❬ PROFINIT ❭

(Databricks) Spark streaming

Martin Oharek

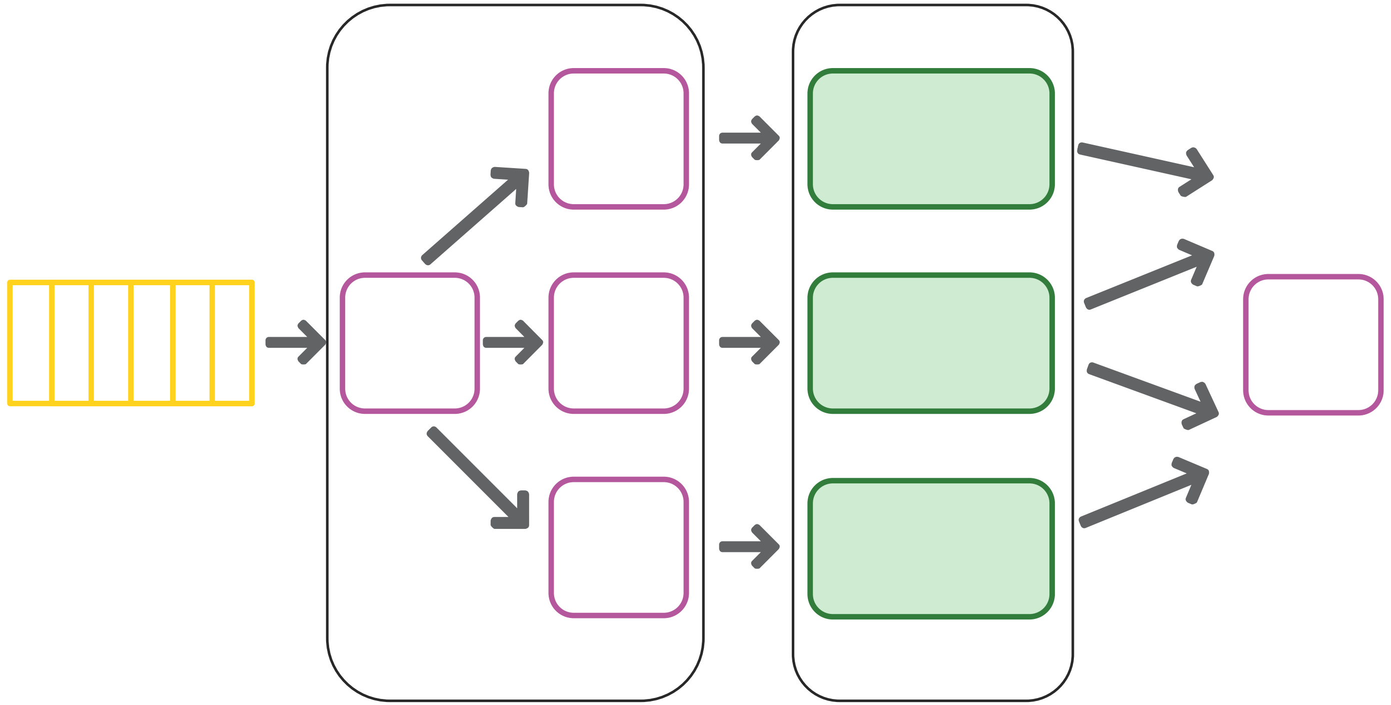
November 8th, 2023

Outline

Outline

1. Streaming intro
2. Basic concept and terminology
3. Sources and sinks for Structured Streaming
4. Operations over Streaming Dataframe
5. Micro-batch vs continuous mode
6. Extras

Streaming intro



Input data stream Message processor Stream processor Storage

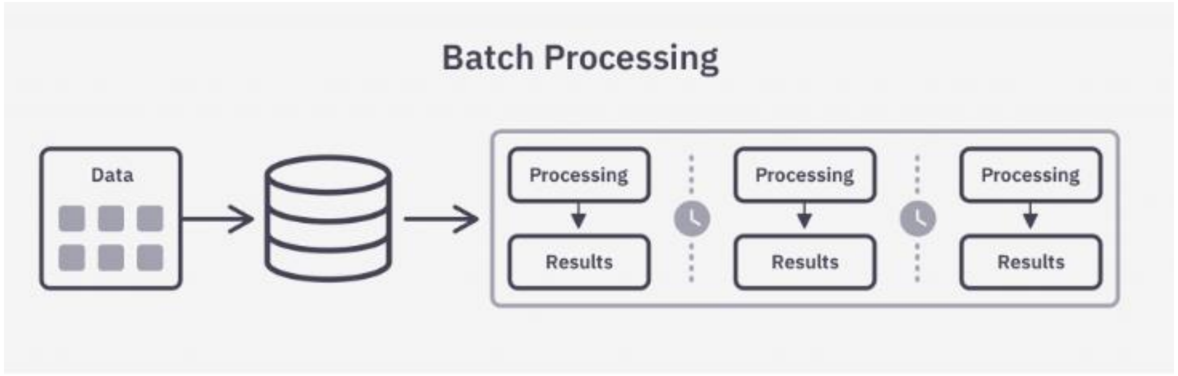
Streaming intro

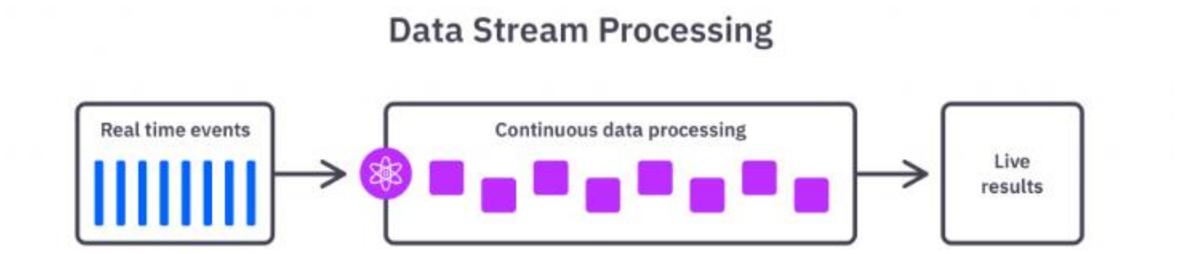
PROFINIT

- Real-time processing of continuous streams of data in motion
- Generally, we can think of three major components:
  - Message processors
    - Deliver data for processing
  - Stream processors
    - Processing layer – runs some computations/application logic on the data
  - Storage/Output
    - Store results, prepare stream for other consumers, send notifications, etc.

