

KQL Quick Start

This page describes a step-by-step process to run queries with KQL.

This is an **internal-use-only** document and still under development!

Please keep in mind that the steps and examples below are not intended to be seen or consumed by any eventual customers, these are just steps for we Confluentians to try out some early functionality and get a very early 'feel' for KQL.

For syntax guide for KQL please refer to [this page](#).

Do note that we've been back and forth on whether to call it 'KSQL' or just 'KQL' so you'll likely see both mentioned below.

We assume you are starting fresh and have no Kafka cluster on your machine.

Step1: Clone the repo and build the project

The first step is to clone the KQL project from the confluent inc repo on git. You can find the project at <https://github.com/confluentinc/ksql>.

After cloning the repo you need to build the project.

```
$ cd ksql
$ mvn clean
install
```

Please note that, at the time of writing, KSQL depends on an unreleased SNAPSHOT build of Kafka 0.10.2 so you will either need access setup to our internal nexus repository from which maven can download that snapshot or, alternatively, just checkout and build Kafka 0.10.2 locally for yourself.

Before continuing please make sure you have **maven** and **git** installed on your machine. If you don't have either of them, you can check out [here](#) to step by step installation and set up guide for maven and git. You need to make sure that our internal nexus repository is set correctly in the [settings.xml](#) file for your maven installation as described [here](#).

****** If you prefer to just try KQL without building it you can use the executable jars from here: [kql](#), [kql-examples](#)

Note that if you use the provided jar files above you won't be able to get the latest fixes and improvements.

Step 2: Start Kafka Cluster

To run the KQL queries you need a kafka cluster. For more details on how to use kafka please refer to <https://kafka.apache.org/quickstart>

Assuming you have downloaded kafka and are in the kafka folder you can start the cluster by starting the zookeeper service and a kafka broker.

To start the zookeeper service use the following command from the KAFKA_HOME directory:

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

To start the kafka broker use the following command from the KAFKA_HOME directory:

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

Step 3: Generate sample data

Assuming you are in your KSQL project folder and successfully built the project you can now use the example data generator tool in the kql-examples module to create some sample topics and publish sample data to them. The build should have created an executable script in this module for you to use, located at kql-examples/target/kql-examples.

The data generation tool takes in a topic name, an Avro schema, the name of a field in the schema to use as a key, an output format, and an optional number of iterations. It then generates random data fitting the given schema with the given key field and writes it to the specified topic in

the requested format. You can also specify a Kafka bootstrap server for the producer; if you don't, it will default to **localhost:9092**.

The data generator tool can generate message data in three different formats:

- JSON
- AVRO
- CSV

Luckily, you don't have to worry about most of that for the purpose of the quickstart—the tool also accepts a useful *quickstart* option, where you only have to give one of "orders", "pageview", or "users", and it will generate data that matches the following topics automatically without you having to specify anything else (although any other options you specify will override the ones for the quickstart option you chose).

The three quickstart topics/schemas are:

Orders:

Default format: JSON

Default topic: orders_kafka_topic_<format> (where <format> is the lowercase name of the format used)

Schema (located in kql-examples/src/main/resources/orders_schema.avro):

Column Name	ORDERTIME	ORDERID (default key)	ITEMID	ORDERUNITS
Column Type	bigint(Long)	integer(Integer)	varchar(String)	Double

Pageview:

Default format: JSON

Default topic: pageview_kafka_topic_<format> (where <format> is the lowercase name of the format used)

Schema (located in kql-examples/src/main/resources/pageview_schema.avro):

Column Name	viewtime (default key)	userid	pageid
Column Type	bigint(Long)	varchar(String)	varchar(String)

Users:

Default format: JSON

Default topic: users_kafka_topic_<format> (where <format> is the lowercase name of the format used)

Schema (located in kql-examples/src/main/resources/users_schema.avro):

Column Name	registertime	userid (default key)	regionid	gender
Column Type	bigint(Long)	integer(Integer)	varchar(String)	varchar(String)

Assuming the KSQL folder is the current directory in the shell you could run the data generator tool with the *orders* quickstart option by using the following command:

```
$ kql-examples/target/kql-examples
quickstart=orders
```

The above command publishes 1000 messages into the *orders* topic in JSON format.

