data virtuality

# DATA VIRTUALITY MASTERCLASS

## Topic: Kafka/KSQLDB showcase

# Welcome to the DV Masterclass! Agenda

2:00pm - 3:00pm: Retrieving Data from Data Virtuality via ODATA

3:00pm - 3:15pm: Break

3:15pm - 4:15pm: Kafka/KSQLDB showcase

4:15pm - 4:20pm: Break

4:20pm - 5:15pm: Procedural Relational Command

# What to expect from this session?

In this track, we will show how to get data from Kafka

- What is Kafka
- What is ksqlDB
- Use cases
- Options to connect
- KSQL SQL syntax specialties and Translator
- Queries from DV

data virtuality

# What is Kafka?

# What is Kafka?

- Kafkas 3 key abilities:
  - To publish (**write**) and subscribe to (**read**) **streams of events**, including continuous import/export of your data from other systems.
  - To store streams of events durably and reliably for as long as you want.
  - To process streams of events as they occur or retrospectively.
- Terminology: **events** are stored in **topics**, configurable how long
  - Topics are partitioned
- Challenge for Data Virtualization:
  - events are active, so we should react, but DV doesn’t care if no one is “asking”
  - new events are coming in while a query is running
  - events will not be available after retention period

# Use cases for Data Virtualization

# Use cases for Data Virtuality

What would be your use cases? Please write them into the chat now!

Use cases from Apache Kafka itself:

- **Messaging**
- **Website Activity Tracking**
- **Metrics (operational monitoring)**
- **Log Aggregation**
- **Stream Processing**
- **Event Sourcing**
- **Commit Log**

Community use cases:

Sebastian wrote: Continuous materialization of changes in DV

# Use cases for Data Virtuality

Commented use cases

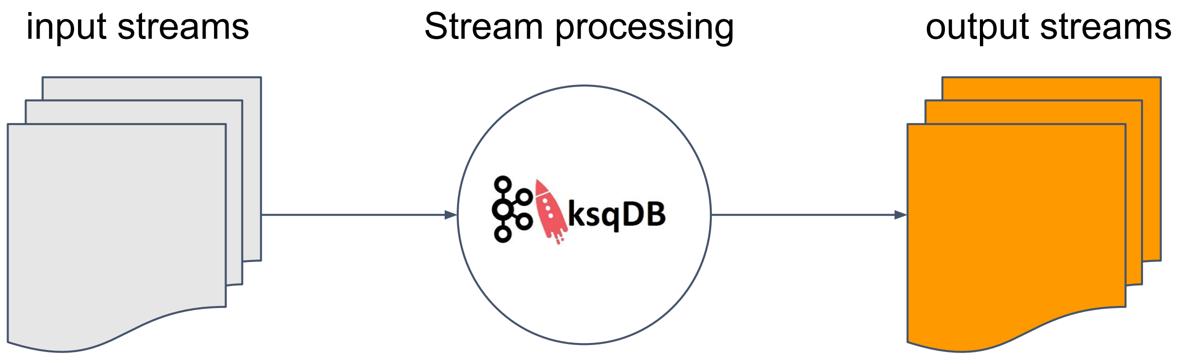
- **Messaging** - sending messages from a DV automation is possible, great use case
- **Website Activity Tracking** - classical use case, we would probably join this to our data, in an analytical way
- **Metrics** - Possible in a pull configuration, also emit metrics from DV
- **Log Aggregation** - Abstraction for Log files
- **Stream Processing** - IMHO very large use case, as we can integrate the results of processed and aggregated streams
- **Event Sourcing** - *is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.*
- **Commit Log** - *Kafka can serve as a kind of external commit-log for a distributed system.*

# What is ksqlDB

# What is ksqlDB?

- ksqlDB is part of the confluent platform (Kafka distribution with extras)
- Provides (among other things) an SQL bridge to kafka, ideal for data virtualization
- ksqlDB is a stream processing database

  capabilities:
  - Derive a new stream from an existing stream
  - Derive a new table from an existing stream
  - Derive a new table from an existing table
  - Derive a new stream from multiple streams



# Streams and Tables

- A **stream** is a an immutable, append-only collection that represents a series of historical facts, or events. Once a row is inserted into a stream, the row can never change. You can append new rows at the end of the stream, but you can't update or delete existing rows.
- A **table** is a mutable collection that models change over time. It uses row keys to display the most recent data for each key. All but the newest rows for each key are deleted periodically. Also, each row has a timestamp, so you can define a windowed table which enables controlling how to group records that have the same key for stateful operations – like aggregations and joins – into time spans. Windows are tracked by record key.

