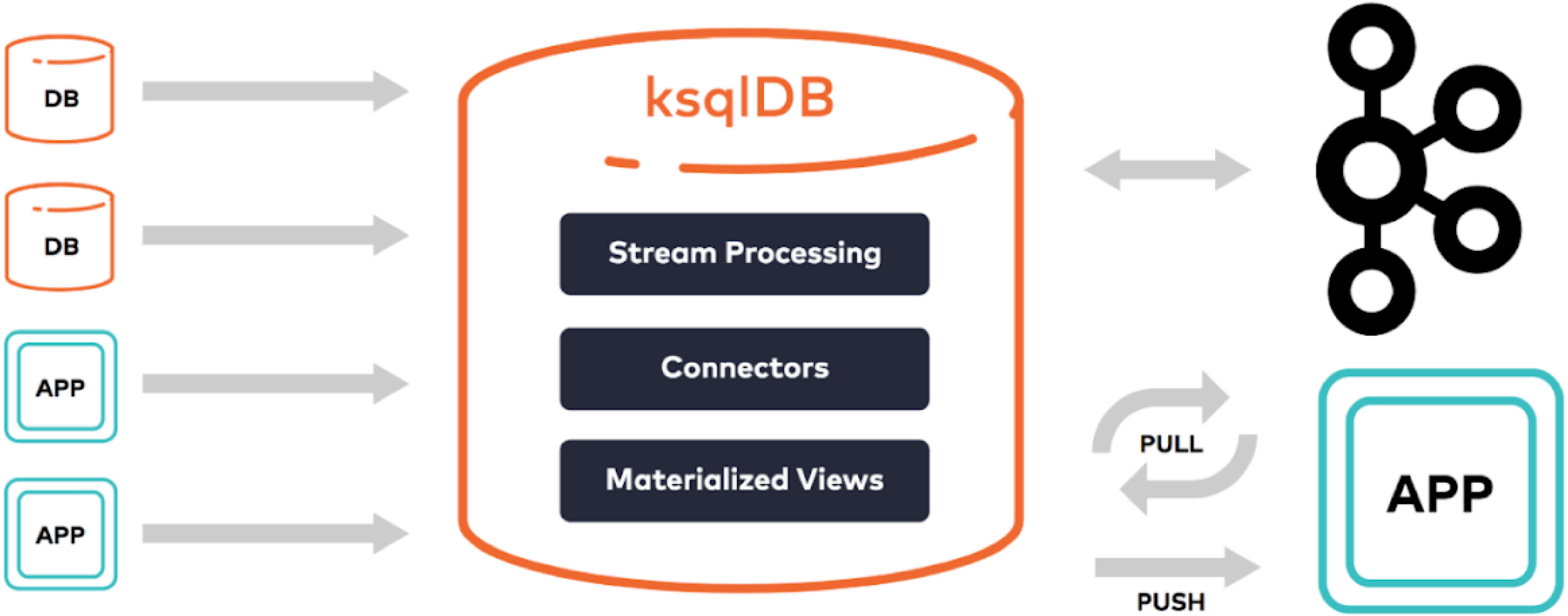
KSQL DB

Stream processing Made Simple with Kafka

ksqlDB

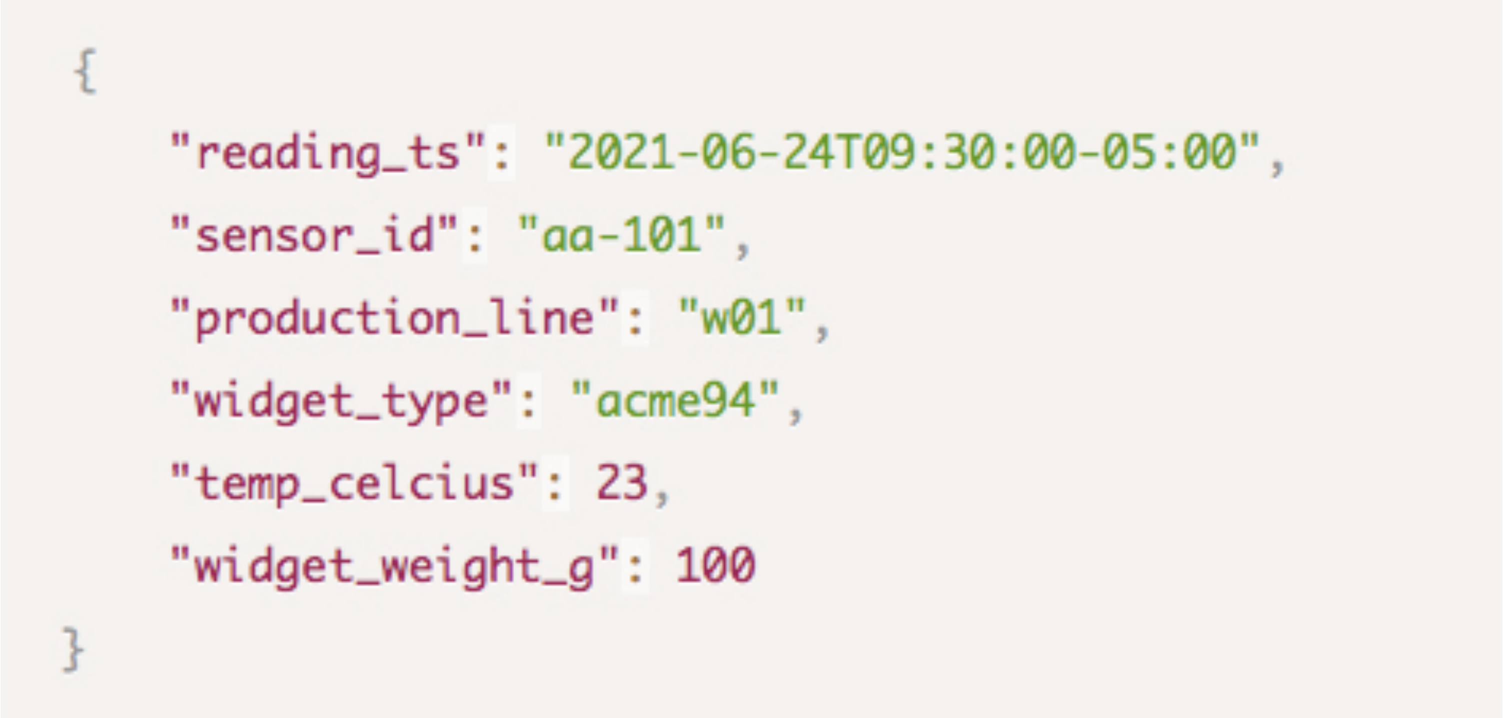
The database purpose-built for stream processing applications.

build stream processing applications on top of Apache Kafka with the ease of building traditional applications on a relational database using SQL



