GHENT UNIVERSITY

imec

POLITECNICO
MILANO 1863

QUANTIA
consulting

Web Stream Processing with OntopStream

The Web Conference 2022

Pieter Bonte
Matteo Belcao
Marco Balduini
Emanuele Della Valle

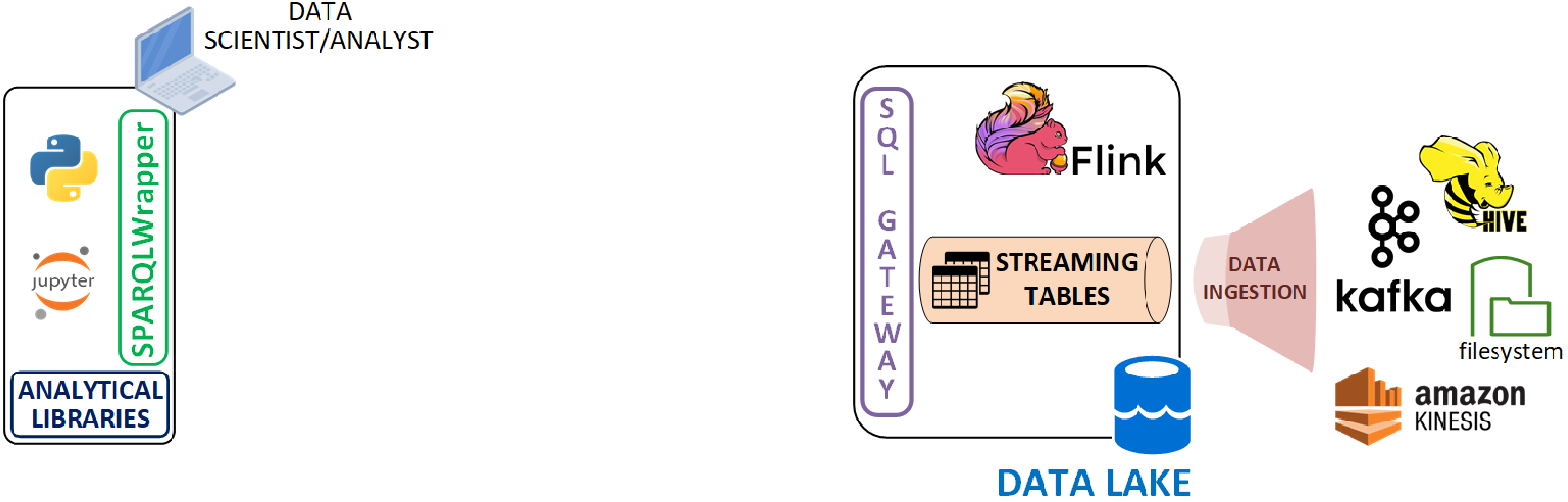
OntopStream

Ontology-Based Data Access

- **Ontology-Based Data Access (OBDA)** softwares aim to solve data integration problems…

- **Virtual Knowledge Graph (VKG)** approach:
  - additional semantic layer on top of the data
  - relational data sources abstraction, exposed as RDF triples
  - SPARQL queries to access the data
  - automatic SPARQL → SQL query rewritings

KG-Empowered Continuous Analytics



DATA SCIENTIST/ANALYST

SPARQLWrapper

jupyter

ANALYTICAL LIBRARIES

SQL GATEWAY

Flink

STREAMING TABLES

DATA LAKE

DATA INGESTION

kafka

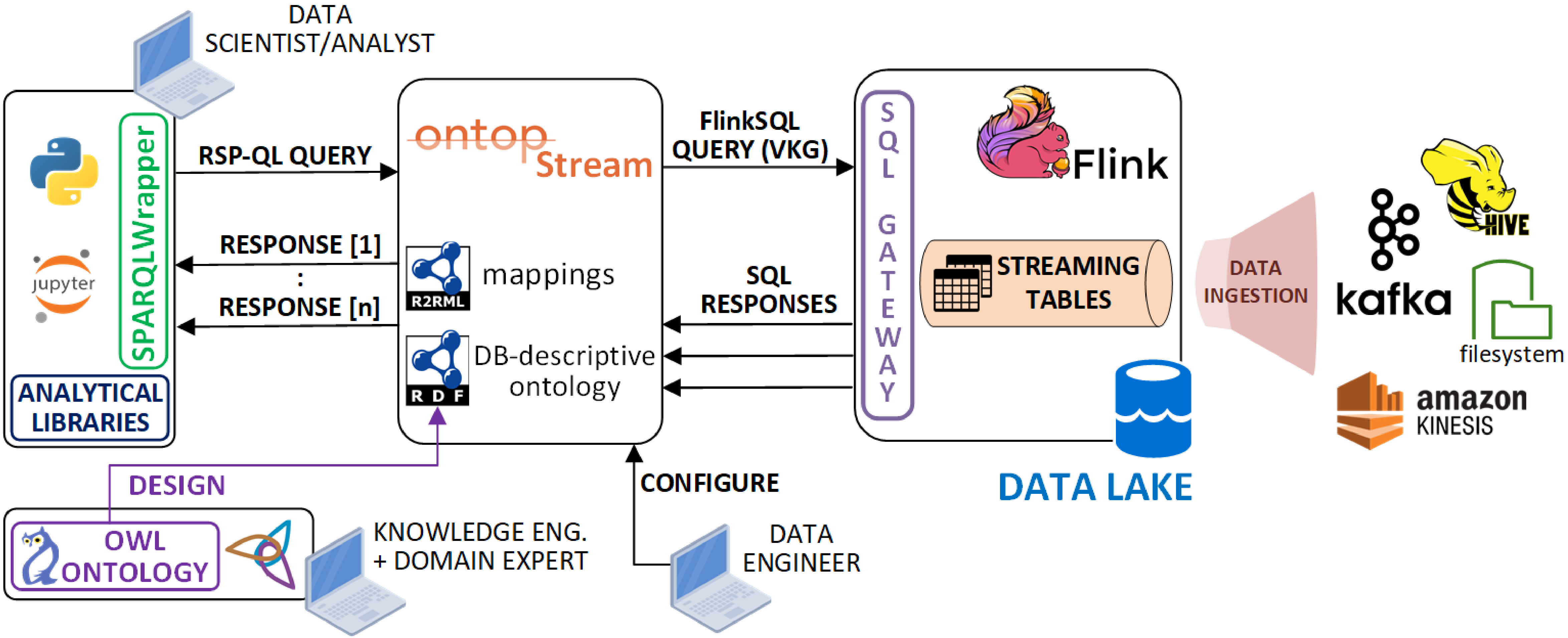
HIVE

filesystem

amazon KINESIS

KG-Empowered Continuous Analytics

Streaming-VKGs as a bridge between Stream Processing and Semantic Techs



OntopStream

- Developed as an extension of the Ontop OBDA system (Java)

- Query relational data streams
    - stored and managed in Apache Flink dynamic tables
    - with RSP-QL continuous queries (windowed / not windowed)

- Get RDF streams of responses

- Two distributions:
    - OntopStream-CLI
    - OntopStream-Endpoint (only HTTP calls)

OntopStream: design decisions

- paradigm shift from traditional OBDA to Streaming-OBDA

- design decisions:

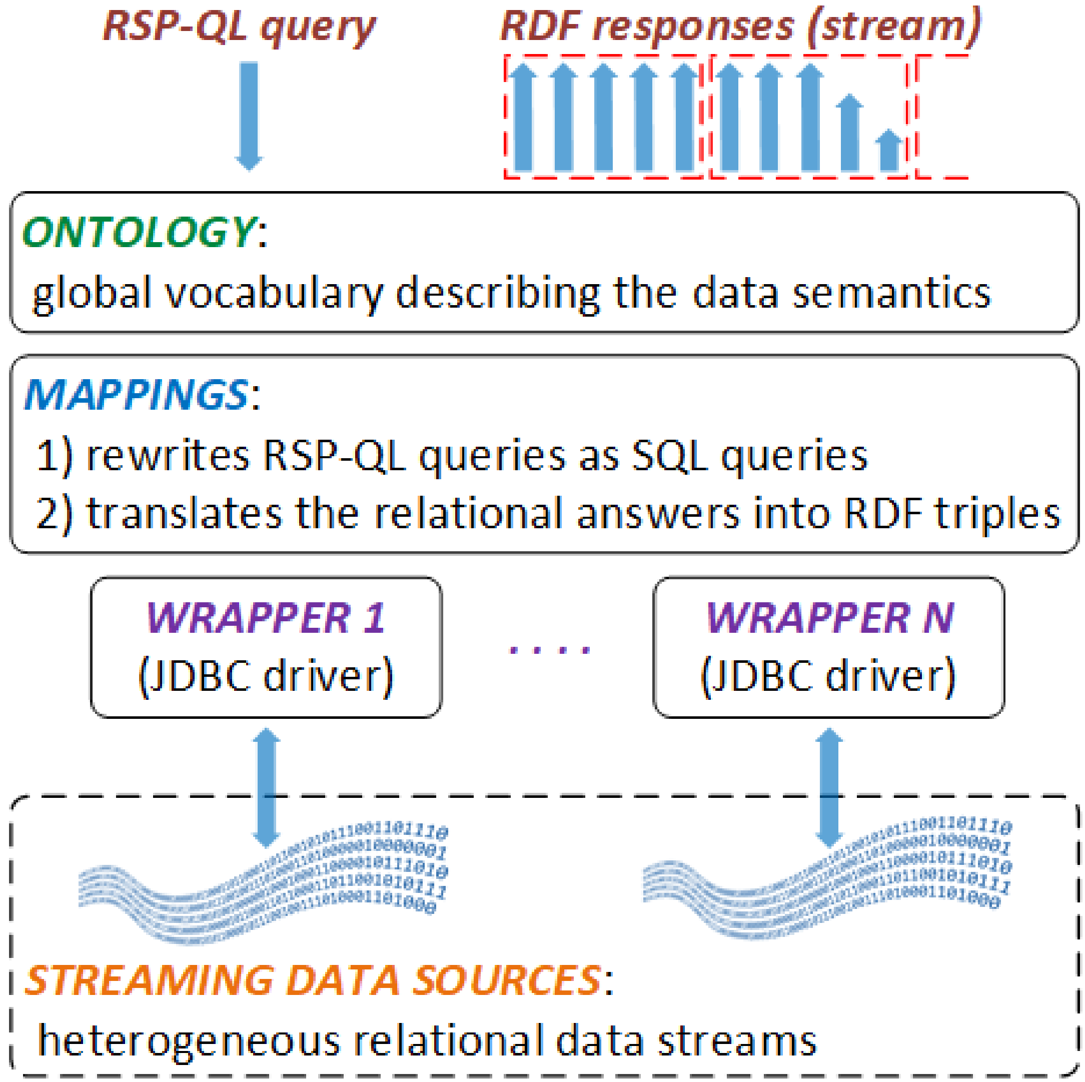
  1. extend the **`Flink JDBC driver`**
  2. re-design part of the **`ontop-engine`** to accept **`RSP-QL`** queries
  3. Streaming Virtual Knowledge Graph query rewriting approach
  4. include support for **`RDF streams`** of query outputs