# ksqlDB Query Types

- **Persistent query**: Persistent queries are server-side queries that run indefinitely processing rows of events.
- **Push query**: A push query is a form of query issued by a client that subscribes to a result as it changes in real-time.
- **Pull query**: A pull query is a form of query issued by a client that retrieves a result as of "now", like a query against a traditional RDBMS.

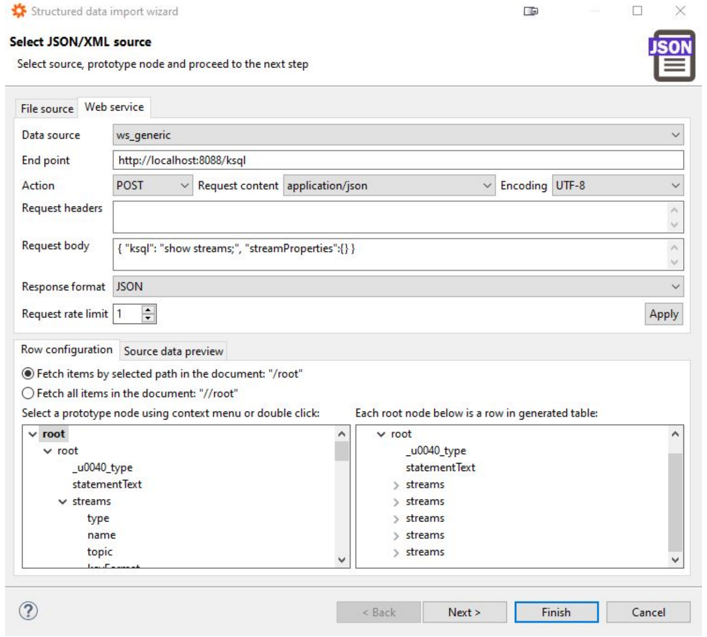
# Options to connect

# Options to connect from DV

- Via REST API
- Via generic JDBC with https://github.com/mmolimar/ksql-jdbc-driver
- Via DV driver implementing https://docs.ksqldb.io/en/latest/developer-guide/ksqldb-clients/java-client/

# Via REST API

- A very powerful way to interact with ksqlDB
- Push queries provide the challenge of a streaming result set: *Execute a push query by sending an HTTP request to the ksqlDB REST API, and the API sends back a chunked response of indefinite length.*
- So we want pull queries



Structured data import wizard

**Select JSON/XML source**

Select source, prototype node and proceed to the next step

File source | Web service

Data source: ws_generic

End point: http://localhost:8088/ksql

Action: POST | Request content: application/json | Encoding: UTF-8

Request headers:

Request body: { "ksql": "show streams;", "streamProperties":{} }

Response format: JSON

Request rate limit: 1

Row configuration | Source data preview

● Fetch items by selected path in the document: "/root"

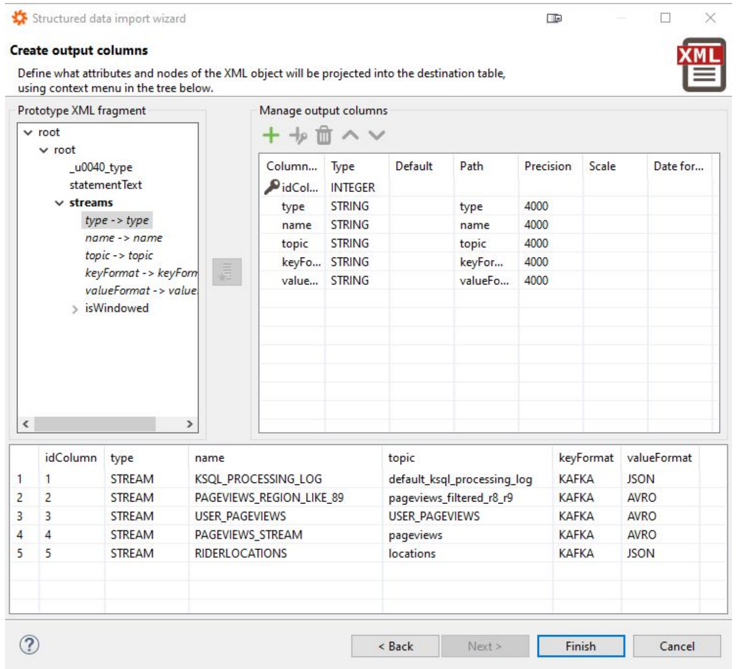
○ Fetch all items in the document: "//root"

