_regions emit changes LIMIT 5;`

Your output should resemble:

```
ksql> SELECT regionid, numusers FROM pageviews_regions emit changes LIMIT 5;
+-----+-----+
|REGIONID          |NUMUSERS
+-----+-----+
|Region_2           |221
|Region_3           |6169
|Region_5           |10659
|Region_2           |11476
|Region_9           |2259
Limit Reached
Query terminated
```

9. Optional: Show all persistent queries.

`SHOW QUERIES;`

Your output should resemble:

Query ID	Kafka Topic	Query String
----------	-------------	--------------

```
-----  
-----  
-----  
----  
CSAS_PAGEVIEWS_FEMALE_1 | PAGEVIEWS_FEMALE | CREATE STREAM  
M pageviews_female AS SELECT * FROM pageviews_enriched WHERE gender =  
'FEMALE';  
CTAS_PAGEVIEWS_REGIONS_3 | PAGEVIEWS_REGIONS | CREATE TABLE  
pageviews_regions WITH (VALUE_FORMAT='avro') AS SELECT gender, region  
id, COUNT(*) AS numusers FROM pageviews_enriched WINDOW TUMBLING  
(size 30 second) GROUP BY gender, regionid HAVING COUNT(*) > 1;  
CSAS_PAGEVIEWS_FEMALE_LIKE_89_2 | PAGEVIEWS_FEMALE_LIKE_89 | 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';  
CSAS_PAGEVIEWS_ENRICHED_0 | PAGEVIEWS_ENRICHED | CREATE STREAM  
pageviews_enriched AS SELECT users_original.userid AS userid, pageid, region  
nid, gender FROM pageviews_original LEFT JOIN users_original ON pagevie  
ws_original.userid = users_original.userid;
```


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

10. 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.

DESCRIBE PAGEVIEWS_REGIONS EXTENDED;

Your output should resemble:

```
Name      : PAGEVIEWS_REGIONS
Type      : TABLE
Key field : KSQL_INTERNAL_COL_0|+|KSQL_INTERNAL_COL_1
Key format : STRING
Timestamp field : Not set - using <ROWTIME>
Value format   : AVRO
Kafka topic  : PAGEVIEWS_REGIONS (partitions: 4, replication: 1)
```

Field | Type

```
ROWTIME | BIGINT    (system)
ROWKEY  | VARCHAR(STRING) (system)
GENDER   | VARCHAR(STRING)
REGIONID | VARCHAR(STRING)
NUMUSERS | BIGINT
```

Queries that write into this TABLE

```
-----  
CTAS_PAGEVIEWS_REGIONS_3 : CREATE TABLE pageviews_regions      WITH (val  
ue_format='avro') AS      SELECT gender, regionid , COUNT(*) AS numusers      FROM  
pageviews_enriched      WINDOW TUMBLING (size 30 second)      GROUP BY gender,  
regionid      HAVING COUNT(*) > 1;
```

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

Local runtime statistics

```
-----  
messages-per-sec: 3.06 total-messages: 1827 last-message: 7/19/18 4:17:55 PM  
UTC  
failed-messages: 0 failed-messages-per-sec: 0 last-failed: n/a  
(Statistics of the local KSQL server interaction with the Kafka topic PAGEVIEWS_REGIONS)  
ksql>
```

You can close the data generation windows/terminals.

----- Lab Ends Here -----

https://ksqldb.io/quickstart.html?_ga=2.53841192.1438767497.1642131382-2002989446.1641377120&_gac=1.255954681.1642171371.CjwKCAiA24SPBhBoEiwAjBgkhg1qFCOJ-Ohq2cWlGrT9c3232dWfPKKpOG6zXpZrNXjqUelgasqp5BoCTEoQAvD_BwE

Any issues related to minimum config clean the zookeeper/kafka-logs and restart the services.

5. KSQL - Kafka Aggregation- 45minutes(D)

In this examples, it will show how to aggregate data from an inbound stream of pageview records, named **pageviews**.

We will demonstrate two features:

- Aggregate Results in a KSQL Table
- Aggregate Over Windows

Load data using users datagen util.

- Create a topic **users_w** (a) - It will be done automatic using datgen.

```
ksql-datagen quickstart=users bootstrap-server=kafkao:9092 format=json topic=users_w  
maxInterval=100
```

- Create stream **users_ws** (b)
- Create kSQL Table using the **users_wt** stream (c)

Let us verify the topic using KSQL;

```
print 'users_w' limit 2;
```

```
ksql> print 'users_w' limit 2;
Format:JSON
{"ROWTIME":1566382936529,"ROWKEY":"User_9","registertime":1496206465489,"userid"
:"User_9","regionid":"Region_3","gender":"FEMALE"}
{"ROWTIME":1566382936573,"ROWKEY":"User_4","registertime":1501550083198,"userid"
:"User_4","regionid":"Region_5","gender":"MALE"}
ksql> █
```

You should be able to view two records as shown above.

(b) Create stream users_ws

```
CREATE STREAM users_ws (userid varchar key,regionid varchar,gender varchar,registertime bigint)
WITH (KAFKA_TOPIC='users_w', VALUE_FORMAT='JSON');
```

Create a KSQL Stream

Choose a topic that contains the data you want in your STREAM

Step 2 of 2

KSQL Cluster

KSQL	
Topic containing data for your STREAM	Name for your STREAM
users_w	

Query the topic as a stream or table	How are your messages encoded?
STREAM	JSON

Key	Timestamp (optional)
userid	

Key Timestamp (optional)

userid	x ▾		▼
--------	-----	--	---

Field(s) you'd like to include in your STREAM

registertime	BIGINT	▼	✖
userid	VARCHAR	▼	✖
regionid	VARCHAR	▼	✖
gender	VARCHAR	▼	✖

+ Add another field

Aggregate Results in a KSQL Table

The result of an aggregate query in KSQL is always a table, because KSQL computes the aggregate for each key, and possibly for each window per key, and updates these results as it processes new input data for a key.

Assume that you want to count the number of pageviews per region. The following query uses the COUNT function to count the pageviews from the time you start the query until you terminate it.

```
SELECT userid, COUNT(*) FROM users_ws GROUP BY userid emit changes;
```

```
|ksql> SELECT userid, COUNT(*) FROM users_ws GROUP BY userid emit changes;
+-----+-----+
|USERID          |KSQL_COL_0
+-----+-----+
|User_4           |3921
|User_3           |3995
|User_2           |4018
|User_1           |4052
|User_5           |3944
|User_6           |3924
|User_9           |3926
|User_8           |3911
|User_7           |3922
|User_4           |12004
|User_3           |11999
|User_8           |11967
|User_7           |11943
|User_5           |11906
|User_2           |11943
|User_9           |11928
|User_1           |12094
|User_6           |11856
^CQuery terminated
ksql>
```

The query uses the CREATE TABLE AS SELECT statement, because the result of the query is a KSQL table.

```
CREATE TABLE pageviews_per_user AS
  SELECT userid,
         COUNT(*) as user_count
    FROM users_ws GROUP BY userid;
```

```
1 CREATE TABLE pageviews_per_user AS
2     SELECT userid,
3            COUNT(*)
4     FROM users_ws GROUP BY userid;
5

● Query properties Run Stop

0 {
1     "@type" : "currentStatus",
2     "statementText" : "CREATE TABLE pageviews_per_user AS\\n SELECT userid,\\n           COUNT(*)\\n      FROM users_ws GROUP BY userid;",
3     "commandId" : "table/PAGEVIEWS_PER_USER/create",
4     "commandStatus":{
5         "status" : "SUCCESS",
6         "message" : "Table created and running"
7     },
8     "commandSequenceNumber" : 21
9 }
```

Determine the value,

`select * from pageviews_per_user ;`

```
ksql> select * from pageviews_per_user ;  
+-----+  
|USERID          |KSQ..._COL_0  
+-----+  
|User_1           |127439  
|User_2           |128239  
|User_3           |127921  
|User_4           |127620  
|User_5           |127910  
|User_6           |127739  
|User_7           |128062  
|User_8           |127497  
|User_9           |128041  
Query terminated  
ksql>
```

Tombstone Records

An important difference between tables and streams is that a record with a non-null key and a null value has a special semantic meaning: in a table, this kind of record is a *tombstone*, which tells KSQL to “DELETE this key from the table”. For a stream, null is a value like any other, with no special meaning.

Aggregate Over Windows

KSQL supports aggregation using tumbling, hopping, and session windows.

In a windowed aggregation, the first seen message is written into the table for a particular key as a null. Downstream applications reading the data will see nulls, and if an application can't handle null values, it may need a separate stream that filters these null records with a WHERE clause.