Streaming Virtual Knowledge Graphs in OntopStream

Data Analyst/Scientist

Knowledge Engineer

Data Engineer



RSP-QL query

RDF responses (stream)

ONTOLOGY:
global vocabulary describing the data semantics

MAPPINGS:
1) rewrites RSP-QL queries as SQL queries
2) translates the relational answers into RDF triples

WRAPPER 1
(JDBC driver)

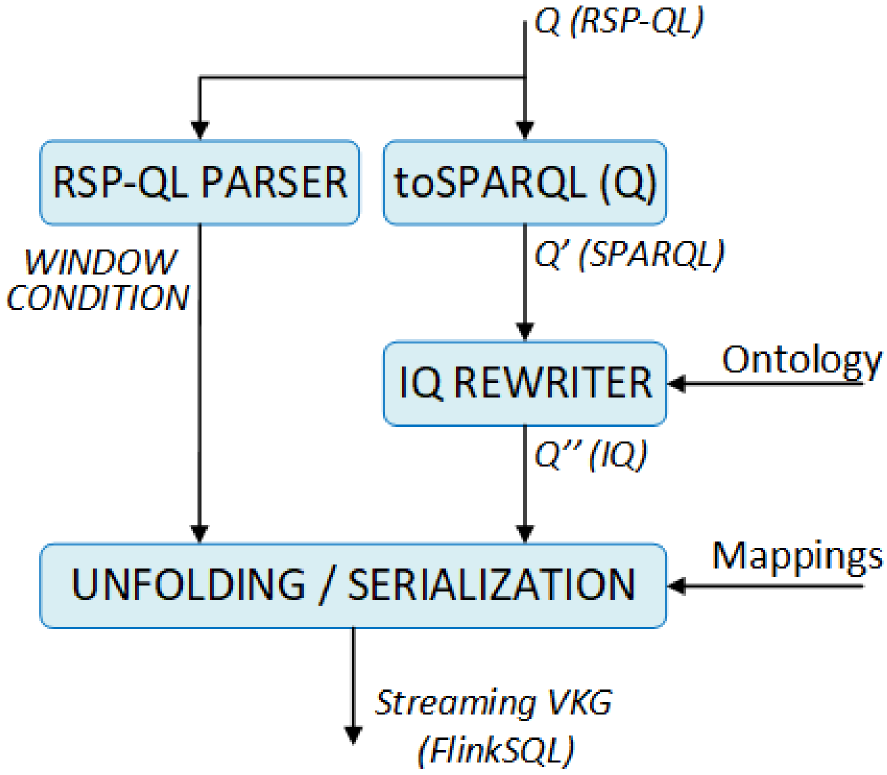
. . . .

WRAPPER N
(JDBC driver)

STREAMING DATA SOURCES:
heterogeneous relational data streams

Streaming Virtual Knowledge Graph query rewriter

- **`rsp4j`** parser to extract window conditions
- Intermediate Query rewriter unchanged
- **`IQ`** representation:
  - created w.r.t to the Ontology **`O`**
  - unfolded in a **`Streaming VKG`** tree
- Each tree node corresponds to a pseudo-SQL statement
- Streaming VKG serialization in a **`FlinkSQL query`**, add window condition **`W`** if existing



Tutorial: pipeline setup

Starting-up the resources

- Requirements: *docker* and *docker-compose* (installation guide)

- Clone the repository: `git clone https://github.com/pbonte/WSP-TheWebConf2022Tutorial.git`

- Start the tutorial environment
  - Open the right folder: `cd exercises/part3`
  - Streaming resources (Flink, Kafka) and JupyterLab:

    Windows: `sudo docker-compose -f flink-kafka-win.yml up -d`

    Mac/UNIX: `sudo docker-compose -f flink-kafka-unix.yml up -d`

  - Flink JDBC Gateway:

    ```
    docker exec -it sql-client /opt/flink-sql-gateway-0.2-SNAPSHOT/bin/sql-gateway.sh --library /opt/sql-client/lib
    ```

    Note: keep the JDBC endpoint alive until you need the service (don't close the terminal window)

Business Scenario: Rental Company

A car rental company has recently decided to **unify the information systems** of **two branches** using ontology-based data access techniques.

Both the branches:

- have a real-time data management infrastructure
- store the rental records in Kafka topics

However, they handles the data differently:

- *Branch A* uses two separate Kafka topics for trucks and cars
- *Branch B* stores all the rentals in a single topic, but the users’ data are kept in a sperate topic

Business Requirements

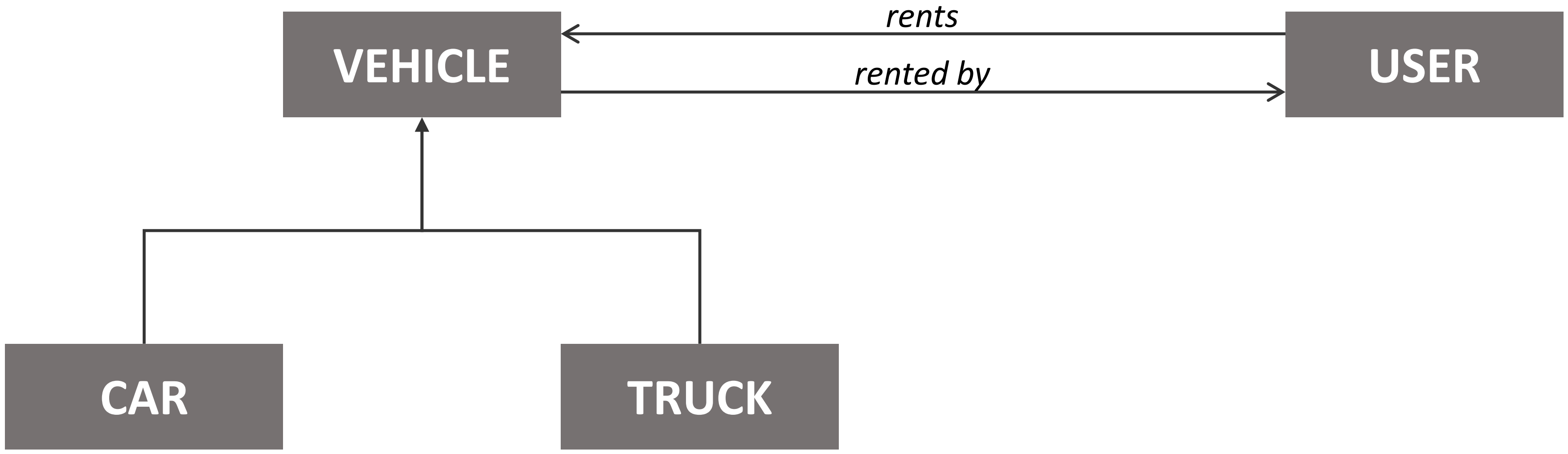
The company is booming, and has in plan to acquire soon new branches.

Therefore, the company wants to make the integration process scalable, so that can be easily extended to all its new branches

They need a data integration solution that:

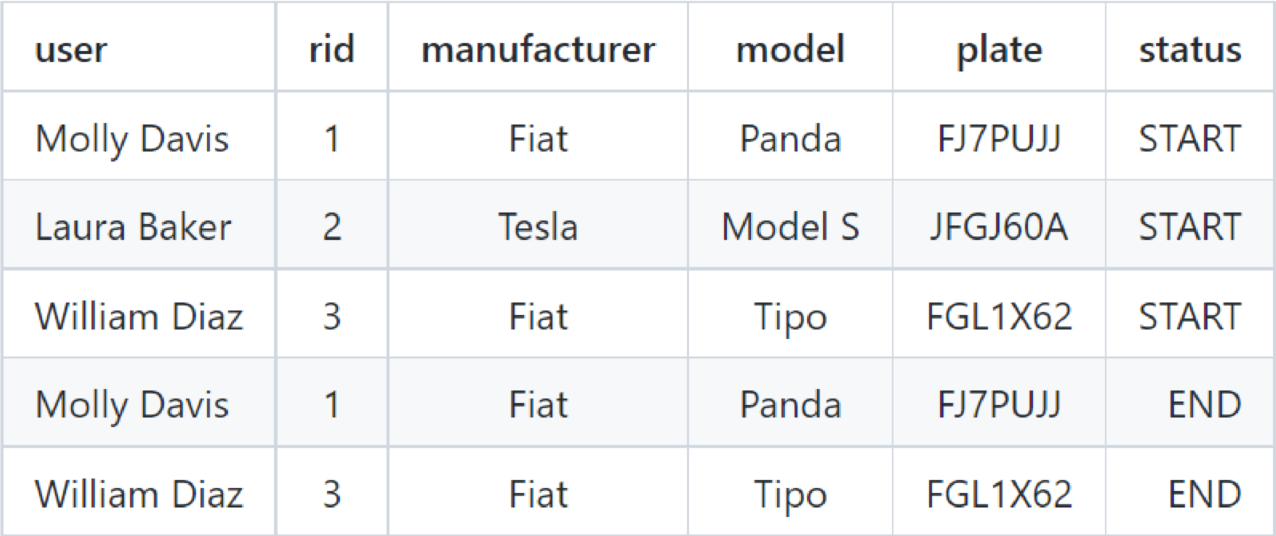
- provides an **unified logical view** of their data
- enables to **query in real-time** their data
- can be used with **python notebooks** for further analyses

Kafka Topics: High-Level View



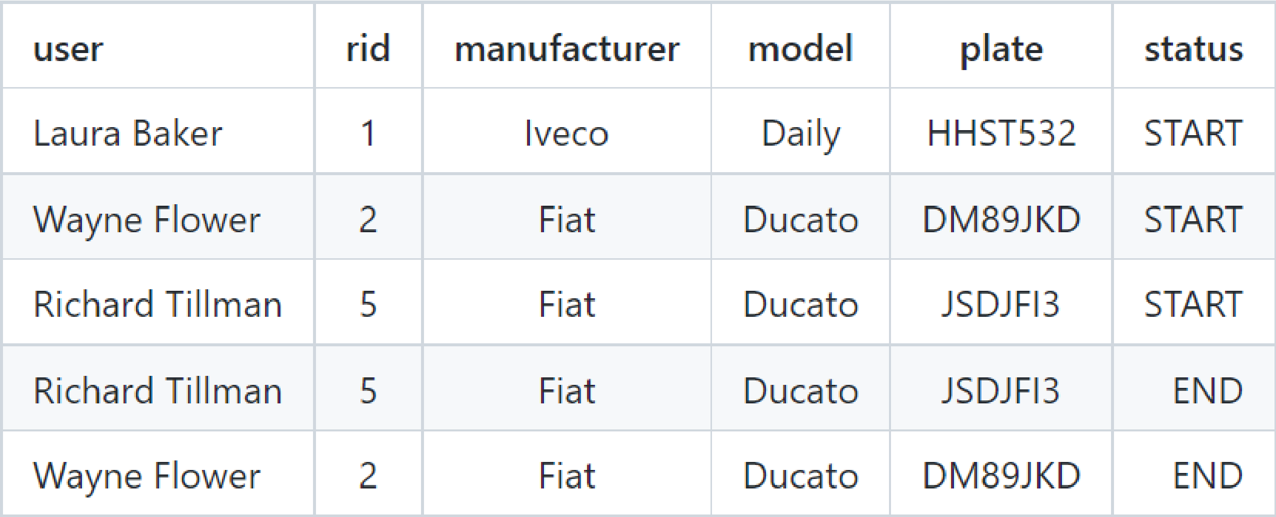
PROBLEM: the two branches uses different data structures…

