Section 3: ksqlDB and KSQL Setup

Lecture 5: KSQL Setup - Mac / Linux / Windows

- install Java (`sudo apt install openjdk-11-jre-headless`)
- check version (`java -version`)
- download Confluent platform from https://www.confluent.io/download/
- unzip and untar (`tar xfz confluent-6*.tar.gz`)
- delete the download (`rm confluent-6*.tar.gz`)
- move to /opt (`sudo mv confluent-6* /opt`)
- create symbolic link (`cd /opt; sudo ln -s confluent-6* confluent`)
- set environment variable for the Confluent Platform directory (`export CONFLUENT_HOME=/opt/confluent`)
- add to path (`export PATH=${PATH}:/opt/confluent/bin`)
- add these same lines to your login profile (end of `~/.bashrc`)

```
export PATH=${PATH}:/opt/confluent/bin
export CONFLUENT_HOME=/opt/confluent
```

Confluent CLI changes from version 5.3 and 6.0

The *Confluent CLI* changed significantly in version 5.3 and 6.0. The most important change is the inclusion of the `local` parameter when interacting with a local development environment. From version 6.0 you'll need to include the `services` parameter

For example, to start the ksql-server

- Prior to 5.3 : `confluent start ksql-server`
- From 5.3 : `confluent local start ksql-server`
- From 6.0 : `confluent local services ksql-server start`

Lecture 7: KSQL Command Line

Create a topic

```
kafka-topics --bootstrap-server localhost:9092 --create --partitions 1 --
replication-factor 1 --topic USERS
```

Get started - KSQL Command Line

At UNIX prompt

```
kafka-console-producer --broker-list localhost:9092 --topic USERS << EOF
Alice,US
Bob,GB
Carol,AU
Dan,US
EOF
```

At KSQL prompt

```
show topics;

-- this will show nothing
print 'USERS';

print 'USERS' from beginning;

print 'USERS' from beginning limit 2;

print 'USERS' from beginning interval 2 limit 2 ;
```

Section 4: ksqlDB and KSQL Basics

Lecture 8: Our first KSQL Stream

Get started - Create a stream with CSV

Special note for 5.4 onwards (ksqlDB)

There are two categories of queries :

- **Push queries**: query the state of the system in motion and continue to output results until they meet a LIMIT condition or are terminated by the user. This was the default behavior in older versions of KSQL. 'EMIT CHANGES' is used to to indicate a query is a push query.
- **Pull queries**: query the current state of the system, return a result, and terminate. Use this to select a result as of "now". New from 5.4. KSQL currently only supports pull queries on materialized aggregate tables (sometimes refered to as *materialized views*). i.e. those created by a 'CREATE TABLE AS SELECT , <aggregate_functions> FROM GROUP BY ' style statement. A query must using a predicate against `ROWKEY`

At KSQL prompt

```
create stream users_stream (name VARCHAR, countrycode VARCHAR) WITH
(KAFKA_TOPIC='USERS', VALUE_FORMAT='DELIMITED');

list streams;
```

```
-- nothing will get shown
select name, countrycode  from users_stream emit changes;
```

`auto.offset.reset` - *Determines what to do when there is no initial offset in Apache Kafka or if the current offset does not exist on the server. The default value in KSQL is latest, which means all Kafka topics are read from the latest available offset. For example, to change it to earliest by using the KSQL command line:*

```
-- default to beginning of time
SET 'auto.offset.reset'='earliest';

-- now will see something
select name, countrycode  from users_stream emit changes;

-- stop after 4 records
select name, countrycode  from users_stream emit changes limit 4;

-- basic aggregate
select countrycode, count(*) from users_stream group by countrycode emit changes;

drop stream if exists users_stream delete topic;

list streams;

show topics;
```

Lecture 9: Create a Stream with JSON

At UNIX prompt

```
kafka-topics --bootstrap-server localhost:9092 --create --partitions 1 --
replication-factor 1 --topic USERPROFILE

kafka-console-producer --broker-list localhost:9092 --topic USERPROFILE << EOF
{"userid": 1000, "firstname":"Alison", "lastname":"Smith", "countrycode":"GB",
"rating":4.7}
EOF

kafka-console-producer --broker-list localhost:9092 --topic USERPROFILE << EOF
{"userid": 1001, "firstname":"Bob", "lastname":"Smith", "countrycode":"US",
"rating":4.2}
EOF
```

At KSQL prompt

```
CREATE STREAM userprofile (userid INT, firstname VARCHAR, lastname VARCHAR,
countrycode VARCHAR, rating DOUBLE) \
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'USERPROFILE');
```

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

select firstname, lastname, countrycode, rating from userprofile emit changes;