Similarly, you can use the following commands to create and populate the *users* and *pageview* topics respectively:

```
$ kql-examples/target/kql-examples quickstart=users
$ kql-examples/target/kql-examples quickstart=pageview
```

Alternatively, if you prefer, you can use each of the above commands with a different value for the 'format' parameter to generate message data in different formats. For example, generating the users preset in avro format and the pageview preset in csv format looks like this:

```
$ kql-examples/target/kql-examples quickstart=users format=avro
$ kql-examples/target/kql-examples quickstart=pageview
format=csv
```

Other examples of overriding the quickstart preset options:

```
# Orders quickstart, but with a different topic name
$ kql-examples/target/kql-examples quickstart=orders topic=special_orders

# Users quickstart, but with a different key field
$ kql-examples/target/kql-examples quickstart=users key=registertime

# Pageview quickstart, but with a different schema (be careful with this one, make sure the key field
name will be valid for the given schema)
$ kql-examples/target/kql-examples quickstart=pageview schema=my_custom_pageview_schema.avro
```

And a final example demonstrating how to specify an alternative Kafka bootstrap server:

```
# Orders quickstart, but sending messages to a non-local server
$ kql-examples/target/kql-examples quickstart=orders
bootstrap-server=162.242.144.98:9092
```

Since the data generation tool relies on the [Avro Random Generator](#) (ARG) internally for producing data to fit a given schema, if you provide a schema of your own you'll also be able to leverage all of the supported annotations that ARG provides to customize exactly what kinds of values you want to generate for your data. For examples of usage you can see the provided quickstart schemas, or check out the README for ARG's GitHub repository.

Note - you only need to run one flavor of the command for each topic! Feel free to pick whichever you prefer for each topic and make a note of your choice - you will need this information in the next step!

Note - in these examples the 'format' represents the format for message value and the format for message key will always be String.

Step 4: Start KQL CLI

Now that you have the kafka cluster up and running and have loaded some sample data it's time to start the KQL CLI. The CLI is an executable jar and so, to simplify the process of using it in the shell, we recommend you rename the executable jar to simply 'kql' first.

If you have build the project yourself, first run the following command:

```
$ cp kql-cli/target/kql
~/
```

This will copy the executable jar into the home folder. Note that you can copy the file to any folder you desire.

If you have downloaded the kql executable jar you do not need to run the above command.

Now you can simply run 'kql' to start the CLI:

```
$ ~/kql
```

Required option '--properties-file' is missing

See the -h or --help flags for usage information

Oops! We need to specify a configuration file for KQL so it knows things like where to find the running Kafka instance to connect to. Assuming your Kafka instance is running at **localhost:9092**, we can create a basic configuration file pretty easily—just create the following file named **kql.properties** in your preferred text editor:

```
#kql.properties
```

```
#Configuration file for KQL CLI
```

```
bootstrap.servers=localhost:9092
```

```
application.id=kql_quickstart
```

Now let's try this again, making sure to specify our new configuration file:

```
$ ~/kql --properties-file kql.properties
```

This will start the KQL command line interface where you can interactively issue KQL commands and queries. You should see something like this (don't mind the lack of --properties-file in the first line, [Chris](#) is just too lazy to update the screenshot):

```
Hojjat-Jafarpours-MBP:kql hojjat$ ~/kql
log4j:WARN No appenders could be found for logger (io.confluent.kql.util.KQLConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
=====
KQL (Kafka Query Language) 0.0.1
=====
kql> █
```

For a brief description of some of the available commands use the `help` command in CLI:

```

Hojjat-Jafarpours-MBP:ksl hojjat$ ~/ksl
log4j:WARN No appenders could be found for logger (io.confluent.ksl.util.KSLConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
=====
KSL (Kafka Query Language) 0.0.1
=====
ksl> help
KSL cli commands:
=====
list topics           ..... Show the list of available topics.
list streams          ..... Show the list of available streams/tables.
describe <stream/table name> ..... Show the schema of the given stream/table.
show queries          ..... Show the list of running queries.
print <topic name>     ..... Print the content of a given topic/stream.
terminate <query id>   ..... Terminate the running query with the given id.

For more information refer to www.ksl.confluent.io

ksl> █

```

Step 5: Registering Catalog Topics

Like most structured-query products, KSL relies on a *catalog* to store metadata about the structure and format of your data to enable query processing. In KSL the catalog is built around two fundamental concepts:

- **Topic:** A topic in KSL describes an actual topic in the Kafka cluster and its properties. Think of it as a "registration" of the Kafka topic, supplemented with some additional metadata, to make it usable from KSL.
- **Stream/Table:** A Stream/Table is a declaration of how the data from a KSL Topic can be parsed and queried. This is where we specify both the schema of the data in the topic and the semantics of how we want to query it ([stream / table duality](#)!). A KSL Table additionally needs to have a state-store specified in addition to the underlying KSL topic. Note that it is perfectly legal and possible to declare multiple Streams or Tables "on top of" the same underlying topic.