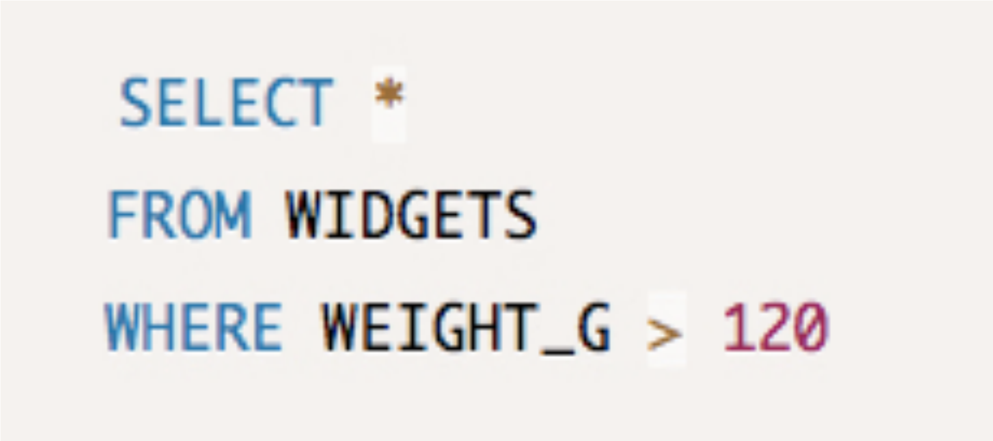
Stream Processing in Action - KSQL

data stream emitted by a production line

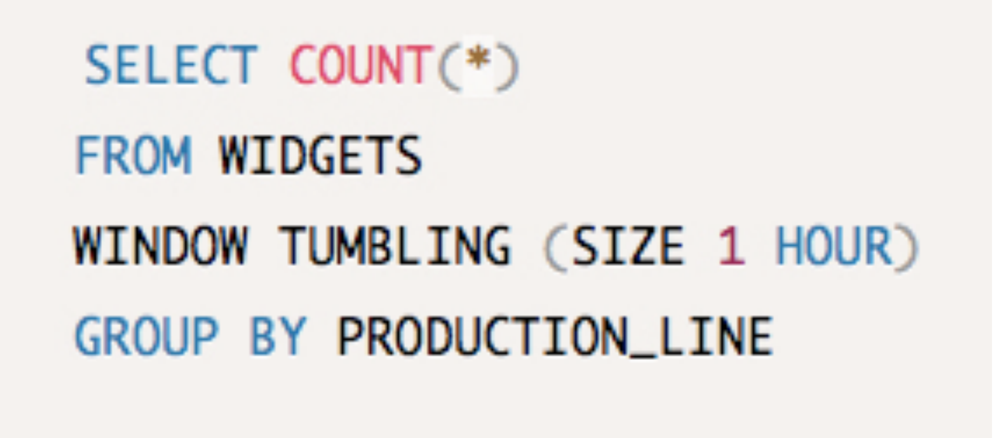


```
{
    "reading_ts": "2021-06-24T09:30:00-05:00",
    "sensor_id": "aa-101",
    "production_line": "w01",
    "widget_type": "acme94",
    "temp_celcius": 23,
    "widget_weight_g": 100
}
```

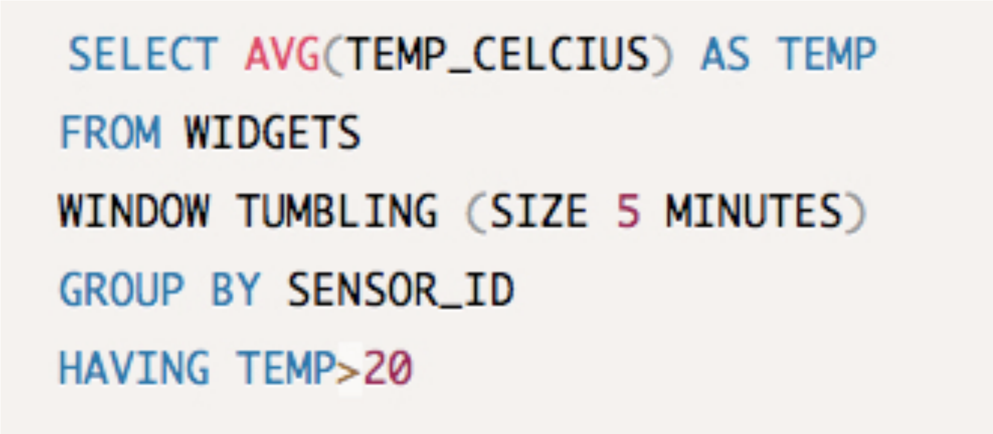
- •Alert if the line produces an item that is over a threshold weight
- •Monitor the rate of item production
- •Detect anomalies in the equipment
- •Store the data for analytics dashboards and ad hoc querying



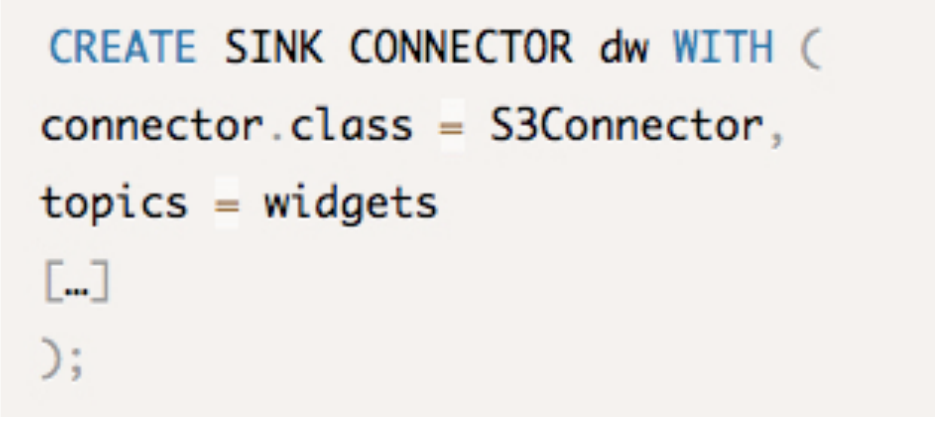
```
SELECT *
FROM WIDGETS
WHERE WEIGHT_G > 120
```



```
SELECT COUNT(*)
FROM WIDGETS
WINDOW TUMBLING (SIZE 1 HOUR)
GROUP BY PRODUCTION_LINE
```

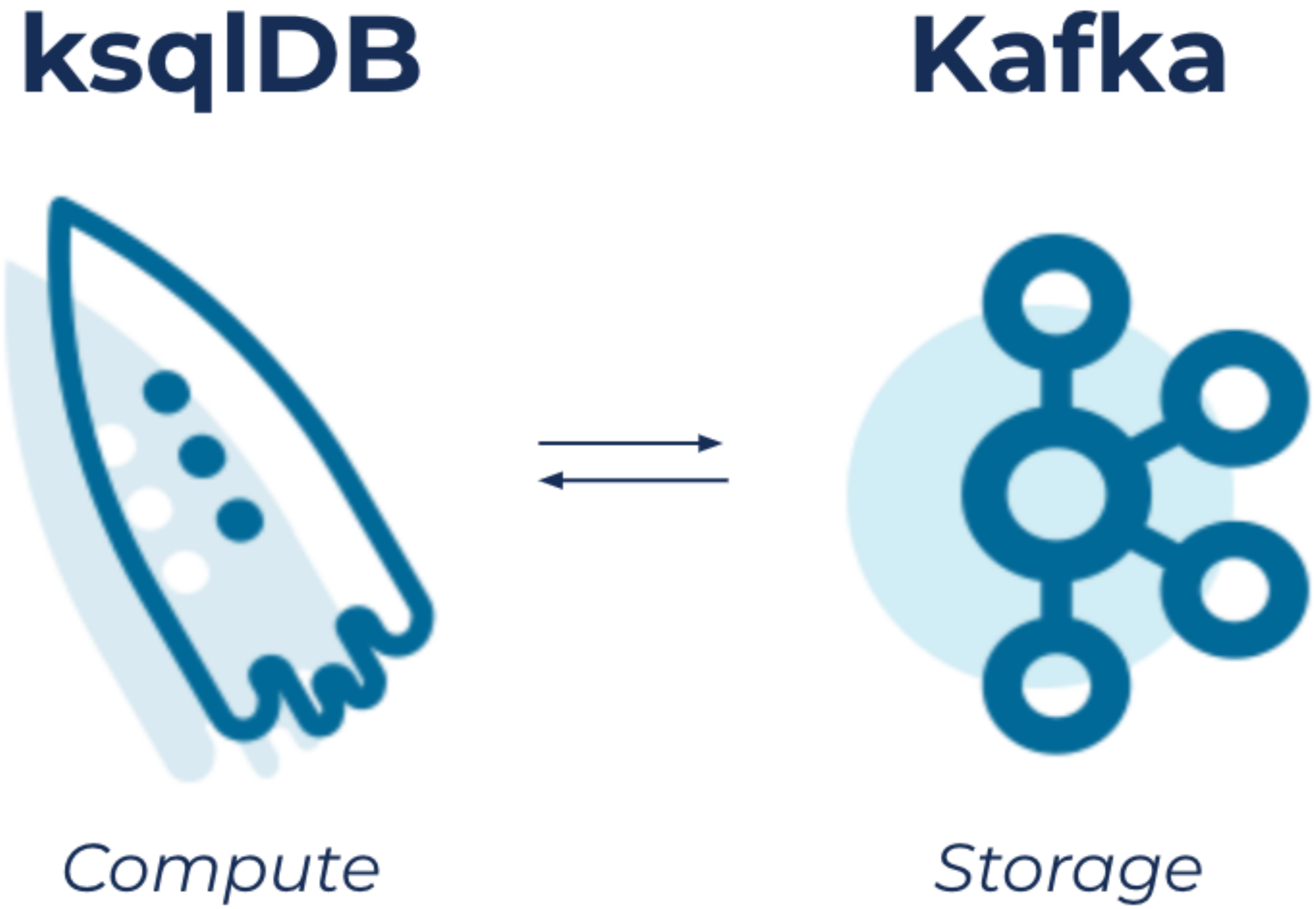


```
SELECT AVG(TEMP_CELCIUS) AS TEMP
FROM WIDGETS
WINDOW TUMBLING (SIZE 5 MINUTES)
GROUP BY SENSOR_ID
HAVING TEMP>20
```



```
CREATE SINK CONNECTOR dw WITH (
connector.class = S3Connector,
topics = widgets
[…]
);
```