Alison | Smith | GB | 4.7
```

Lecture 10: KSQL Datagen - Generating Streams

At UNIX prompt

```
ksql-datagen schema=./datagen/userprofile.avro format=json topic=USERPROFILE
key=userid msgRate=1 iterations=100
```

At KSQL prompt

```
-- Review a stream - every 5th row
print 'USERPROFILE' interval 5;
```

Lecture 11: Manipulate a Stream

At KSQL prompt

```
ksql> describe userprofile;

Name                 : USERPROFILE
 Field       | Type
----------------------------------------
 ROWTIME     | BIGINT           (system)  <-- NOTE
 ROWKEY      | VARCHAR(STRING)  (system)  <-- NOTE
 USERID      | INTEGER
 FIRSTNAME   | VARCHAR(STRING)
 LASTNAME    | VARCHAR(STRING)
 COUNTRYCODE | VARCHAR(STRING)
 RATING      | DOUBLE


select rowtime, firstname from userprofile emit changes;
```

- Review *Scalar functions* at https://docs.confluent.io/current/ksql/docs/developer-guide/syntax-reference.html#scalar-functions

```
select  TIMESTAMPTOSTRING(rowtime, 'dd/MMM HH:mm') as createtime, firstname + ' '
+ ucase(lastname)  as full_name
from userprofile emit changes;
```

Lecture 12: Streams from streams and functions

Create a stream from a stream At KSQL prompt

```
select firstname + ' '
+ ucase( lastname)
+ ' from ' + countrycode
+ ' has a rating of ' + cast(rating as varchar) + ' stars. '
+ case when rating < 2.5 then 'Poor'
       when rating between 2.5 and 4.2 then 'Good'
       else 'Excellent'
   end as description
from userprofile emit changes;

Bob FAWCETT from IN has a rating of 4.4 stars. | Excellent
Heidi COEN from US has a rating of 4.9 stars. | Excellent
Bob FAWCETT from IN has a rating of 2.2 stars. | Poor
```

At KSQL prompt

Review the script `user_profile_pretty.ksql`

```
list streams;

run script 'user_profile_pretty.ksql';

list streams;

describe extended user_profile_pretty;

select description from user_profile_pretty emit changes;

drop stream user_profile_pretty;

terminate CSAS_USER_PROFILE_PRETTY_0;

drop stream user_profile_pretty;

list streams;
```

```
drop stream IF EXISTS user_profile_pretty DELETE TOPIC;
```

Lecture 13: ksqlDB Tables

Create a table

At UNIX prompt

```
kafka-topics --bootstrap-server localhost:9092 --create --partitions 1 --
replication-factor 1 --topic COUNTRY-CSV

-- version 5.5 and later
kafka-console-producer --broker-list localhost:9092 --topic COUNTRY-CSV --property
"parse.key=true"  --property "key.separator=:" << EOF
AU:Australia
IN:India
GB:UK
US:United States
EOF
```

At KSQL prompt

```
-- version 5.5 and later
CREATE TABLE COUNTRYTABLE  (countrycode VARCHAR PRIMARY KEY, countryname VARCHAR)
WITH (KAFKA_TOPIC='COUNTRY-CSV', VALUE_FORMAT='DELIMITED');

show tables;

describe COUNTRYTABLE;

describe extended COUNTRYTABLE;

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

select countrycode, countryname from countrytable emit changes;

-- Note the countryname is "UK"
select countrycode, countryname from countrytable where countrycode='GB' emit
changes limit 1;

-- This does not exist
select countrycode, countryname from countrytable where countrycode='FR' emit
changes;
```

Update a table

One record updated (UK->United Kingdom), one record added (FR)

At UNIX prompt

```
kafka-console-producer --broker-list localhost:9092 --topic COUNTRY-CSV --property
"parse.key=true"  --property "key.separator=:" << EOF
GB:United Kingdom
FR:France
EOF
```

At KSQL prompt

```
select countrycode, countryname from countrytable emit changes;

-- Note the countryname has changed to "United Kingdom"
select countrycode, countryname from countrytable where countrycode='GB' emit
changes limit 1;

-- And now appears
select countrycode, countryname from countrytable where countrycode='FR' emit
changes;
```

Section 5: ksqlDB and KSQL Intermediate

Lecture 14: KSQL Joins

Join user stream to country table

At KSQL prompt