The following commands, entered in the KSL CLI, will create three KSL topics in your local catalog and associate them to the Kafka topics which we created with the data generator tool:

****Note** Before registering avro topics we need to have access to the corresponding avro schema file in order that KSL can deserialize the messages. You can use the schema files provided in `./ksl-examples/src/main/resources` for any of the three topics you chose to create in avro format:

```
$ cp ./ksl-examples/src/main/resources/*.avro /tmp/
```

```

ksl> CREATE TOPIC orders_topic WITH (format = 'avro',
avroschemafile='/tmp/order_schema.avro',kafka_topic='orders_kafka_topic_avro');

ksl> CREATE TOPIC users_topic WITH (format = 'json', kafka_topic='users_kafka_topic_json');

ksl> CREATE TOPIC pageview_topic WITH (format = 'json', kafka_topic='pageview_kafka_topic_json');

```

Note - Be sure to edit these commands to reflect your actual choice of CSV / JSON / AVRO when you created the topics with the generator tool earlier. As you may surmise from the patterns in the examples here, the name of the actual Kafka topic that gets created has the chosen format appended to it so it's a bit less confusing when testing 😊. Just run the regular ``kafka-topics --list`` tool if you need to remind yourself what you've actually created...

Note that if a topic is in avro format you need to provide the path for its' avro schema file in the CREATE TOPIC command. In the above example commands you can see that the `orders_topic` is in avro format so we had to provide the path for the schema file.

After creating the above topics you should be able to see and verify the list of available topics in the system using the ``list topics`` command:

```

Hojjat-Jafarpours-MBP:kql hojjat$ ~/kql
log4j:WARN No appenders could be found for logger (io.confluent.kql.util.KQLConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
=====
KQL (Kafka Query Language) 0.0.1
=====
kql> help
KQL cli commands:
=====
list topics           ..... Show the list of available topics.
list streams         ..... Show the list of available streams/tables.
describe <stream/table name> ..... Show the schema of the given stream/table.
show queries         ..... Show the list of running queries.
print <topic name>    ..... Print the content of a given topic/stream.
terminate <query id>  ..... Terminate the running query with the given id.

For more information refer to www.kql.confluent.io

kql> CREATE TOPIC orders_topic WITH (format = 'avro', avroschemafile='/Users/hojjat/avro_order_schema.avro',kafka_topic='orders_kafka_topic_avro');
kql> CREATE TOPIC users_topic WITH (format = 'json', kafka_topic='users_kafka_topic_json');
kql> CREATE TOPIC pageview_topic WITH (format = 'json', kafka_topic='pageview_kafka_topic_json');
kql> list topics

```

KQL Topic	Corresponding Kafka Topic	Topic Format	Notes
PAGEVIEW_TOPIC	pageview_kafka_topic_json	JSON	
ORDERS_TOPIC	orders_kafka_topic_avro	AVRO	
USERS_TOPIC	users_kafka_topic_json	JSON	Avro schema path: /Users/hojjat/avro_order_schema.avro

```

( 3 rows)
kql>

```

Step 6: Creating Streams and Tables

Now we can go ahead and create streams and tables for our newly created topics. We can use the `CREATE STREAM` and `CREATE TABLE` statements as follows:

```
kql> CREATE STREAM orders (ordertime bigint, orderid varchar, itemid varchar, orderunits double) WITH (topicname = 'orders_topic' , key='ordertime');
```

```
kql> CREATE STREAM pageview (viewtime bigint, userid varchar, pageid varchar) WITH (topicname = 'pageview_topic',key='viewtime');
```

```
kql> CREATE TABLE users (usertime bigint, userid varchar, regionid varchar, gender varchar) WITH (topicname = 'users_topic', key='userid', statestore='user_statestore');
```

Note that you need to know the schema of the data in the topic to create streams/tables. Also note that you need to specify the `statestore` property for CREATE TABLE statement.