Aggregate Records Over a Tumbling Window

This query computes the pageview count per region per minute:

```
CREATE TABLE pageviews_per_user_per_minute AS
  SELECT regionid,
         COUNT(*) as count_regionid
    FROM users_ws
   WINDOW TUMBLING(SIZE 1 MINUTE)
  GROUP BY regionid;
```

```
1 CREATE TABLE pageviews_per_user_per_minute AS
2     SELECT regionid,
3            COUNT(*)
4     FROM users_ws
5     WINDOW TUMBLING (SIZE 1 MINUTE)
6     GROUP BY regionid;
7
8
```

● Query properties Run Stop

```
0 {
1     "@type": "currentStatus",
2     "statementText": "CREATE TABLE pageviews_per_user_per_minute AS\n    SELECT regionid,\n           COUNT(*)\n    FROM users_ws\n    WINDOW TUMBLING (SIZE 1 MINUTE)\n   ",
3     "commandId": "table/PAGEVIEWS_PER_USER_PER_MINUTE/create",
4     "commandStatus": {
5         "status": "SUCCESS",
6         "message": "Table created and running"
7     },
8     "commandSequenceNumber": 22
9 }
```

Verify the records:

select * from pageviews_per_user_per_minute ;

```

ksql> select * from pageviews_per_user_per_minute ;
+-----+-----+-----+-----+
| REGIONID | WINDOWSTART | WINDOWEND | COUNT_REGIONID |
+-----+-----+-----+-----+
| Region_1 | 1687405560000 | 1687405620000 | 145220 |
| Region_2 | 1687405560000 | 1687405620000 | 145152 |
| Region_3 | 1687405560000 | 1687405620000 | 145700 |
| Region_4 | 1687405560000 | 1687405620000 | 144857 |
| Region_5 | 1687405560000 | 1687405620000 | 145491 |
| Region_6 | 1687405560000 | 1687405620000 | 145183 |
| Region_7 | 1687405560000 | 1687405620000 | 145393 |
| Region_8 | 1687405560000 | 1687405620000 | 145341 |
| Region_9 | 1687405560000 | 1687405620000 | 145426 |
Query terminated
ksql>

```

If you want to view for a particular region for example, we would like to fetch record related to regionId is equal to Region_1.

`select * from pageviews_per_user_per_minute where regionid='Region_1';`

```

ksql> select * from pageviews_per_user_per_minute where regionid='Region_1';
1566386957480 | Region_1 : Window{start=1566386940000 end=-} | Region_1 | 32
1566386959670 | Region_1 : Window{start=1566386940000 end=-} | Region_1 | 37
1566386962160 | Region_1 : Window{start=1566386940000 end=-} | Region_1 | 44
1566386964123 | Region_1 : Window{start=1566386940000 end=-} | Region_1 | 51
^C1566386966089 | Region_1 : Window{start=1566386940000 end=-} | Region_1 | 54
Query terminated
ksql>

```

To count the pageviews for “Region_6” by female users every 30 seconds, you can change the previous query to the following:

```
CREATE TABLE pageviews_per_region_per_30secs AS
  SELECT regionid,
    COUNT(*)
  FROM users_ws
  WINDOW TUMBLING (SIZE 30 SECONDS)
  WHERE UCASE(gender)='FEMALE' AND LCASE(regionid)='region_6'
  GROUP BY regionid;
```

The screenshot shows a code editor interface with a light gray background. On the left, there is a vertical line of numbers from 1 to 9, each aligned with a line of code. On the right, there is a toolbar with a 'Run' button. Below the code area, there is a section labeled 'Query properties' with a 'Run' button.

```
1 CREATE TABLE pageviews_per_region_per_30secs AS
2   SELECT regionid,
3     COUNT(*)
4   FROM users_ws
5   WINDOW TUMBLING (SIZE 30 SECONDS)
6   WHERE UCASE(gender)='FEMALE' AND LCASE(regionid)='region_6'
7   GROUP BY regionid;
8
9
```

● Query properties Run

```
0 {
1   "@type": "currentStatus",
2   "statementText": "CREATE TABLE pageviews_per_region_per_30secs AS\\n  SELECT regionid,\\n    COUNT(*)\\n  FROM users_ws\\n  WINDOW TUMBLING (\\n    SIZE 30 SECONDS\\n  )\\n  WHERE UCASE(gender)='FEMALE' AND LCASE(regionid)='region_6'\\n  GROUP BY regionid;",
3   "commandId": "table/PAGEVIEWS_PER_REGION_PER_30SECS/create",
4   "commandStatus": {
5     "status": "SUCCESS",
6     "message": "Table created and running"
7   },
8   "commandSequenceNumber": 23
9 }
```

```
select * from pageviews_per_region_per_30secs;
```

Execute the above query to get the result set.

```
ksql>
ksql> select * from pageviews_per_region_per_30secs;
1566387771722 | Region_6 : Window{start=1566387750000 end=-} | Region_6 | 16
1566387773324 | Region_6 : Window{start=1566387750000 end=-} | Region_6 | 17
1566387775769 | Region_6 : Window{start=1566387750000 end=-} | Region_6 | 18
1566387776749 | Region_6 : Window{start=1566387750000 end=-} | Region_6 | 19
1566387779913 | Region_6 : Window{start=1566387750000 end=-} | Region_6 | 21
1566387780450 | Region_6 : Window{start=1566387780000 end=-} | Region_6 | 1
1566387781489 | Region_6 : Window{start=1566387780000 end=-} | Region_6 | 2
1566387784203 | Region_6 : Window{start=1566387780000 end=-} | Region_6 | 4
^CQuery terminated
ksql> █
```

Aggregate Records Over a Hopping Window

This query computes the count for a hopping window of 30 seconds that advances by 10 seconds.

UCASE and LCASE functions are used to convert the values of `gender` and `regionid` columns to uppercase and lowercase, so that you can match them correctly. KSQL also supports the LIKE operator for prefix, suffix, and substring matching.

```
CREATE TABLE pageviews_per_region_per_30secs10secs AS
SELECT regionid,
```

```
COUNT(*)
FROM users_ws
WINDOW HOPPING (SIZE 30 SECONDS, ADVANCE BY 10 SECONDS)
WHERE UCASE(gender)='FEMALE' AND LCASE (regionid) LIKE '%_6'
GROUP BY regionid;
```

The screenshot shows a database interface with a code editor and a results panel.

Code Editor:

```
1 CREATE TABLE pageviews_per_region_per_30secs10secs AS
2     SELECT regionid,
3            COUNT(*)
4     FROM users_ws
5     WINDOW HOPPING (SIZE 30 SECONDS, ADVANCE BY 10 SECONDS)
6     WHERE UCASE(gender)='FEMALE' AND LCASE (regionid) LIKE '%_6'
7     GROUP BY regionid;
8
9
```

Results Panel:

```
0 {
1   "@type": "currentStatus",
2   "statementText": "CREATE TABLE pageviews_per_region_per_30secs10secs AS\nSELECT regionid,\n      COUNT(*)\n    FROM users_ws\n    WINDOW HOPP",
3   "commandId": "table/PAGEVIEWS_PER_REGION_PER_30SECS10SECS/create",
4   "commandStatus": {
5     "status": "SUCCESS",
6     "message": "Table created and running"
7   },
8   "commandSequenceNumber": 24
9 }
```

Run

```
select * from pageviews_per_region_per_30secs10secs;
```

```
ksql> select * from pageviews_per_region_per_30secs10secs;
1566387927042 | Region_6 : Window{start=1566387900000 end=-} | Region_6 | 21
1566387927042 | Region_6 : Window{start=1566387910000 end=-} | Region_6 | 15
1566387927042 | Region_6 : Window{start=1566387920000 end=-} | Region_6 | 7
1566387929672 | Region_6 : Window{start=1566387900000 end=-} | Region_6 | 22
1566387929672 | Region_6 : Window{start=1566387910000 end=-} | Region_6 | 16
1566387929672 | Region_6 : Window{start=1566387920000 end=-} | Region_6 | 8
1566387931175 | Region_6 : Window{start=1566387910000 end=-} | Region_6 | 17
1566387931175 | Region_6 : Window{start=1566387920000 end=-} | Region_6 | 9
1566387931175 | Region_6 : Window{start=1566387930000 end=-} | Region_6 | 1
1566387935410 | Region_6 : Window{start=1566387910000 end=-} | Region_6 | 21
1566387935410 | Region_6 : Window{start=1566387920000 end=-} | Region_6 | 13
1566387935410 | Region_6 : Window{start=1566387930000 end=-} | Region_6 | 5
1566387939245 | Region_6 : Window{start=1566387910000 end=-} | Region_6 | 22
1566387939245 | Region_6 : Window{start=1566387920000 end=-} | Region_6 | 14
1566387939245 | Region_6 : Window{start=1566387930000 end=-} | Region_6 | 6
^CQuery terminated
ksql>
```

Aggregate Records Over a Session Window

The following query counts the number of pageviews per region for session windows, with a session inactivity gap of 60 seconds. This query *sessionizes* the input data and performs the counting step per region.

```
CREATE TABLE pageviews_per_region_per_session AS
SELECT regionid,
       COUNT(*)
  FROM users_ws
WINDOW SESSION (60 SECONDS)
GROUP BY regionid;
```

```
1 CREATE TABLE pageviews_per_region_per_session AS
2     SELECT regionid,
3            COUNT(*)
4     FROM users_ws
5     WINDOW SESSION (60 SECONDS)
6     GROUP BY regionid;
7
8
9
```

● Query properties

Run Stop

```
0 {
1   "@type": "currentStatus",
2   "statementText": "CREATE TABLE pageviews_per_region_per_session AS\\n SELECT regionid,\\n           COUNT(*)\\n      FROM users_ws\\n      WINDOW SESSION (60 SECONDS)\\n      G",
3   "commandId": "table/PAGEVIEWS_PER_REGION_PER_SESSION/create",
4   "commandStatus": {
5     "status": "SUCCESS",
6     "message": "Table created and running"
7   },
8   "commandSequenceNumber": 25
9 }
```

Verify the record:

select * from pageviews_per_region_per_session emit changes;

```
ksql> select * from pageviews_per_region_per_session;
1566388589563 | Region_9 : Window{start=1566388558855 end=1566388589563} | Region_9 | 71
1566388590276 | Region_3 : Window{start=1566388557991 end=1566388590276} | Region_3 | 66
1566388590619 | Region_2 : Window{start=1566388558275 end=1566388590619} | Region_2 | 69
1566388590703 | Region_5 : Window{start=1566388558422 end=1566388590703} | Region_5 | 68
1566388590931 | Region_8 : Window{start=1566388559473 end=1566388590931} | Region_8 | 72
1566388591014 | Region_6 : Window{start=1566388557993 end=1566388591014} | Region_6 | 74
1566388590645 | Region_4 : Window{start=1566388558141 end=1566388590645} | Region_4 | 78
1566388590764 | Region_1 : Window{start=1566388558079 end=1566388590764} | Region_1 | 84
1566388590832 | Region_7 : Window{start=1566388558257 end=1566388590832} | Region_7 | 82
```

----- Lab Ends Here -----

6. Kafka – UDAF – 60 Minutes

In this lab we will learn how to write UDAF and consume in KSQL. UDAFs can be used for computing aggregates against multiple rows of data.