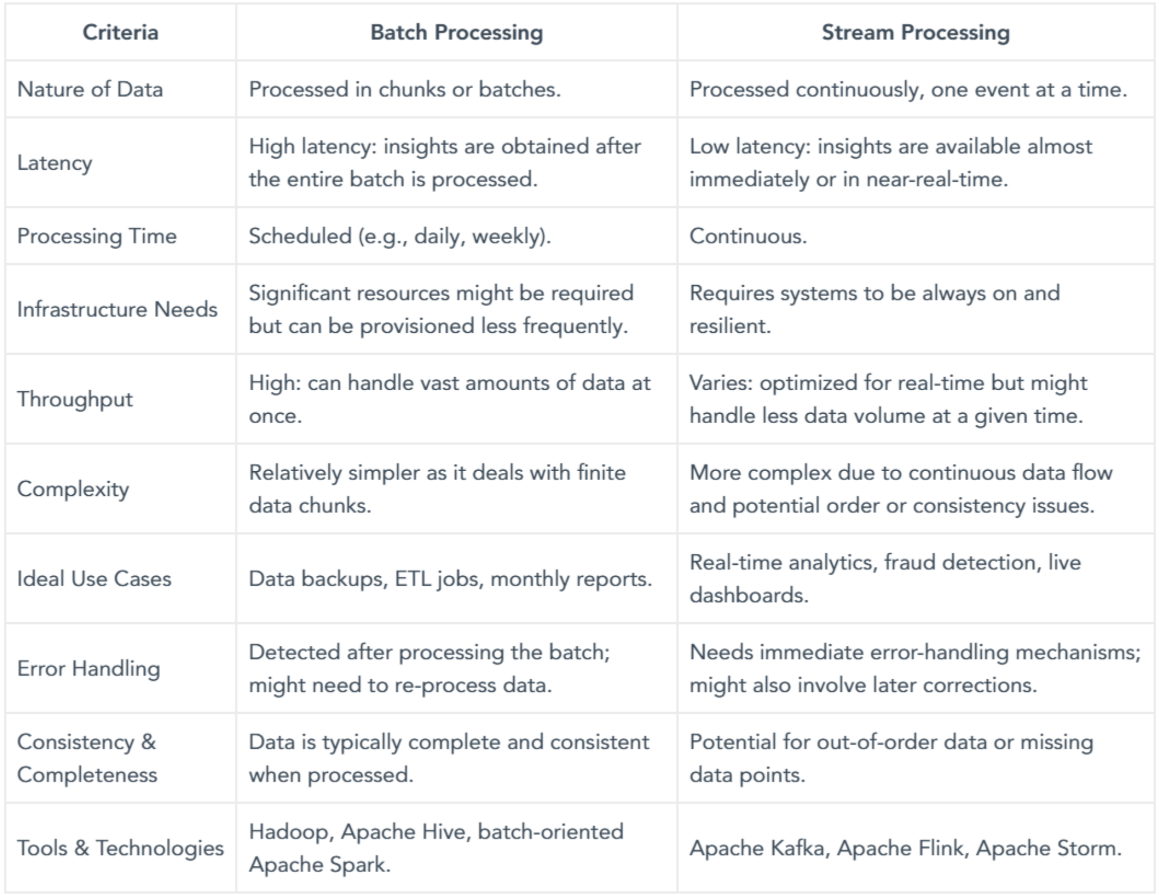
Streaming intro

PROFINIT





Batch processing vs Stream Processing



Criteria	Batch Processing	Stream Processing
Nature of Data	Processed in chunks or batches.	Processed continuously, one event at a time.
Latency	High latency: insights are obtained after the entire batch is processed.	Low latency: insights are available almost immediately or in near-real-time.
Processing Time	Scheduled (e.g., daily, weekly).	Continuous.
Infrastructure Needs	Significant resources might be required but can be provisioned less frequently.	Requires systems to be always on and resilient.
Throughput	High: can handle vast amounts of data at once.	Varies: optimized for real-time but might handle less data volume at a given time.
Complexity	Relatively simpler as it deals with finite data chunks.	More complex due to continuous data flow and potential order or consistency issues.
Ideal Use Cases	Data backups, ETL jobs, monthly reports.	Real-time analytics, fraud detection, live dashboards.
Error Handling	Detected after processing the batch; might need to re-process data.	Needs immediate error-handling mechanisms; might also involve later corrections.
Consistency & Completeness	Data is typically complete and consistent when processed.	Potential for out-of-order data or missing data points.
Tools & Technologies	Hadoop, Apache Hive, batch-oriented Apache Spark.	Apache Kafka, Apache Flink, Apache Storm.

Streaming in Spark

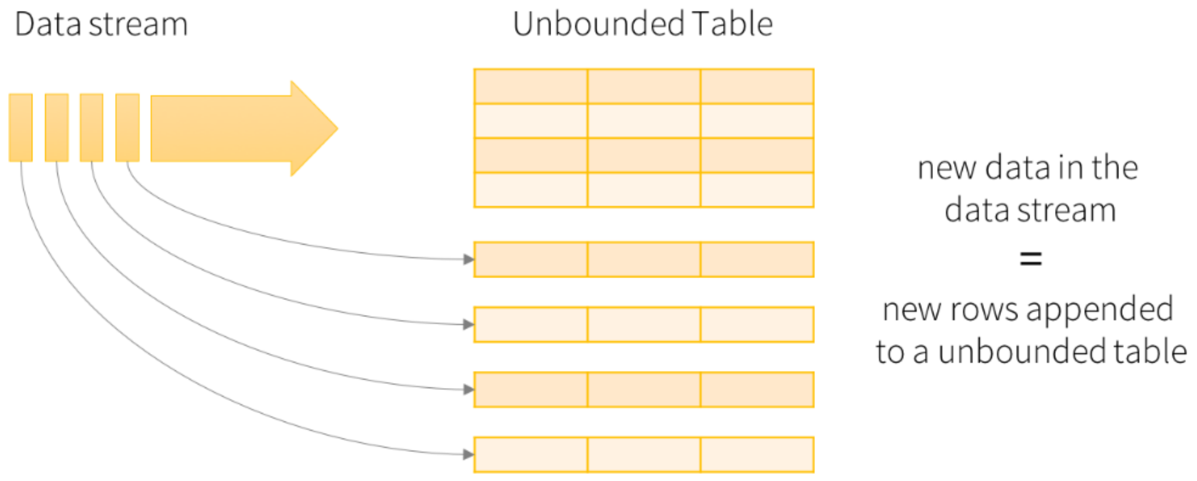
PROFINIT

- Spark streaming
  - https://spark.apache.org/docs/latest/streaming-programming-guide.html#overview
  - Legacy project, no longer updated
  - DStreams – low level RDD streaming API
- **Spark Structured Streaming**
  - https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#overview
  - Streaming API built on the Spark SQL engine, optimizations of execution plans available
  - Unified API for batch/streaming – code can be reused
  - You can use DataSet/DataFrame API in Java, Scala, Python and R
  - Internally, queries are processed in **micro-batches**
  - Since Spark 2.3 – introduced **continous processing**

Basic concept and terminology

Basic concept

- Treat a live data stream as a table that is being continuously appended


Data stream
Unbounded Table
new data in the data stream
=
new rows appended to a unbounded table

Data stream as an unbounded table

Example

PROFINIT

- Create stream and add logic

```
# Create DataFrame representing the stream of input lines from connection to localhost:9999
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Split the lines into words
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)

# Generate running word count
wordCounts = words.groupBy("word").count()
```