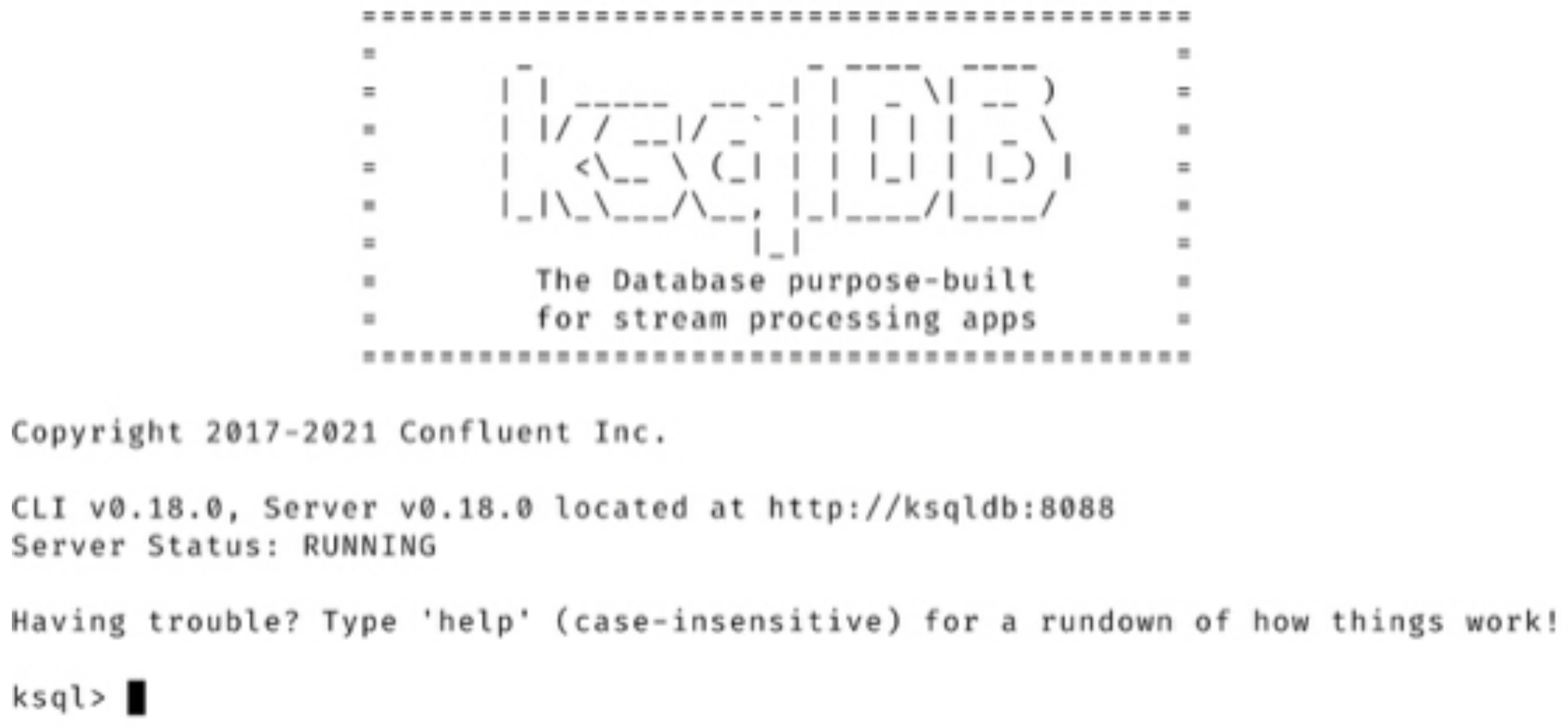
How does ksqlDB work?



- ksqlDB allows us to read, filter, transform, or otherwise process streams and tables of events, which are backed by Kafka topics.
- We can also join streams and/or tables to meet the needs of our application.
- And we can do all of this using familiar SQL syntax.