You can see the created streams/tables in the catalog using the `list streams` command:

```

Hojjat-Jafarpours-MBP:kql hojjat$ ~/kql
log4j:WARN No appenders could be found for logger (io.confluent.kql.util.KQLConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
=====
KQL (Kafka Query Language) 0.0.1
=====
kql> help
KQL cli commands:
=====
list topics           ..... Show the list of available topics.
list streams         ..... Show the list of available streams/tables.
describe <stream/table name> ..... Show the schema of the given stream/table.
show queries         ..... Show the list of running queries.
print <topic name>    ..... Print the content of a given topic/stream.
terminate <query id>  ..... Terminate the running query with the given id.

For more information refer to www.kql.confluent.io

kql> CREATE TOPIC orders_topic WITH (format = 'avro', avroschemafile='/Users/hojjat/avro_order_schema.avro',kafka_topic='orders_kafka_topic_avro');
kql> CREATE TOPIC users_topic WITH (format = 'json', kafka_topic='users_kafka_topic_json');
kql> CREATE TOPIC pageview_topic WITH (format = 'json', kafka_topic='pageview_kafka_topic_json');
kql> list topics

```

KQL Topic	Corresponding Kafka Topic	Topic Format	Notes
PAGEVIEW_TOPIC	pageview_kafka_topic_json	JSON	
ORDERS_TOPIC	orders_kafka_topic_avro	AVRO	
USERS_TOPIC	users_kafka_topic_json	JSON	Avro schema path: /Users/hojjat/avro_order_schema.avro

```

( 3 rows)
kql> CREATE STREAM orders (ordertime bigint, orderid varchar, itemid varchar, orderunits double) WITH (topicname = 'orders_topic' , key='ordertime');
kql> CREATE STREAM pageview (viewtime bigint, userid varchar, pageid varchar) WITH (topicname = 'pageview_topic',key='viewtime');
kql> CREATE TABLE users (usertime bigint, userid varchar, regionid varchar, gender varchar) WITH (topicname = 'users_topic', key='userid', statestore='user_statestore');
kql> list streams

```

Name	KQL Topic	Topic Key	Topic Type	Topic Format
ORDERS	ORDERS_TOPIC	ordertime	KSTREAM	AVRO
PAGEVIEW	PAGEVIEW_TOPIC	viewtime	KSTREAM	JSON
USERS	USERS_TOPIC	userid	KTABLE	JSON

```

( 3 rows)
kql>

```

Step 7: Catalog Export / Import

In order to not lose the catalog data in case of unexpected termination of the CLI tool we can export the catalog contents to a local file and use that to later restart the CLI. This way we don't have to create the topics and streams again.

To export the catalog file use the following command with an appropriate file path for your machine:

```
kql> export catalog to  
'/tmp/catalogfile.json';
```

Now that you have the catalog file, if you later restart the KQL CLI you can use this file to pre-populate the catalog like this:

```
$ ~/kql --properties-file kql.properties --catalog-file  
/tmp/catalogfile.json
```