Example

PROFINIT

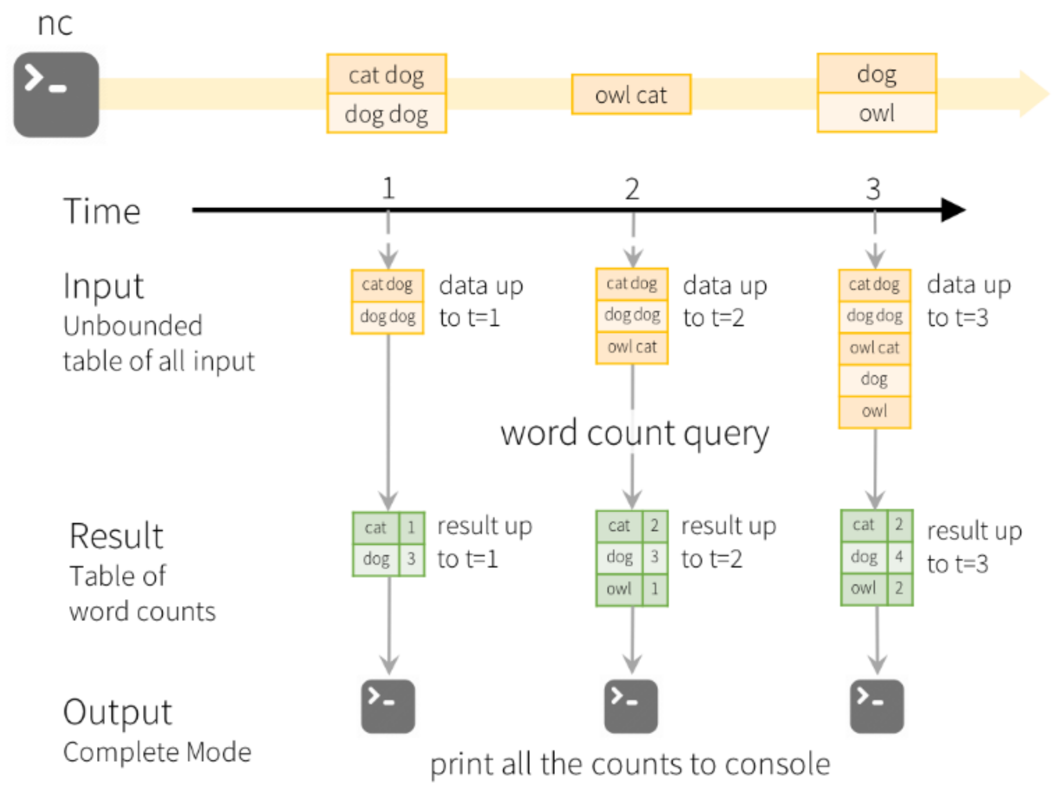
- Start receiving data

```
 # Start running the query that prints the running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

Basic concept

PROFINIT



Output modes

PROFINIT

- **Append**
  - Only the new rows appended in the Result Table since the last trigger will be written to the external storage. This is applicable only on the queries where existing rows in the Result Table are not expected to change.
- **Complete**
  - The entire updated Result Table will be written to the external storage. It is up to the storage connector to decide how to handle writing of the entire table.
- **Update**
  - Only the rows that were updated in the Result Table since the last trigger will be written to the external storage (available since Spark 2.1.1).

Basic concept

- **Structured streaming does not materialize an entire table**
  - Process latest data incrementally, update the result and discard
  - Keeps only minimal intermediate state that is required to update the result
- Faul-tolerant
  - Checkpoints
    - Metadata/data checkpoints saved to durable/fault-tolerant storage (S3, HDFS,..)
  - Write-ahead logs
    - Capture ingested data, but not yet processed by query

Basic concept

- Handling recovery after system failure in streaming systems:
  - Fault-tolerance semantics:
    - **At least once**: Each message is guaranteed to be processed, but it may get processed more than once
    - **At most once**: Each message may or may not be processed. If a message is processed, it's only processed once
    - **Exactly once**: Each message is guaranteed to be processed once and only once

How to achieve exactly-once delivery (simplified)?

- Streaming source
  - In case of failure, data should be **replayable** in the source system
- Checkpointing and write ahead logs
  - Store current state to durable, fault-tolerant storage to recover in case of driver/executor failures
- Idempotent processing and sinks
  - Ensure that data is not duplicated when reprocessed/retried after failure
  - Ensure correct final state of the system after data is reprocessed

Sources and Sinks for Structured Streaming

Data sources for Structured Streaming

PROFINIT

File source

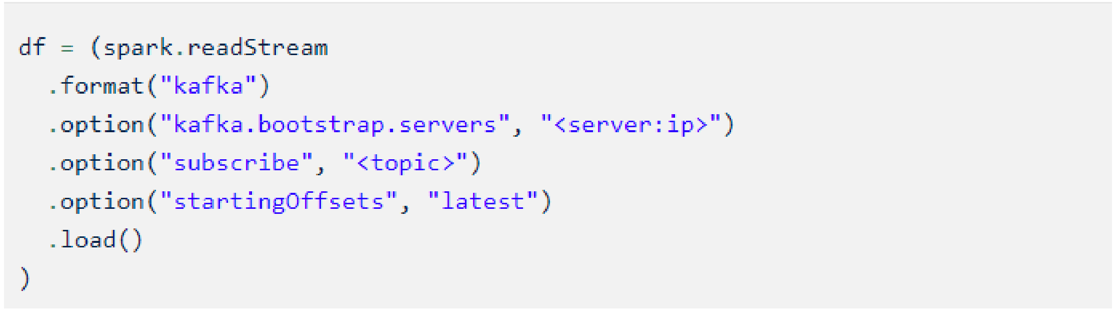
- Reads file written in a directory as a stream of data
- CSV, JSON, ORC, Parquet, S3,…

```
userSchema = StructType().add("name", "string").add("age", "integer")
csvDF = spark \
    .readStream \
    .option("sep", ";") \
    .schema(userSchema) \
    .csv("/path/to/directory")  # Equivalent to format("csv").load("/path/to/directory")
```

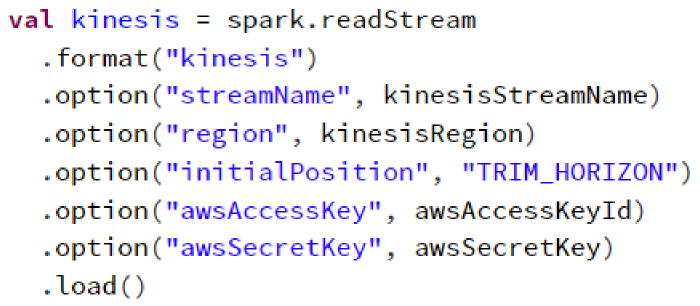
Data sources for Structured Streaming

Messaging services

- Kafka, Kinesis, Event Hubs,…



```
df = (spark.readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "<server:ip>")
  .option("subscribe", "<topic>")
  .option("startingOffsets", "latest")
  .load()
)
```



```
val kinesis = spark.readStream
  .format("kinesis")
  .option("streamName", kinesisStreamName)
  .option("region", kinesisRegion)
  .option("initialPosition", "TRIM_HORIZON")
  .option("awsAccessKey", awsAccessKeyId)
  .option("awsSecretKey", awsSecretKey)
  .load()
```

Data sources for Structured Streaming

PROFINIT

Delta table

- Incremental read of delta tables
- You can use **change data feed (CDF)** of Delta Lake table to upsert changes in downstream tables
- Configure input rate
  - **maxFilesPerTrigger**: How many new files to be considered in every micro-batch (default 1000).
  - **maxBytesPerTrigger**: How much data gets processed in each micro-batch. This is not set by default.

```
spark.readStream.format("delta")
  .option("startingVersion", "5")
  .load("/tmp/delta/user_events")
```

```
spark.readStream.format("delta") \
  .option("readChangeFeed", "true") \
  .table("myDeltaTable")
```

Data sources for Structured Streaming

PROFINIT

- **Ingesting supported files from cloud object storage**
  - Classic file source or **Databricks Auto Loader (**cloudFiles format)
  - Auto Loader scales to support (near) real-time ingestion of millions of files per hour
  - S3, Azure Blob Storage, Google Cloud Storage,…

```
(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "json")
  .option("cloudFiles.schemaLocation", checkpoint_path)
  .load(file_path)
```

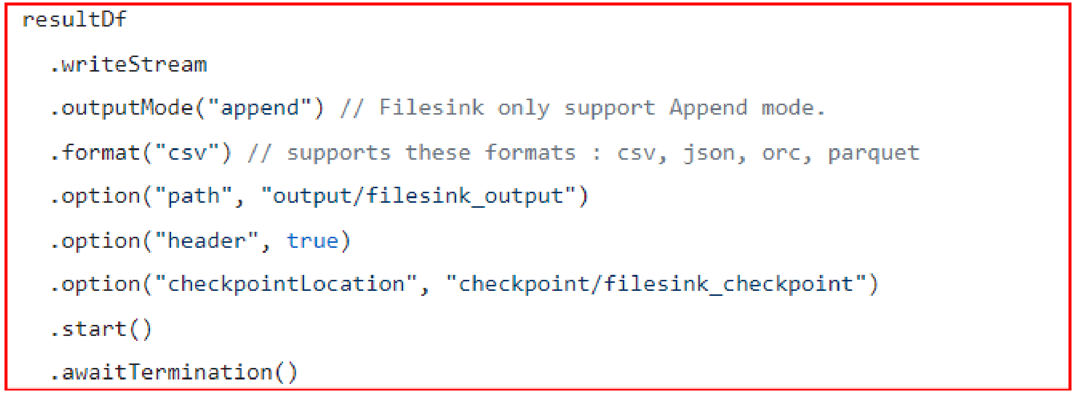
  - cloudFiles.schemaLocation option: supports schema inference and evolution

Data sources for Structured Streaming

- Benefits of Auto Loader over „classic“ file source:
  - Efficient and more performant file discovery
  - Schema inference and evolution
  - Cheap file discovery
- File detection modes
  - Directory listing
    - Used by default
    - Reduced number of API calls by listing files in subdirectories
  - File notification service
    - Leverages file notifications and queue service in cloud infrastructure account
    - Better for large input directories / high volume of files, more difficult to set up

Output Sinks for Structured Streaming

- File sink