It depends on: KSQL - Kafka Aggregation

This UDAF performs some basic math and adding the computations to a Map object.

Returning a **Map** is one method for returning multiple values from a KSQL function. Using the example above for your own UDAF, take note of the following methods:

- **initialize**: used to specify the initial value of your aggregation
- **aggregate**: performs the actual aggregation by looking at the current row's value (i.e., the **currentValue** argument), as well as the current aggregation value (i.e., **aggregateValue** argument), and generates a new aggregate
- **merge**: describes how to merge two aggregations into one (e.g., when using session windows)

Create a class: **SummaryStatsUdaf** in the package - **com.osteck.ksql.udf**

<!----- UDAF Begins Here-----→

```
package com.osteck.ksql.udf;

import java.util.HashMap;
import java.util.Map;

import io.confluent.ksql.function.udaf.Udaf;
import io.confluent.ksql.function.udaf.UdafDescription;
import io.confluent.ksql.function.udaf.UdafFactory;

/**
 * In this example, we implement a UDAF for computing some summary
 * statistics
 * for a stream of doubles.
 * </pre>
 */
@UdafDescription(name = "summary_stats_big", description = "Example
UDAF that computes some summary stats for a stream of BigInt",
version = "1.0", author = "Henry P")
public final class SummaryStatsUdaf {

    private SummaryStatsUdaf() {
```

```
}

@UdafFactory(description = "compute summary stats for BigInt")
// Can be used with stream aggregations. The input of our
aggregation will be
// doubles,Intermediate will be a Map too
// and the output will be a map
public static Udaf<Long,Map<String, Long>, Map<String, Long>>
createUdaf() {

    return new Udaf<Long, Map<String, Long>, Map<String,
Long>>() {

        /**
         * Specify an initial value for our aggregation
         *
         * @return the initial state of the aggregate.
         */
        @Override
        public Map<String, Long> initialize() {
            final Map<String, Long> stats = new HashMap<>();
            stats.put("mean", Long.valueOf(0));
            stats.put("sample_size", Long.valueOf(0));
            stats.put("sum", Long.valueOf(0));
            return stats;
        }
    }
}
```

```
}

 /**
 * Perform the aggregation whenever a new record appears
in our stream.
 *
 * @param newValue
 *          the new value to add to the {@code
aggregateValue}.
 * @param aggregateValue
 *          the current aggregate.
 * @return the new aggregate value.
 */
@Override
public Map<String, Long> aggregate(final Long newValue,
final Map<String, Long> aggregateValue) {
    final Long sampleSize = Long.valueOf(1) +
(aggregateValue.getOrDefault
                           ("sample_size",
Long.valueOf(0)));
    final Long sum = newValue +
(aggregateValue.getOrDefault("sum", Long.valueOf(0)));
    // calculate the new aggregate
```

```
aggregateValue.put("mean", sum / (sampleSize));
aggregateValue.put("sample_size", sampleSize);
aggregateValue.put("sum", sum);
return aggregateValue;
}

/**
 * Called to merge two aggregates together.
 *
 * @param aggOne
 *          the first aggregate
 * @param aggTwo
 *          the second aggregate
 * @return the merged result
 */
@Override
public Map<String, Long> merge(final Map<String, Long>
aggOne, final Map<String, Long> aggTwo) {
    final Long sampleSize =
aggOne.getOrDefault("sample_size", Long.valueOf(0)) + (
                    aggTwo.getOrDefault("sample_size",
Long.valueOf(0)));
    final Long sum = aggOne.getOrDefault("sum",
Long.valueOf(0)) +
```

```
        (aggTwo.getOrDefault("sum",
Long.valueOf(0)));
        // calculate the new aggregate
final Map<String, Long> newAggregate = new
HashMap<>();
newAggregate.put("mean", sum / (sampleSize));
newAggregate.put("sample_size", sampleSize);
newAggregate.put("sum", sum);
return newAggregate;
}

@Override
public Map<String, Long> map(Map<String, Long> agg) {
    // TODO Auto-generated method stub
    return agg;
}
};

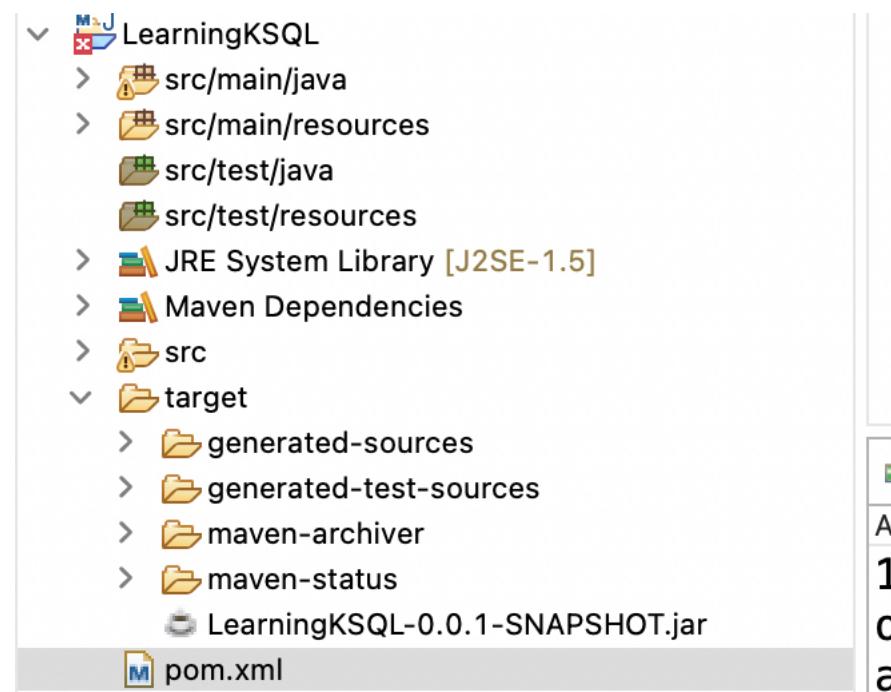
}
}

<!----- UDAF Ends Here ----->
```

Once the UDAF logic is ready, then it's time to deploy your KSQL functions to a KSQL server. To begin, build the project by running the following command in the project root directory:

Maven → Run As → Maven Install

The following file will be generated and need to be deployed in the kafka cluster.



Ar
1
0
a

Now, simply copy this JAR file to the KSQL extension directory (see the `ksql.extension.dir` property in the `ksql-server.properties` file) and restart your KSQL server so that it can pick up the new JAR containing your custom KSQL function. If the

entry is not there; configure it by entering the following line in the [ksql-server.properties file](#).

`ksql.extension.dir=/opt/scripts/ksql`

Create the folder structure and grant the access.

```
[root@kafka0 ksql]# pwd  
/opt/scripts/ksql  
[root@kafka0 ksql]# ls  
LearningKSQL-0.0.1-SNAPSHOT.jar  
[root@kafka0 ksql]#
```

Hints[mkdir, chmod 777]

Copy the jar.

Restart the confluent

Once KSQL has finished restarting and has connected to a running Apache Kafka® cluster, you can verify that the new functions exist by running the [DESCRIBE FUNCTION](#) command from the CLI:

`DESCRIBE FUNCTION SUMMARY_STATS_BIG ;`

```
Query terminated
ksql> DESCRIBE FUNCTION SUMMARY_STATS_BIG ;

Name      : SUMMARY_STATS_BIG
Author    : Henry P
Version   : 1.0
Overview  : Example UDAF that computes some summary stats for a stream of BigInt
Type      : AGGREGATE
Jar       : /opt/scripts/ksql/LearningKSQL-0.0.1-SNAPSHOT.jar
Variations :

        Variation  : SUMMARY_STATS_BIG(val BIGINT)
        Returns    : MAP<VARCHAR, BIGINT>
        Description: compute summary stats for BigInt
ksql> █
```

Let us generate data for using this UDAF in our query.

Use the following avro schema to generate the data.

Create **impressions.avro** file in **/opt/scripts** folder and execute the following command from that folder only.

<!-----Schema File : impressions.avro Begins-----→

```
{  
    "namespace": "streams",  
    "name": "impressions",  
    "type": "record",  
    "fields": [  
        {"name": "impresssiontime", "type": {  
            "type": "long",  
            "format_as_time": "unix_long",  
            "arg.properties": {  
                "iteration": { "start": 1, "step": 10}  
            }  
        }},  
        {"name": "impressionid", "type": {  
            "type": "string",  
            "arg.properties": {  
                "regex": "impression_[1-9][0-9][0-9]"  
            }  
        }},  
        {"name": "userid", "type": {  
            "type": "string",  
            "arg.properties": {  
                "regex": "user_[1-9][0-9]?"  
            }  
        }},  
        {"name": "adid", "type": {  
            "type": "string",  
            "arg.properties": {  
                "regex": "ad_[1-9][0-9]?"  
            }  
        }}]
```

```

        "regex": "ad_[1-9][0-9]?"}
    },
    {"name": "impressionscore", "type": {
        "type": "long",
        "arg.properties": {
            "iteration": { "start": 1, "step": 2}
        }
    }}
]
}
<!----- Schema File : impressions.avro Ends ----->

```

```
#cd /opt/scripts
ksql-dagagen schema=impressions.avro format=avro topic=impressions key=impressionid bootstrap-
server=kafka:9092
```

When you have a custom schema registered, you can generate test data that's made up of random values that satisfy the schema requirements. In the `impressions` schema, advertisement identifiers are two-digit random numbers between 10 and 99, as specified by the regular expression `ad_[1-9][0-9]`. It has `impressionscore` which is the score of impression for the advertisement. We will calculate stats on this column using the UDAF we have define earlier.