Select a prototype node using context menu or double click:

```
root
  root
    _u0040_type
    statementText
    streams
      type
      name
      topic
```

Each root node below is a row in generated table:

```
root
  _u0040_type
  statementText
  > streams
  > streams
  > streams
  > streams
  > streams
```



Structured data import wizard

**Create output columns**

Define what attributes and nodes of the XML object will be projected into the destination table, using context menu in the tree below.

Prototype XML fragment

```
root
  root
    _u0040_type
    statementText
    streams
      type -> type
      name -> name
      topic -> topic
      keyFormat -> keyForm
      valueFormat -> value
    > isWindowed
```

Manage output columns

| Column... | Type | Default | Path | Precision | Scale | Date for... |
|---|---|---|---|---|---|---|
| idCol... | INTEGER | | | | | |
| type | STRING | | type | 4000 | | |
| name | STRING | | name | 4000 | | |
| topic | STRING | | topic | 4000 | | |
| keyFo... | STRING | | keyFor... | 4000 | | |
| value... | STRING | | valueFo... | 4000 | | |

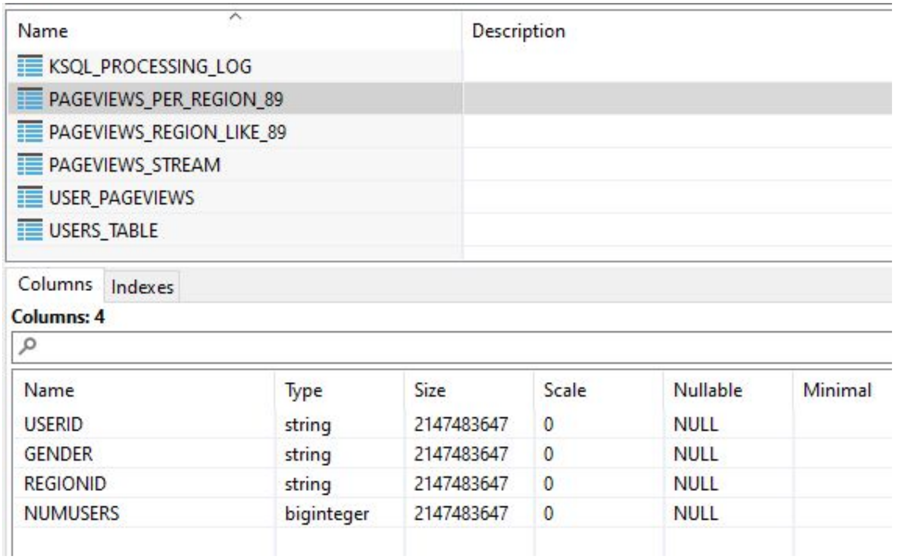
| | idColumn | type | name | topic | keyFormat | valueFormat |
|---|---|---|---|---|---|---|
| 1 | 1 | STREAM | KSQL_PROCESSING_LOG | default_ksql_processing_log | KAFKA | JSON |
| 2 | 2 | STREAM | PAGEVIEWS_REGION_LIKE_89 | pageviews_filtered_r8_r9 | KAFKA | AVRO |
| 3 | 3 | STREAM | USER_PAGEVIEWS | USER_PAGEVIEWS | KAFKA | AVRO |
| 4 | 4 | STREAM | PAGEVIEWS_STREAM | pageviews | KAFKA | AVRO |
| 5 | 5 | STREAM | RIDERLOCATIONS | locations | KAFKA | JSON |