```
select up.firstname, up.lastname, up.countrycode, ct.countryname
from USERPROFILE up
left join COUNTRYTABLE ct on ct.countrycode=up.countrycode emit changes;

create stream up_joined as
select up.firstname
+ ' ' + ucase(up.lastname)
+ ' from ' + ct.countryname
+ ' has a rating of ' + cast(rating as varchar) + ' stars.' as description
, up.countrycode
from USERPROFILE up
left join COUNTRYTABLE ct on ct.countrycode=up.countrycode;

select description from up_joined emit changes;
```

Lecture 15: Pull Queries

Pull Queries

Pull Query - new in ksqlDB (5.4 onwards)

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

-- from 5.5 onwards
CREATE STREAM driverLocations (driverId VARCHAR KEY, countrycode VARCHAR, city
VARCHAR, driverName VARCHAR)
  WITH (kafka_topic='driverlocations', value_format='json', partitions=1);

INSERT INTO driverLocations (driverId, countrycode, city, driverName) VALUES ('1',
'AU', 'Sydney', 'Alice');
INSERT INTO driverLocations (driverId, countrycode, city, driverName) VALUES ('2',
'AU', 'Melbourne', 'Bob');
INSERT INTO driverLocations (driverId, countrycode, city, driverName) VALUES ('3',
'GB', 'London', 'Carole');
INSERT INTO driverLocations (driverId, countrycode, city, driverName) VALUES ('4',
'US', 'New York', 'Derek');
```

```
create table countryDrivers as select countrycode, count(*) as numDrivers from
driverLocations group by countrycode;
```

```
-- note: as a pull query we don't use "emit"
select countrycode, numdrivers from countryDrivers where countrycode='AU';

INSERT INTO driverLocations (driverId, countrycode, city, driverName) VALUES ('5',
'AU', 'Sydney', 'Emma');


-- note: as a pull query we don't use "emit"
select countrycode, numdrivers from countryDrivers where countrycode='AU';
```

Lecture 16: Kafka Connect with ksqlDB

Kafka Connect with ksqlDB. You will be running this example using docker. First we need to stop the local Confluent platform

```
confluent local services stop
```

Now, start Postgres and Confluent platform together using docker

```
docker-compose up -d
```

Start ksqlDB KSQL CLI

```
docker-compose exec  ksqldb-cli ksql http://ksqldb-server:8088
```

Kafka Connect

```
cat postgres-setup.sql

docker-compose exec postgres psql -U postgres -f /postgres-setup.sql
```

To look at the Postgres table

```
docker-compose exec postgres psql -U postgres -c "select * from carusers;"
```

```
CREATE SOURCE CONNECTOR postgres_jdbc_source WITH(
  "connector.class"='io.confluent.connect.jdbc.JdbcSourceConnector',
  "connection.url"='jdbc:postgresql://postgres:5432/postgres',
  "mode"='incrementing',
  "incrementing.column.name"='ref',
  "table.whitelist"='carusers',
  "connection.password"='password',
  "connection.user"='postgres',
  "topic.prefix"='db-',
  "key"='username');
```

```
print 'db-carusers' from beginning;
```

In another window, insert a new database row

```
docker exec -it postgres psql -U postgres -c "INSERT INTO carusers (username)
VALUES ('Derek');"
```

Lecture 17: Data Encodings

Data Formats

Imagine a *complaints* stream of unhappy customers. Explore the different data formats (CSV, JSON, AVRO)

Column	AVRO Type	KSQL Type

Column	AVRO Type	KSQL Type
customer_name	string	VARCHAR
complaint_type	string	VARCHAR
trip_cost	float	DOUBLE
new_customer	boolean	BOOLEAN

Lecture 18: CSV Delimited Data

At UNIX prompt

```
kafka-topics --bootstrap-server localhost:9092 --create --partitions 1 --
replication-factor 1 --topic COMPLAINTS_CSV

kafka-console-producer --broker-list localhost:9092 --topic COMPLAINTS_CSV << EOF
Alice, Late arrival, 43.10, true
EOF
```

At KSQL prompt

```
CREATE STREAM complaints_csv (customer_name VARCHAR, complaint_type VARCHAR,
trip_cost DOUBLE, new_customer BOOLEAN) \
  WITH (VALUE_FORMAT = 'DELIMITED', KAFKA_TOPIC = 'COMPLAINTS_CSV');

select * from complaints_csv emit changes;
```

CSV - experience with bad data

At UNIX prompt

```
kafka-console-producer --broker-list localhost:9092 --topic COMPLAINTS_CSV << EOF
Alice, Bob and Carole, Bad driver, 43.10, true
EOF
```

Lecture 19: JSON Data

JSON - At UNIX prompt

```
kafka-topics --bootstrap-server localhost:9092 --create --partitions 1 --
replication-factor 1 --topic COMPLAINTS_JSON

kafka-console-producer --broker-list localhost:9092 --topic COMPLAINTS_JSON << EOF
```

```
{"customer_name":"Alice, Bob and Carole", "complaint_type":"Bad driver",
"trip_cost": 22.40, "new_customer": true}
EOF
```