After a few startup messages, your output should resemble:

```
impression_162 --> ([[ 1566555549459 | 'impression_162' | 'user_81' | 'ad_38' | 575 ]) ts:156655  
5549459  
impression_712 --> ([[ 1566555549895 | 'impression_712' | 'user_33' | 'ad_26' | 577 ]) ts:156655  
5549895  
impression_279 --> ([[ 1566555549920 | 'impression_279' | 'user_91' | 'ad_92' | 579 ]) ts:156655  
5549920  
impression_306 --> ([[ 1566555550302 | 'impression_306' | 'user_34' | 'ad_78' | 581 ]) ts:156655  
5550303  
impression_522 --> ([[ 1566555550475 | 'impression_522' | 'user_40' | 'ad_96' | 583 ]) ts:156655  
5550476  
impression_733 --> ([[ 1566555550636 | 'impression_733' | 'user_80' | 'ad_31' | 585 ]) ts:156655  
5550637  
impression_318 --> ([[ 1566555551108 | 'impression_318' | 'user_91' | 'ad_17' | 587 ]) ts:156655  
5551109  
impression_632 --> ([[ 1566555551319 | 'impression_632' | 'user_25' | 'ad_80' | 589 ]) ts:156655  
5551319  
impression_959 --> ([[ 1566555551366 | 'impression_959' | 'user_28' | 'ad_65' | 591 ]) ts:156655  
5551366
```

By now, you should have a topic by the name, impressions as shown below.

The screenshot shows the Apache Kafka Control Center interface. On the left, there's a sidebar with 'MONITORING' and 'MANAGEMENT' sections. Under 'MONITORING', 'System health', 'Data streams', and 'Consumer lag' are listed. Under 'MANAGEMENT', 'Kafka Connect', 'Clusters', and 'Topics' are listed, with 'Topics' being the active tab, indicated by a purple background.

The main area is titled 'MANAGEMENT > Topics'. It features a search bar with the text 'imp' and a checkbox labeled 'Show internal topics'. Below this, a table lists a single topic:

| Name | Pa | Tot |
|-------------|-----|-----|
| impressions | ... | 1 |

Consume the Test Data Stream

In the KSQL CLI or in Control Center, register the **impressions** stream:

CREATE STREAM impressions (viewtime BIGINT, key VARCHAR, userid VARCHAR, adid VARCHAR, impressionscore BIGINT) WITH (KAFKA_TOPIC='impressions', VALUE_FORMAT='avro');

You can query the stream now.

```
select * from impressions;
```

```
1566555480855 | impression_410 | null | null | user_74 | ad_22 | 1
1566555480926 | impression_631 | null | null | user_50 | ad_10 | 3
1566555481142 | impression_705 | null | null | user_31 | ad_12 | 5
1566555481574 | impression_365 | null | null | user_91 | ad_72 | 7
1566555481894 | impression_610 | null | null | user_91 | ad_14 | 9
1566555481970 | impression_526 | null | null | user_45 | ad_88 | 11
1566555482437 | impression_663 | null | null | user_15 | ad_24 | 13
1566555482725 | impression_939 | null | null | user_26 | ad_95 | 15
1566555482856 | impression_654 | null | null | user_39 | ad_73 | 17
```

At this point, invoking our UDF/UDAF is simply a matter of adding it to our KSQL query:

```
SELECT userid , summary_stats_big ( impressionscore )
FROM impressions GROUP BY userid EMIT CHANGES;
```

```
user_64 | {sample_size=1, mean=1223, sum=1223}
user_12 | {sample_size=1, mean=1225, sum=1225}
user_14 | {sample_size=1, mean=1227, sum=1227}
user_60 | {sample_size=1, mean=1229, sum=1229}
user_99 | {sample_size=1, mean=1231, sum=1231}
user_23 | {sample_size=1, mean=1233, sum=1233}
user_18 | {sample_size=1, mean=1237, sum=1237}
user_57 | {sample_size=2, mean=1237, sum=2474}
user_29 | {sample_size=1, mean=1241, sum=1241}
user_96 | {sample_size=1, mean=1243, sum=1243}
user_47 | {sample_size=1, mean=1245, sum=1245}
user_15 | {sample_size=1, mean=1247, sum=1247}
user_96 | {sample_size=2, mean=1246, sum=2492}
user_19 | {sample_size=1, mean=1251, sum=1251}
user_24 | {sample_size=1, mean=1253, sum=1253}
```

As you can observe for each userid it returns a stat calculated using the UDAF we have define earlier.

----- Lab Ends Here -----

7. Testing and Troubleshooting – 60 Minutes

Testing Lab:

In this exercise, let us write a unit test for aggregation operation in Kafka Streams application.

Define Avro schema and store in the following file.

avro/electronic_order.avsc

Update the following in the above file.

```
{  
  "namespace": "com.test.avro",  
  "type": "record",  
  "name": "ElectronicOrder",  
  "fields": [  
    {"name": "order_id", "type": "string"},  
    {"name": "electronic_id", "type": "string"},  
    {"name": "user_id", "type": "string"},
```

```
{"name": "price", "type": "double", "default": 0.0 },  
 {"name": "time", "type": "long" }  
]  
}
```

The Utility class to read stream properties file along with some helper methods are define in the below java file.

com/utils/stream/StreamsUtils.java

```
package com.utils.stream;
```

```
import java.io.FileInputStream;
```

```
import java.io.IOException;
```

```
import java.util.HashMap;
```

```
import java.util.Map;
```

```
import java.util.Properties;
```

```
import org.apache.avro.specific.SpecificRecord;
```

```
import org.apache.kafka.clients.admin.NewTopic;
```

```
import org.apache.kafka.clients.producer.Callback;

import io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde;

public class StreamsUtils {

    public static final String PROPERTIES_FILE_PATH =
"src/main/resources/streams.properties";

    public static final short REPLICATION_FACTOR = 3;
    public static final int PARTITIONS = 6;

    public static Properties loadProperties() throws
IOException {
        Properties properties = new Properties();
        try (FileInputStream fis = new
FileInputStream("src/main/resources/streams.properties")) {
```

```
        properties.load(fis);
        return properties;
    }

    public static Map<String, Object> propertiesToMap(final
Properties properties) {
    final Map<String, Object> configs = new HashMap<>();
    properties.forEach((key, value) ->
configs.put((String)key, (String)value));
    return configs;
}

public static <T extends SpecificRecord>
SpecificAvroSerde<T> getSpecificAvroSerde(final Map<String, Object>
serdeConfig) {
```

```
    final SpecificAvroSerde<T> specificAvroSerde = new
SpecificAvroSerde<>();
    specificAvroSerde.configure(serdeConfig, false);
    return specificAvroSerde;
}

public static Callback callback() {
    return (metadata, exception) -> {
        if(exception != null) {
            System.out.printf("Producing records
encountered error %s %n", exception);
        } else {
            System.out.printf("Record produced - offset -
%d timestamp - %d %n", metadata.offset(), metadata.timestamp());
        }
    }
}
```

```
    };  
}  
  
public static NewTopic createTopic(final String topicName){  
    return new NewTopic(topicName, PARTITIONS,  
REPLICATION_FACTOR);  
}  
}
```

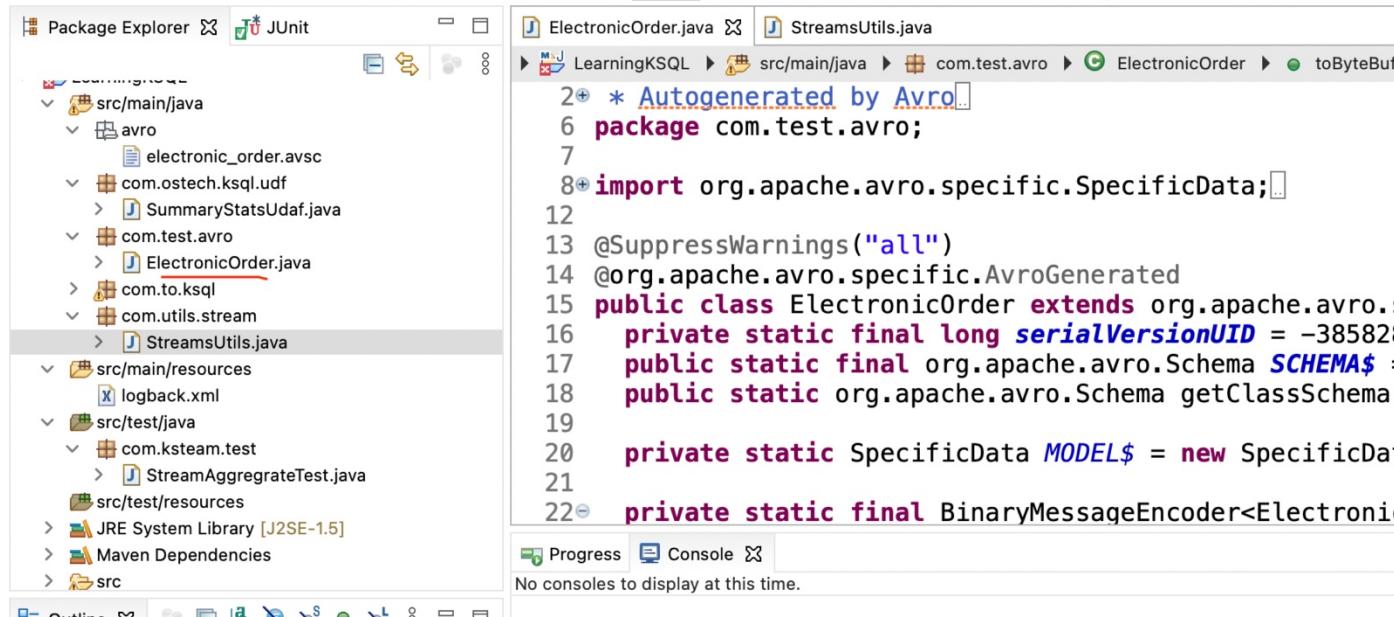
Generate Resources from Avro Schema.