```
resultDf
   .writeStream
   .outputMode("append") // Filesink only support Append mode.
   .format("csv") // supports these formats : csv, json, orc, parquet
   .option("path", "output/filesink_output")
   .option("header", true)
   .option("checkpointLocation", "checkpoint/filesink_checkpoint")
   .start()
   .awaitTermination()
```

Output Sinks for Structured Streaming

- Kafka sink

```
(spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "<server:ip>")
    .option("subscribe", "<topic>")
    .option("startingOffsets", "latest")
    .load()
    .join(spark.read.table("<table-name>"), on="<id>", how="left")
    .writeStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "<server:ip>")
    .option("topic", "<topic>")
    .option("checkpointLocation", "<checkpoint-path>")
    .start()
)
```

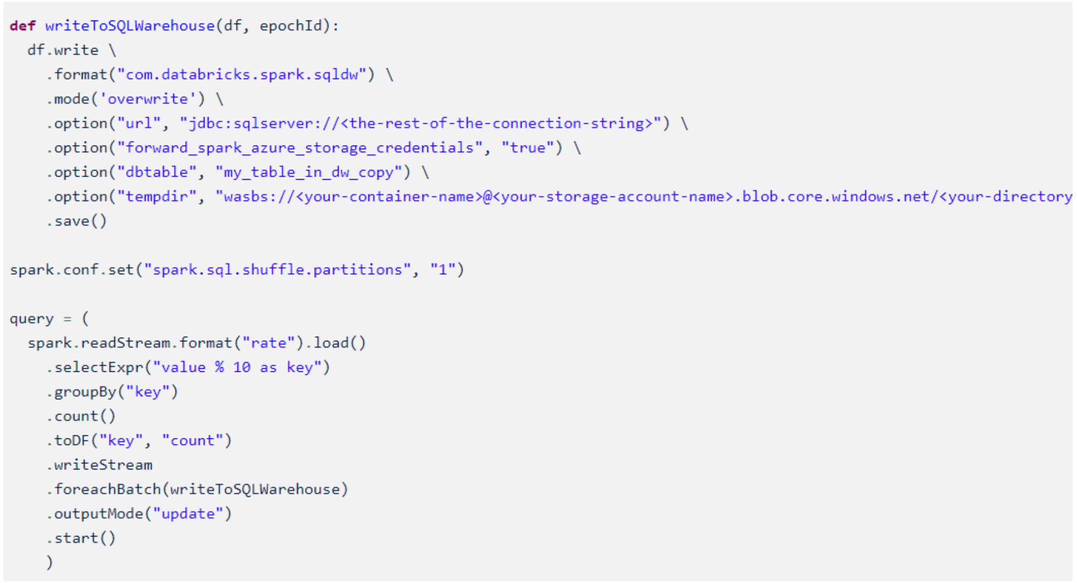
Output Sinks for Structured Streaming

- Delta Lake table sink
  - Good practice to ingest streaming data from external sources to Delta Lake
    - Secured „exactly-once“ processing, enabled by transaction log

```
(spark.readStream
    .table("<table-name1>")
    .join(spark.read.table("<table-name2>"), on="<id>", how="left")
    .writeStream
    .trigger(availableNow=True)
    .option("checkpointLocation", "<checkpoint-path>")
    .toTable("<table-name3>")
)
```

Output Sinks for Structured Streaming

- forEachBatch
  - Data are processed with custom logic per **micro-batch** -> can use (theoretically) **arbitrary sink**
  - Provides only at-least-once guarantees
    - You can use batchId to deduplicate data and secure exactly-once, but requires additional effort
  - Does not work with continuous-processing mode



```python
def writeToSQLWarehouse(df, epochId):
  df.write \
    .format("com.databricks.spark.sqldw") \
    .mode('overwrite') \
    .option("url", "jdbc:sqlserver://<the-rest-of-the-connection-string>") \
    .option("forward_spark_azure_storage_credentials", "true") \
    .option("dbtable", "my_table_in_dw_copy") \
    .option("tempdir", "wasbs://<your-container-name>@<your-storage-account-name>.blob.core.windows.net/<your-directory
    .save()

spark.conf.set("spark.sql.shuffle.partitions", "1")

query = (
  spark.readStream.format("rate").load()
    .selectExpr("value % 10 as key")
    .groupBy("key")
    .count()
    .toDF("key", "count")
    .writeStream
    .foreachBatch(writeToSQLWarehouse)
    .outputMode("update")
    .start()
    )
```

Operations over Streaming Dataframe

Operations over streaming dataframe

- Most operations work just the same as in the case of „classic“ dataframe
  - Select, where, groupBy,…
  - Temporary tables + SQL
- Joins
  - Stream dataframe – Static dataframe
  - Stream dataframe – Stream dataframe

Operations over streaming dataframe

- We generally distinguish between two types of operations:
  - **Stateless**
    - E.g. Filter - needs only information available in current micro-batch
  - **Stateful**
    - Such as count of keys over 5 minute period (aggregation), drop duplicates, etc. - need to preserve state and get information about previous data/results

Operations over streaming dataframe

PROFINIT

- Without restriction, state can become unbounded - will quickly introduce latency or even errors
- We should setup some threshold for how long to continue processing updates for a given state – **watermark**

Operations over streaming dataframe

- **Append** with watermark
  - Rows are written to the target table once the watermark threshold has passed. Old state is dropped once the threshold has passed.
- **Update** with watermark
  - Rows are written to the target table as results are calculated, and can be updated and overwritten as new data arrives. Old state is dropped once the threshold has passed.
- Complete
  - Aggregation state is **not** dropped. The target table is rewritten with each trigger.

Operations over streaming dataframe - windows

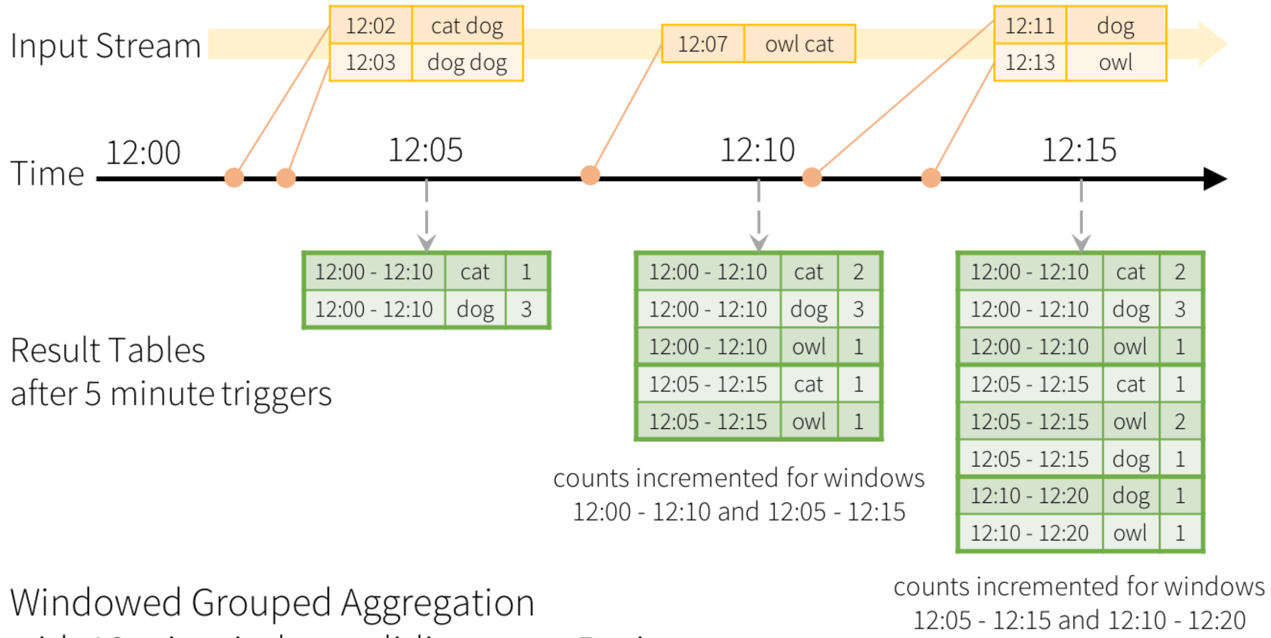
PROFINIT

- Window operations over event-time