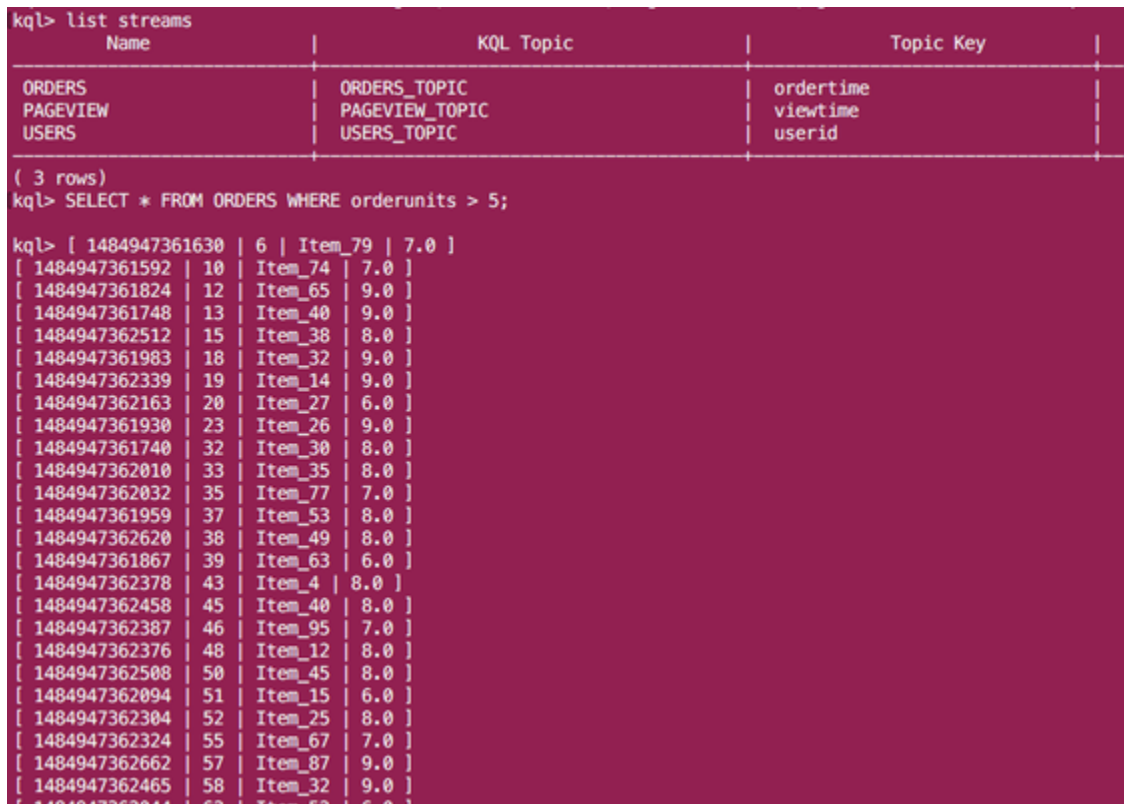
Step 8: Writing Queries

Now that we have all of our catalog metadata entries for topics and streams/tables defined we can move ahead to the fun part - writing continuous queries in KQL! Currently we have two types of KQL queries depending on the sink of the query. If the results of the query are to be shown in the console we can use a simple SELECT statement. On the other hand if we want to generate the results of a query into another KQL topic we need to use either the CREATE STREAM AS SELECT or CREATE TABLE AS SELECT statements.

Let's say we want to filter the orders stream with the following condition: **order_units > 5** and see the results in the **console**. This could be very useful when inspecting the data during iterative development for example. We would need to execute the following query:

```
kql> SELECT * FROM ORDERS WHERE  
orderunits > 5;
```

This will run a continuous query and keep printing the results in the console. Here is a screenshot of running this query:



Name	KQL Topic	Topic Key
ORDERS	ORDERS_TOPIC	ordertime
PAGEVIEW	PAGEVIEW_TOPIC	viewtime
USERS	USERS_TOPIC	userid

(3 rows)

```
kql> SELECT * FROM ORDERS WHERE orderunits > 5;
```

```
kql> [ 1484947361630 | 6 | Item_79 | 7.0 ]  
[ 1484947361592 | 10 | Item_74 | 7.0 ]  
[ 1484947361824 | 12 | Item_65 | 9.0 ]  
[ 1484947361748 | 13 | Item_40 | 9.0 ]  
[ 1484947362512 | 15 | Item_38 | 8.0 ]  
[ 1484947361983 | 18 | Item_32 | 9.0 ]  
[ 1484947362339 | 19 | Item_14 | 9.0 ]  
[ 1484947362163 | 20 | Item_27 | 6.0 ]  
[ 1484947361930 | 23 | Item_26 | 9.0 ]  
[ 1484947361740 | 32 | Item_30 | 8.0 ]  
[ 1484947362010 | 33 | Item_35 | 8.0 ]  
[ 1484947362032 | 35 | Item_77 | 7.0 ]  
[ 1484947361959 | 37 | Item_53 | 8.0 ]  
[ 1484947362620 | 38 | Item_49 | 8.0 ]  
[ 1484947361867 | 39 | Item_63 | 6.0 ]  
[ 1484947362378 | 43 | Item_4 | 8.0 ]  
[ 1484947362458 | 45 | Item_40 | 8.0 ]  
[ 1484947362387 | 46 | Item_95 | 7.0 ]  
[ 1484947362376 | 48 | Item_12 | 8.0 ]  
[ 1484947362508 | 50 | Item_45 | 8.0 ]  
[ 1484947362094 | 51 | Item_15 | 6.0 ]  
[ 1484947362304 | 52 | Item_25 | 8.0 ]  
[ 1484947362324 | 55 | Item_67 | 7.0 ]  
[ 1484947362662 | 57 | Item_87 | 9.0 ]  
[ 1484947362465 | 58 | Item_32 | 9.0 ]  
[ 1484947362044 | 62 | Item_52 | 6.0 ]
```

****** Note that this is a continuous query and will keep running in the console until you tell it to stop! If you add more data to the orders topic by re-running the data generator tool you will see additional results of the query execution on the new data in the console.