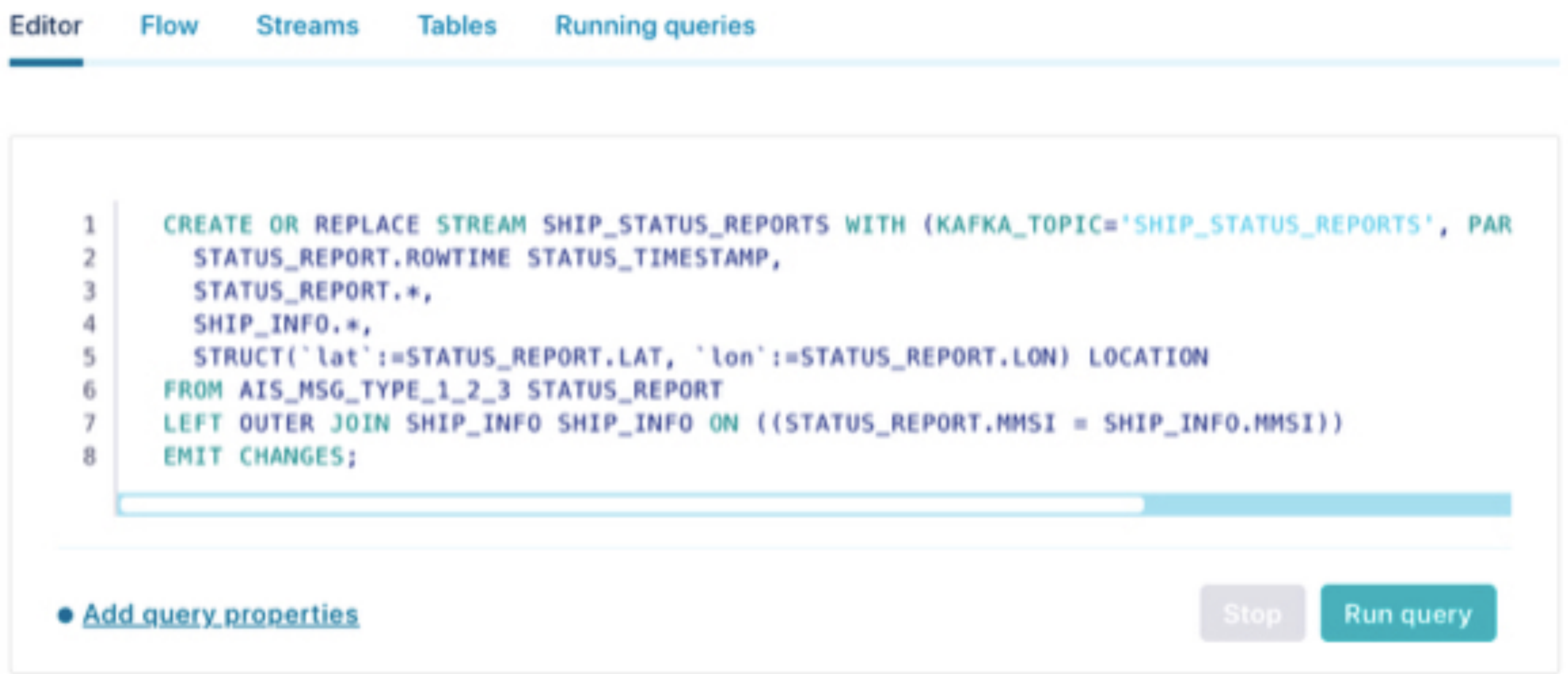
Interacting with ksqlDB

Command Line Interface (CLI)



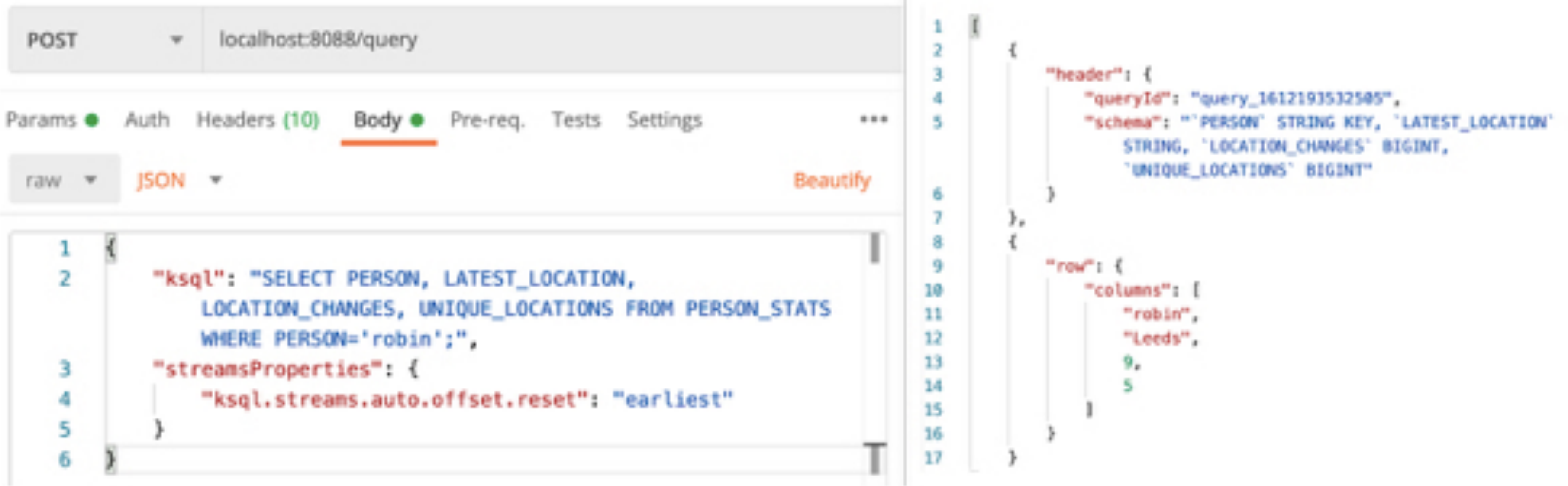
```
===========================================
=       _              _ ____  ____       =
=      | | _____  __ _| |  _ \| __ )      =
=      | |/ / __|/ _` | | | | |  _ \      =
=      |   <\__ \ (_| | | |_| | |_) |     =
=      |_|\_\___/\__, |_|____/|____/      =
=                   |_|         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

Copyright 2017-2021 Confluent Inc.

CLI v0.18.0, Server v0.18.0 located at http://ksqldb:8088
Server Status: RUNNING

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

ksql> █
```

Web UI



```
CREATE OR REPLACE STREAM SHIP_STATUS_REPORTS WITH (KAFKA_TOPIC='SHIP_STATUS_REPORTS', PAR
  STATUS_REPORT.ROWTIME STATUS_TIMESTAMP,
  STATUS_REPORT.*,
  SHIP_INFO.*,
  STRUCT(`lat`:=STATUS_REPORT.LAT, `lon`:=STATUS_REPORT.LON) LOCATION
FROM AIS_MSG_TYPE_1_2_3 STATUS_REPORT
LEFT OUTER JOIN SHIP_INFO SHIP_INFO ON ((STATUS_REPORT.MMSI = SHIP_INFO.MMSI))
EMIT CHANGES;
```

REST API



POST localhost:8088/query

```
{
    "ksql": "SELECT PERSON, LATEST_LOCATION,
        LOCATION_CHANGES, UNIQUE_LOCATIONS FROM PERSON_STATS
        WHERE PERSON='robin';",
    "streamsProperties": {
        "ksql.streams.auto.offset.reset": "earliest"
    }
}
```

```
[
    {
        "header": {
            "queryId": "query_1612193532505",
            "schema": "`PERSON` STRING KEY, `LATEST_LOCATION`
                STRING, `LOCATION_CHANGES` BIGINT,
                `UNIQUE_LOCATIONS` BIGINT"
        }
    },
    {
        "row": {
            "columns": [
                "robin",
                "Leeds",
                9,
                5
            ]
        }
    }
```

Using ksqlDB

create a **stream** called MOVEMENTS

```
CREATE STREAM MOVEMENTS (PERSON VARCHAR KEY, LOCATION VARCHAR)  WITH
(VALUE_FORMAT='JSON', PARTITIONS=1, KAFKA_TOPIC='movements');
```

```
INSERT INTO MOVEMENTS VALUES ('Allison', 'Denver');
INSERT INTO MOVEMENTS VALUES ('Robin', 'Leeds');
INSERT INTO MOVEMENTS VALUES ('Robin', 'Ilkley');
INSERT INTO MOVEMENTS VALUES ('Allison', 'Boulder');
```