-
  ```
  windowedCounts = words
      .groupBy(
          window(words.timestamp, "10 minutes", "5 minutes"),
          words.word
      )
      .count()
  ```
- 10 minute window, slide every 5 minutes

Operations over streaming dataframe - windows

PROFINIT



12:02	cat dog	
12:03	dog dog	
12:07	owl cat	
12:11	dog	
12:13	owl	

12:00 - 12:10	cat	1
12:00 - 12:10	dog	3

12:00 - 12:10	cat	2
12:00 - 12:10	dog	3
12:00 - 12:10	owl	1
12:05 - 12:15	cat	1
12:05 - 12:15	owl	1

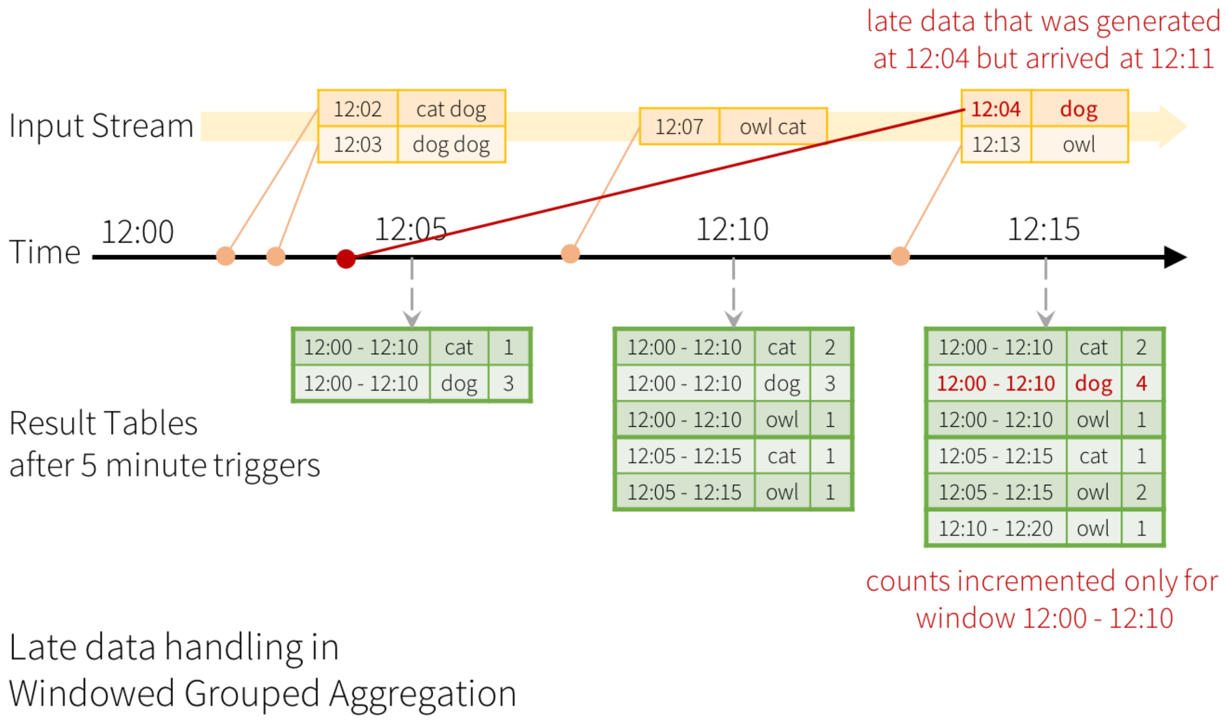
counts incremented for windows 12:00 - 12:10 and 12:05 - 12:15

12:00 - 12:10	cat	2
12:00 - 12:10	dog	3
12:00 - 12:10	owl	1
12:05 - 12:15	cat	1
12:05 - 12:15	owl	2
12:05 - 12:15	dog	1
12:10 - 12:20	dog	1
12:10 - 12:20	owl	1

counts incremented for windows 12:05 - 12:15 and 12:10 - 12:20

Windowed Grouped Aggregation
with 10 min windows, sliding every 5 mins

Operations over streaming dataframe - windows


late data that was generated at 12:04 but arrived at 12:11
Input Stream
12:02 | cat dog
12:03 | dog dog
12:07 | owl cat
12:04 | dog
12:13 | owl
Time
12:00
12:05
12:10
12:15
Result Tables after 5 minute triggers
counts incremented only for window 12:00 - 12:10
Late data handling in Windowed Grouped Aggregation

12:00 - 12:10	cat	1
12:00 - 12:10	dog	3

12:00 - 12:10	cat	2
12:00 - 12:10	dog	3
12:00 - 12:10	owl	1
12:05 - 12:15	cat	1
12:05 - 12:15	owl	1

12:00 - 12:10	cat	2
12:00 - 12:10	dog	4
12:00 - 12:10	owl	1
12:05 - 12:15	cat	1
12:05 - 12:15	owl	2
12:10 - 12:20	owl	1

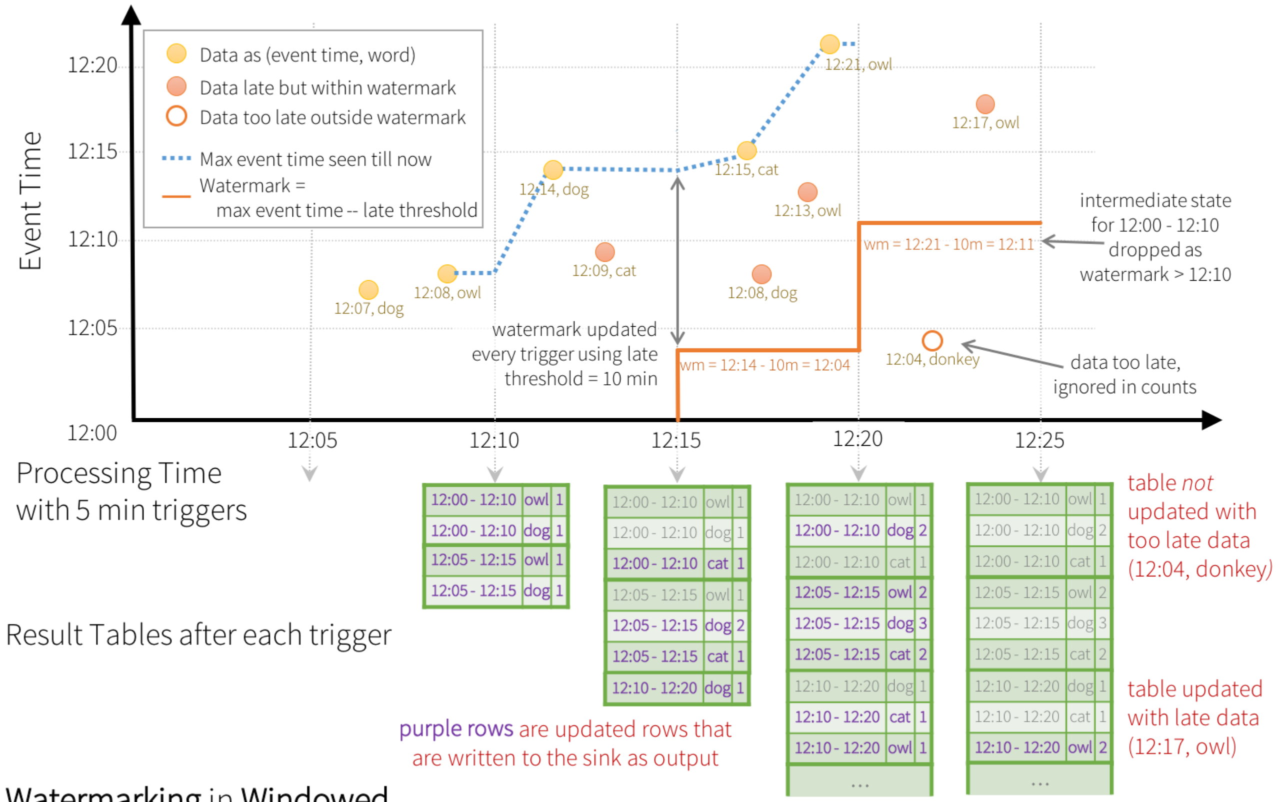
Operations over streaming dataframe - windows

PROFINIT

- Introduce watermark to handle late data
  - Old data arriving after watermark threshold has passed is not taken into account
- Conditions to use watermark:
  - Update or append mode
  - Aggregation should use event_time based column or window function over event_time column
  - Watermark should be specified over same column as given aggregation
  - **withWatermark** clause must precede given aggregation