Kafka topics: Branch A



user	rid	manufacturer	model	plate	status
Molly Davis	1	Fiat	Panda	FJ7PUJJ	START
Laura Baker	2	Tesla	Model S	JFGJ60A	START
William Diaz	3	Fiat	Tipo	FGL1X62	START
Molly Davis	1	Fiat	Panda	FJ7PUJJ	END
William Diaz	3	Fiat	Tipo	FGL1X62	END

DEALER1_CARS

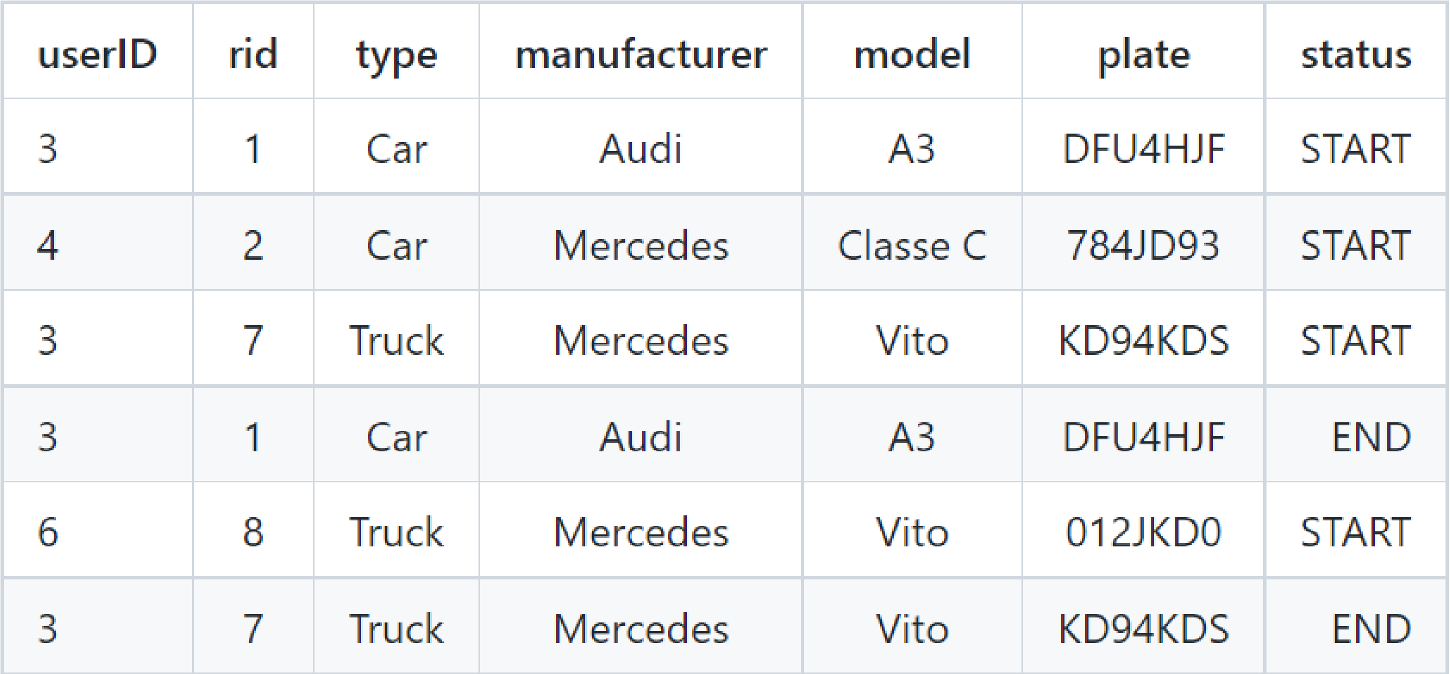


user	rid	manufacturer	model	plate	status
Laura Baker	1	Iveco	Daily	HHST532	START
Wayne Flower	2	Fiat	Ducato	DM89JKD	START
Richard Tillman	5	Fiat	Ducato	JSDJFI3	START
Richard Tillman	5	Fiat	Ducato	JSDJFI3	END
Wayne Flower	2	Fiat	Ducato	DM89JKD	END

DEALER1_TRUCKS

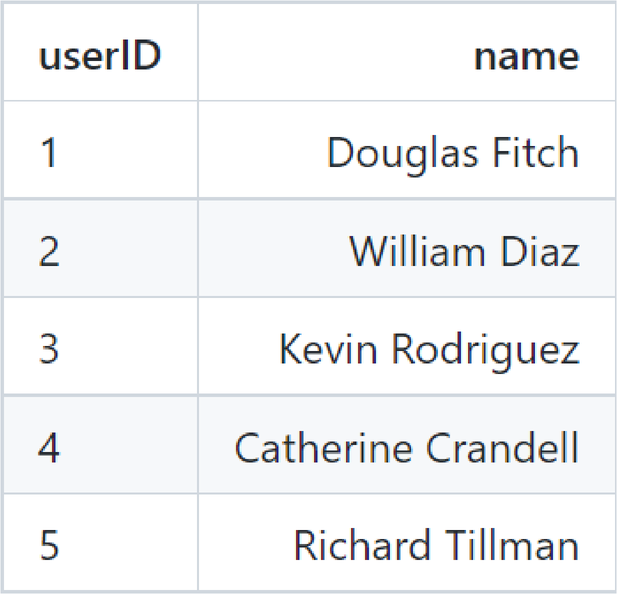
Kafka topics: Branch B

DEALER2_VEHICLES



userID	rid	type	manufacturer	model	plate	status
3	1	Car	Audi	A3	DFU4HJF	START
4	2	Car	Mercedes	Classe C	784JD93	START
3	7	Truck	Mercedes	Vito	KD94KDS	START
3	1	Car	Audi	A3	DFU4HJF	END
6	8	Truck	Mercedes	Vito	012JKD0	START
3	7	Truck	Mercedes	Vito	KD94KDS	END

DEALER2_USERS



userID	name
1	Douglas Fitch
2	William Diaz
3	Kevin Rodriguez
4	Catherine Crandell
5	Richard Tillman

Kafka topics: Flink ingestion

- Data acquisition in Flink can be automated

- Design the topics ingestion in Flink:
  - Flink streaming tables
    - queried with FlinkSQL continuous queries, recorded in Flink
  - Kafka connector for Flink (Table & SQL API):
    - files*:*
      - ***sql-client-conf.yaml***: Kafka → Flink
      - ***sql-gateway-defaults.yaml***: Flink JDBC Gateway
    - **table schema**: topics fields, datatypes, watermarks, …
    - **connector properties**: Kafka address, schema registry, …

Example: DEALER2_VEHICLES topic

```
- name: D2_VEHICLES
  type: source
  update-mode: append
  schema:
  - name: userID
    type: BIGINT
  - name: rid
    type: BIGINT
  - name: type
    type: STRING
  - name: manufacturer
    type: STRING
  - name: model
    type: STRING
  - name: plate
    type: STRING
  - name: status
    type: STRING
  - name: ts
    type: STRING
```

TABLE NAME

data: SOURCE → FLINK

append new data, when is available from the source (Kafka)

TABLE SCHEMA

Reference: [Apache Kafka SQL Connector]()

Example: DEALER2_VEHICLES topic

TABLE SCHEMA

```
- name: parsed_timestamp
  type: TIMESTAMP
  rowtime:
   timestamps:
    type: "from-field"
    from: "ts"
   watermarks:
    type: "periodic-bounded"
    delay: "5"
```

reference field for time-based operations (WINDOWS)

SOURCE CONNECTOR

```
connector:
  property-version: 1
  type: kafka
  version: universal
  topic: DEALER2_VEHICLES
  startup-mode: earliest-offset
  properties:
  - key: bootstrap.servers
   value: kafka:9092
 format:
  property-version: 1
  type: json
  schema: "ROW(userID BIGINT, rid BIGINT, type STRING, manufacturer STRING,
          model STRING, plate STRING, status STRING, ts STRING)"
```

source type

source address

topic schema

Relational Streaming Data Integration…

Now, we have a Flink streaming table for each Kafka topic

- *DEALER1_CARS* and *DEALER1_TRUCKS*
- *DEALER2_VEHICLES* and *DEALER2_USERS*

The data streams are still not integrated!!!

Relational Streaming Data Integration…

Now, we have a Flink streaming table for each Kafka topic

- *DEALER1_CARS* and *DEALER1_TRUCKS*
- *DEALER2_VEHICLES* and *DEALER2_USERS*

We can use **OntopStream** to create a **unified logical view** of the data streams…

Flink relational streams:

- exposed to OntopStream using the ***Flink JDBC Gateway***
- can be queried with ***FlinkSQL*** continuous queries

Relational Streaming Data Integration…

- Onpstream automates:
  - **RSP-QL → FlinkSQL** query rewriting
  - **relational → RDF** response streams translation

- To use OntopStream for the streaming data integration tasks we need:
  1. **Ontology**: provides the unified logical view to the user
     - Classes
     - Object Properties
     - Data Properties
  2. **Streaming-VKG mappings**: bridges the ontology with data streams (Kafka messages in Flink)
  3. **JDBC connection** configuration

1) Ontology Design

Ontology design made using Protégé, an open-source graphical ontology editor.