At KSQL prompt

```
CREATE STREAM complaints_json (customer_name VARCHAR, complaint_type VARCHAR,
trip_cost DOUBLE, new_customer BOOLEAN) \
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'COMPLAINTS_JSON');

select * from complaints_json emit changes;
```

JSON - experience with bad data

At UNIX prompt

```
kafka-console-producer --broker-list localhost:9092 --topic COMPLAINTS_JSON << EOF
{"customer_name":"Bad Data", "complaint_type":"Bad driver", "trip_cost": 22.40,
"new_customer": ShouldBeABoolean}
EOF
```

Review the KSQL Server logs `confluent local services ksql-server log`

Now look at the *KSQL Server log*. We can see bad data is noticed; but hidden in a conversion error message

```
  at [Source: (byte[])"{"customer_name":"Bad Data", "complaint_type":"Bad driver",
"trip_cost": 22.40, "new_customer": ShouldBeABoolean}"; line: 1, column: 105]
 Caused by: com.fasterxml.jackson.core.JsonParseException: Unrecognized token
'ShouldBeABoolean': was expecting ('true', 'false' or 'null')
```

Lecture 20: Avro Data

At UNIX prompt

```
kafka-topics --bootstrap-server localhost:9092 --create --partitions 1 --
replication-factor 1 --topic COMPLAINTS_AVRO

kafka-avro-console-producer  --broker-list localhost:9092 --topic COMPLAINTS_AVRO
\
--property value.schema='
{
  "type": "record",
  "name": "myrecord",
  "fields": [
      {"name": "customer_name",  "type": "string" }
```

```
    , {"name": "complaint_type", "type": "string" }
    , {"name": "trip_cost", "type": "float" }
    , {"name": "new_customer", "type": "boolean"}
  ]
}' << EOF
{"customer_name":"Carol", "complaint_type":"Late arrival", "trip_cost": 19.60,
"new_customer": false}
EOF
```

At KSQL prompt

```
-- Note no columns or data type specified
create stream complaints_avro with (kafka_topic='COMPLAINTS_AVRO',
value_format='AVRO');

describe extended complaints_avro;
```

AVRO - experience with bad data

At UNIX prompt - note bad data is noted at serialization time

```
kafka-avro-console-producer  --broker-list localhost:9092 --topic COMPLAINTS_AVRO
\
--property value.schema='
{
  "type": "record",
  "name": "myrecord",
  "fields": [
      {"name": "customer_name",  "type": "string" }
    , {"name": "complaint_type", "type": "string" }
    , {"name": "trip_cost", "type": "float" }
    , {"name": "new_customer", "type": "boolean"}
  ]
}' << EOF
{"customer_name":"Bad Data", "complaint_type":"Bad driver", "trip_cost": 22.40,
"new_customer": ShouldBeABoolean}
EOF
```

Lecture 21: Avro Schema Evolution

At UNIX prompt

```
# Optional : strart Confluent Control Center
confluent local services start

curl -s -X GET http://localhost:8081/subjects/COMPLAINTS_AVRO-value/versions
```

```
kafka-avro-console-producer  --broker-list localhost:9092 --topic COMPLAINTS_AVRO
\
--property value.schema='
{
  "type": "record",
  "name": "myrecord",
  "fields": [
      {"name": "customer_name",  "type": "string" }
    , {"name": "complaint_type", "type": "string" }
    , {"name": "trip_cost", "type": "float" }
    , {"name": "new_customer", "type": "boolean"}
    , {"name": "number_of_rides", "type": "int", "default" : 1}
  ]
}' << EOF
{"customer_name":"Ed", "complaint_type":"Dirty car", "trip_cost": 29.10,
"new_customer": false, "number_of_rides": 22}
EOF

curl -s -X GET http://localhost:8081/subjects/COMPLAINTS_AVRO-value/versions

curl -s -X GET http://localhost:8081/subjects/COMPLAINTS_AVRO-value/versions/1 |
jq '.'

curl -s -X GET http://localhost:8081/subjects/COMPLAINTS_AVRO-value/versions/2 |
jq '.'
```

