Emergency (911) calls: views

We read in the first 10K entries from our dataset (size: ~660K). We have shown the functionality of our code using these 10K entries rather than 600K entries. All output will work with these 10K entries.

This cell contains two helper functions: one to delete a Kafka topic and one to test a streaming query and show the output.

In [134]:

```
from kafka.admin import KafkaAdminClient, NewTopic
from kafka.errors import UnknownTopicOrPartitionError

def delete_kafka_topic(name):
    admin_client = KafkaAdminClient(bootstrap_servers="localhost:9092")
    try:
        admin_client.delete_topics([name])
    except UnknownTopicOrPartitionError:
        pass


def test_query(sdf, mode="append", rows=None, wait=2, sort=None):
    try:
        tq = (
            # Create an output stream
            sdf.writeStream
            # Only write new rows to the output
            .outputMode(mode)
            # Write output stream to an in-memory Spark table (a DataFrame)
            .format("memory")
            # The name of the output table will be the same as the name of the query
            .queryName("test_query")
            # Submit the query to Spark and execute it
            .start()
        )

        tq.processAllAvailable()

        sleep(wait)
        while(tq.status.get("isTriggerActive") == True):
            print(f"DataAvailable: {tq.status['isDataAvailable']},\tTriggerActive: {tq.s
tatus['isTriggerActive']}\t{tq.status['message']}")
            sleep(wait)

        # When the status says "Waiting for data to arrive", that means the query
        # has finished its current iteration and is waiting for new messages from
        # Kafka.
        print(f"DataAvailable: {tq.status['isDataAvailable']},\tTriggerActive: {tq.statu
s['isTriggerActive']}\t{tq.status['message']}")

        memory_sink = spark.table("test_query")

        if sort:
            memory_sink = memory_sink.sort(*sort)

        # Show result table in Jupyter Notebook. Since Jupyter Notebooks have native supp
ort for showing pandas tables,
        # we convert the Spark DataFrame.
        if rows:
            display(memory_sink)
            display(memory_sink.take(10))
        else:
            display(memory_sink)
            display(memory_sink.toPandas())
```

```
    finally:
        # Always try to stop the query but it doesn't matter if it fails.
        try:
            tq.stop()
        except:
            pass
```

Create a Spark session.

In [135]:

```
from IPython.display import display, clear_output
from time import sleep

import os
os.environ['PYSPARK_SUBMIT_ARGS'] = '--packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.0.0 pyspark-shell'

import pyspark
from pyspark import SparkContext
from pyspark.sql.session import SparkSession
from pyspark.sql.types import *
from pyspark.sql.functions import *

# Create a local Spark cluster with two executors (if it doesn't already exist)
spark = SparkSession.builder.master('local[2]').getOrCreate()
sc = spark.sparkContext
```

The first step is to read the cleaned data from Kafka. This is very similar to the code reading the uncleaned data, with the single exception that `timestamp` is now part of the json struct.

We read the `ingest-cleaned` topic from kafka and decode the json in the following cell.

In [136]:

```
input = (
    spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", "localhost:9092")
    .option("subscribe", "ingest-cleaned")
    .option("startingOffsets", "earliest")
    .load()
)

# decode JSON
schema = StructType([
    StructField("lat", DoubleType()),
    StructField("lng", DoubleType()),
    StructField("desc", StringType()),
    StructField("zip", IntegerType()),
    StructField("title", StringType()),
    StructField("timeStamp", TimestampType()),
    StructField("twp", StringType()),
    StructField("addr", StringType()),
    StructField("majorTitle", StringType()),
    StructField("minorTitle", StringType()),
    StructField("hour", IntegerType()),
    StructField("date", DateType()),
])

decoded_json_stream = (
    input
    .withColumn("value", input["value"].cast("string"))
    .select("value")
    .withColumn("nineoneone", from_json(col("value"), schema))
    .select("nineoneone.*")
)

test_query(decoded_json_stream, mode="append")
```

```
DataAvailable: False, TriggerActive: False Waiting for data to arrive
```

```
DataFrame[lat: double, lng: double, desc: string, zip: int, title: string, timeStamp: tim
estamp, twp: string, addr: string, majorTitle: string, minorTitle: string, hour: int, dat
e: date]
```