In order to stop the current query at any time you can type "**close**" command in the CLI. Go ahead and do that now.

Now let's assume that instead of simply inspecting the *orders* topic data we want to export the results of our query into a new topic. To achieve

this goal we can use the CREATE STREAM AS SELECT statement as follows:

```
kql> CREATE STREAM bigorders_avro AS SELECT * FROM orders WHERE orderunits > 5 ;
```

This will result in creation of a new Kafka topic along with a new KQL topic and stream. The following screenshot shows the execution of the above command:

```
[ 1484947367163 | 981 | Item_42 | 6.0 |
[ 1484947366913 | 985 | Item_35 | 9.0 |
[ 1484947367218 | 986 | Item_20 | 6.0 |
[ 1484947367317 | 990 | Item_15 | 8.0 |
[ 1484947367729 | 991 | Item_69 | 8.0 |
[ 1484947367580 | 995 | Item_52 | 7.0 |
close
kql> CREATE STREAM bigorders_avro AS SELECT * FROM orders WHERE orderunits > 5 ;
kql> list streams
+-----+-----+-----+-----+-----+
| Name | KQL Topic | Topic Key | Topic Type | Topic Format |
+-----+-----+-----+-----+-----+
| ORDERS | ORDERS_TOPIC | ordertime | KSTREAM | AVRO |
| PAGEVIEW | PAGEVIEW_TOPIC | viewtime | KSTREAM | JSON |
| USERS | USERS_TOPIC | userid | KTABLE | JSON |
| BIGORDERS_AVRO | BIGORDERS_AVRO | ordertime | KSTREAM | AVRO |
+-----+-----+-----+-----+-----+
( 4 rows)
kql> list topics;
+-----+-----+-----+-----+-----+
| KQL Topic | Corresponding Kafka Topic | Topic Format | Notes |
+-----+-----+-----+-----+-----+
| PAGEVIEW_TOPIC | pageview_kafka_topic_json | JSON | Avro schema path: /Users/hojjat/avro_order_schema.avro |
| ORDERS_TOPIC | orders_kafka_topic_avro | AVRO | Avro schema path: /tmp/BIGORDERS_AVRO.avro |
| BIGORDERS_AVRO | BIGORDERS_AVRO | AVRO | |
| USERS_TOPIC | users_kafka_topic_json | JSON | |
+-----+-----+-----+-----+-----+
( 4 rows)
kql>
```

As you can see we now have a new KQL topic, BIGORDERS_AVRO, and a new stream, BIGORDERS_AVRO. Note that by default the format for the new topic is the same as the format for the query source. If the newly-created topic is in avro format, you can specify a path to which a generated avro schema file for the results will be saved in a WITH clause of the CREATE STREAM AS SELECT statement. If the path is not specified KQL will store the corresponding avro schema file in your /tmp folder.

Now let's assume we want to run the same query but store the results in JSON format and name the resultant underlying Kafka topic *bigorders_topic*. The following statement will accomplish this:

```
kql> CREATE STREAM bigorders_json WITH (format = 'json', kafka_topic='bigorders_topic') AS SELECT * FROM orders WHERE orderunits > 5 ;
```

At any time, you can use the `show queries` command to list out the currently-executing continuous transformations that have been started by your CREATE STREAM AS... or CREATE TABLE AS ... statements.

Now that we have covered the basic concepts of the KQL CLI you can try the following queries and explore their results:

```
kql> CREATE STREAM bigorders_item2 WITH(format = 'avro',
avroschemafile='/Users/hojjat/avro_bigorders_item2_schema.avro') AS SELECT ordertime, itemid,
concat(LCASE(itemid),'_'), orderunits FROM orders WHERE itemid = 'Item_2';

kql> CREATE STREAM enrichedpageview_female AS SELECT users.userid AS userid, pageid, regionid, gender
FROM pageview LEFT JOIN users ON pageview.userid = users.userid WHERE gender = 'FEMALE';

kql> CREATE STREAM enrichedpageview_female_region8 AS SELECT * FROM enrichedpageview_female WHERE
regionid = 'Region_8';

kql> CREATE STREAM enrichedpageview_female_region8_0 AS SELECT userid, pageid, regionid FROM
enrichedpageview_female WHERE pageid LIKE '%8';
```

Step 9: Scalar Functions

You may have noticed in one of the four examples right above that we included a couple of functions (`concat` and `lcase`) in the first CREATE STREAM statement. There's a list of the currently-implemented functions available on the [KQL Syntax Guide](#) page. You should be able to freely mix-and-match and nest these functions together to create your own examples.