At KSQL prompt

```
ksql> describe complaints_avro;

Name                 : COMPLAINTS_AVRO
 Field          | Type
-------------------------------------------
 ROWTIME        | BIGINT           (system)
 ROWKEY         | VARCHAR(STRING)  (system)
 CUSTOMER_NAME  | VARCHAR(STRING)
 COMPLAINT_TYPE | VARCHAR(STRING)
 TRIP_COST      | DOUBLE
 NEW_CUSTOMER   | BOOLEAN
-------------------------------------------


ksql> create stream complaints_avro_v2 with (kafka_topic='COMPLAINTS_AVRO',
value_format='AVRO');

ksql> describe complaints_avro_v2;

Name                 : COMPLAINTS_AVRO_V2
 Field           | Type
-------------------------------------------
 ROWTIME         | BIGINT           (system)
```

```
 ROWKEY          | VARCHAR(STRING)  (system)
 CUSTOMER_NAME   | VARCHAR(STRING)
 COMPLAINT_TYPE  | VARCHAR(STRING)
 TRIP_COST       | DOUBLE
 NEW_CUSTOMER    | BOOLEAN
 NUMBER_OF_RIDES | INTEGER                       <-- *** NOTE new column
-------------------------------------------
```

Lecture 22: Nested JSON

Imagine we have data like this

```
{
  "city": {
    "name": "Sydney",
    "country": "AU",
    "latitude": -33.8688,
    "longitude": 151.2093
  },
  "description": "light rain",
  "clouds": 92,
  "deg": 26,
  "humidity": 94,
  "pressure": 1025.12,
  "rain": 1.25
}
```

At UNIX prompt

```
kafka-topics --bootstrap-server localhost:9092 --create --partitions 1 --
replication-factor 1 --topic WEATHERNESTED

cat demo-weather.json | kafka-console-producer --broker-list localhost:9092 --
topic WEATHERNESTED
```

Extract like this - At KSQL prompt

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


CREATE STREAM weather
      (city STRUCT <name VARCHAR, country VARCHAR, latitude DOUBLE, longitude
DOUBLE>,
       description VARCHAR,
       clouds BIGINT,
       deg BIGINT,
```

```
        humidity BIGINT,
        pressure DOUBLE,
        rain DOUBLE)
WITH (KAFKA_TOPIC='WEATHERNESTED', VALUE_FORMAT='JSON');

SELECT city->name AS city_name, city->country AS city_country, city->latitude as
latitude, city->longitude as longitude, description, rain from weather emit
changes;
```

Lecture 23: Build a rekeyed table

- create a table based on rekeyed `city` field from `weather` stream
- At KSQL prompt

```
create stream weatherraw with (value_format='AVRO') as SELECT city->name AS
city_name, city->country AS city_country, city->latitude as latitude, city-
>longitude as longitude, description, rain from weather ;

list streams;
-- note AVRO

describe extended weatherraw;
```

Now notice the *Key field*

```
ksql> describe extended weatherraw;
>

Name                 : WEATHERRAW
Type                 : STREAM
Key field            :                <- *** NOTE BLANK ***
Key format           : STRING
Timestamp field      : Not set - using <ROWTIME>
Value format         : AVRO
Kafka topic          : WEATHERRAW (partitions: 4, replication: 1)
```

```
create stream weatherrekeyed as select * from weatherraw partition by city_name;

describe extended weatherrekeyed;
```

Now notice the *Key field*

```
ksql> describe extended weatherrekeyed;
>
```

```
Name                 : WEATHERREKEYED
Type                 : STREAM
Key field            : CITY_NAME    <- ***  Keyed on city ***
Key format           : STRING
Timestamp field      : Not set - using <ROWTIME>
Value format         : AVRO
Kafka topic          : WEATHERREKEYED (partitions: 4, replication: 1)
```

```
-- prior to 5.5
create table weathernow with (kafka_topic='WEATHERREKEYED', value_format='AVRO',
key='CITY_NAME');

-- from 5.5 onwards
create table weathernow (city_name varchar primary key, city_country varchar,
latitude double, longitude double, description varchar, rain double) with
(kafka_topic='WEATHERREKEYED', value_format='AVRO');


select * from weathernow emit changes;

select * from weathernow where city_name = 'San Diego' emit changes;
```

Let's make it sunny! At UNIX prompt

```
cat demo-weather-changes.json | kafka-console-producer --broker-list
localhost:9092 --topic WEATHERNESTED
```

At KSQL prompt

```
select * from weathernow where city_name = 'San Diego' emit changes;
```

Lecture 24: Repartition a Stream

When you use KSQL to join streaming data, you must ensure that your streams and tables are co-partitioned, which means that input records on both sides of the join have the same configuration settings for partitions.

At UNIX prompt

```
kafka-topics --bootstrap-server localhost:9092 --create --partitions 2 --
replication-factor 1 --topic DRIVER_PROFILE

kafka-console-producer --broker-list localhost:9092 --topic DRIVER_PROFILE << EOF
```

```
{"driver_name":"Mr. Speedy", "countrycode":"AU", "rating":2.4}
EOF
```

At KSQL prompt