# Via JDBC: Create a ksqlDB Connection

- Connector implements the java client in [https://docs.ksqldb.io/en/latest/developer-guide/ksqldb-clients/java-client/](https://docs.ksqldb.io/en/latest/developer-guide/ksqldb-clients/java-client/)
  The client supports pull and push queries; inserting new rows of data into existing ksqlDB streams; creation and management of new streams, tables, and persistent queries; and also admin operations such as listing streams, tables, and topics.

```
call SYSADMIN.createConnection ( 'ksqldb', 'ksqldb', 'host=localhost,port=8088' ) ;
call SYSADMIN.createDatasource ( 'ksqldb', 'ksqldb', '', 'supportsNativeQueries=true' ) ;
```

- Metadata is being loaded.



| Name | Description |
|---|---|
| KSQL_PROCESSING_LOG | |
| PAGEVIEWS_PER_REGION_89 | |
| PAGEVIEWS_REGION_LIKE_89 | |
| PAGEVIEWS_STREAM | |
| USER_PAGEVIEWS | |
| USERS_TABLE | |

Columns: 4

| Name | Type | Size | Scale | Nullable | Minimal |
|---|---|---|---|---|---|
| USERID | string | 2147483647 | 0 | NULL | |
| GENDER | string | 2147483647 | 0 | NULL | |
| REGIONID | string | 2147483647 | 0 | NULL | |
| NUMUSERS | biginteger | 2147483647 | 0 | NULL | |

# Demo: Creating and Querying objects in ksqlDB

# Create and query ksqlDB Stream

We are following the confluent platform quickstart
[https://docs.confluent.io/platform/current/platform-quickstart.html#quick-start-for-cp](https://docs.confluent.io/platform/current/platform-quickstart.html#quick-start-for-cp)

```
call "ksqldb.native"(
  'CREATE or replace STREAM pageviews_stream WITH (KAFKA_TOPIC=''pageviews'',
VALUE_FORMAT=''AVRO'');', 'stmt');;
```

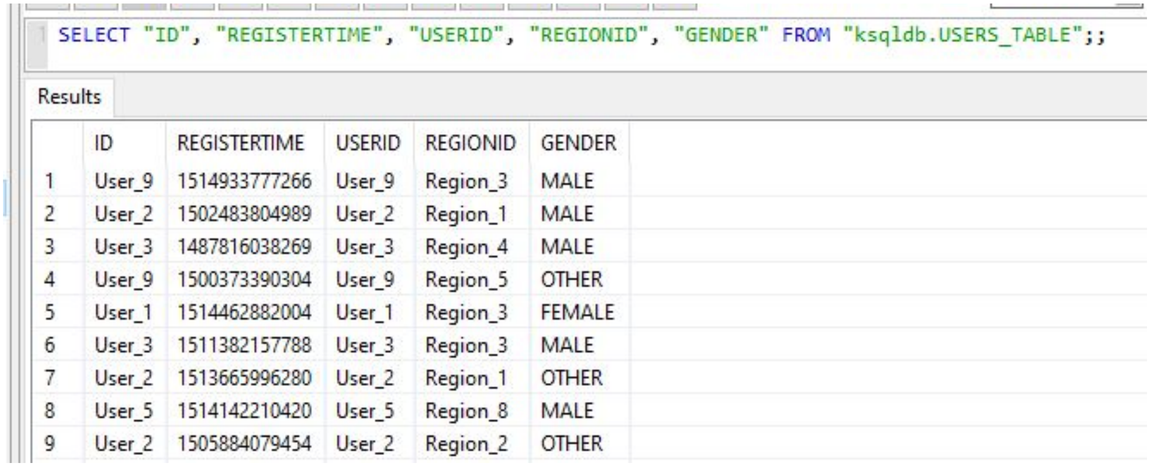
note the second parameter, it can either be **stmt, query** or **stream**

```
-- direct select
SELECT "VIEWTIME", "USERID", "PAGEID" FROM "ksqldb.PAGEVIEWS_STREAM" LIMIT 10;;

-- native way:
select
       *
    from
        ( call "ksqldb.native" ( 'SELECT VIEWTIME, USERID, PAGEID FROM pageviews_stream EMIT
CHANGES LIMIT 10;', 'query' ) ) a
       , ARRAYTABLE ( a.tuple COLUMNS "VIEWTIME" bigint, "USERID" string, "PAGEID" string ) as w;;

select
       cast (tuple as string)
    from
        ( call "ksqldb.native" ( 'SELECT VIEWTIME, USERID, PAGEID FROM pageviews_stream EMIT
CHANGES LIMIT 10;', 'query' ) ) a
```