Using ksqlDB

```
#show streams;
```

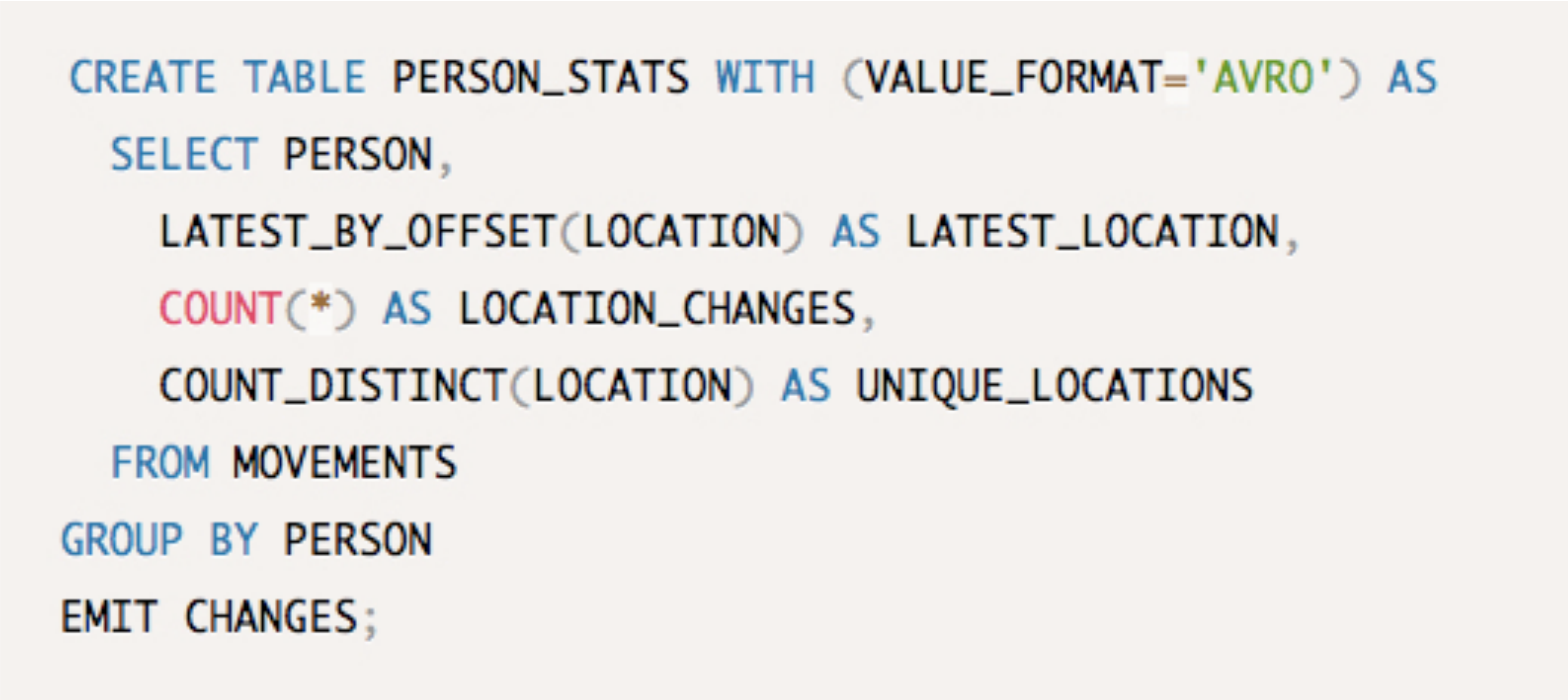
```
#SET 'auto.offset.reset' = 'earliest';
```

(to see all of the data that’s already in the stream, and not just new messages as they arrive)

Now run the following SELECT to show all of the events in MOVEMENTS:

```
#SELECT * FROM MOVEMENTS EMIT CHANGES;
```

Using ksqlDB



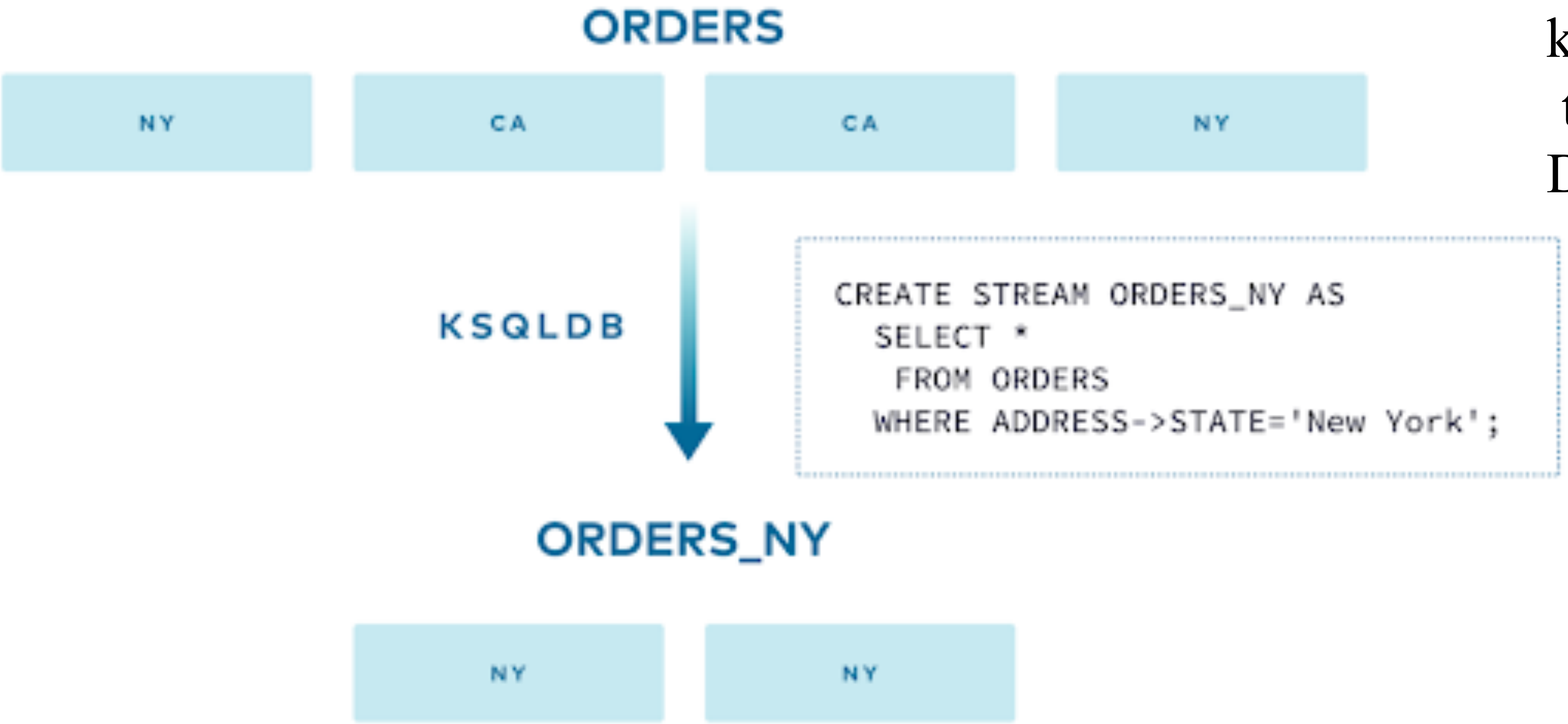
```
CREATE TABLE PERSON_STATS WITH (VALUE_FORMAT='AVRO') AS
  SELECT PERSON,
    LATEST_BY_OFFSET(LOCATION) AS LATEST_LOCATION,
    COUNT(*) AS LOCATION_CHANGES,
    COUNT_DISTINCT(LOCATION) AS UNIQUE_LOCATIONS
  FROM MOVEMENTS
GROUP BY PERSON
EMIT CHANGES;
```

#show tables;
This will verify that the table has been created.