```
CREATE STREAM DRIVER_PROFILE (driver_name VARCHAR, countrycode VARCHAR, rating
DOUBLE)
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'DRIVER_PROFILE');



select dp.driver_name, ct.countryname, dp.rating
from DRIVER_PROFILE dp
left join COUNTRYTABLE ct on ct.countrycode=dp.countrycode emit changes;

Can't join DRIVER_PROFILE with COUNTRYTABLE since the number of partitions don't
match. DRIVER_PROFILE partitions = 2; COUNTRYTABLE partitions = 1. Please
repartition either one so that the number of partitions match.
```

We can fix this by co-partitioning, use the PARTITION BY clause. At KSQL prompt

```
create stream driverprofile_rekeyed with (partitions=1) as select * from
DRIVER_PROFILE partition by driver_name;

select dp2.driver_name, ct.countryname, dp2.rating
from DRIVERPROFILE_REKEYED dp2
left join COUNTRYTABLE ct on ct.countrycode=dp2.countrycode emit changes;
```

Lecture 25: Merging Streams

Merging Streams; Concat Topics with INSERT

- create stream of requested rides in Europe using data gen
- create stream of requested rides in USA using data gen
- combine into single stream of all requested rides using *INSERT*

At UNIX prompt

```
ksql-datagen schema=./datagen/riderequest-europe.avro  format=avro
topic=riderequest-europe key=rideid msgRate=1 iterations=100

ksql-datagen schema=./datagen/riderequest-america.avro format=avro
topic=riderequest-america key=rideid msgRate=1 iterations=100
```

At KSQL prompt

```
create stream rr_america_raw with (kafka_topic='riderequest-america',
value_format='avro');

create stream rr_europe_raw with (kafka_topic='riderequest-europe',
value_format='avro');

select * from rr_america_raw emit changes;

select * from rr_europe_raw emit changes;

create stream rr_world as select 'Europe' as data_source, * from rr_europe_raw;

insert into rr_world      select 'Americas' as data_source, * from rr_america_raw;

select * from rr_world emit changes;
```

Lecture 26: Windowing

- how many requests are arriving each time period
- At KSQL prompt

```
select data_source, city_name, count(*)
from rr_world
window tumbling (size 60 seconds)
group by data_source, city_name emit changes;
```

```
select data_source, city_name, COLLECT_LIST(user)
from rr_world
window tumbling (size 60 seconds)
group by data_source, city_name emit changes;
```

```
select data_source, city_name, COLLECT_LIST(user)
from rr_world WINDOW SESSION (60 SECONDS)
group by data_source, city_name emit changes;

select TIMESTAMPTOSTRING(WindowStart, 'HH:mm:ss')
, TIMESTAMPTOSTRING(WindowEnd, 'HH:mm:ss')
, data_source
, TOPK(city_name, 3)
, count(*)
FROM rr_world
WINDOW TUMBLING (SIZE 1 minute)
group by data_source
emit changes;
```

Lecture 27: Geospatial

- create stream - distance of car to waiting rider
- At KSQL prompt

```
select * from rr_world emit changes;

describe rr_world;

create stream requested_journey as
select rr.latitude as from_latitude
, rr.longitude as from_longitude
, rr.user
, rr.city_name as city_name
, w.city_country
, w.latitude as to_latitude
, w.longitude as to_longitude
, w.description as weather_description
, w.rain
from rr_world rr
left join weathernow w on rr.city_name = w.city_name;


create stream ridetodest as
select user
, city_name
, city_country
, weather_description
, rain
, GEO_DISTANCE(from_latitude, from_longitude, to_latitude, to_longitude, 'km') as
dist
from requested_journey;
```

```
select user + ' is travelling ' + cast(round(dist) as varchar) +' km to ' +
city_name + ' where the weather is reported as ' + weather_description
from ridetodest emit changes;

Alice is at (52,0) and is travelling 215 km to Manchester where it is SUNNY
Heidi is at (51,-1) and is travelling 88 km to London where it is heavy rain
Grace is at (50,-1) and is travelling 138 km to London where it is heavy rain
```

Section 6: ksqlDB and KSQL Extensions - UDF & UDAF

Lecture 28: Extending KSQL - UDF / UDAF

UDF - Build and deploy KSQL User Defined Anomoly Functions - write a UDF to calculare drive time based on

- distance to travel
- weather conditions

Compile Code to Create Anomoly Functions

- Have a look at the file `java/src/main/java/com/vsimon/kafka/streams/TaxiWait.java`
- If you don't want to compile the code; just copy the JAR from `java/pre-compiled/ksql-udf-taxi-1.0.jar`
- Download Maven and follow the installation instructions (https://maven.apache.org/)