# Create and query ksqlDB Table

```
call "ksqldb.native"(
 'CREATE OR REPLACE TABLE users_table (id VARCHAR PRIMARY KEY)
  WITH (KAFKA_TOPIC=''users'', VALUE_FORMAT=''AVRO'');', 'stmt');;
```

- Tables in ksqlDB need a primary key!
- Result will be the same for each query
- Result will be in fetching mode in DV until cancelled



```
SELECT "ID", "REGISTERTIME", "USERID", "REGIONID", "GENDER" FROM "ksqldb.USERS_TABLE";;
```

Results

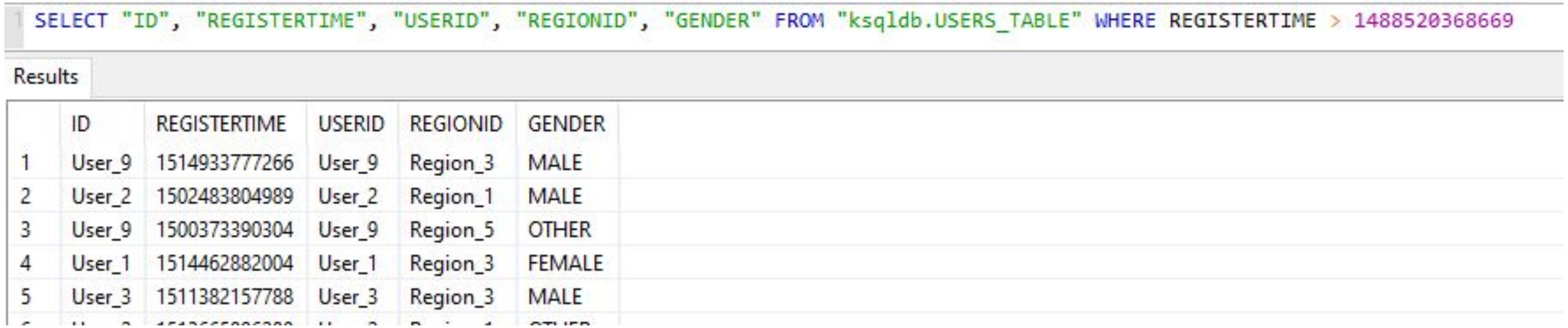
| | ID | REGISTERTIME | USERID | REGIONID | GENDER |
|---|---|---|---|---|---|
| 1 | User_9 | 1514933777266 | User_9 | Region_3 | MALE |
| 2 | User_2 | 1502483804989 | User_2 | Region_1 | MALE |
| 3 | User_3 | 1487816038269 | User_3 | Region_4 | MALE |
| 4 | User_9 | 1500373390304 | User_9 | Region_5 | OTHER |
| 5 | User_1 | 1514462882004 | User_1 | Region_3 | FEMALE |
| 6 | User_3 | 1511382157788 | User_3 | Region_3 | MALE |
| 7 | User_2 | 1513665996280 | User_2 | Region_1 | OTHER |
| 8 | User_5 | 1514142210420 | User_5 | Region_8 | MALE |
| 9 | User_2 | 1505884079454 | User_2 | Region_2 | OTHER |