ElectronicOrder.java object is being used in the stream application to map the record event in the stream application.

It will be generated from the above define Avro schema.

Pom.xml → Generate sources.

It will generate class file as shown below:



The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer view displays the project structure under 'src/main/java'. It includes packages like 'avro' (containing 'electronic_order.avsc'), 'com.ostech.ksql.udf' (containing 'SummaryStatsUdaf.java'), 'com.test.avro' (containing 'ElectronicOrder.java'), 'com.to.ksql', and 'com.utils.stream' (containing 'StreamsUtils.java'). On the right, the code editor window shows the generated Java code for 'ElectronicOrder.java'. The code is annotated with 'Autogenerated by Avro' and includes imports for org.apache.avro specific data, suppression of warnings, and annotations for AvroGenerated. It defines a class 'ElectronicOrder' extending org.apache.avro.specific.SpecificData, with static fields for serialVersionUID, SCHEMA\$, and MODEL\$. A BinaryMessageEncoder<ElectronicOrder> is also defined.

```

2+ * Autogenerated by Avro...
6 package com.test.avro;
7
8+ import org.apache.avro_specific.SpecificData;
12
13 @SuppressWarnings("all")
14 @org.apache.avro_specific.AvroGenerated
15 public class ElectronicOrder extends org.apache.avro.s
16     private static final long serialVersionUID = -385828
17     public static final org.apache.avro.Schema SCHEMA$ =
18     public static org.apache.avro.Schema getClassSchema(
19
20     private static SpecificData MODEL$ = new SpecificDat
21
22+ private static final BinaryMessageEncoder<Electronic

```

Finally, let us define the Unit Test Program

com/ksteam/test/StreamAggregateTest.java

Import the following package.

package com.ksteam.test;

```
import static org.junit.Assert.assertEquals;  
import io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde;  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Map;  
import java.util.Properties;  
  
import org.apache.kafka.common.serialization.Serde;  
import org.apache.kafka.common.serialization.Serdes;  
import org.apache.kafka.streams.StreamsBuilder;  
import org.apache.kafka.streams.StreamsConfig;  
import org.apache.kafka.streams.TestInputTopic;  
import org.apache.kafka.streams.TestOutputTopic;  
import org.apache.kafka.streams.TopologyTestDriver;
```

```
import org.apache.kafka.streams.kstream.Consumed;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.kstream.Produced;
import org.junit.Test;

import com.test.avro.ElectronicOrder;
import com.utils.stream.StreamsUtils;
```

Add the following test code.

```
@Test
public void shouldAggregateRecords() {
```

```
final Properties streamsProps = new Properties();
streamsProps.put(StreamsConfig.APPLICATION_ID_CONFIG,
"aggregate-test");
streamsProps.put("schema.registry.url",
"mock://aggregation-test");

final String inputTopicName = "input";
final String outputTopicName = "output";
final Map<String, Object> configMap =
    StreamsUtils.propertiesToMap(streamsProps);

final SpecificAvroSerde<ElectronicOrder> electronicSerde =
    StreamsUtils.getSpecificAvroSerde(configMap);
final Serde<String> stringSerde = Serdes.String();
final Serde<Double> doubleSerde = Serdes.Double();
```

```
final StreamsBuilder builder = new StreamsBuilder();
final KStream<String, ElectronicOrder> electronicStream =
    builder.stream(inputTopicName,
Consumed.with(Serdes.String(), electronicSerde));

electronicStream.groupByKey().aggregate(() -> 0.0,
    (key, order, total) -> total + order.getPrice(),
    Materialized.with(stringSerde, doubleSerde))
    .toStream().to(outputTopicName,
Produced.with(Serdes.String(), Serdes.Double()));

try (final TopologyTestDriver testDriver = new
TopologyTestDriver(builder.build(), streamsProps)) {
    final TestInputTopic<String, ElectronicOrder>
inputTopic =
```

```
testDriver.createInputTopic(inputTopicName,  
    stringSerde.serializer(),  
    electronicSerde.serializer());  
  
final TestOutputTopic<String, Double> outputTopic =  
    testDriver.createOutputTopic(outputTopicName,  
        stringSerde.deserializer(),  
        doubleSerde.deserializer());  
  
final List<ElectronicOrder> orders = new ArrayList<>();  
  
orders.add(ElectronicOrder.newBuilder().setElectronicId("one").setOrder  
    id("1").setUserId("vandeley").setTime(5L).setPrice(5.0).build());  
  
orders.add(ElectronicOrder.newBuilder().setElectronicId("one").setOrder  
    id("2").setUserId("penny-  
    packer").setTime(5L).setPrice(15.0).build());
```

```
orders.add(ElectronicOrder.newBuilder().setElectronicId("one").setOrder  
Id("3").setUserId("romanov").setTime(5L).setPrice(25.0).build())  
);  
  
        // Successful Test case  
        List<Double> expectedValues = List.of(5.0, 20.0, 45.0);  
        orders.forEach(order ->  
inputTopic.pipeInput(String.valueOf(order.getElectronicId()),  
order));  
  
        List<Double> actualValues =  
outputTopic.readValuesToList();  
        assertEquals(expectedValues, actualValues);  
  
    }  
  
}
```

That completes the definition of the Test program.

Note:

Begin by adding a configuration to use a mock Schema Registry under your existing `streamsProps` configuration:

```
streamsProps.put("schema.registry.url", "mock://aggregation-test");
```

This is an in-memory version of your Schema Registry, suitable for unit testing.

Next, under your `groupByKey` statement, create a `TopologyTestDriver` with the application topology and configuration. (Note that we use a `try-with-resources` block to ensure that the test driver is closed at the end of the test, which is important for cleaning up state).

Create a test input topic with the `TopologyTestDriver` factory method `createInputTopic`; you'll use this in the test to drive input records into the Kafka Streams application. Provide a topic name and also SerDes, since a Kafka Streams application expects to see records in byte array format.

```
try (final TopologyTestDriver testDriver = new
TopologyTestDriver(builder.build(), streamsProps)) {
    final TestInputTopic<String, ElectronicOrder> inputTopic =
        testDriver.createInputTopic(inputTopicName,
            stringSerde.serializer(),
            electronicSerde.serializer());
```

Now create a `TestOutputTopic` with another `TopologyTestDriver` factory method. The `TestOutputTopic` captures results from the Kafka Streams application under test. Add the `outputTopicName` and SerDes. (Note that the Kafka Streams application output is also in byte array format.)

```
final TestOutputTopic<String, Double> outputTopic =
    testDriver.createOutputTopic(outputTopicName,
        stringSerde.deserializer(),
        doubleSerde.deserializer());
```

Next, you need some sample events for the test. Create a list of `ElectronicOrder` objects for input into the topology. Here, each object is created using the builder provided by the generated Avro object code.

```
final List<ElectronicOrder> orders = new ArrayList<>();
orders.add(ElectronicOrder.newBuilder().setElectronicId("one").setOrderId("1")
    .setUserId("vandeley").setTime(5L).setPrice(5.0).build());
orders.add(ElectronicOrder.newBuilder().setElectronicId("one").setOrderId("2")
    .setUserId("penny-packer").setTime(5L).setPrice(15.0).build());
orders.add(ElectronicOrder.newBuilder().setElectronicId("one").setOrderId("3")
    .setUserId("romanov").setTime(5L).setPrice(25.0).build());
```

Create the list of expected aggregation results. Note that the `TopologyTestDriver` does not buffer out anything, so each input record emits a new update (similar to the behavior you see by setting the cache to zero bytes in the Kafka Streams application). Next, run some events into your test topology using `inputTopic.pipeInput`.

```
List<Double> expectedValues = List.of(5.0, 20.0, 45.0);  
orders.forEach(order -> inputTopic.pipeInput(order.getElectronicId(),  
order));
```

Get the output from Kafka Streams and use the `outputTopic.readValuesToList` method to read only the values, output into a list (since we only care about the aggregation results). Finally, confirm the results of the test by asserting that the `actualValues` list equals `expectedValues`.

```
List<Double> actualValues = outputTopic.readValuesToList();  
assertEquals(expectedValues, actualValues);
```