	lat	lng	desc	zip	title	timeStamp	twp	addr	majo
0	40.297876	-75.581294	REINDEER CT & DEAD END; NEW HANOVER; Station ...	19525.0	EMS: BACK PAINS/INJURY	2015-12-10 17:10:52	NEW HANOVER	REINDEER CT & DEAD END	
1	40.258061	-75.264680	BRIAR PATH & WHITEMARSH LN; HATFIELD TOWNSHIP...	19446.0	EMS: DIABETIC EMERGENCY	2015-12-10 17:29:21	HATFIELD TOWNSHIP	BRIAR PATH & WHITEMARSH LN	
2	40.121182	-75.351975	HAWS AVE; NORRISTOWN; 2015-12-10 @ 14:39:21-St...	19401.0	Fire: GAS-ODOR/LEAK	2015-12-10 14:39:21	NORRISTOWN	HAWS AVE	
3	40.116153	-75.343513	AIRY ST & SWEDE ST; NORRISTOWN; Station 308A;...	19401.0	EMS: CARDIAC EMERGENCY	2015-12-10 16:47:36	NORRISTOWN	AIRY ST & SWEDE ST	
4	40.251492	-75.603350	CHERRYWOOD CT & DEAD END; LOWER POTTSGROVE; S...	NaN	EMS: DIZZINESS	2015-12-10 16:56:52	LOWER POTTSGROVE	CHERRYWOOD CT & DEAD END	
...	...	...	...	...	...	...	...	...	
9995	40.075536	-75.304635	3RD AVE & FAYETTE ST; CONSHOHOCKEN; 2016-01-06...	19428.0	Fire: FIRE ALARM	2016-01-06 17:22:10	CONSHOHOCKEN	3RD AVE & FAYETTE ST	
9996	40.211663	-75.275969	3RD ST & E MONTGOMERY AVE; NORTH WALES; Stati...	19454.0	EMS: SEIZURES	2016-01-06 17:23:43	NORTH WALES	3RD ST & E MONTGOMERY AVE	
9997	40.069013	-75.134458	OLD YORK RD & ACADEMY LN; CHELTENHAM; 2016-01-...	19027.0	Traffic: VEHICLE ACCIDENT -	2016-01-06 17:24:15	CHELTENHAM	OLD YORK RD & ACADEMY LN	T
9998	40.312619	-75.312583	SCHOOL LN & LINCOLN AVE; SOUDERTON; Station 3...	18964.0	EMS: DIABETIC EMERGENCY	2016-01-06 17:30:57	SOUDERTON	SCHOOL LN & LINCOLN AVE	
9999	40.371453	-75.484076	3RD ST & JEFFERSON ST; RED HILL; Station 369;...	18076.0	EMS: OVERDOSE	2016-01-06 17:27:35	RED HILL	3RD ST & JEFFERSON ST	

10000 rows × 12 columns

Oef 1

DataFrame: Show the number of calls of each type (= majorTitle) in a day. This view shows results before the day is over (using update mode), but uses as little RAM as possible. We group by date and majorTitle (i.e. the type of call) and we create a watermarked window over 1 day.

Watermarking helps a Stream Processing Engine to deal with lateness. It is a threshold to specify how long the system waits for late events. If an arriving event lies within our watermark, it gets used to update a query. Otherwise, if it's older than the watermark, it will be dropped and not further processed by the Streaming Engine.