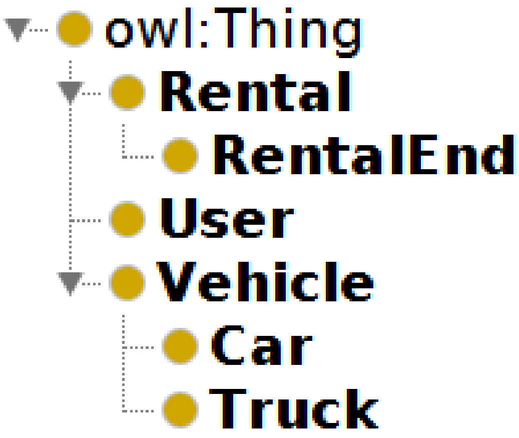
Classes:

- express the **logical concepts** of the unified logical view
- The **Car** and **Truck** concepts are expressed as subclasses of **Vehicle**

  i.e., a *Tesla Model X* is a Car, but also a transportation Vehicle

- **RentalEnd** is a specialization of (subclass) **Rental**

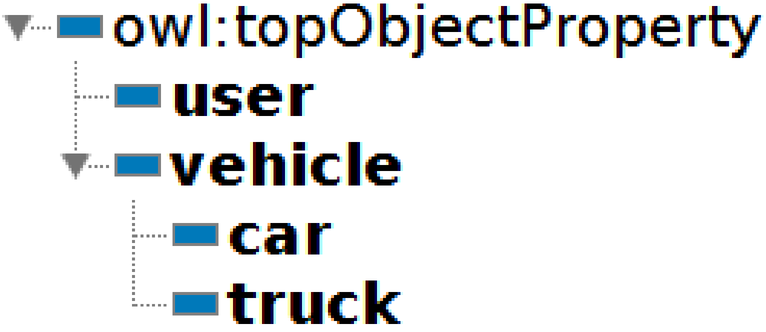
  it will be useful later for queries about ended rentals



1) Ontology Design

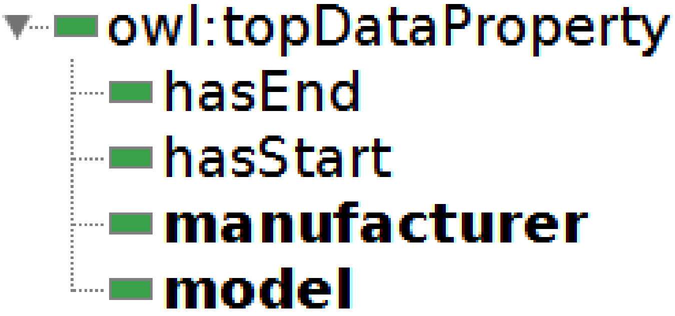
Object Properties:

- ease the mapping process
- express implicit domain/range restrictions on Class istances:
  - the **user** property range is **User**
  - the **vehicle** property range is **Vehicle**



Data Properties:

- expose the Kafka messages entries
  - Vehicle details (manufacturer, model)
  - Timestamps
  - Users personal information (name)



2) Streaming-VKG mappings

OntopStream answers **RSP-QL queries** with **RDF streams** of semantically-enriched responses based on:

- ontological concepts
- relational data streams: retrieved through **Streaming VKG queries** registered in Flink

Streaming-VKG mapping

- binding between a set of RDF statements and FlinkSQL selection query
- connects the ontological layer terms to data streams (in this tutorial, Kafka messages in Flink)
- consists of:
  - **MappingID**: friendly name to identify the mapping
  - **Source**: FlinkSQL query for the data extraction from the Flink streaming tables
  - **Target**: one or more RDF statements corresponding to the VKG generated by the single entry obtained from the data extracted with the Source query

2) Streaming-VKG mappings: Baranch A

Entities:

- **Rental:** each rental ID in the stream
- **Vehicle**: plate numbers
- **User:** client names

Kind of rented vehicle?

- **D1_CARS** table stores **Cars** data
- **D1_TRUCKS** table stores **Trucks** data

Start or ended lease?

- the **status** field refers to the rental state
- we can use a **WHERE** clause in the source query to filter out rentals by their status:
    - **status=’START’** retrieves the starting rentals Kafka messages
    - **status=’END’** retrieves the ended rentals Kafka messages

2) Streaming-VKG mappings: Baranch A

RentalEnd subclass of **Rental**:

- to ease the complexity of queries asking only for ended rentals, we use the subclass specialization

Started rentals [rentals.obda]

```
mappingId   DEALER1−CarRental
target      :D1_C{rid} a :Rental; :user :{user}; :hasStart {ts}^^xsd:dateTime; :car :{plate}. :{plate} a :Car; :manufacturer {manufacturer}; :model {model}.
source      SELECT rid, user, ts, plate, manufacturer, model FROM D1_CARS WHERE status=’START’

mappingId   DEALER1−TruckRental
target      :D1_T{rid} a :Rental; :user :{user}; :hasStart {ts}^^xsd:dateTime; :truck :{plate}. :{plate} a :Truck; :manufacturer {manufacturer}; :model {model}.
source      SELECT rid, user, ts, plate, manufacturer, model FROM D1_TRUCKS WHERE status=’START’
```

Ended rentals [rentals.obda]

```
mappingId   DEALER1−CarRentalEnd
target      :D1_C{rid} a :RentalEnd; :hasEnd {ts}^^xsd:dateTime; :car :{plate}.
source      SELECT rid,ts,plate FROM D1_CARS WHERE status=’END’

mappingId   DEALER1−TruckRentalEnd
target      :D1_T{rid} a :RentalEnd; :hasEnd {ts}^^xsd:dateTime; :truck :{plate}.
source      SELECT rid,ts,plate FROM D1_TRUCKS WHERE status=’END’’
```

2) Streaming-VKG mappings: Branch B

Entities:

- **Rental:** each rental ID in the stream
- **Vehicle**: plate numbers
- **User:** client names

Kind of rented vehicle?

- the **type** field refers to the kind of vehicle in the **D2_VEHICLES** table
- for <u>starting rentals</u>, we can use a **WHERE** clause in the source query to determine the vehicle:
  - **type=‘Car’** retrieves **Car** rental entries
  - **type=‘Truck’** retrieves **Truck** rental entries
- for <u>ending rentals</u>, since the vehicle class is determined in the starting rental messages:
  - use the generic **vehicle** object property (property range is **Vehicle**)

Start or ended lease? (same as Branch A)

- use the **WHERE** clause in the source query to filter out rentals by their **status** field

2) Streaming-VKG mappings: Branch B

Users are kept in a separate topic:

- need to combine the Flink streaming tables **D2_VEHICLES** and **D2_USERS**
- FlinkSQL source query with a **JOIN** over the **userID** field

RentalEnd subclass of **Rental**: (same as Branch A)

- to ease the complexity of queries asking only for ended rentals, we use the subclass specialization

Started/Ended rentals [rentals.obda]

```
mappingId   DEALER2−CarRental
target      :D2_{rid} a :Rental; :user :{name}; :hasStart {ts}^^xsd:dateTime; :car :{plate}. :{plate} a :Car; :manufacturer {manufacturer}; :model {model}.
source      SELECT rid,name,ts,plate,manufacturer,model FROM D2_VEHICLES,D2_USERS WHERE D2_VEHICLES.userID=D2_USERS.userID
                        AND type=’Car’ AND status=’START’

mappingId   DEALER2−TruckRental
target      :D2_{rid} a :Rental; :user :{name}; :hasStart {ts}^^xsd:dateTime; :truck :{plate}. :{plate} a :Truck; :manufacturer {manufacturer}; :model {model}.
source      SELECT rid,name,ts,plate,manufacturer,model FROM D2_VEHICLES,D2_USERS WHERE D2_VEHICLES.userID=D2_USERS.userID
                        AND type=’Truck’ AND status=’START’

mappingId   DEALER2−RentalEnd
target      :D2_{rid} a :RentalEnd; :hasEnd {ts}^^xsd:dateTime; :vehicle :{plate}.
source      SELECT rid,ts,plate FROM D2_VEHICLES,D2_USERS WHERE D2_VEHICLES.userID=D2_USERS.userID AND status=’END’
```

3) JDBC connection

- OntopStream interacts with Apache Flink:
  - through **JDBC calls**
  - using a **custom JDBC driver**

- Before starting OntopStream, we need to configure the connection to the **Flink JDBC Gateway**

- The configuration must be specified in a **property file**, passed as input to OntopStream on its startup

rentals.propery

```
jdbc.url=jdbc:flink://sql-client:8083?planner=blink
jdbc.driver=com.ververica.flink.table.jdbc.FlinkDriver
jdbc.user=
jdbc.name=test-RSE-streaming
jdbc.fetchSize=1
jdbc.password=
```

Tutorial: OntopStream hands-on

OntopStream startup

- The OntopStream docker image is available on DockerHub

  hub.docker.com/r/chimerasuite/ontop-stream

- We can now start the OntopStream endpoint using the command:

```
docker-compose -f ontop.yml up -d
```

- If we look at the configuration in the **ontop.yml** file we can see the three input files:
  - **tutorial.owl**: contained the ontology describing the user unified logical view
  - **tutorial.obda**: the Streaming-VKG mappings we've designed
  - **tutorial.properties**: the JDBC connection properties

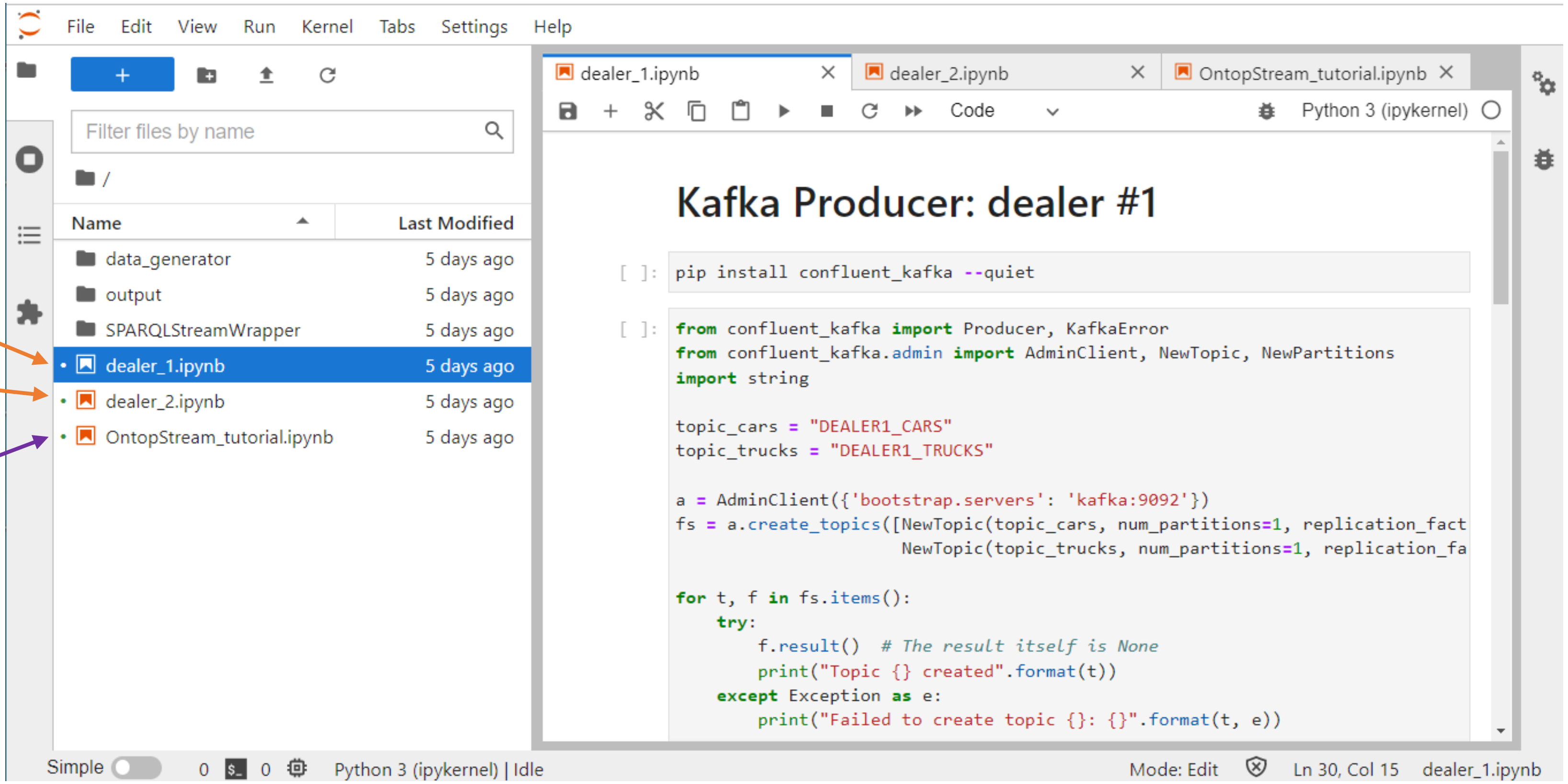
JupyterLab setup

- We're finally ready for querying the streams of data using a python notebook
- The platform is accessible from `http://<IP-ADDRESS>:8888/lab?token=TEST`