-
  ```
  windowedCounts = words
      .withWatermark("timestamp", "10 minutes")
      .groupBy(
          window(words.timestamp, "10 minutes", "5 minutes"),
          words.word
      )
      .count()
  ```

Operations over streaming dataframe - windows



Event Time
12:20
12:15
12:10
12:05
12:00
Data as (event time, word)
Data late but within watermark
Data too late outside watermark
Max event time seen till now
Watermark = max event time -- late threshold
12:07, dog
12:08, owl
12:14, dog
12:09, cat
12:15, cat
12:08, dog
12:13, owl
12:21, owl
12:04, donkey
12:17, owl
watermark updated every trigger using late threshold = 10 min
wm = 12:14 - 10m = 12:04
wm = 12:21 - 10m = 12:11
intermediate state for 12:00 - 12:10 dropped as watermark > 12:10
data too late, ignored in counts
12:05
12:10
12:15
12:20
12:25
Processing Time with 5 min triggers
Result Tables after each trigger
purple rows are updated rows that are written to the sink as output
table *not* updated with too late data (12:04, donkey)
table updated with late data (12:17, owl)

12:00 - 12:10	owl	1
12:00 - 12:10	dog	1
12:05 - 12:15	owl	1
12:05 - 12:15	dog	1

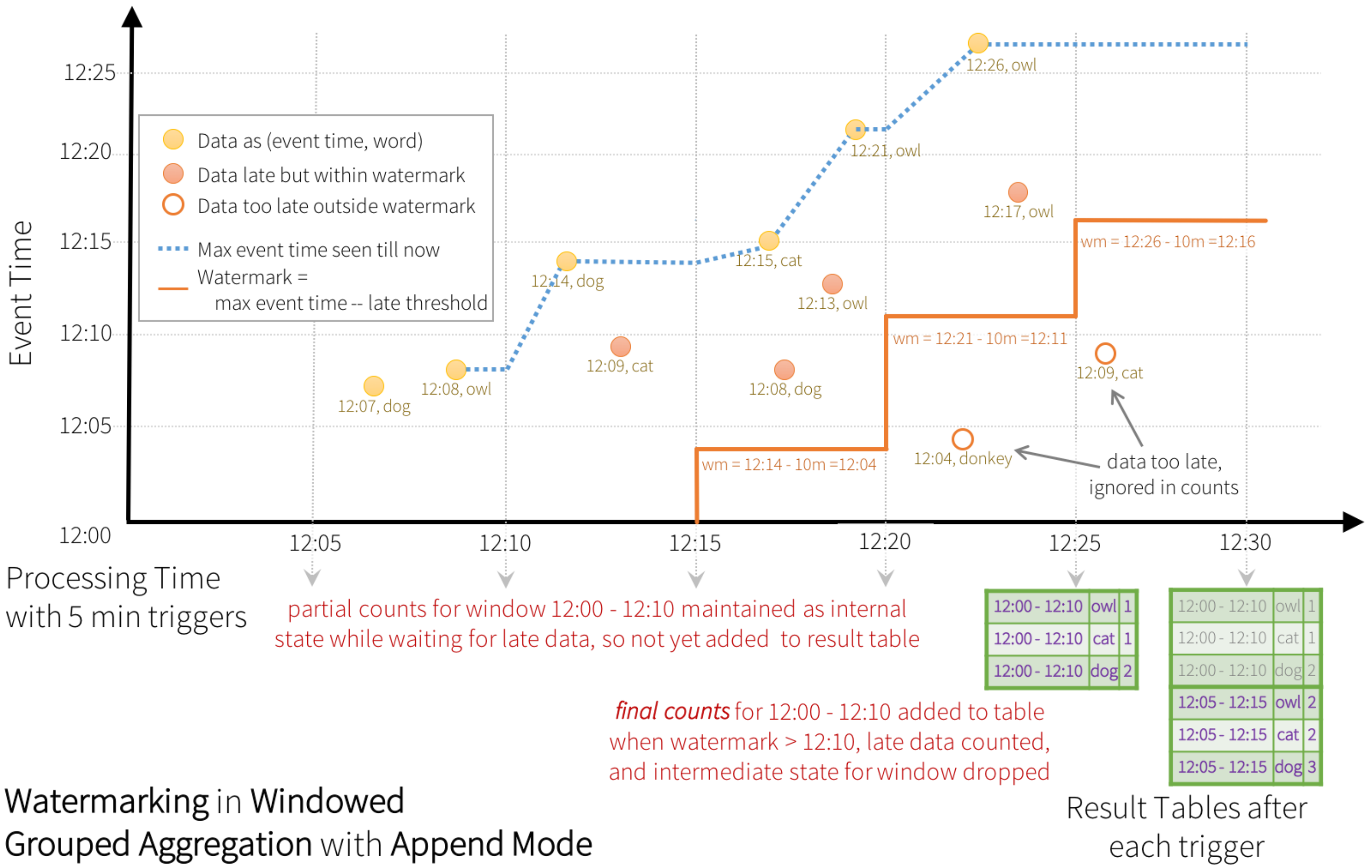
12:00 - 12:10	owl	1
12:00 - 12:10	dog	1
12:00 - 12:10	cat	1
12:05 - 12:15	owl	1
12:05 - 12:15	dog	2
12:05 - 12:15	cat	1
12:10 - 12:20	dog	1

12:00 - 12:10	owl	1
12:00 - 12:10	dog	2
12:00 - 12:10	cat	1
12:05 - 12:15	owl	2
12:05 - 12:15	dog	3
12:05 - 12:15	cat	2
12:10 - 12:20	dog	1
12:10 - 12:20	cat	1
12:10 - 12:20	owl	1
...		

12:00 - 12:10	owl	1
12:00 - 12:10	dog	2
12:00 - 12:10	cat	1
12:05 - 12:15	owl	2
12:05 - 12:15	dog	3
12:05 - 12:15	cat	2