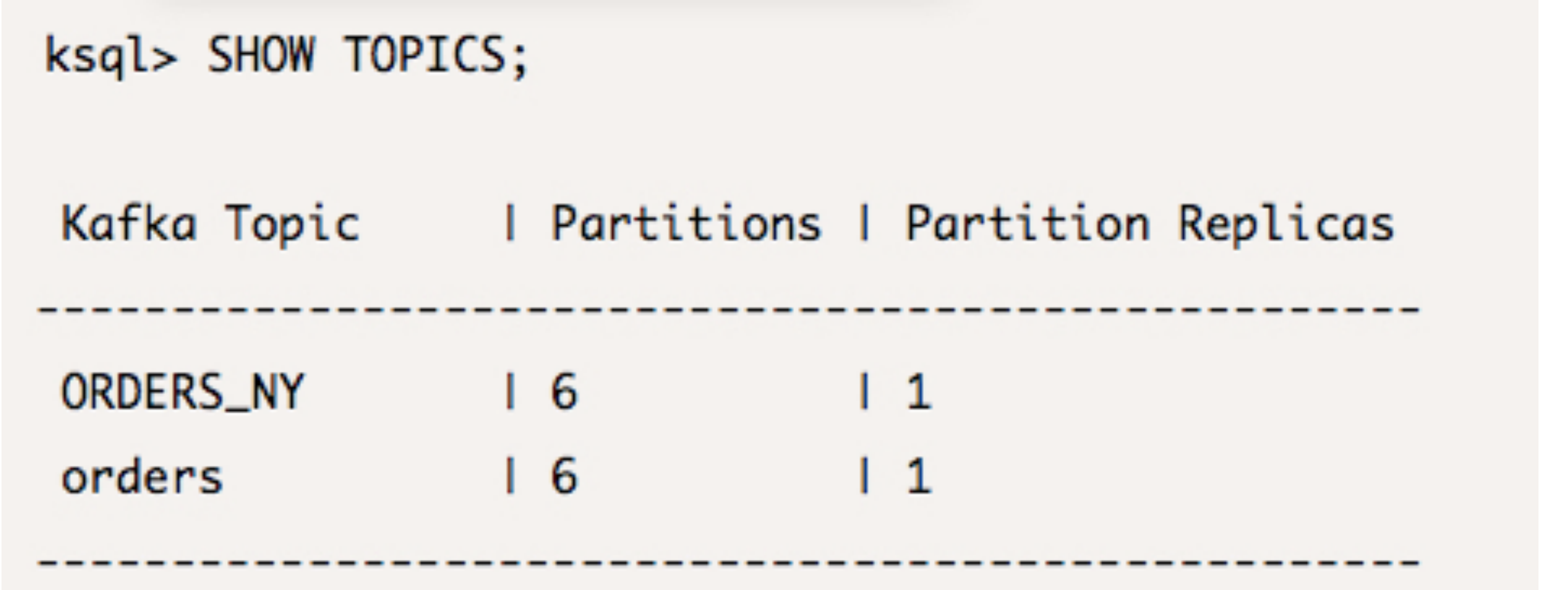
SELECT * FROM PERSON_STATS WHERE person ='Allison';

Filtering with ksqlDB

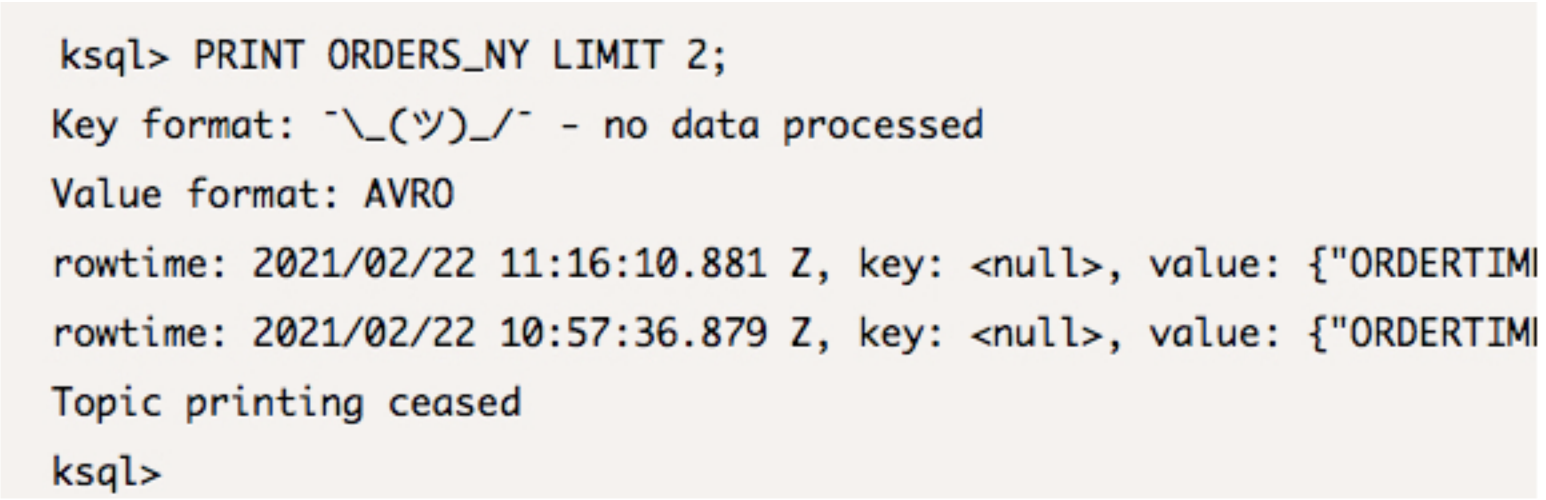
ksqlDB streams are backed by Kafka topics, this means that you are writing this data Directly to a Kafka topic:



```
CREATE STREAM ORDERS_NY AS
  SELECT *
   FROM ORDERS
  WHERE ADDRESS->STATE='New York';
```



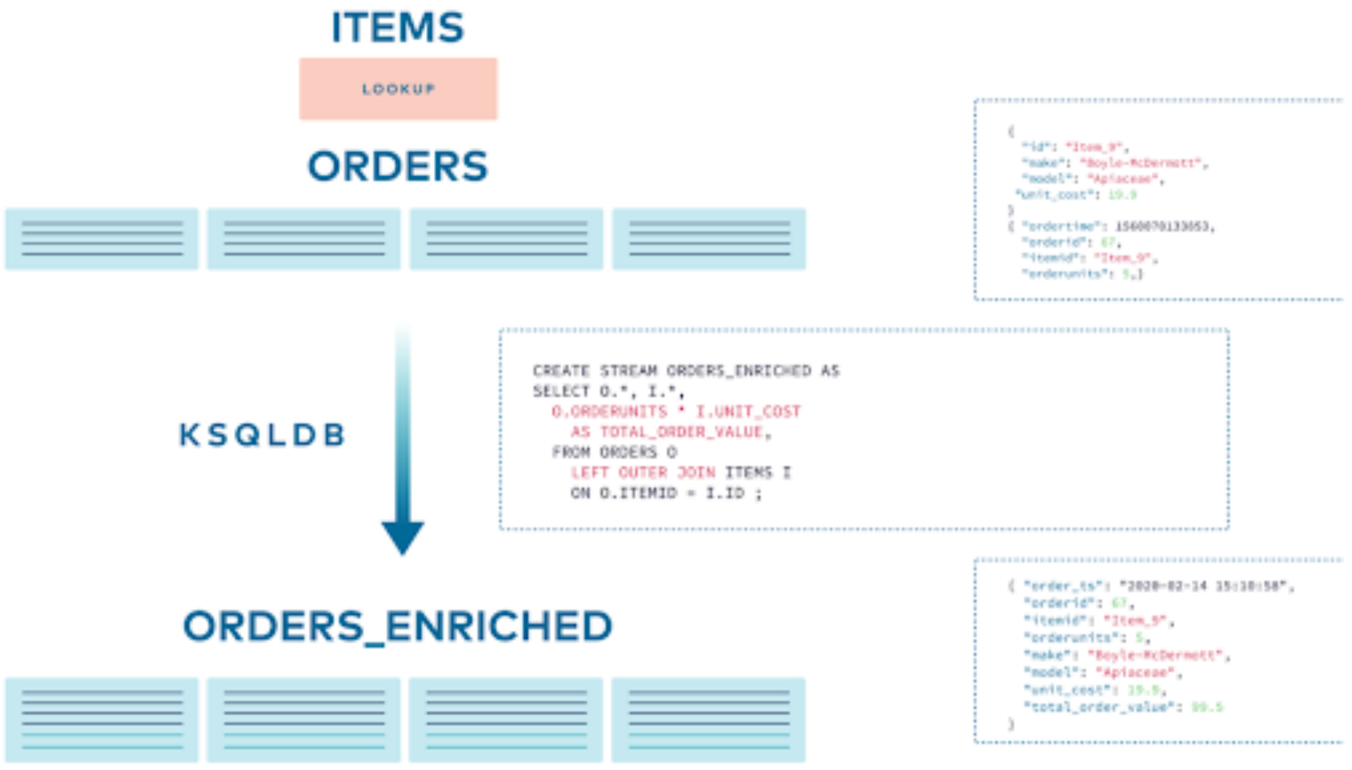
```
ksql> SHOW TOPICS;

 Kafka Topic       | Partitions | Partition Replicas
---------------------------------------------------
 ORDERS_NY         | 6          | 1
 orders            | 6          | 1
---------------------------------------------------
```



```
ksql> PRINT ORDERS_NY LIMIT 2;
Key format: ¯\_(ツ)_/¯ - no data processed
Value format: AVRO
rowtime: 2021/02/22 11:16:10.881 Z, key: <null>, value: {"ORDERTIM
rowtime: 2021/02/22 10:57:36.879 Z, key: <null>, value: {"ORDERTIM
Topic printing ceased
ksql>
```