Execute Test Case using pom.xml

Pom.xml → Run as → Maven Test

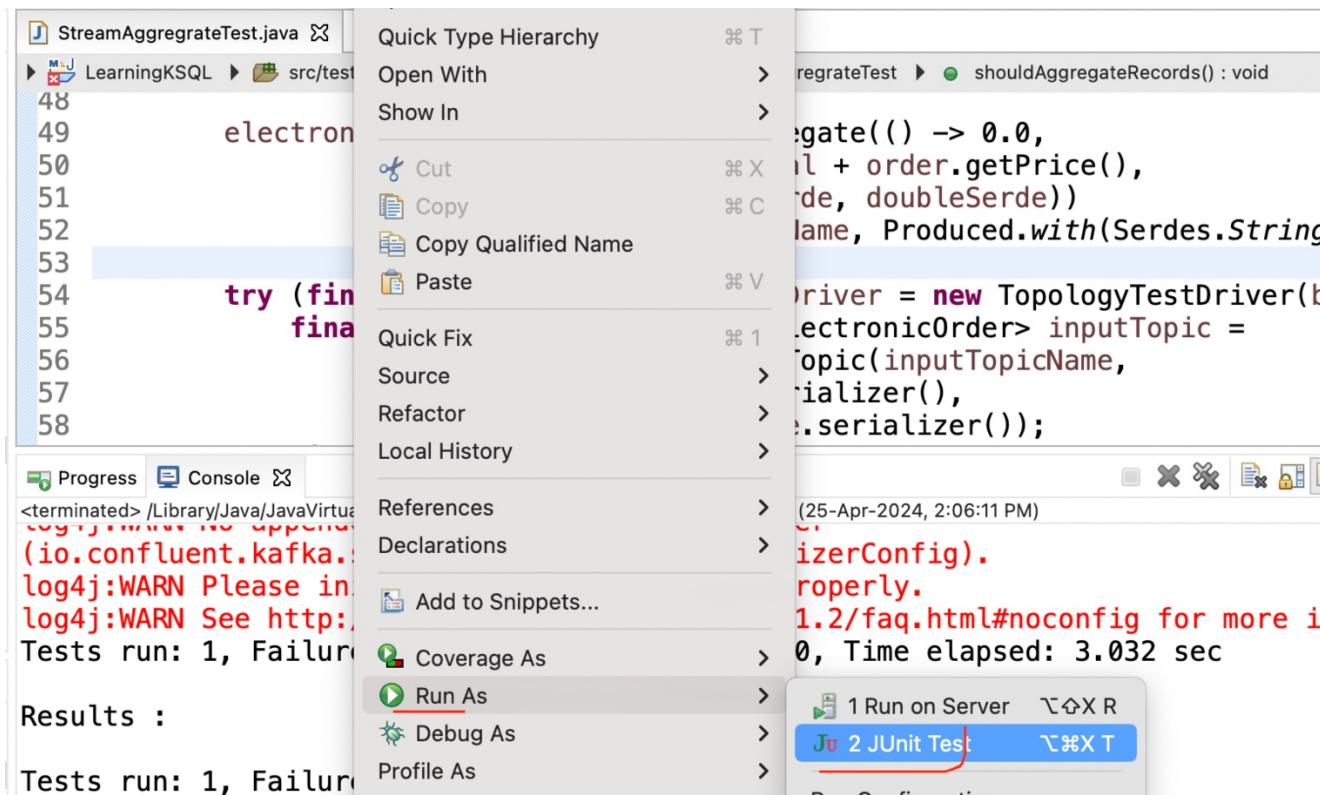
```
<terminated> /Library/Java/JavaVirtualMachines/jdk-11.0.9.jdk/Contents/Home/bin/java (25-Apr-2024, 2:06:11 PM)
log4j:WARN No appenders could be found for logger.
(io.confluent.kafka.serializers.KafkaAvroSerializerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.032 sec
```

Results :

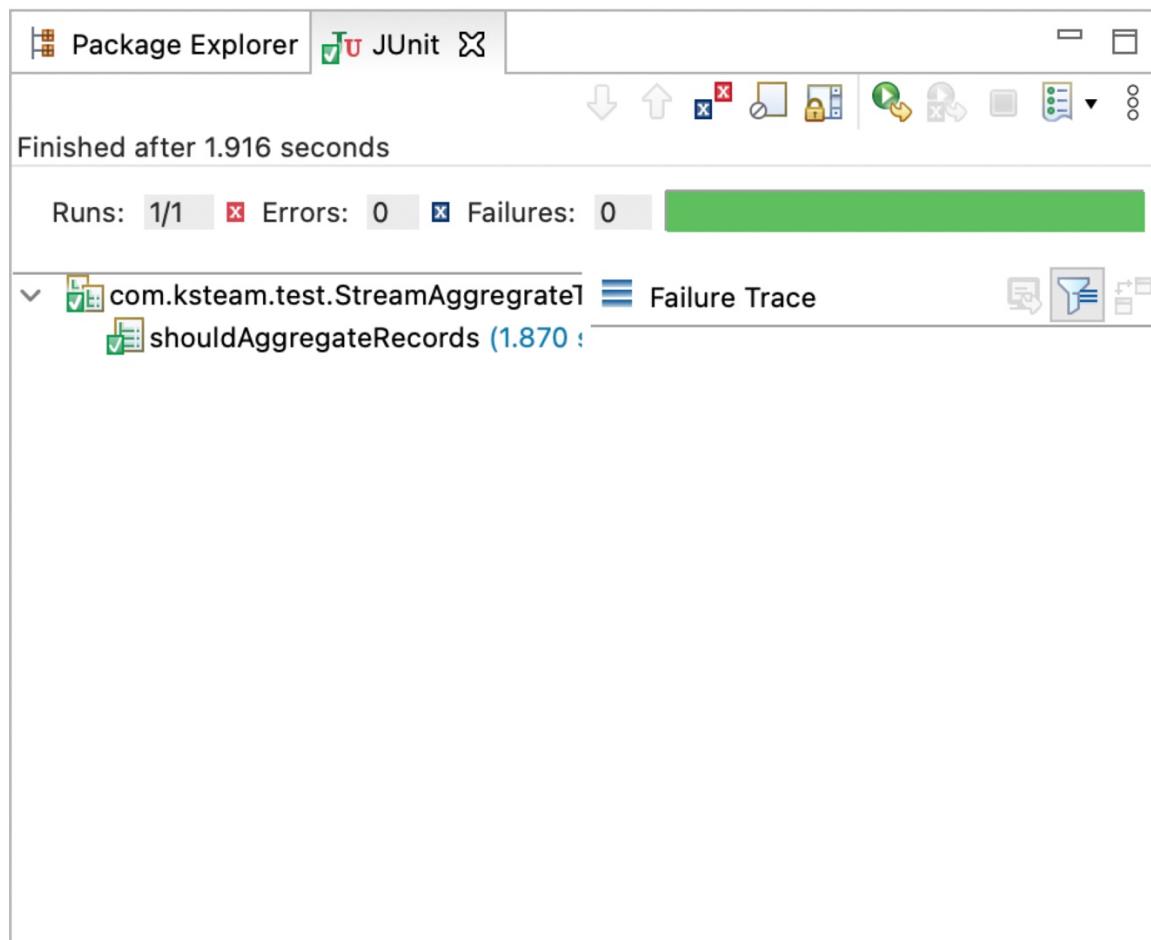
```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  8.450 s
[INFO] Finished at: 2024-04-25T14:06:22+05:30
[INFO] -----
```

Or Run as Junit application.



It should output as shown below:



Troubleshooting:

In this section, build an example workload that enables experimenting with pull queries.

In the ksqlDB CLI, run the following statement to create an input stream.

```
CREATE STREAM pq_pageviews (user_id INTEGER KEY, url STRING, status INTEGER) WITH  
(kafka_topic='pq_pageviews', value_format='json', partitions=1);
```

Create a materialized view over this input stream. The materialized view aggregates events from the `pageviews` stream, grouped on the events' `url` field:

```
CREATE TABLE pq_pageviews_metrics AS  
SELECT url, COUNT(*) AS num_views  
FROM pq_pageviews  
GROUP BY url  
EMIT CHANGES;
```

Populate the materialized view by writing events to the `pageviews` input stream.

```
INSERT INTO pq_pageviews (user_id, url, status) VALUES (0, 'https://confluent.io', 200);  
INSERT INTO pq_pageviews (user_id, url, status) VALUES (1, 'https://confluent.io/blog', 200);
```

Now that the materialized view contains some data, issue a pull query against it to retrieve the latest count for a given URL.

```
SELECT * FROM pq_pageviews_metrics WHERE url = 'https://confluent.io';
```

Your output should resemble:

```
ksql> SELECT * FROM pq_pageviews_metrics WHERE url = 'https://confluent.io';
+-----+-----+
| URL          | NUM.Views |
+-----+-----+
| https://confluent.io | 1           |
Query terminated
ksql>
```

This pull query should return precisely one row. Each time the pull query runs, it will return the latest value for the targeted row.