```
cd java
mvn clean package
ls target/ksql-udf-taxi-1.0.jar
```

Deploy KSQL User Defined Functions

Find the location of your extension directory. From KSQL

```
ksql> LIST PROPERTIES;

 Property               | Effective Value
-------------------------------------------
 . . .
 ksql.extension.dir     | ext                <--  *** Look for this
 . . .
```

```
# Stop (just the) KSQL-Server
confluent local services ksql-server stop

# Create an ext (extensions) directory in ${CONFLUENT_HOME}/ext
mkdir /opt/confluent/ext

# build ksql-udf-taxi.jar as above and copy into ext directory
cp target/ksql-udf-taxi-1.0.jar /opt/confluent/ext

# or to use the pre-compile one
cp pre-compiled/ksql-udf-taxi-1.0.jar /opt/confluent/ext

# Restart KSQL server
confluent local services ksql-server start
```

Check KSQL User Defined Functions Available

Start `ksql` client and verify

```
ksql> list functions;

 Function Name     | Type
-------------------------------
  . . .
 SUM               | AGGREGATE
 TAXI_WAIT         | SCALAR          <--- You need this one
 TIMESTAMPTOSTRING | SCALAR


ksql> DESCRIBE FUNCTION TAXI_WAIT;

Name        : TAXI_WAIT
Overview    : Return expected wait time in minutes
Type        : scalar
Jar         : /etc/ksql/ext/ksql-udf-taxi-1.0.jar
Variations  :

    Variation   : TAXI_WAIT(VARCHAR, DOUBLE)
    Returns     : DOUBLE
    Description : Given weather and distance return expected wait time in minutes
```

Lecture 29: Using the UDF / UDAF

Use the UDF

```
describe ridetodest;

select user
, round(dist) as dist
, weather_description
, round(TAXI_WAIT(weather_description, dist)) as taxi_wait_min
from ridetodest emit changes;


select user
+ ' will be waiting ' + cast(round(TAXI_WAIT(weather_description, dist)) as
varchar)
+ ' minutes for their trip of '
+ cast(round(dist) as varchar) +' km to ' + city_name
+ ' where it is ' + weather_description
from ridetodest emit changes;

Heidi will be waiting 14 minutes for their trip of 358 km to Bristol where it is
light rain
Bob will be waiting 4 minutes for their trip of 218 km to Manchester where it is
SUNNY
```

```
Frank will be waiting 15 minutes for their trip of 193 km to London where it is
heavy rain
```

Section 7: ksqlDB and KSQL in Production

Lecture 30: Moving to Productions-Headless for KSQL

Headless KSQL server cluster is *not* aware of anys streams or tables you defined in other (interactive) KSQL clusters.

```
confluent stop ksql-server

/opt/confluent/bin/ksql-server-start /opt/confluent/etc/ksql/ksql-
server.properties  --queries-file ./where-is-bob.ksql

# show CLI does not work
ksql

# check if BOB topic exists
kafka-topics --zookeeper localhost:2181 --list --topic BOB

kafka-avro-console-consumer --bootstrap-server localhost:9092 --topic BOB
```

Lecture 31: Explain Plan

Explain

```
create stream my_stream
as select firstname
from userprofile;

show queries;

explain CSAS_MY_STREAM_1;


create table my_table
as select firstname, count(*) as cnt
from userprofile
group by firstname;

show queries;

explain CTAS_MY_TABLE_0;
```

Kafka Streams Topology Visualizer

Converts an ASCII Kafka Topology description into a hand drawn diagram.

- See https://zz85.github.io/kafka-streams-viz/

Lecture 32: Scaling and Load Balancing

Multi Server with docker

```
docker-compose  -f docker-compose-prod.yml up -d
ksql-datagen schema=./datagen/userprofile.avro format=json topic=USERPROFILE
key=userid maxInterval=1000 iterations=10000
```

In KSQL

```
CREATE STREAM userprofile (userid INT, firstname VARCHAR, lastname VARCHAR,
countrycode VARCHAR, rating DOUBLE)
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'USERPROFILE');

create stream up_lastseen as
SELECT TIMESTAMPTOSTRING(rowtime, 'dd/MMM HH:mm:ss') as createtime, firstname
from userprofile;
```

```
kafka-console-consumer --bootstrap-server localhost:9092  --topic UP_LASTSEEN


docker-compose -f docker-compose-prod.yml ps

# stop 1
docker-compose -f docker-compose-prod.yml stop ksql-server-1

# re-start 1
docker-compose -f docker-compose-prod.yml start ksql-server-1

# stop 2
docker-compose -f docker-compose-prod.yml stop ksql-server-2

# stop 1
docker-compose -f docker-compose-prod.yml stop ksql-server-1

# start 2
docker-compose -f docker-compose-prod.yml start ksql-server-1

# start 1
docker-compose -f docker-compose-prod.yml start ksql-server-1
```

Lecture 33: Configuration Settings

Understanding settings