12:10 - 12:20	dog	1
12:10 - 12:20	cat	1
12:10 - 12:20	owl	2
...		

Watermarking in **Windowed Grouped Aggregation** with **Update Mode**

Operations over streaming dataframe - windows



12:00 - 12:10	owl	1
12:00 - 12:10	cat	1
12:00 - 12:10	dog	2

12:00 - 12:10	owl	1
12:00 - 12:10	cat	1
12:00 - 12:10	dog	2
12:05 - 12:15	owl	2
12:05 - 12:15	cat	2
12:05 - 12:15	dog	3

Watermarking in Windowed Grouped Aggregation with Append Mode

Trigger action

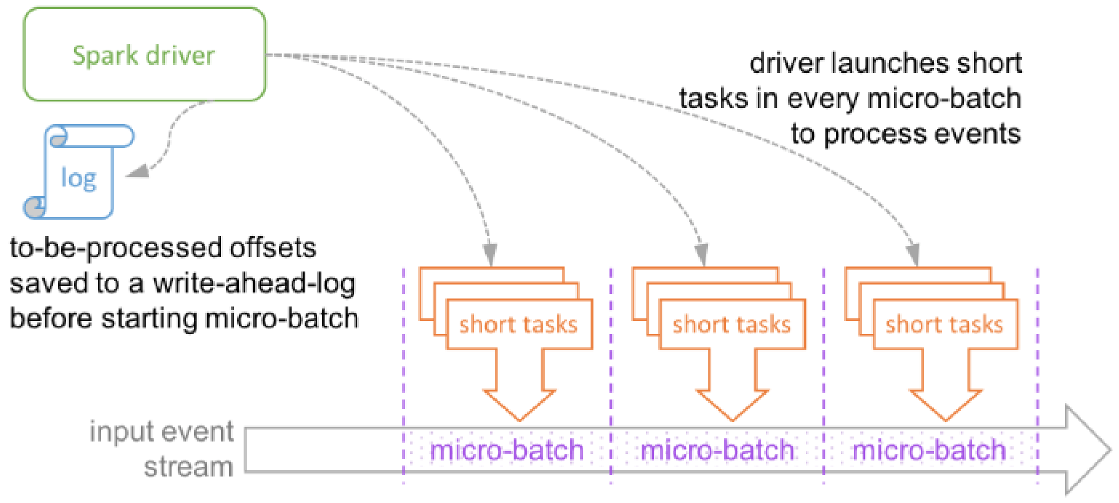
PROFINIT

- Allow flexibility over time interval triggers – if you don‘t need real time processing, there is no need to have it – **control cost**
- E.g. Update database every hour,…
- .trigger(**processingTime**='10 seconds')
  - Default 500ms
  - Micro-batch mode
- .trigger(**once**=True)
  - Process all available data in single batch and exit, now deprecated
- .trigger(**availableNow**=True)
  - Process all available data in multiple micro-batches and exit
  - Better scalability then „once“
- .trigger(**continuous**= '1 second')
  - Low-latency, continuous mode

Micro-batch vs continuous mode

Micro-batch vs continuous mode

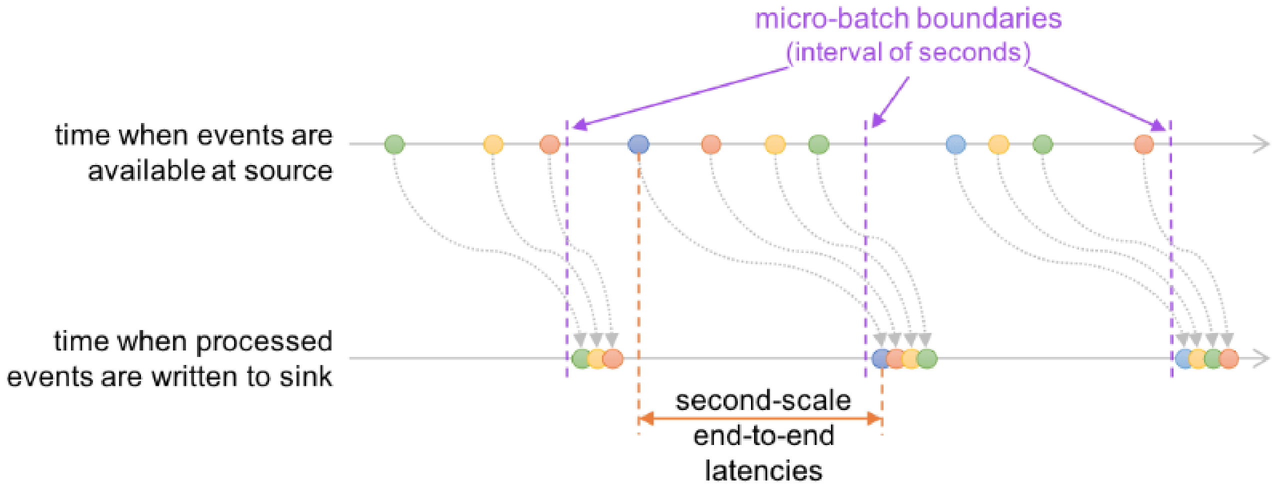
- Micro batch processing


Spark driver
log
driver launches short tasks in every micro-batch to process events
to-be-processed offsets saved to a write-ahead-log before starting micro-batch
short tasks
short tasks
short tasks
input event stream
micro-batch
micro-batch
micro-batch

Micro-batch Processing uses periodic tasks to process events

Micro-batch vs continuous mode

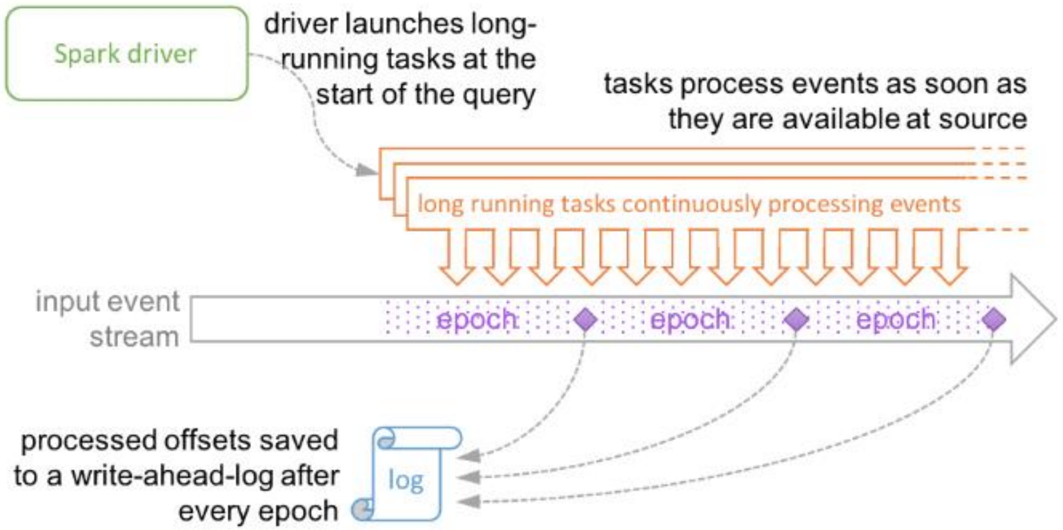
- Micro batch processing