In [137]:

```
# met watermark
```

```
window_1day_wm = (
    decoded_json_stream
    .withWatermark("timeStamp", "2 hour")
    .groupBy(
        'date', 'majorTitle',
        window(col("timeStamp"), "1 day")
    )
    .agg(
        count("majorTitle").alias("numMajorTypes")
    )
)
test_query(window_1day_wm, mode='update', sort=['window', 'majorTitle'])
```

```
DataAvailable: False, TriggerActive: True Waiting for data to arrive

DataFrame[date: date, majorTitle: string, window: struct<start:timestamp,end:timestamp>,
numMajorTypes: bigint]
```

	date	majorTitle	window	numMajorTypes
0	2015-12-10	EMS	(2015-12-10 00:00:00, 2015-12-11 00:00:00)	58
1	2015-12-10	Fire	(2015-12-10 00:00:00, 2015-12-11 00:00:00)	15
2	2015-12-10	Traffic	(2015-12-10 00:00:00, 2015-12-11 00:00:00)	41
3	2015-12-11	EMS	(2015-12-11 00:00:00, 2015-12-12 00:00:00)	186
4	2015-12-11	Fire	(2015-12-11 00:00:00, 2015-12-12 00:00:00)	68
...	...	...	...	...
79	2016-01-05	Fire	(2016-01-05 00:00:00, 2016-01-06 00:00:00)	77
80	2016-01-05	Traffic	(2016-01-05 00:00:00, 2016-01-06 00:00:00)	171
81	2016-01-06	EMS	(2016-01-06 00:00:00, 2016-01-07 00:00:00)	146
82	2016-01-06	Fire	(2016-01-06 00:00:00, 2016-01-07 00:00:00)	58
83	2016-01-06	Traffic	(2016-01-06 00:00:00, 2016-01-07 00:00:00)	116

84 rows × 4 columns

Oef 2

Kafka: For each hour of the day, record if there were more calls than on the previous day. Use a static DataFrame for the historical records.

We will do this by joining a Streaming DataFrame with a static/regular DataFrame. The historical data will come from the static DataFrame.

In [138]:

```
# met watermark
window_wm = (
    decoded_json_stream
    .withWatermark("timeStamp", "2 hour")
    .groupBy(
        'date', 'hour',
        window(col("timeStamp"), "1 hour")
    )
    .agg(
        count("title").alias("numCalls")
    )
)
test_query(window_wm, mode="update", sort=["date", "window"])
```

```
DataAvailable: False, TriggerActive: False Waiting for data to arrive

DataFrame[date: date, hour: int, window: struct<start:timestamp,end:timestamp>, numCalls:
bigint]
```

	date	hour	window	numCalls
0	2015-12-10	14	(2015-12-10 14:00:00, 2015-12-10 15:00:00)	1
1	2015-12-10	15	(2015-12-10 15:00:00, 2015-12-10 16:00:00)	1
2	2015-12-10	16	(2015-12-10 16:00:00, 2015-12-10 17:00:00)	6
3	2015-12-10	17	(2015-12-10 17:00:00, 2015-12-10 18:00:00)	16
4	2015-12-10	18	(2015-12-10 18:00:00, 2015-12-10 19:00:00)	26
...	...	...	...	...
633	2016-01-06	13	(2016-01-06 13:00:00, 2016-01-06 14:00:00)	27
634	2016-01-06	14	(2016-01-06 14:00:00, 2016-01-06 15:00:00)	26
635	2016-01-06	15	(2016-01-06 15:00:00, 2016-01-06 16:00:00)	29
636	2016-01-06	16	(2016-01-06 16:00:00, 2016-01-06 17:00:00)	22
637	2016-01-06	17	(2016-01-06 17:00:00, 2016-01-06 18:00:00)	20

638 rows × 4 columns

Create static historical dataframe:

In [139]:

```
nineoneonedf = spark.read.format("csv").option("header", "true").load("../911.csv").limi
t(10000)
nineoneonedf = nineoneonedf.withColumn('historicalHour', pyspark.sql.functions.hour(col(
"timeStamp")).cast(IntegerType()))
nineoneonedf = nineoneonedf.withColumn('historicalDate', pyspark.sql.functions.to_date(c
ol("timeStamp")))

static_prev = (
    nineoneonedf
    .groupBy('historicalDate', 'historicalHour')
    .agg(
        count("title").alias("historicalNumCalls")
    )
)

static_prev.toPandas().sort_values(by=['historicalDate', 'historicalHour'], ignore_index
=True)
```