Query Restarts

If your cluster is repeatedly hitting query restarts, you can see details about the underlying failures by running the ksqlDB EXPLAIN statement:

```
EXPLAIN <QUERY ID>;
```

How to find the Query ID

```
show queries;
```

```
ksql> show queries;

Query ID          | Query Type | Status    | Sink Name           | Sink Kafka Topic | Query String
-----+-----+-----+-----+-----+-----+
CTAS_PQ_PAGEVIEWS_METRICS_5 | PERSISTENT | RUNNING:1 | PQ_PAGEVIEWS_METRICS | PQ_PAGEVIEWS_METRICS | CREATE TABLE PQ_PAGEVIEWS_METRIC
CS WITH (KAFKA_TOPIC='PQ_PAGEVIEWS_METRICS', PARTITIONS=1, REPLICAS=1) AS SELECT    PQ_PAGEVIEWS.URL URL,    COUNT(*) NUM_VIEWS FROM PQ
_PAGEVIEWS PQ_PAGEVIEWS GROUP BY PQ_PAGEVIEWS.URL EMIT CHANGES;
-----+-----+-----+-----+-----+-----+
For detailed information on a Query run: EXPLAIN <Query ID>;
ksql> ■
```

```
#Explain CTAS_PQ_PAGEVIEWS_METRICS_5
```

```
--> Aggregate-GroupBy-repartition-sink
<-- Aggregate-GroupBy
Sink: Aggregate-GroupBy-repartition-sink (topic: Aggregate-GroupBy-repartition)
<-- Aggregate-GroupBy-repartition-filter

Sub-topology: 1
Source: Aggregate-GroupBy-repartition-source (topics: [Aggregate-GroupBy-repartition])
--> KSTREAM-AGGREGATE-0000000005
Processor: KSTREAM-AGGREGATE-0000000005 (stores: [Aggregate-Aggregate-Materialize])
--> Aggregate-Aggregate-ToOutputSchema
<-- Aggregate-GroupBy-repartition-source
Processor: Aggregate-Aggregate-ToOutputSchema (stores: [])
--> Aggregate-Project
<-- KSTREAM-AGGREGATE-0000000005
Processor: Aggregate-Project (stores: [])
--> KTABLE-TOSTREAM-0000000011
<-- Aggregate-Aggregate-ToOutputSchema
Processor: KTABLE-TOSTREAM-0000000011 (stores: [])
--> KSTREAM-SINK-0000000012
<-- Aggregate-Project
Sink: KSTREAM-SINK-0000000012 (topic: PQ_PAGEVIEWS_METRICS)
<-- KTABLE-TOSTREAM-0000000011
```

Overridden Properties

| Property | Value |
|-------------------|--------|
| auto.offset.reset | latest |

ksql> █

Get Details about the stream

DESCRIBE pq_pageviews_metrics;

```
ksql> DESCRIBE pq_pageviews_metrics;

Name          : PQ_PAGEVIEWS_METRICS
Field	Type
URL        | VARCHAR(STRING) (primary key)
NUM.Views | BIGINT
-----|-----
For runtime statistics and query details run: DESCRIBE <Stream,Table> EXTENDED;
ksql> |
```

DESCRIBE pq_pageviews_metrics EXTENDED;

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

Runtime statistics by host
-----
Host          | Metric           | Value    | Last Message
-----
kafka0:8088   | messages-per-sec | 0        | 2024-04-27T06:09:49.806Z
kafka0:8088   | total-messages   | 2        | 2024-04-27T06:09:49.806Z
-----
(Statistics of the local KSQL server interaction with the Kafka topic PQ_PAGEVIEWS_METRICS)

Consumer Groups summary:

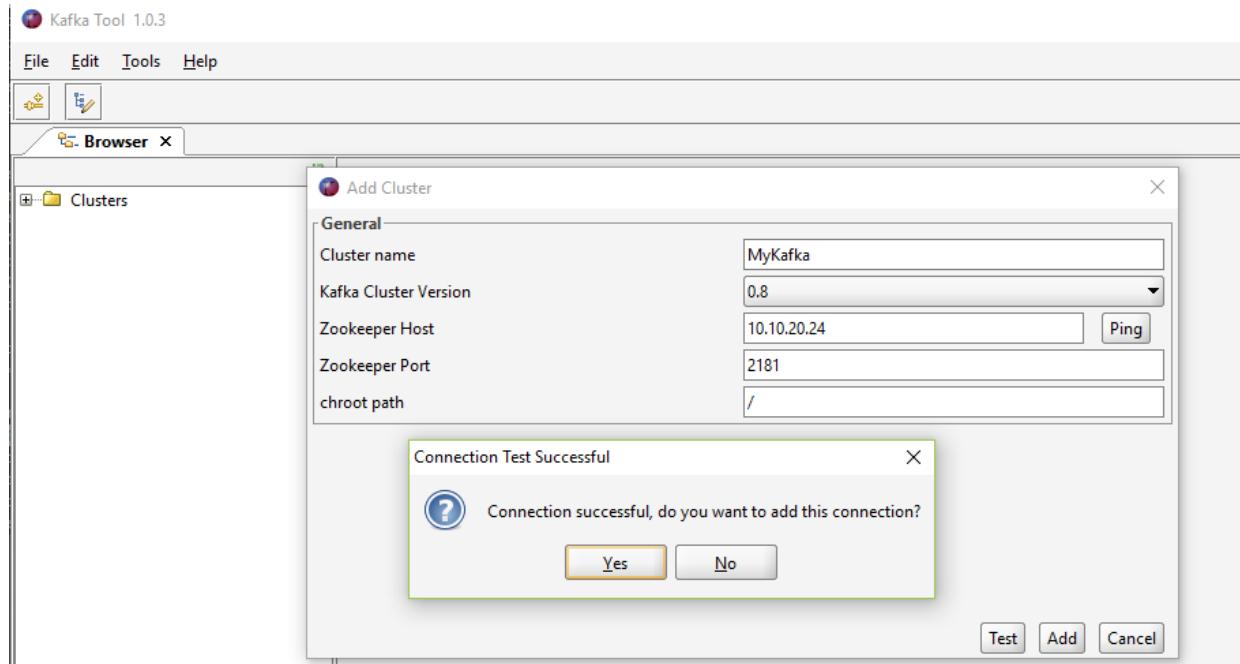
Consumer Group      : _confluent-ksql-default_query_CTAS_PQ_PAGEVIEWS_METRICS_5
Kafka topic         : _confluent-ksql-default_query_CTAS_PQ_PAGEVIEWS_METRICS_5-Aggregate-GroupBy-repartition
Max lag             : 0

Partition | Start Offset | End Offset | Offset | Lag
-----
0        | 2              | 2          | 2      | 0
-----
Kafka topic      : pq_pageviews
Max lag          : 0

Partition | Start Offset | End Offset | Offset | Lag
-----
0        | 0              | 2          | 2      | 0
-----
ksql> █
```

----- Lab Ends Here -----

8. Kafkatools



9. Errors

I. LEADER_NOT_AVAILABLE

{test=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)

```
[2018-05-15 23:46:40,132] WARN [Producer clientId=console-producer] Error while
fetching metadata with correlation id 14 : {test=LEADER_NOT_AVAILABLE} (org.apac
he.kafka.clients.NetworkClient)
[2018-05-15 23:46:40,266] WARN [Producer clientId=console-producer] Error while
fetching metadata with correlation id 15 : {test=LEADER_NOT_AVAILABLE} (org.apac
he.kafka.clients.NetworkClient)
^C[2018-05-15 23:46:40,394] WARN [Producer clientId=console-producer] Error whil
e fetching metadata with correlation id 16 : {test=LEADER_NOT_AVAILABLE} (org.ap
ache.kafka.clients.NetworkClient)
[root@tos opt]# {test=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkCl
ient)
bash: syntax error near unexpected token `org.apache.kafka.clients.NetworkClient
,
```

Solutions: /opt/kafka/config/server.properties

Update the following information.

```
# it uses the value for "listeners" if configured. Otherwise, it will use the v
alue
# returned from java.net.InetAddress.getCanonicalHostName().
advertised.listeners=PLAINTEXT://localhost:9092
# More listener names to security protocols, the default is for them to be the s
```

java.util.concurrent.ExecutionException:

org.apache.kafka.common.errors.TimeoutException: Expiring 1 record(s) for my-kafka-topic-6: 30037 ms has passed since batch creation plus linger time

at

```
org.apache.kafka.clients.producer.internals.FutureRecordMetadata.valueOrError(FutureRecordMetadata.java:94)
```

at

```
org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMetadata.java:64)
```

at

```
org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMetadata.java:29)
```

at com.tos.kafka.MyKafkaProducer.runProducer(MyKafkaProducer.java:97)

at com.tos.kafka.MyKafkaProducer.main(MyKafkaProducer.java:18)

Caused by: org.apache.kafka.common.errors.TimeoutException: Expiring 1 record(s) for my-kafka-topic-6: 30037 ms has passed since batch creation plus linger time.

Solution:

Update the following in all the server properties: /opt/kafka/config/server.properties

```
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://tos.master.com:9093

# Hostname and port the broker will advertise to producers and consumers. If not
# set,
# it uses the value for "listeners" if configured. Otherwise, it will use the v
alue
# returned from java.net.InetAddress.getCanonicalHostName().
advertised.listeners=PLAINTEXT://tos.master.com:9093

# Maps listener names to security protocols, the default is for them to be the s
ame. See the config documentation for more details
#listener.security.protocol.map=PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_
PLAINTEXT,SASL_SSL:SASL_SSL
```

It's should be updated with your hostname and restart the broker

Changes in the following file, if the hostname is to be changed.

//kafka/ Server.properties and control center

/apps/confluent/etc/confluent-control-center/control-center-dev.properties

/apps/confluent/etc/ksql/ksql-server.properties

/tmp/confluent.8A2Ii7O4/connect/connect.properties

Update localhost to resolve to the ip in /etc/hosts.

In case the hostname doesn't start, update with ip address and restart the broker.

10. Annexure Code:**II. DumplogSegment**

```
/opt/kafka/bin/kafka-run-class.sh kafka.tools.DumpLogSegments --deep-iteration --print-data-log --files \
```

```
/tmp/kafka-logs/my-kafka-connect-0/oooooooooooooooooooo.log | head -n 4
```

95 Kafka – Dev Ops

```
[root@tos test-topic-0]# more 00000000000000000000.log
[root@tos test-topic-0]# cd ..
[root@tos kafka-logs]# cd my-kafka-connect-0/
[root@tos my-kafka-connect-0]# ls
00000000000000000000.index      0000000000000000000011.snapshot
00000000000000000000.log        leader-epoch-checkpoint
00000000000000000000.timeindex
[root@tos my-kafka-connect-0]# more *log
\kafka Connector.--More-- (53%)