micro-batch boundaries
(interval of seconds)
time when events are available at source
time when processed events are written to sink
second-scale end-to-end latencies

Second-scale end-to-end latencies with Micro-batch Processing

Micro-batch vs continuous mode

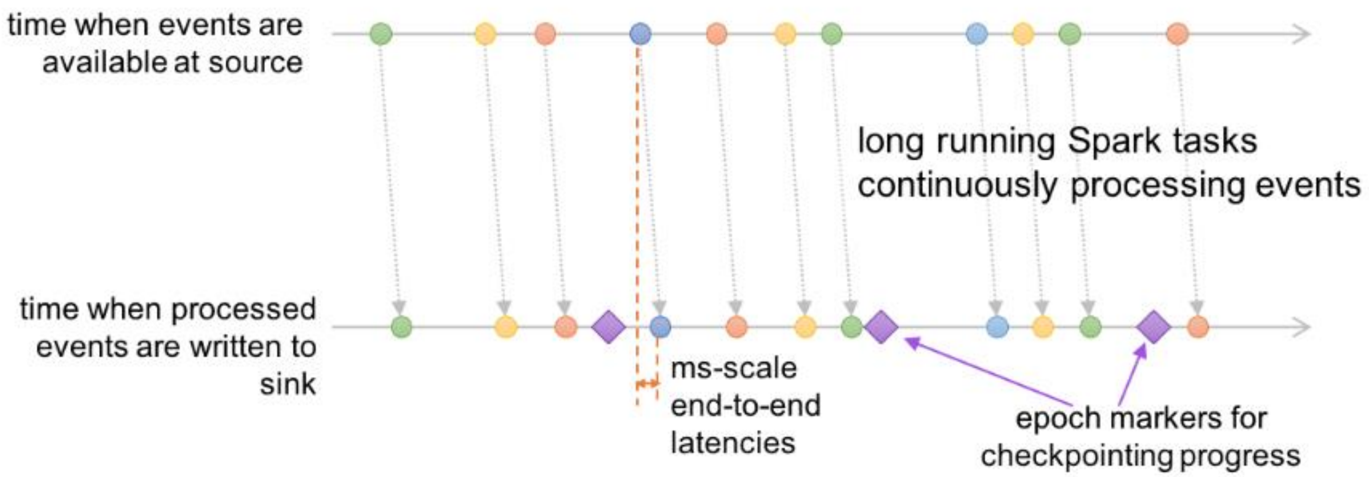
- Continuous processing


Spark driver
driver launches long-running tasks at the start of the query
tasks process events as soon as they are available at source
long running tasks continuously processing events
input event stream
epoch
epoch
epoch
processed offsets saved to a write-ahead-log after every epoch
log

Continuous Processing uses long-running tasks to continuously process events

Micro-batch vs continuous mode

- Continuous processing


time when events are available at source
long running Spark tasks continuously processing events
time when processed events are written to sink
ms-scale end-to-end latencies
epoch markers for checkpointing progress

Millisecond-scale end-to-end latencies with Continuous Processing

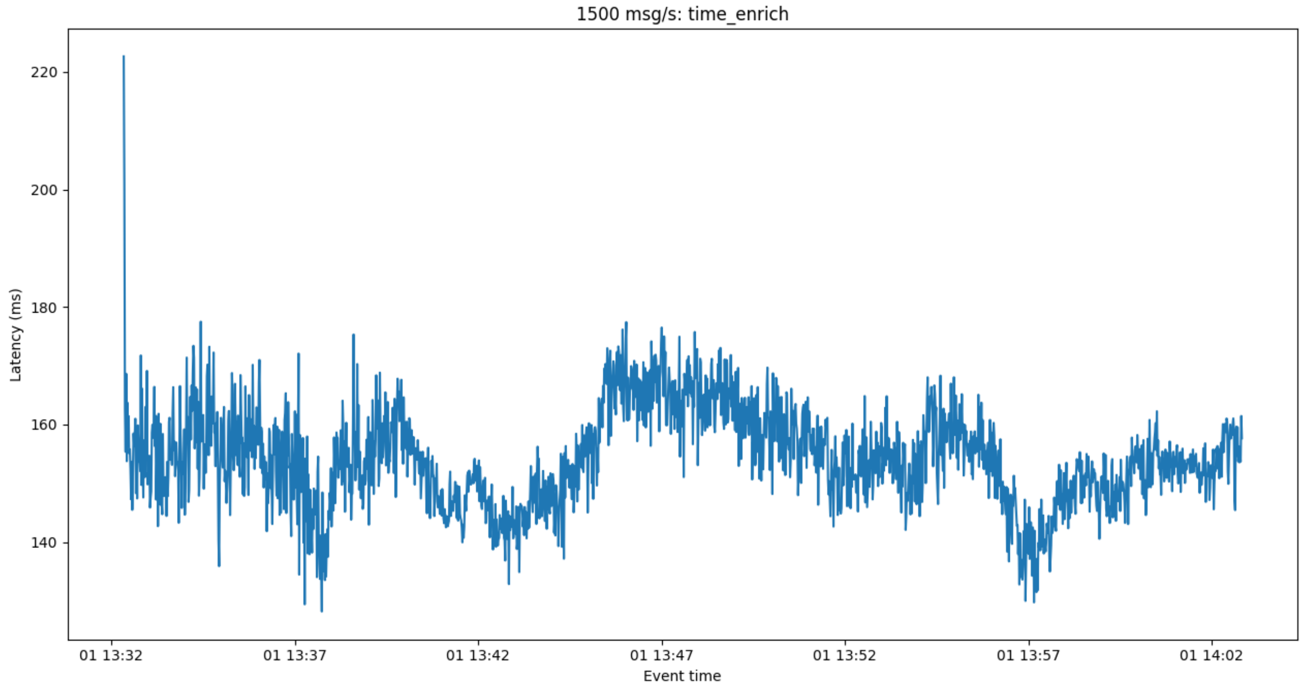
Extras

Other streaming services

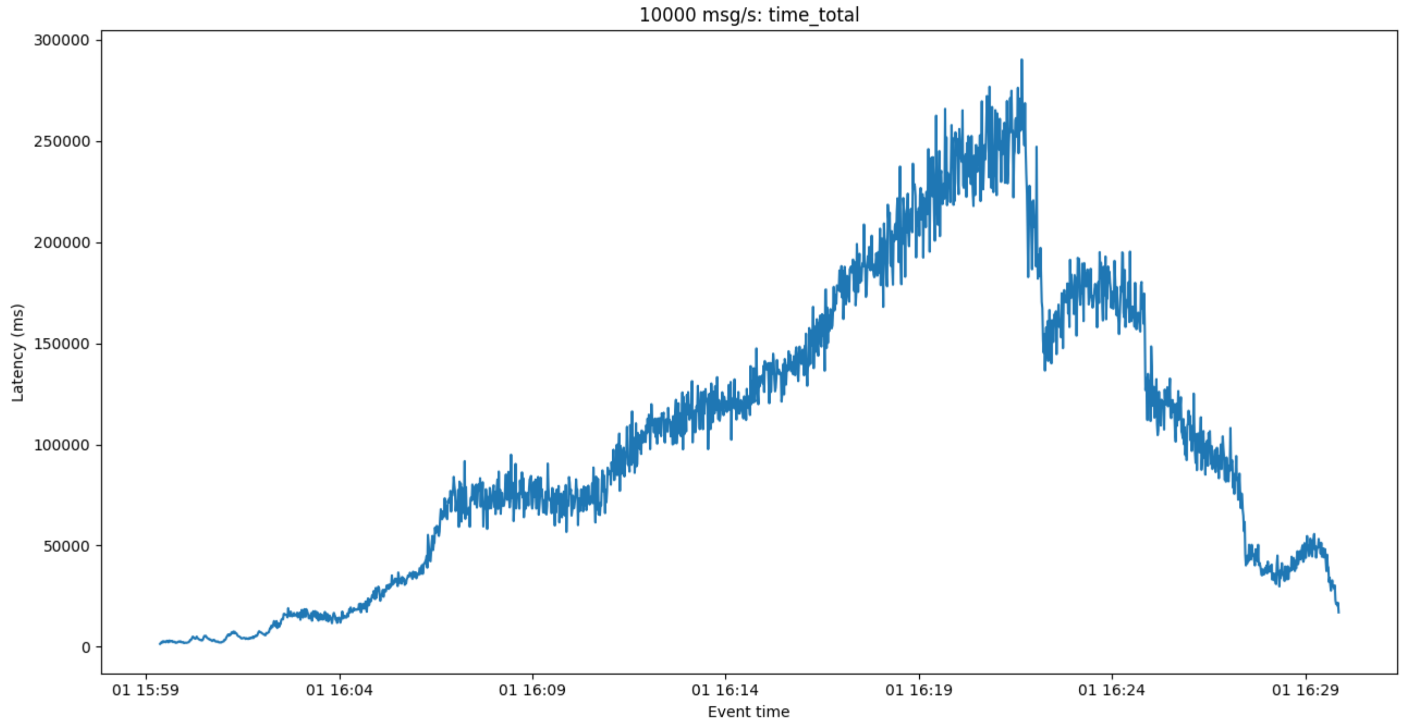
- Apache Flink
  - Real-time streaming
  - Also supports batch processing
- Apache Storm
  - Real-time streaming
- Kafka Streams

Resilient/stable streaming example

PROFINIT



Latency cumulation



PROFINIT

Q&A

PROFINIT

Thanks for attention!

Profinit EU, s.r.o.
Tychonova 2, 160 00 Praha 6

Tel.: + 420 224 316 016, web: www.profinit.eu

LINKEDIN
linkedin.com/company/profinit

TWITTER
@profinit_EU

FACEBOOK
facebook.com/Profinit.EU

YOUTUBE
Profinit EU, s.r.o.