```
confluent stop
confluent destroy

cd /opt/confluent/etc/ksql
vi ksql-server.properties

# add this line anywhere in file
ksql.service.id=myservicename

confluent start ksql-server
```

Start KSQL

```
LIST PROPERTIES;

 Property                                             | Default override |
Effective Value
------------------------------------------------------------------------------------------
-----------------
 ksql.schema.registry.url                             | SERVER           |
http://localhost:8081
 ksql.streams.auto.offset.reset                       | SERVER           |
latest
 ksql.service.id                                      | SERVER           |
myservicename          <-- *** Note: this is the one we changed

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

LIST PROPERTIES;


 Property                                             | Default override |
Effective Value
------------------------------------------------------------------------------------------
------------------
 ksql.schema.registry.url                             | SERVER           |
http://localhost:8081
 ksql.streams.auto.offset.reset                       | SESSION          |
earliest                <-- *** Note both the override and Value cahnges
 ksql.service.id                                      | SERVER           |
myservicename
```

Lecture 34: State Stores

State Stores

Start KSQL

```
LIST PROPERTIES;

# Look for ksql.streams.state.dir

 Property                                                | Default override |
Effective Value
------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
------------
 ksql.streams.state.dir                                  | SERVER           |
/var/folders/1p/3whlrkzx4bs3fkd55_600x4c0000gp/T/confluent.V2kB1p2N/ksql-
server/data/kafka-streams
```

At UNIX

```
ksql-datagen schema=./datagen/userprofile.avro format=json topic=USERPROFILE
key=userid maxInterval=5000 iterations=100
```

At KSQL

```
CREATE STREAM userprofile (userid INT, firstname VARCHAR, lastname VARCHAR,
countrycode VARCHAR, rating DOUBLE) \
  WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'USERPROFILE');
```

At UNIX

```
# note: this will show nothing
find /var/folders/1p/3whlrkzx4bs3fkd55_600x4c0000gp/T/confluent.V2kB1p2N/ksql-
server/data/kafka-streams -type f
```

Run a stateful operation, which should require RocksDB to persist to disk

```
select countrycode, count(*) from userprofile group by countrycode;
```

At UNIX

```
# note: this will now show files
find /var/folders/1p/3whlrkzx4bs3fkd55_600x4c0000gp/T/confluent.V2kB1p2N/ksql-
server/data/kafka-streams -type f
```

Complex State Stores example

```
set 'ksql.sink.partitions' = '1';

kafka-topics --bootstrap-server localhost:9092 --create --partitions 1 --
replication-factor 1 --topic userrequests
kafka-topics --bootstrap-server localhost:9092 --create --partitions 1 --
replication-factor 1 --topic browsertype
kafka-topics --bootstrap-server localhost:9092 --create --partitions 1 --
replication-factor 1 --topic location
kafka-topics --bootstrap-server localhost:9092 --create --partitions 1 --
replication-factor 1 --topic cartype

create table browsertype (browser_code bigint, browsername varchar) with
(kafka_topic='browsertype', value_format='json', key='browser_code');

create table location (location_code bigint, locationname varchar) with
(kafka_topic='location', value_format='json', key='location_code');

create table cartype (car_code bigint, carname varchar) with
(kafka_topic='cartype', value_format='json', key='car_code');


create stream userrequests (browser_code bigint, location_code bigint, car_code
bigint, silly varchar) with (kafka_topic='userrequests', value_format='json');

create stream user_browser as select us.LOCATION_CODE, us.CAR_CODE, us.silly,
bt.browsername from userrequests us left join browsertype bt on
bt.browser_code=us.browser_code;


create stream user_browser_location as select ub.CAR_CODE, ub.silly,
ub.browsername, l.locationname from user_browser ub left join location l on
ub.location_code = l.location_code;

create stream user_browser_location_car as select ubl.silly, ubl.browsername,
ubl.locationname, c.carname from user_browser_location ubl left join cartype c on
ubl.CAR_CODE = c.car_code;
```

Lecture 35: Testing ksqlDB applications

Testing

```
cd testing

ksql-test-runner --sql-file ksql-statements.ksql --input-file input.json --output-
file output.json

ksql-test-runner --sql-file ksql-statements.ksql --input-file input.json --output-
```

```
file output.json  | grep  ">>>"

ksql-test-runner --sql-file ksql-statements-enhanced.ksql --input-file input.json
--output-file output.json  | grep  ">>>"
```