Lookups and Joins with ksqlDB

Joins can be between streams of events, between streams and tables, or between tables and tables.
Instead of making calls out to external systems to enrich the data with lookups (which introduces complexity, and can be inefficient), the data is ingested, using Kafka Connect, from the external system into an Apache Kafka topic, from which the join can be performed natively using ksqlDB.

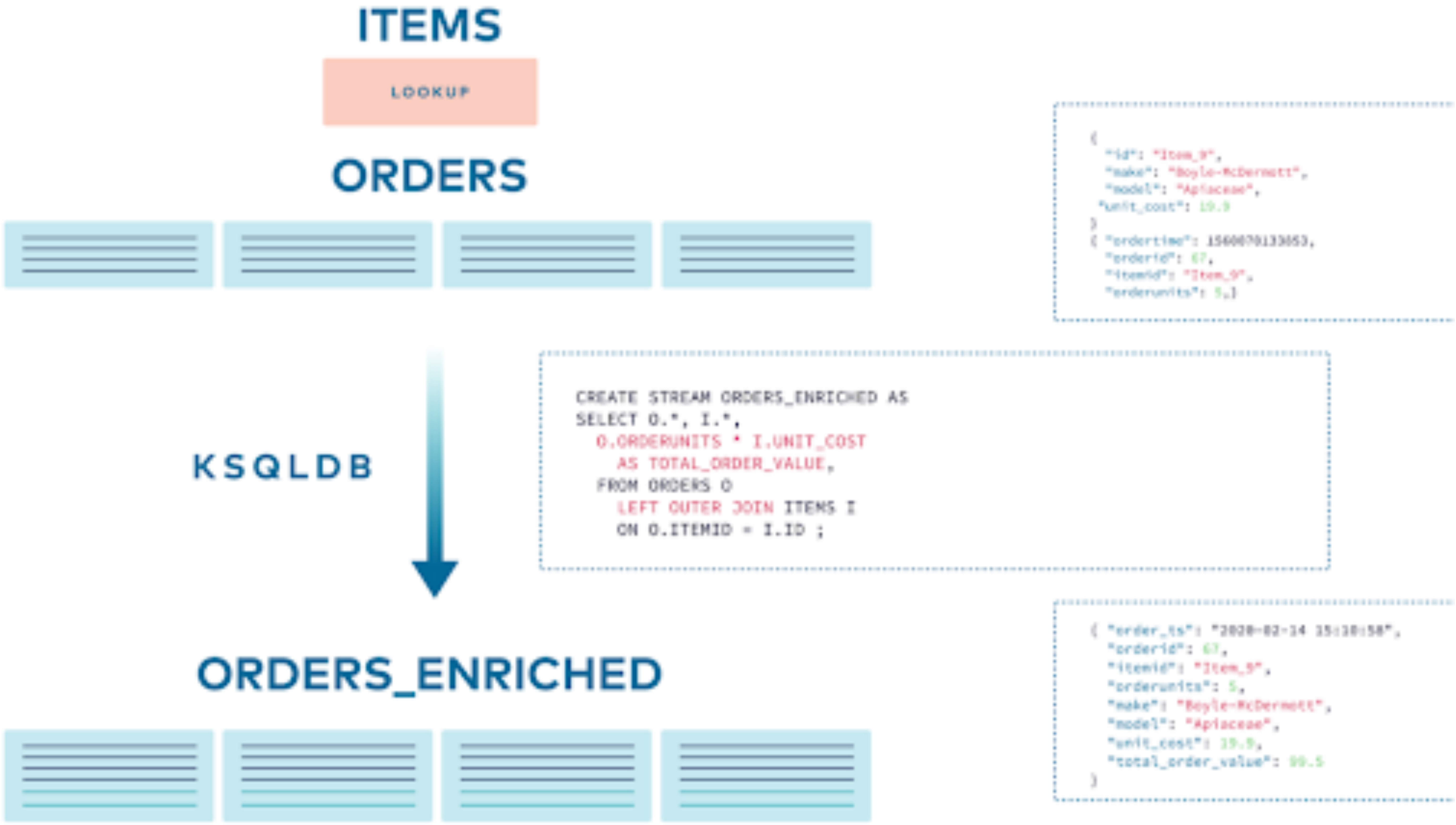


```
{
  "id": "Item_9",
  "make": "Boyle-McDermott",
  "model": "Apiacese",
 "unit_cost": 19.9
}
{ "ordertime": 1560070133853,
  "orderid": 67,
  "itemid": "Item_9",
  "orderunits": 5,}
```

```
CREATE STREAM ORDERS_ENRICHED AS
SELECT O.*, I.*,
  O.ORDERUNITS * I.UNIT_COST
    AS TOTAL_ORDER_VALUE,
  FROM ORDERS O
    LEFT OUTER JOIN ITEMS I
    ON O.ITEMID = I.ID ;
```

```
{ "order_ts": "2020-02-14 15:10:58",
  "orderid": 67,
  "itemid": "Item_9",
  "orderunits": 5,
  "make": "Boyle-McDermott",
  "model": "Apiacese",
  "unit_cost": 19.9,
  "total_order_value": 99.5
}
```

Lookups and Joins with ksqlDB



```
{
  "id": "Item_9",
  "make": "Boyle-McDermott",
  "model": "Apiaceae",
 "unit_cost": 19.9
}
{ "ordertime": 1560070133053,
  "orderid": 67,
  "itemid": "Item_9",
  "orderunits": 5,}
```

```
CREATE STREAM ORDERS_ENRICHED AS
SELECT O.*, I.*,
  O.ORDERUNITS * I.UNIT_COST
    AS TOTAL_ORDER_VALUE,
  FROM ORDERS O
    LEFT OUTER JOIN ITEMS I
    ON O.ITEMID = I.ID ;
```

```
{ "order_ts": "2020-02-14 15:30:58",
  "orderid": 67,
  "itemid": "Item_9",
  "orderunits": 5,
  "make": "Boyle-McDermott",
  "model": "Apiaceae",
  "unit_cost": 19.9,
  "total_order_value": 99.5
}
```