# Pull Queries

- Using the KSQLDB directly, see below challenge
  - `select * from USERS_TABLE WHERE REGISTERTIME > 1488520368669;`
    - *WHERE clause missing key column for disjunct: (REGISTERTIME > 1488520368669). See https://cnfl.io/queries for more info. Add EMIT CHANGES if you intended to issue a push query. Pull queries require a WHERE clause that: - includes a key equality expression, e.g. `SELECT * FROM X WHERE <key-column>=Y;`. - in the case of a multi-column key, is a conjunction of equality expressions that cover all key columns. If more flexible queries are needed, table scans can be enabled by setting ksql.query.pull.table.scan.enabled=true. Statement: select * from USERS_TABLE WHERE REGISTERTIME > 1488520368669;*
  - `select * from USERS_TABLE where ID='User_3';`
    - *The `USERS_TABLE` table isn't queryable. To derive a queryable table, you can do 'CREATE TABLE QUERYABLE_USERS_TABLE AS SELECT * FROM USERS_TABLE'. See https://cnfl.io/queries for more info. Add EMIT CHANGES if you intended to issue a push query. Statement: select * from USERS_TABLE Where ID='User_3';*

But our connector takes care of this:

```
SELECT "ID", "REGISTERTIME", "USERID", "REGIONID", "GENDER" FROM "ksqldb.USERS_TABLE" WHERE REGISTERTIME > 1488520368669
```



```
SELECT "ID", "REGISTERTIME", "USERID", "REGIONID", "GENDER" FROM "ksqldb.USERS_TABLE" WHERE REGISTERTIME > 1488520368669
```

| | ID | REGISTERTIME | USERID | REGIONID | GENDER |
|---|---|---|---|---|---|
| 1 | User_9 | 1514933777266 | User_9 | Region_3 | MALE |
| 2 | User_2 | 1502483804989 | User_2 | Region_1 | MALE |
| 3 | User_9 | 1500373390304 | User_9 | Region_5 | OTHER |
| 4 | User_1 | 1514462882004 | User_1 | Region_3 | FEMALE |
| 5 | User_3 | 1511382157788 | User_3 | Region_3 | MALE |

# Create and query a joined stream

```
 call "ksqldb.native"(
  'CREATE STREAM user_pageviews
  AS SELECT users_table.id AS userid, pageid, regionid, gender
    FROM pageviews_stream
    LEFT JOIN users_table ON pageviews_stream.userid = users_table.id EMIT CHANGES;', 'stmt');;

;;

  SELECT "USERID", "PAGEID", "REGIONID", "GENDER" FROM "ksqldb.USER_PAGEVIEWS" limit 10;;
```

- JOIN operation is happening on the ksqlDB side

# Further operations: Filtered Stream

```
 call "ksqldb.native"(
  'CREATE OR REPLACE STREAM pageviews_region_like_89
  WITH (KAFKA_TOPIC=''pageviews_filtered_r8_r9'', VALUE_FORMAT=''AVRO'')
    AS SELECT * FROM user_pageviews
    WHERE regionid LIKE ''%_8'' OR regionid LIKE ''%_9''
EMIT CHANGES;', 'stmt');;
```

- Derived stream
- ksqlDB processing

# Further operations: Windowed Table

```
 call "ksqldb.native"(
  'CREATE TABLE pageviews_per_region_89 WITH (KEY_FORMAT=''JSON'')
  AS SELECT userid, gender, regionid, COUNT(*) AS numusers
    FROM pageviews_region_like_89
    WINDOW TUMBLING (SIZE 30 SECOND)
    GROUP BY userid, gender, regionid
    HAVING COUNT(*) > 1
EMIT CHANGES;', 'stmt');;
```

- Here, we look into the last 30 seconds of events
- Persistent

# Creating Tables / Writing into Streams

```
call ks.native('create table test1(id integer primary key, a varchar) WITH
(kafka_topic=''locations'', value_format=''json'', partitions=1);','stmt');;

call ks.native('insert into test1 values(1, ''aaa'');','stmt');

call "ksqldb.native"('INSERT INTO riderLocations (profileId, latitude, longitude) VALUES
(''c2309eec'', 37.7877, -122.4205);', 'stmt');;
```

- This enables you to write data from DV into a kafka stream

# Any feedback / questions?

# Thank you!

Please feel free to contact us at:
presales@datavirtuality.com

or

visit us at:
datavirtuality.com