Step 10: Non-interactive Mode

So far we've been iteratively developing and testing our queries in the KSQL CLI. Let's suppose that we are now ready to move on and deploy one or more of them to run continuously somewhere. To do that we need two things:

1. Export your catalog to a file:

```
kql> export catalog to  
'/tmp/my_catalog.json';
```

2. Exit the CLI (type ``exit`` to close out cleanly) and create a new text file containing the query or queries to be run. Note that the catalog itself only contains the metadata about your Tables and Streams, not the actual statements used to populate them with data, so you will need a full `CREATE TABLE AS SELECT` statement here. (In a future iteration we will also add an ``INSERT INTO foo SELECT ... FROM bar`` statement which will feel more natural here but for now we have to repeat the full CTAS).

```
$ echo "create stream lights_out_big_orders as select * from orders where orderunits > 6;" >  
/tmp/ctas_1.sql
```

Now that we have armed ourselves with both our catalog and query files we are ready to go:

```
$ ~/kql --properties-file kql.properties --catalog-file /tmp/my_catalog.json --query-file  
/tmp/ctas_1.sql
```

Note - now that we are no longer in the interactive development mode the normal rules of topic offsets will apply, meaning that we need to run the data generator for our orders topic again to produce more records into it to be picked up by our continuous query. (You can run the generator tool for each topic as many times as you like, it will simply append more records every time).

Of course there's no longer any console output for us to check what is happening so, to verify for ourselves what just happened, let's open a new terminal and resort to the regular Kafka tools:

```
$ kafka-topics.sh --zookeeper localhost:2181  
--list
```

If you copy/pasted the exact query above, you should see an entry in the list of topics for `LIGHTS_OUT_BIG_ORDERS`, which is the newly-created topic holding the stream of big orders data. Go ahead and verify the contents:

```
$ kafka-console-consumer.sh --zookeeper localhost:2181 --topic LIGHTS_OUT_BIG_ORDERS  
--from-beginning
```

If you leave the console consumer running for a little longer and go back and run the data generator one more time you should see the newly-produced records show up in the console consumer almost immediately.

Appendix: Fun Extras

If you made it this far - congratulations! We've added a couple of fun experimental steps here for your amusement...

Playing with BASH

The non-interactive mode of the KSQL tool can also be used to return results onto STDOUT rather than producing them into another topic. Simply use a query file with a regular `SELECT` statement rather than a CTAS:

```
$ echo "select * from orders where orderunits > 3 and orderunits < 8;" >  
/tmp/medium_orders.sql
```

Go ahead and launch this new query file in the non-interactive mode:

```
$ ~/kql --property-file kql.properties --catalog-file /tmp/my_catalog.json --query-file  
/tmp/medium_orders.sql
```

And remember to run the generator tool again to create new records! You should see the matching orders scroll by in your terminal.

This means you can treat your KQL query just like any other shell command when it comes to piping it's output into something else... Let's try an example:

```
$ ~/kql --property-file kql.properties --catalog-file /tmp/my_catalog.json --query-file /tmp/medium_orders.sql | sed "s/Item/Medium_Item/"
```

I can imagine having some fun linking this up to a topic fed from a twitter or irc Connector 😊

Showing the power of Streams

It should hopefully be obvious that Kafka Streams is doing much of the heavy lifting under the hood in all of these examples. You can easily demonstrate that with a little experiment:

1. Increase the partition count of the *orders* topic to something higher than 1:

```
$ kafka-topics.sh --zookeeper localhost:2181 --alter --partitions 2 --topic orders_kafka_topic_json
```

2. Re-launch the same version of our SELECT query which returns data onto STDOUT, only this time do it in two different terminal sessions :

```
$ ~/kql --property-file kql.properties --catalog-file /tmp/my_catalog.json --query-file /tmp/medium_orders.sql
```

3. Re-run the data generator a couple of times and you should be able to see that each instance of the query is processing records from a different partition of the input topic. If you subsequently kill one of the terminals and leave the other running you should see the internal Streams app rebalance both partitions into the surviving instance. Pretty cool!