[root@tos my-kafka-connect-0]# pwd
/tmp/kafka-logs/my-kafka-connect-0
[root@tos my-kafka-connect-0]# /opt/kafka/bin/kafka-run-class.sh kafka.tools.DumpLogSegments --deep-iteration --print-data-log --files \
> /tmp/kafka-logs/my-kafka-connect-0/00000000000000000000.log | head -n 4
Dumping /tmp/kafka-logs/my-kafka-connect-0/00000000000000000000.log
Starting offset: 0
offset: 0 position: 0 CreateTime: 1530552634675 isvalid: true keysize: -1 valuesize: 31 magic: 2 compresscodec: NONE producerId: -1 producerEpoch: -1 sequence: -1 isTransactional: false headerKeys: [] payload: This Message is from Test File .
offset: 1 position: 0 CreateTime: 1530552634677 isvalid: true keysize: -1 valuesize: 43 magic: 2 compresscodec: NONE producerId: -1 producerEpoch: -1 sequence: -1 isTransactional: false headerKeys: [] payload: It will be consumed by the Kafka Connector.
[root@tos my-kafka-connect-0]#
```

III. Data Generator – JSON

Streaming Json Data Generator

Downloading the generator

You can always find the [most recent release](#) over on github where you can download the bundle file that contains the runnable application and example configurations. Head there now and download a release to get started!

Configuration

The generator runs a Simulation which you get to define. The Simulation can specify one or many Workflows that will be run as part of your Simulation. The Workflows then generates Events and these Events are then sent somewhere. You will also need to define Producers that are used to send the Events generated by your Workflows to some destination. These destinations could be a log file, or something more complicated like a Kafka Queue.

You define the configuration for the json-data-generator using two configuration files. The first is a Simulation Config. The Simulation Config defines the Workflows that should be run and different Producers that events should be sent to. The second is a Workflow configuration (of which you can have multiple). The Workflow defines the frequency of Events and Steps that the Workflow uses to generate the Events. It is the Workflow that defines the format and content of your Events as well.

For our example, we are going to pretend that we have a programmable [Jackie Chan](#) robot. We can command Jackie Chan though a programmable interface that happens to take json as an input via a Kafka queue and you can command him to perform different fighting moves in different martial arts styles. A Jackie Chan command might look like this:

```
{  
  "timestamp": "2015-05-20T22:05:44.789Z",  
  "style": "DRUNKEN_BOXING",  
  "action": "PUNCH",  
  "weapon": "CHAIR",  
  "target": "ARMS",  
  "strength": 8.3433  
}
```

[view raw example](#) [JackieChanCommand.json](#) hosted with [GitHub](#)

Now, we want to have some fun with our awesome Jackie Chan robot, so we are going to make him do random moves using our json-data-generator! First we need to define a Simulation Config and then a Workflow that Jackie will use.

SIMULATION CONFIG

Let's take a look at our example Simulation Config:

```
{  
  "workflows": [ {  
    "workflowName": "jackieChan",  
    "workflowFilename": "jackieChanWorkflow.json"  
  },  
  "producers": [ {  
    "type": "kafka",  
    "broker.server": "192.168.59.103",  
    "broker.port": 9092,  
    "topic": "jackieChanCommand",  
    "flatten": false,  
    "sync": false  
  }]  
}
```

{

[view rawjackieChanSimConfig.json](#) hosted with by [GitHub](#)

As you can see, there are two main parts to the Simulation Config. The Workflows name and list the workflow configurations you want to use. The Producers are where the Generator will send the events to. At the time of writing this, we have three supported Producers:

- A Logger that sends events to log files
- A [Kafka](#) Producer that will send events to your specified Kafka Broker
- A [Tranquility](#) Producer that will send events to a [Druid](#) cluster.

You can find the full configuration options for each on the [github](#) page. We used a Kafka producer because that is how you command our Jackie Chan robot.

WORKFLOW CONFIG

The Simulation Config above specifies that it will use a Workflow called jackieChanWorkflow.json. This is where the meat of your configuration would live. Let's take a look at the example Workflow config and see how we are going to control Jackie Chan:

{

```
"eventFrequency": 400,
```

```
"varyEventFrequency": true,  
"repeatWorkflow": true,  
"timeBetweenRepeat": 1500,  
"varyRepeatFrequency": true,  
"steps": [  
    "config": [{}  
        "timestamp": "now()",  
        "style": "random('KUNG_FU','WUSHU','DRUNKEN_BOXING')",  
        "action": "random('KICK','PUNCH','BLOCK','JUMP')",  
        "weapon": "random('BROAD_SWORD','STAFF','CHAIR','ROPE')",  
        "target": "random('HEAD','BODY','LEGS','ARMS')",  
        "strength": "double(1.0,10.0)"  
    }  
],  
"duration": 0  
}]
```

```
}
```

[view rawjackieChanWorkflow.json](#) hosted with [GitHub](#)

The Workflow defines many things that are all defined on the github page, but here is a summary:

- At the top are the properties that define how often events should be generated and if / when this workflow should be repeated. So this is like saying we want Jackie Chan to do a martial arts move every 400 milliseconds (he's FAST!), then take a break for 1.5 seconds, and do another one.
- Next, are the Steps that this Workflow defines. Each Step has a config and a duration. The duration specifies how long to run this step. The config is where it gets interesting!

WORKFLOW STEP CONFIG

The Step Config is your specific definition of a json event. This can be any kind of json object you want. In our example, we want to generate a Jackie Chan command message that will be sent to his control unit via Kafka. So we define the command message in our config, and since we want this to be fun, we are going to randomly generate what kind of style, move, weapon, and target he will use.

You'll notice that the values for each of the object properties look a bit funny. These are special Functions that we have created that allow us to generate values for each of the properties. For instance, the "random('KICK','PUNCH','BLOCK','JUMP')" function will randomly choose one of the values and output it as the value of the "action" property in the

command message. The “now()” function will output the current date in an ISO8601 date formatted string. The “double(1.0,10.0)” will generate a random double between 1 and 10 to determine the strength of the action that Jackie Chan will perform. If we wanted to, we could make Jackie Chan perform combo moves by defining a number of Steps that will be executed in order.

There are many more Functions available in the generator with everything from random string generation, counters, random number generation, dates, and even support for randomly generating arrays of data. We also support the ability to reference other randomly generated values. For more info, please check out the [full documentation](#) on the github page.

Once we have defined the Workflow, we can run it using the json-data-generator. To do this, do the following:

1. If you have not already, go ahead and [download the most recent release](#) of the json-data-generator.
2. Unpack the file you downloaded to a directory.

```
(tar -xvf json-data-generator-1.4.0-bin.tar -C /apps )
```

3. Copy your custom configs into the conf directory
4. Then run the generator like so:
 1. java -jar json-data-generator-1.4.0.jar jackieChanSimConfig.json

You will see logging in your console showing the events as they are being generated. The jackieChanSimConfig.json generates events like these:

```
{"timestamp":"2015-05-20T22:21:18.036Z","style":"WUSHU","action":"BLOCK","weapon":"CHAIR","target":"BODY","strength":4.7912}  
{"timestamp":"2015-05-20T22:21:19.247Z","style":"DRUNKEN_BOXING","action":"PUNCH","weapon":"BROAD_SWORD","target":"ARMS","strength":3.0248}  
{"timestamp":"2015-05-20T22:21:20.947Z","style":"DRUNKEN_BOXING","action":"BLOCK","weapon":"ROPE","target":"HEAD","strength":6.7571}  
{"timestamp":"2015-05-20T22:21:22.715Z","style":"WUSHU","action":"KICK","weapon":"BROAD_SWORD","target":"ARMS","strength":9.2062}  
{"timestamp":"2015-05-20T22:21:23.852Z","style":"KUNG_FU","action":"PUNCH","weapon":"BROAD_SWOR D","target":"HEAD","strength":4.6202}  
{"timestamp":"2015-05-20T22:21:25.195Z","style":"KUNG_FU","action":"JUMP","weapon":"ROPE","target":"ARMS","strength":7.5303}
```

```
{"timestamp":"2015-05-20T22:21:26.492Z","style":"DRUNKEN_BOXING","action":"PUNCH","weapon":"STAFF","target":"HEAD","strength":1.1247}  
{"timestamp":"2015-05-20T22:21:28.042Z","style":"WUSHU","action":"BLOCK","weapon":"STAFF","target":"ARMS","strength":5.5976}  
{"timestamp":"2015-05-20T22:21:29.422Z","style":"KUNG_FU","action":"BLOCK","weapon":"ROPE","target":"ARMS","strength":2.152}  
{"timestamp":"2015-05-20T22:21:30.782Z","style":"DRUNKEN_BOXING","action":"BLOCK","weapon":"STAFF","target":"ARMS","strength":6.2686}  
{"timestamp":"2015-05-20T22:21:32.128Z","style":"KUNG_FU","action":"KICK","weapon":"BROAD_SWORD","target":"BODY","strength":2.3534}
```

[view rawjackieChanCommands.json](#) hosted with [GitHub](#)

If you specified to repeat your Workflow, then the generator will continue to output events and send them to your Producer simulating a real world client, or in our case, continue to make Jackie Chan show off his awesome skills. If you also had a Chuck Norris robot, you could add another Workflow config to your Simulation and have the two robots fight it

out! Just another example of how you can use the generator to simulate real world situations.

IV. Configuration

```
export KSQL_HEAP_OPTS="-Xms1G -Xmx2G"
```

V. Resources

<https://developer.ibm.com/hadoop/2017/04/10/kafka-security-mechanism-saslplain/>

<https://sharebigdata.wordpress.com/2018/01/21/implementing-sasl-plain/>

<https://developer.ibm.com/code/howtos/kafka-authn-authz>

<https://github.com/confluentinc/kafka-streams-examples/tree/4.1.x/>

<https://github.com/spring-cloud/spring-cloud-stream-samples/blob/master/kafka-streams-samples/kafka-streams-table-join/src/main/java/kafka/streams/table/join/KafkaStreamsTableJoin.java>