Kafka producer #1

Kafka producer #2

OntopStream demo



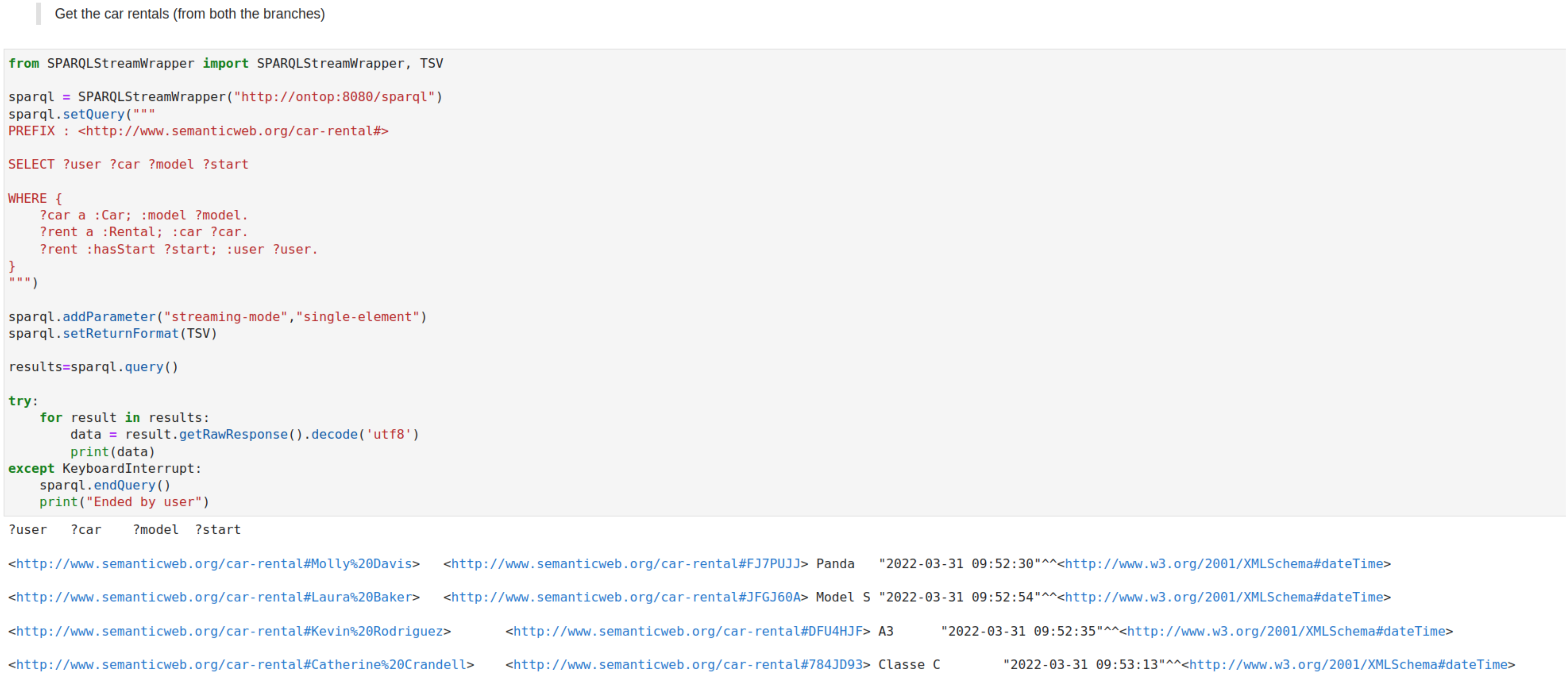
Name	Last Modified
data_generator	5 days ago
output	5 days ago
SPARQLStreamWrapper	5 days ago
dealer_1.ipynb	5 days ago
dealer_2.ipynb	5 days ago
OntopStream_tutorial.ipynb	5 days ago

Kafka Producer: dealer #1

```
[ ]: pip install confluent_kafka --quiet
```

```
[ ]: from confluent_kafka import Producer, KafkaError
     from confluent_kafka.admin import AdminClient, NewTopic, NewPartitions
     import string

     topic_cars = "DEALER1_CARS"
     topic_trucks = "DEALER1_TRUCKS"

     a = AdminClient({'bootstrap.servers': 'kafka:9092'})
     fs = a.create_topics([NewTopic(topic_cars, num_partitions=1, replication_fact
                           NewTopic(topic_trucks, num_partitions=1, replication_fa

     for t, f in fs.items():
         try:
             f.result()  # The result itself is None
             print("Topic {} created".format(t))
         except Exception as e:
             print("Failed to create topic {}: {}".format(t, e))
```

First Query

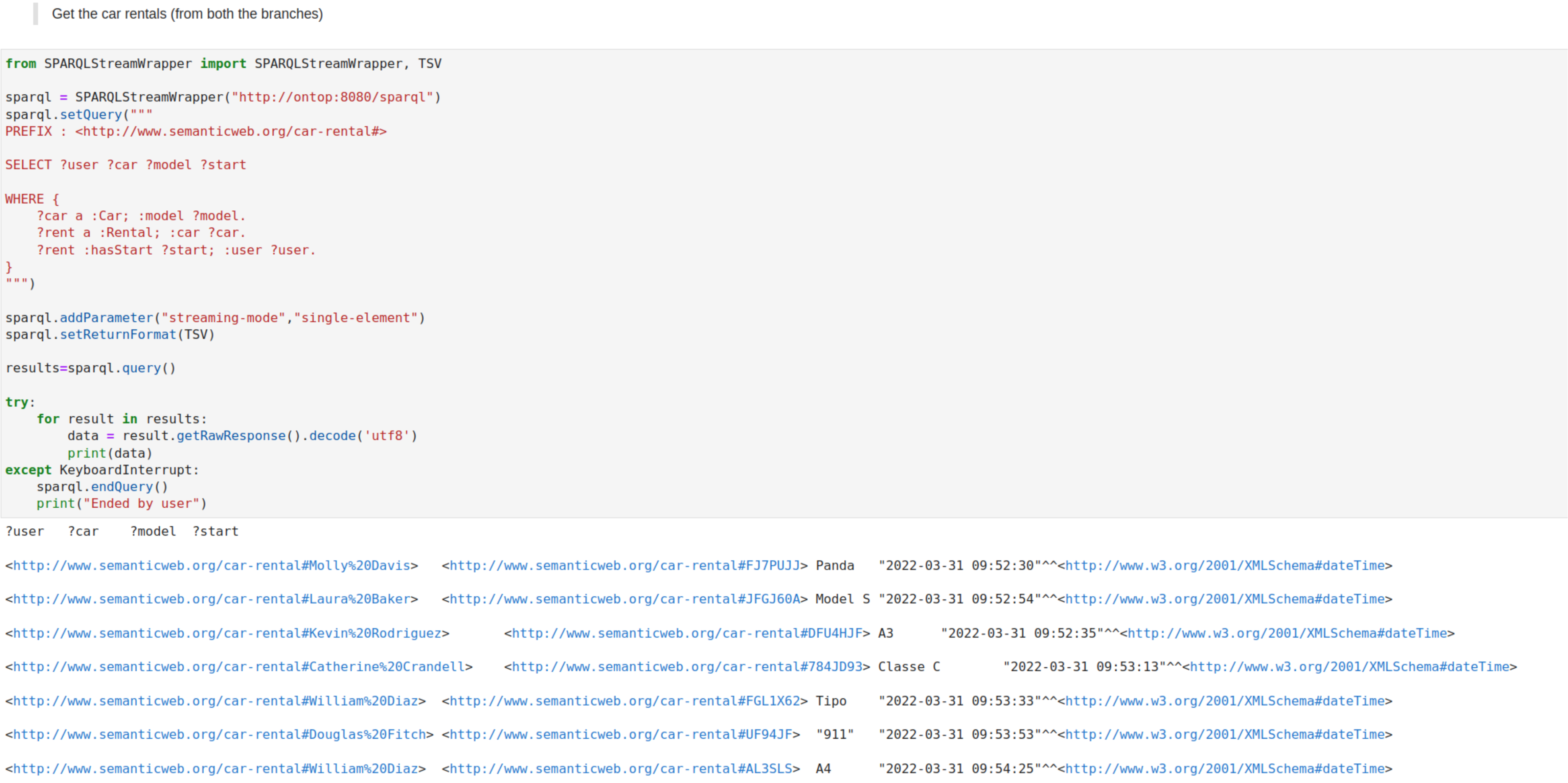


> Get the car rentals (from both the branches)

```python
from SPARQLStreamWrapper import SPARQLStreamWrapper, TSV

sparql = SPARQLStreamWrapper("http://ontop:8080/sparql")
sparql.setQuery("""
PREFIX : <http://www.semanticweb.org/car-rental#>

SELECT ?user ?car ?model ?start

WHERE {
    ?car a :Car; :model ?model.
    ?rent a :Rental; :car ?car.
    ?rent :hasStart ?start; :user ?user.
}
""")

sparql.addParameter("streaming-mode","single-element")
sparql.setReturnFormat(TSV)

results=sparql.query()

try:
    for result in results:
        data = result.getRawResponse().decode('utf8')
        print(data)
except KeyboardInterrupt:
    sparql.endQuery()
    print("Ended by user")
```

```
?user   ?car    ?model  ?start

<http://www.semanticweb.org/car-rental#Molly%20Davis>   <http://www.semanticweb.org/car-rental#FJ7PUJJ> Panda   "2022-03-31 09:52:30"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Laura%20Baker>   <http://www.semanticweb.org/car-rental#JFGJ60A> Model S "2022-03-31 09:52:54"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Kevin%20Rodriguez>       <http://www.semanticweb.org/car-rental#DFU4HJF> A3      "2022-03-31 09:52:35"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Catherine%20Crandell>    <http://www.semanticweb.org/car-rental#784JD93> Classe C        "2022-03-31 09:53:13"^^<http://www.w3.org/2001/XMLSchema#dateTime>
```

First Query

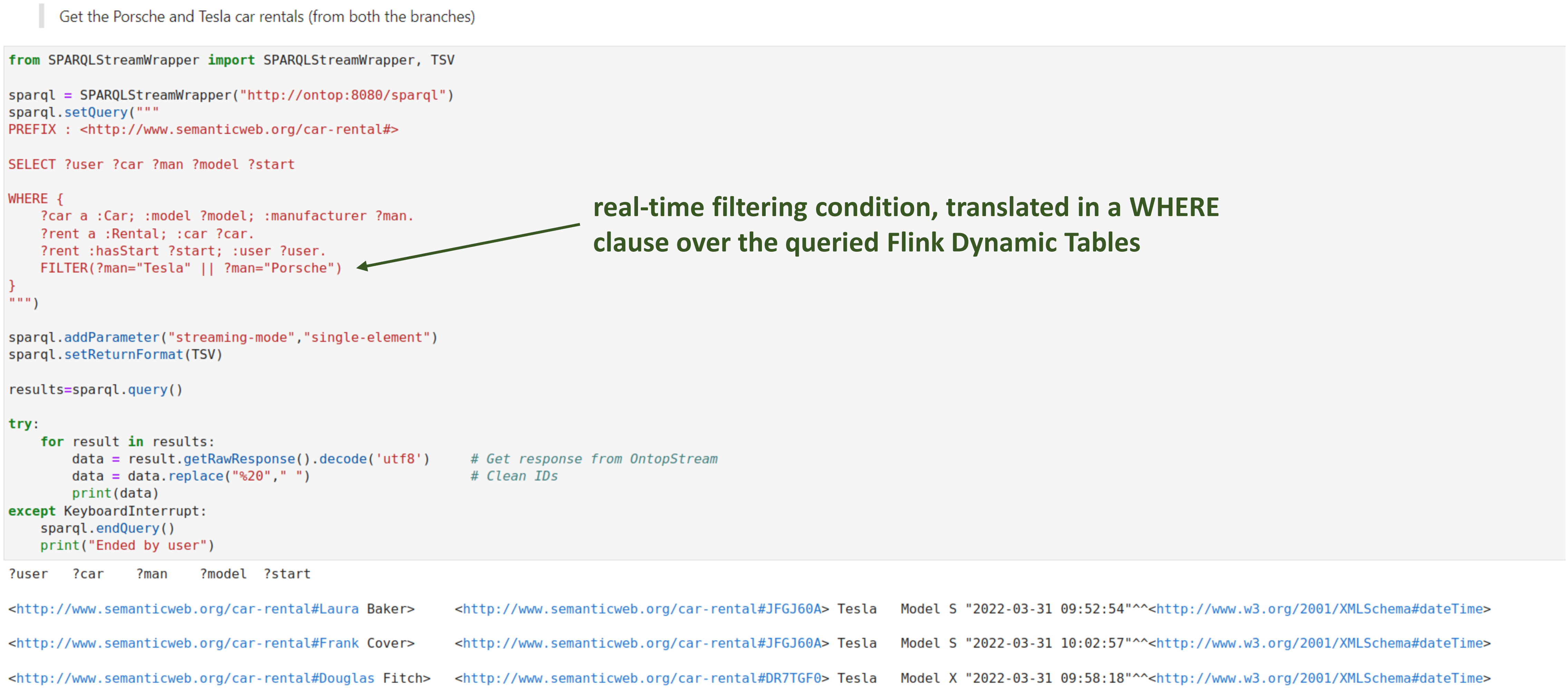


> Get the car rentals (from both the branches)

```
from SPARQLStreamWrapper import SPARQLStreamWrapper, TSV

sparql = SPARQLStreamWrapper("http://ontop:8080/sparql")
sparql.setQuery("""
PREFIX : <http://www.semanticweb.org/car-rental#>

SELECT ?user ?car ?model ?start

WHERE {
    ?car a :Car; :model ?model.
    ?rent a :Rental; :car ?car.
    ?rent :hasStart ?start; :user ?user.
}
""")

sparql.addParameter("streaming-mode","single-element")
sparql.setReturnFormat(TSV)

results=sparql.query()

try:
    for result in results:
        data = result.getRawResponse().decode('utf8')
        print(data)
except KeyboardInterrupt:
    sparql.endQuery()
    print("Ended by user")
```

```
?user   ?car    ?model  ?start

<http://www.semanticweb.org/car-rental#Molly%20Davis>   <http://www.semanticweb.org/car-rental#FJ7PUJJ> Panda   "2022-03-31 09:52:30"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Laura%20Baker>   <http://www.semanticweb.org/car-rental#JFGJ60A> Model S "2022-03-31 09:52:54"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Kevin%20Rodriguez>       <http://www.semanticweb.org/car-rental#DFU4HJF> A3      "2022-03-31 09:52:35"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Catherine%20Crandell>    <http://www.semanticweb.org/car-rental#784JD93> Classe C        "2022-03-31 09:53:13"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#William%20Diaz>  <http://www.semanticweb.org/car-rental#FGL1X62> Tipo    "2022-03-31 09:53:33"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Douglas%20Fitch> <http://www.semanticweb.org/car-rental#UF94JF>  "911"   "2022-03-31 09:53:53"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#William%20Diaz>  <http://www.semanticweb.org/car-rental#AL3SLS>  A4      "2022-03-31 09:54:25"^^<http://www.w3.org/2001/XMLSchema#dateTime>
```

Second Query – Real-Time Filtering



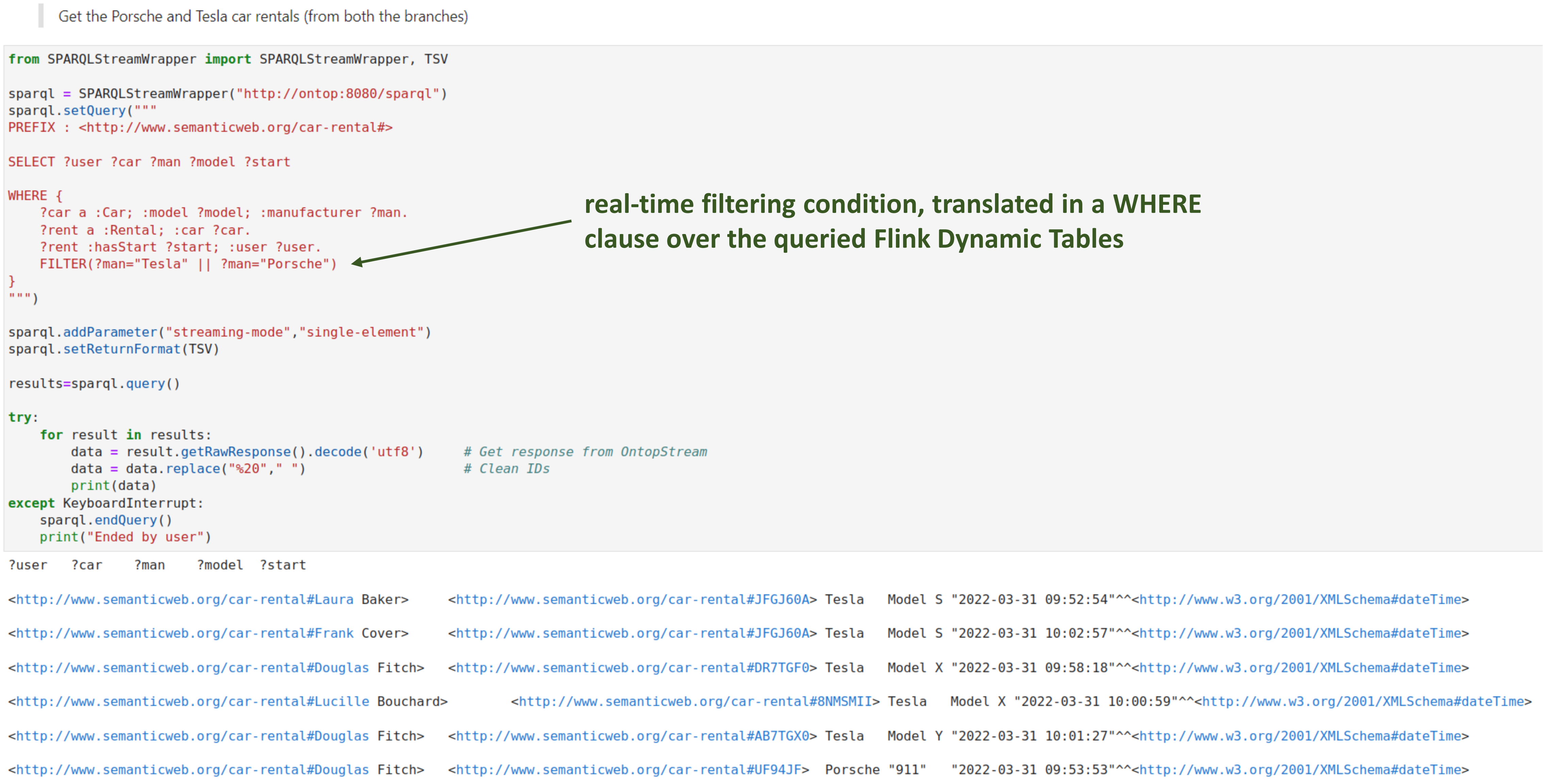
> Get the Porsche and Tesla car rentals (from both the branches)

```
from SPARQLStreamWrapper import SPARQLStreamWrapper, TSV

sparql = SPARQLStreamWrapper("http://ontop:8080/sparql")
sparql.setQuery("""
PREFIX : <http://www.semanticweb.org/car-rental#>

SELECT ?user ?car ?man ?model ?start

WHERE {
    ?car a :Car; :model ?model; :manufacturer ?man.
    ?rent a :Rental; :car ?car.
    ?rent :hasStart ?start; :user ?user.
    FILTER(?man="Tesla" || ?man="Porsche")
}
""")

sparql.addParameter("streaming-mode","single-element")
sparql.setReturnFormat(TSV)

results=sparql.query()

try:
    for result in results:
        data = result.getRawResponse().decode('utf8')     # Get response from OntopStream
        data = data.replace("%20"," ")                     # Clean IDs
        print(data)
except KeyboardInterrupt:
    sparql.endQuery()
    print("Ended by user")
```

real-time filtering condition, translated in a WHERE clause over the queried Flink Dynamic Tables

```
?user   ?car    ?man    ?model  ?start

<http://www.semanticweb.org/car-rental#Laura Baker>    <http://www.semanticweb.org/car-rental#JFGJ60A> Tesla   Model S "2022-03-31 09:52:54"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Frank Cover>    <http://www.semanticweb.org/car-rental#JFGJ60A> Tesla   Model S "2022-03-31 10:02:57"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Douglas Fitch>  <http://www.semanticweb.org/car-rental#DR7TGF0> Tesla   Model X "2022-03-31 09:58:18"^^<http://www.w3.org/2001/XMLSchema#dateTime>
```

Second Query – Real-Time Filtering



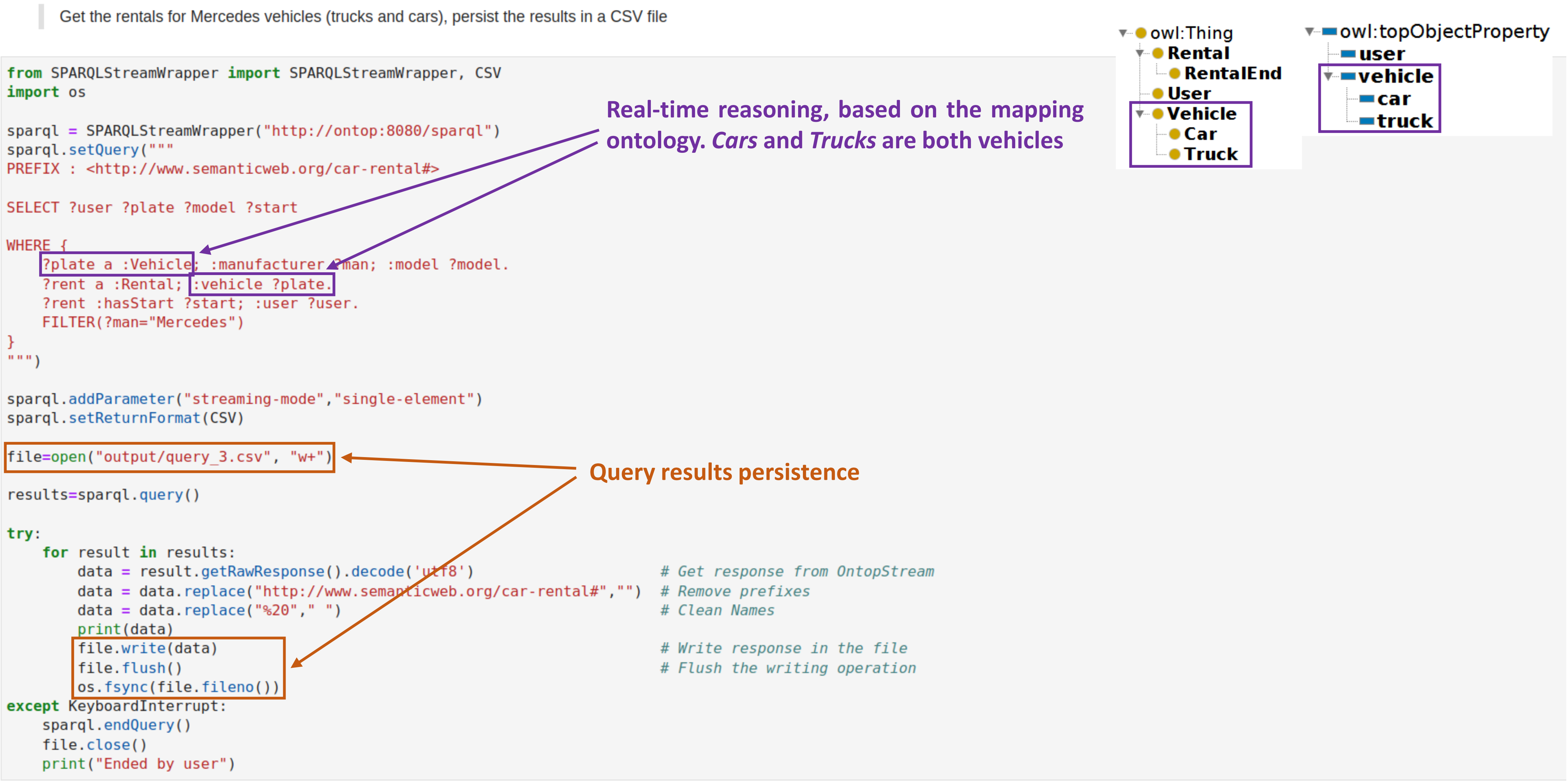
> Get the Porsche and Tesla car rentals (from both the branches)

```
from SPARQLStreamWrapper import SPARQLStreamWrapper, TSV

sparql = SPARQLStreamWrapper("http://ontop:8080/sparql")
sparql.setQuery("""
PREFIX : <http://www.semanticweb.org/car-rental#>

SELECT ?user ?car ?man ?model ?start

WHERE {
    ?car a :Car; :model ?model; :manufacturer ?man.
    ?rent a :Rental; :car ?car.
    ?rent :hasStart ?start; :user ?user.
    FILTER(?man="Tesla" || ?man="Porsche")
}
""")

sparql.addParameter("streaming-mode","single-element")
sparql.setReturnFormat(TSV)

results=sparql.query()

try:
    for result in results:
        data = result.getRawResponse().decode('utf8')     # Get response from OntopStream
        data = data.replace("%20"," ")                    # Clean IDs
        print(data)
except KeyboardInterrupt:
    sparql.endQuery()
    print("Ended by user")
```

```
?user   ?car    ?man    ?model  ?start

<http://www.semanticweb.org/car-rental#Laura Baker>    <http://www.semanticweb.org/car-rental#JFGJ60A> Tesla   Model S "2022-03-31 09:52:54"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Frank Cover>    <http://www.semanticweb.org/car-rental#JFGJ60A> Tesla   Model S "2022-03-31 10:02:57"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Douglas Fitch>  <http://www.semanticweb.org/car-rental#DR7TGF0> Tesla   Model X "2022-03-31 09:58:18"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Lucille Bouchard>       <http://www.semanticweb.org/car-rental#8NMSMII> Tesla   Model X "2022-03-31 10:00:59"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Douglas Fitch>  <http://www.semanticweb.org/car-rental#AB7TGX0> Tesla   Model Y "2022-03-31 10:01:27"^^<http://www.w3.org/2001/XMLSchema#dateTime>

<http://www.semanticweb.org/car-rental#Douglas Fitch>  <http://www.semanticweb.org/car-rental#UF94JF>  Porsche "911"   "2022-03-31 09:53:53"^^<http://www.w3.org/2001/XMLSchema#dateTime>
```

real-time filtering condition, translated in a WHERE clause over the queried Flink Dynamic Tables

Third Query – Reasoning...



> Get the rentals for Mercedes vehicles (trucks and cars), persist the results in a CSV file

```
from SPARQLStreamWrapper import SPARQLStreamWrapper, CSV
import os

sparql = SPARQLStreamWrapper("http://ontop:8080/sparql")
sparql.setQuery("""
PREFIX : <http://www.semanticweb.org/car-rental#>

SELECT ?user ?plate ?model ?start

WHERE {
    ?plate a :Vehicle; :manufacturer ?man; :model ?model.
    ?rent a :Rental; :vehicle ?plate.
    ?rent :hasStart ?start; :user ?user.
    FILTER(?man="Mercedes")
}
""")

sparql.addParameter("streaming-mode","single-element")
sparql.setReturnFormat(CSV)

file=open("output/query_3.csv", "w+")

results=sparql.query()

try:
    for result in results:
        data = result.getRawResponse().decode('utf8')                     # Get response from OntopStream
        data = data.replace("http://www.semanticweb.org/car-rental#","")  # Remove prefixes
        data = data.replace("%20"," ")                                     # Clean Names
        print(data)
        file.write(data)                                                   # Write response in the file
        file.flush()                                                       # Flush the writing operation
        os.fsync(file.fileno())
except KeyboardInterrupt:
    sparql.endQuery()
    file.close()
    print("Ended by user")
```

Real-time reasoning, based on the mapping ontology. *Cars* and *Trucks* are both vehicles

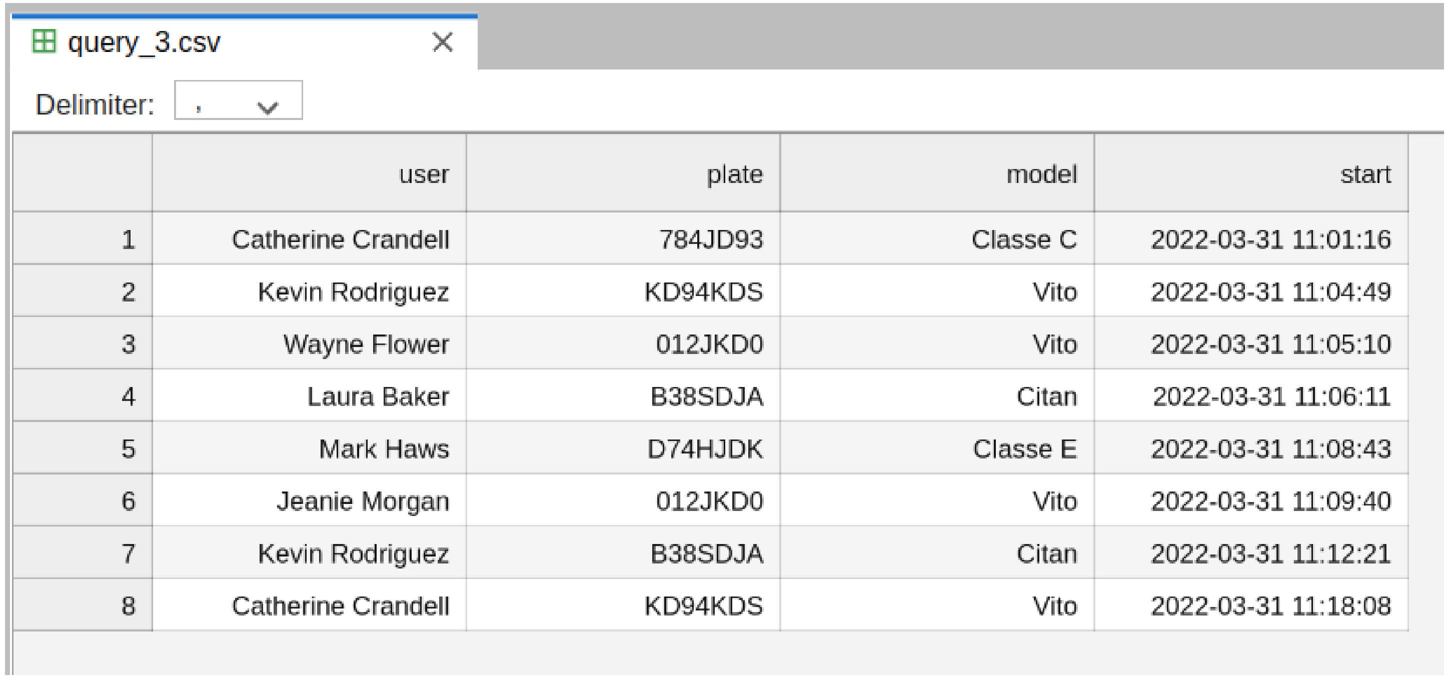
Query results persistence

Third Query – Reasoning…

Car (branch 1)

Truck (branch 2)

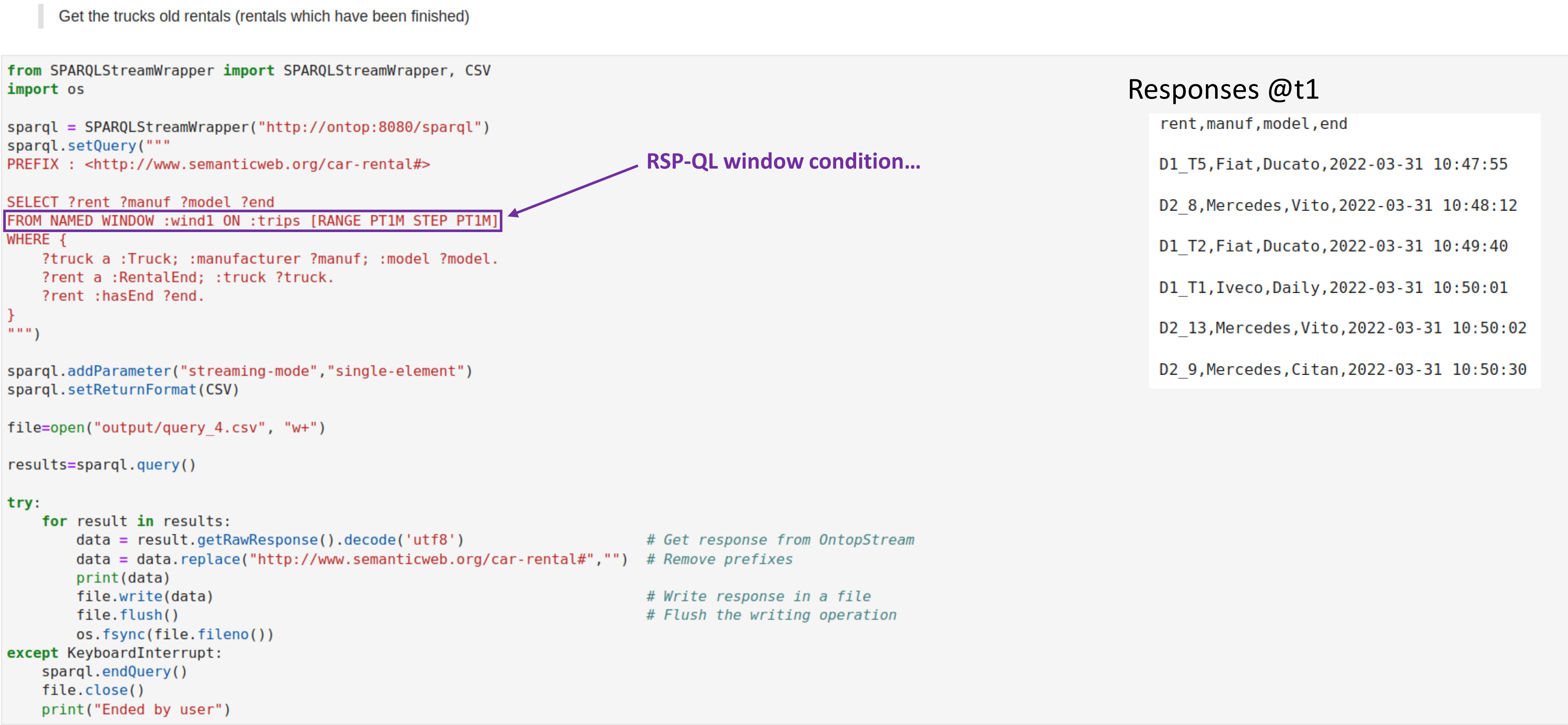
```
user,plate,model,start

Catherine Crandell,784JD93,Classe C,2022-03-31 11:01:16

Kevin Rodriguez,KD94KDS,Vito,2022-03-31 11:04:49

Wayne Flower,012JKD0,Vito,2022-03-31 11:05:10

Laura Baker,B38SDJA,Citan,2022-03-31 11:06:11

Mark Haws,D74HJDK,Classe E,2022-03-31 11:08:43

Jeanie Morgan,012JKD0,Vito,2022-03-31 11:09:40

Kevin Rodriguez,B38SDJA,Citan,2022-03-31 11:12:21

Catherine Crandell,KD94KDS,Vito,2022-03-31 11:18:08
```



Delimiter: ,

	user	plate	model	start
1	Catherine Crandell	784JD93	Classe C	2022-03-31 11:01:16
2	Kevin Rodriguez	KD94KDS	Vito	2022-03-31 11:04:49
3	Wayne Flower	012JKD0	Vito	2022-03-31 11:05:10
4	Laura Baker	B38SDJA	Citan	2022-03-31 11:06:11
5	Mark Haws	D74HJDK	Classe E	2022-03-31 11:08:43
6	Jeanie Morgan	012JKD0	Vito	2022-03-31 11:09:40
7	Kevin Rodriguez	B38SDJA	Citan	2022-03-31 11:12:21
8	Catherine Crandell	KD94KDS	Vito	2022-03-31 11:18:08

Fourth Query – Windowing (future developments)



> Get the trucks old rentals (rentals which have been finished)

```
from SPARQLStreamWrapper import SPARQLStreamWrapper, CSV
import os

sparql = SPARQLStreamWrapper("http://ontop:8080/sparql")
sparql.setQuery("""
PREFIX : <http://www.semanticweb.org/car-rental#>

SELECT ?rent ?manuf ?model ?end
FROM NAMED WINDOW :wind1 ON :trips [RANGE PT1M STEP PT1M]
WHERE {
    ?truck a :Truck; :manufacturer ?manuf; :model ?model.
    ?rent a :RentalEnd; :truck ?truck.
    ?rent :hasEnd ?end.
}
""")

sparql.addParameter("streaming-mode","single-element")
sparql.setReturnFormat(CSV)

file=open("output/query_4.csv", "w+")

results=sparql.query()

try:
    for result in results:
        data = result.getRawResponse().decode('utf8')           # Get response from OntopStream
        data = data.replace("http://www.semanticweb.org/car-rental#","")  # Remove prefixes
        print(data)
        file.write(data)                                          # Write response in a file
        file.flush()                                              # Flush the writing operation
        os.fsync(file.fileno())
except KeyboardInterrupt:
    sparql.endQuery()
    file.close()
    print("Ended by user")
```

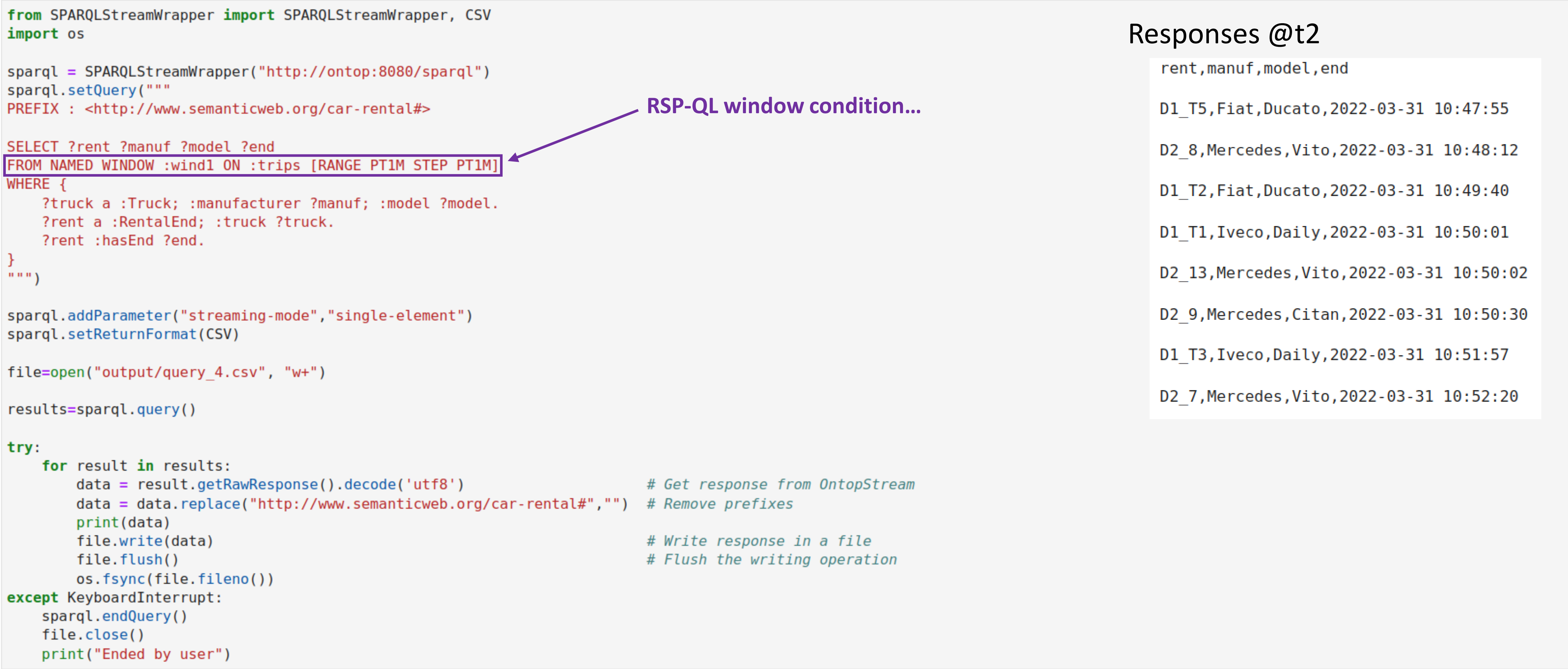
RSP-QL window condition…

Responses @t1

```
rent,manuf,model,end

D1_T5,Fiat,Ducato,2022-03-31 10:47:55

D2_8,Mercedes,Vito,2022-03-31 10:48:12

D1_T2,Fiat,Ducato,2022-03-31 10:49:40

D1_T1,Iveco,Daily,2022-03-31 10:50:01

D2_13,Mercedes,Vito,2022-03-31 10:50:02

D2_9,Mercedes,Citan,2022-03-31 10:50:30
```

Fourth Query – Windowing (future developments)

> Get the trucks old rentals (rentals which have been finished)



```python
from SPARQLStreamWrapper import SPARQLStreamWrapper, CSV
import os

sparql = SPARQLStreamWrapper("http://ontop:8080/sparql")
sparql.setQuery("""
PREFIX : <http://www.semanticweb.org/car-rental#>

SELECT ?rent ?manuf ?model ?end
FROM NAMED WINDOW :wind1 ON :trips [RANGE PT1M STEP PT1M]
WHERE {
    ?truck a :Truck; :manufacturer ?manuf; :model ?model.
    ?rent a :RentalEnd; :truck ?truck.
    ?rent :hasEnd ?end.
}
""")

sparql.addParameter("streaming-mode","single-element")
sparql.setReturnFormat(CSV)

file=open("output/query_4.csv", "w+")

results=sparql.query()

try:
    for result in results:
        data = result.getRawResponse().decode('utf8')          # Get response from OntopStream
        data = data.replace("http://www.semanticweb.org/car-rental#","")  # Remove prefixes
        print(data)
        file.write(data)                                       # Write response in a file
        file.flush()                                           # Flush the writing operation
        os.fsync(file.fileno())
except KeyboardInterrupt:
    sparql.endQuery()
    file.close()
    print("Ended by user")
```

RSP-QL window condition…

Responses @t2

```
rent,manuf,model,end

D1_T5,Fiat,Ducato,2022-03-31 10:47:55

D2_8,Mercedes,Vito,2022-03-31 10:48:12

D1_T2,Fiat,Ducato,2022-03-31 10:49:40

D1_T1,Iveco,Daily,2022-03-31 10:50:01

D2_13,Mercedes,Vito,2022-03-31 10:50:02

D2_9,Mercedes,Citan,2022-03-31 10:50:30

D1_T3,Iveco,Daily,2022-03-31 10:51:57

D2_7,Mercedes,Vito,2022-03-31 10:52:20
```

Thank you !!