Join - Example

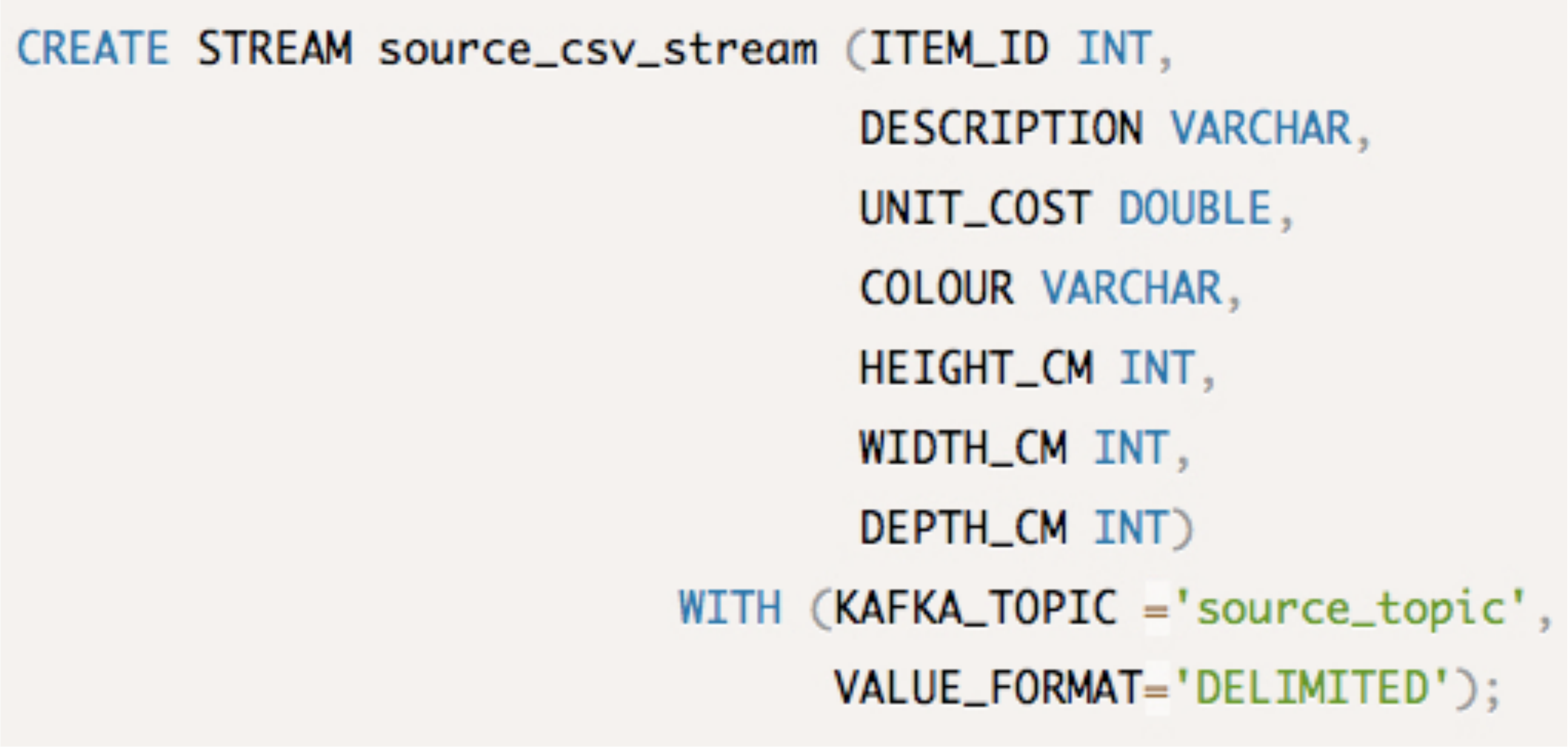
joining order events to a table that contains information about the item being ordered:

```
CREATE STREAM ORDERS_ENRICHED AS

SELECT
O.*, I.*, O.ORDERUNITS * I.UNIT_COST AS TOTAL_ORDER_VALUE

FROM ORDERS O
LEFT OUTER JOIN ITEMS I
 ON O.ITEMID = I.ID ;
```

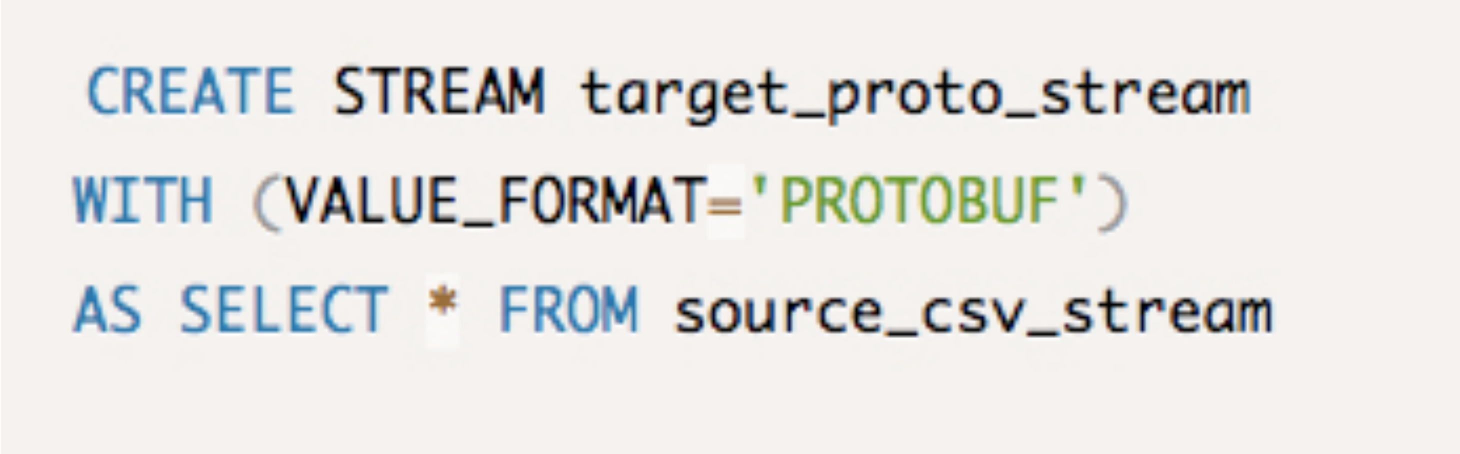
Converting Data Formats with ksqlDB

There are different ways to serialize data written to Apache Kafka topics. Common options include Avro, Protobuf, and JSON.
You can use ksqlDB to create a new stream of data identical to the source but serialized differently.

To write a stream of data from its CSV source to a stream using Protobuf, you would first declare the schema of the CSV data:



```
CREATE STREAM source_csv_stream (ITEM_ID INT,
                                 DESCRIPTION VARCHAR,
                                 UNIT_COST DOUBLE,
                                 COLOUR VARCHAR,
                                 HEIGHT_CM INT,
                                 WIDTH_CM INT,
                                 DEPTH_CM INT)
                           WITH (KAFKA_TOPIC ='source_topic',
                                VALUE_FORMAT='DELIMITED');
```



```
 CREATE STREAM target_proto_stream
WITH (VALUE_FORMAT='PROTOBUF')
AS SELECT * FROM source_csv_stream
```

Push Queries and Pull Queries

Push queries are identified by the EMIT CHANGES clause. By running a push query, the client will receive a message for every change that occurs on the stream (that is, every new message).

Pull queries return the current state to the client, and then terminate. In that sense, they are much more akin to a SELECT statement executed on a regular RDBMS. They can only be used against ksqlDB tables with materialized state, that is, a table in which there is an aggregate function. Currently, tables declared against an existing Apache Kafka topic cannot be queried with a pull query (as of ksqlDB v0.15 / February 2021).

Lab : **Workflow using KSQL - CLI – 90 Minutes**