Out[139]:

	historicalDate	historicalHour	historicalNumCalls
0	2015-12-10	14	1
1	2015-12-10	15	1
2	2015-12-10	16	6
3	2015-12-10	17	16
4	2015-12-10	18	26
...	...	...	...
633	2016-01-06	13	27
634	2016-01-06	14	26
635	2016-01-06	15	29
636	2016-01-06	16	22
637	2016-01-06	17	20

638 rows × 3 columns

The first day will not be included in our output, because there is no previous day to compare it with. One can include this day using a left join instead, but this will produce NaNs for the values of the historical columns.

Thus, we will use a simple inner join instead.

In [140]:

```
# Je kan streams met streams joinen
from datetime import datetime, timedelta

windows_and_historical = (
    window_wm
    .join(static_prev, (static_prev.historicalDate == (date_sub(window_wm.date,1))) & (s
tatic_prev.historicalHour == (window_wm.hour)) , "inner")
)

test_query(windows_and_historical, mode="append", sort=["date"])
```

```
DataAvailable: False, TriggerActive: False Waiting for data to arrive

DataFrame[date: date, hour: int, window: struct<start:timestamp,end:timestamp>, numCalls:
bigint, historicalDate: date, historicalHour: int, historicalNumCalls: bigint]
```

	date	hour	window	numCalls	historicalDate	historicalHour	historicalNumCalls
0	2015-12-11	17	(2015-12-11 17:00:00, 2015-12-11 18:00:00)	39	2015-12-10	17	16
1	2015-12-11	14	(2015-12-11 14:00:00, 2015-12-11 15:00:00)	23	2015-12-10	14	1
2	2015-12-11	16	(2015-12-11 16:00:00, 2015-12-11 17:00:00)	27	2015-12-10	16	6
3	2015-12-11	15	(2015-12-11 15:00:00, 2015-12-11 16:00:00)	28	2015-12-10	15	1
4	2015-12-11	18	(2015-12-11 18:00:00, 2015-12-11 19:00:00)	24	2015-12-10	18	26
...	...	...	...	...	...	...	...
598	2016-01-06	10	(2016-01-06 10:00:00, 2016-01-06 11:00:00)	22	2016-01-05	10	19
599	2016-01-06	11	(2016-01-06 11:00:00, 2016-01-06 12:00:00)	17	2016-01-05	11	20
600	2016-01-06	12	(2016-01-06 12:00:00, 2016-01-06 13:00:00)	30	2016-01-05	12	37
601	2016-01-06	13	(2016-01-06 13:00:00, 2016-01-06 14:00:00)	27	2016-01-05	13	37
602	2016-01-06	14	(2016-01-06 14:00:00, 2016-01-06 15:00:00)	26	2016-01-05	14	27

603 rows × 7 columns

We now add a boolean column, where the value for a row will be True if the day has more calls than the same hour of the previous day.

In [141]:

```
windows_and_historical = windows_and_historical.withColumn('hasMoreCalls', when(col('numc
alls') > col('historicalNumCalls'), True).otherwise(False))
```

In [142]:

```
# as an example to show that the above works:
test_query(windows_and_historical, mode="append", sort=["date"])
```

```
DataAvailable: False, TriggerActive: False Waiting for data to arrive

DataFrame[date: date, hour: int, window: struct<start:timestamp,end:timestamp>, numCalls:
bigint, historicalDate: date, historicalHour: int, historicalNumCalls: bigint, hasMoreCal
ls: boolean]
```

	date	hour	window	numCalls	historicalDate	historicalHour	historicalNumCalls	hasMoreCalls
0	2015-12-11	17	(2015-12-11 17:00:00, 2015-12-11 18:00:00)	39	2015-12-10	17	16	True
1	2015-12-11	14	(2015-12-11 14:00:00, 2015-12-11 15:00:00)	23	2015-12-10	14	1	True
2	2015-12-11	16	(2015-12-11 16:00:00, 2015-12-11 17:00:00)	27	2015-12-10	16	6	True

