Setup

- install Java; check version (`java -version`)
- download Confluent platform from https://www.confluent.io/download/
- unzip and untar (`tar xvfz confluent-5*.tar.gz`)
- move to /opt
- create symbolic link (`ln -s confluent-5* confluent`)
- add to path (`export PATH=${PATH}:/opt/confluent/bin`)
- install the Confluent CLI (see https://docs.confluent.io/current/cli/installing.html#scripted-installation)
  - `curl -L https://cnfl.io/cli | sh -s -- -b /opt/confluent/bin`
- Start
  - `confluent local start ksql-server`
  - or if using Docker `docker-compose up -d`

Confluent CLI changes from version 5.3

The *Confluent CLI* changed significantly in version 5.3. The most important change is the inclusion of the `local` paramter when interacting with a local development environment.

For example, to start the ksql-server

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

Create a topic

```
kafka-topics --zookeeper localhost:2181 --create --partitions 1 --
replication-factor 1 --topic USERS
```

Or you can use the more modern syntax

```
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 ;

```

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
-- 5.3 and earlier
select name, countrycode  from users_stream;

-- 5.4 onwards
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
-- 5.3 and earlier
select name, countrycode  from users_stream;

-- 5.4 onwards
select name, countrycode  from users_stream emit changes;

-- stop after 4 records
-- 5.3 and earlier
select name, countrycode  from users_stream limit 4;

-- 5.4 onwards
select name, countrycode  from users_stream emit changes limit 4;

-- basic aggregate
-- 5.3 and earlier
select countrycode, count(*) from users_stream group by countrycode;

-- 5.4 onwards
select countrycode, count(*) from users_stream group by countrycode emit
changes;

drop stream if exists users_stream delete topic;

list streams;

show topics;
```

Create a stream with JSON

At UNIX prompt

```
kafka-topics --zookeeper localhost:2181 --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';

-- 5.3 and earlier
select firstname, lastname, countrycode, rating from userprofile;

-- 5.4 onwards
select firstname, lastname, countrycode, rating from userprofile emit
changes;

Alison | Smith | GB | 4.7
```

Manipulate a stream

Run a data gen

At UNIX prompt

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

At KSQL prompt

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

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


-- 5.3 and earlier
select rowtime, firstname from userprofile;

-- 5.4 onwards
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
from userprofile emit changes;

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

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 query CSAS_USER_PROFILE_PRETTY_4;

drop stream user_profile_pretty;

list streams;

drop stream IF EXISTS user_profile_pretty DELETE TOPIC;
```

Create a table

At UNIX prompt

```
kafka-topics --zookeeper localhost:2181 --create --partitions 1 --
replication-factor 1 --topic COUNTRY-CSV

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

At KSQL prompt

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

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:GB,United Kingdom
FR: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;
```

Join

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
from USERPROFILE up
left join COUNTRYTABLE ct on ct.countrycode=up.countrycode;

select * from up_joined emit changes;
```

Pull Queries

Pull Query - new in ksqlDB (5.4 onwards)

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

CREATE STREAM driverLocations (driverId INTEGER, countrycode VARCHAR, city
VARCHAR, driverName VARCHAR)
  WITH (kafka_topic='driverlocations', key='driverId',
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 rowkey='AU';

INSERT INTO driverLocations (driverId, countrycode, city, driverName)
VALUES (5, 'AU', 'Sydney', 'Emma');
select countrycode, numdrivers from countryDrivers where rowkey='AU';

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

Kafka Connect with ksqlDB

To run Postgres and Confluent platform

```
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');"
```

Data Formats

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

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

CSV Delimited

At UNIX prompt

```
kafka-topics --zookeeper localhost:2181 --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
```

JSON

At UNIX prompt

```
kafka-topics --zookeeper localhost:2181 --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;
```

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 log ksql-server`

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')
```

AVRO

At UNIX prompt

```
kafka-topics --zookeeper localhost:2181 --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
```

AVRO Schema Evolution

At UNIX prompt

```
# Optional : strart Confluent Control Center
confluent local 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
---------------------------------------------
```

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 --zookeeper localhost:2181 --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;
```

Build 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)
```

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

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;
```

Repartition

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 --zookeeper localhost:2181 --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;
```

Mergin 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 maxInterval=5000 iterations=100

ksql-datagen schema=./datagen/riderequest-america.avro format=avro
topic=riderequest-america key=rideid maxInterval=5000 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;
```

Windows

- 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;
```

Geospacial

- 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
```

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 stop ksql-server

# 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 start ksql-server
```

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
```

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
```