	date	hour	window	numCalls	historicalDate	historicalHour	historicalNumCalls	hasMoreCalls
3	2015-12-11	15	(2015-12-11 15:00:00, 2015-12-11 16:00:00)	28	2015-12-10	15	1	True
4	2015-12-11	18	(2015-12-11 18:00:00, 2015-12-11 19:00:00)	24	2015-12-10	18	26	False
...	...	...	...	...	...	...	...	...
598	2016-01-06	10	(2016-01-06 10:00:00, 2016-01-06 11:00:00)	22	2016-01-05	10	19	True
599	2016-01-06	11	(2016-01-06 11:00:00, 2016-01-06 12:00:00)	17	2016-01-05	11	20	False
600	2016-01-06	12	(2016-01-06 12:00:00, 2016-01-06 13:00:00)	30	2016-01-05	12	37	False
601	2016-01-06	13	(2016-01-06 13:00:00, 2016-01-06 14:00:00)	27	2016-01-05	13	37	False
602	2016-01-06	14	(2016-01-06 14:00:00, 2016-01-06 15:00:00)	26	2016-01-05	14	27	False

603 rows × 8 columns

Finally, we want to write our results to the `ingest-cleaned-oef2` Kafka topic. This Kafka output stream expects a dataframe, a value and an optional key column.

To create the `value` column, we first create a struct from all columns in the dataframe by using the `struct` function, serialize the result to json using `to_json`, and keep only the value column using `select` and `alias`.

In [147]:

```
output_stream = windows_and_historical.select(to_json(struct("*")).alias("value"))
```

In [148]:

```
import shutil
shutil.rmtree('checkpoints-cleanup')
```

In [149]:

```
try:
    # In case the previous query wasn't stopped
    tq.stop()
    # Remove old checkpoint dir, otherwise you'll get weird runtime faults
    os.rmdir("checkpoints-cleanup")
except:
    pass

# Prepare df for Kafka and write to kafka
tq = (
    output_stream
    .writeStream.format("kafka")
    .option("kafka.bootstrap.servers", "localhost:9092")
    .option("topic", "ingest-cleaned-oef2")
    .option("checkpointLocation", "checkpoints-cleanup")
    .start()
)

sleep(2)
display(tq.status)
```

```
{'message': 'Processing new data',
 'isDataAvailable': True,
 'isTriggerActive': True}
```

Oef 3

DataFrame: recreate the data for one of the graphs you created in the lab 1 project task

This graph would show how many EMS calls with minorTitle 'ASSAULT VICTIM' there were per week.

In [157]:

```
decoded_json_stream = decoded_json_stream.withColumn('year', year(to_timestamp(col('timeS
tamp'), 'dd/MM/yyyy')).cast(IntegerType()))

window_oef3_wm = (
    decoded_json_stream
    .withWatermark("timeStamp", "2 hour")
    .where(col('majorTitle') == 'EMS')
    .where(col('minorTitle') == ' ASSAULT VICTIM')
    .groupBy(
        'year',
        window(col("timeStamp"), "1 week")
    )
    .agg(
        count("minorTitle").alias("numAssaultCallsPerWeek")
    )
)

test_query(window_oef3_wm, mode="append", sort=["year", "window"])
```

```
DataAvailable: False, TriggerActive: False Waiting for data to arrive

DataFrame[year: int, window: struct<start:timestamp,end:timestamp>, numAssaultCallsPerWee
k: bigint]
```

	year	window	numAssaultCallsPerWeek
0	2015	(2015-12-10 00:00:00, 2015-12-17 00:00:00)	16
1	2015	(2015-12-17 00:00:00, 2015-12-24 00:00:00)	14
2	2015	(2015-12-24 00:00:00, 2015-12-31 00:00:00)	18

In [ ]: