Spark Structured Streaming

Real Time Processing with Spark Structured Streaming

What will we cover

- Differences between batch and stream processing
- Need for stream processing
- Challenges of stream processing
- Design and Concepts of Spark Structured Streaming
- Sources, Sinks, Output Modes and Triggers
- Operations
    - Window Operations, Watermarking
    - Join Operations
    - Deduplication
- Hands on lab to see various operations in action
- Considerations while running Spark Structured Streaming in Production
    - Recovering from checkpoints
    - Monitoring
- Addressing challenges of stream processing with Apache Spark Structured Streaming
- UI Enhancement in Spark 3.0
- Path to expertise

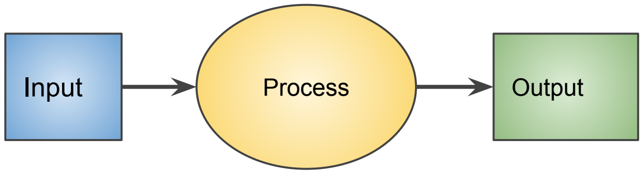
Pre requisites

- Basics of Spark SQL
- Basics of Python / pyspark
- Spark Core: Desirable

Processing: Batch vs Stream

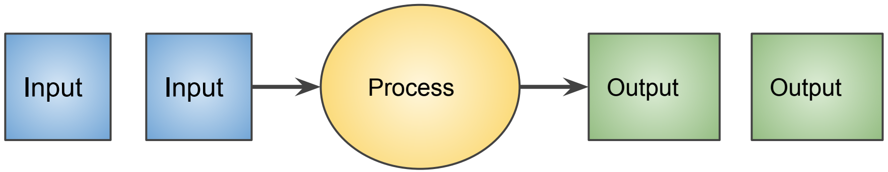
Batch

- Fixed set of inputs
- Process starts and then ends



Stream

- Unbounded stream of inputs
- Processing keeps on running



Need for Stream Processing

Notification and Alerting
Trigger alert when an event or series of events occur

Real time reporting
Real-time dashboard

Incremental ETL
Incorporate new data quickly and update data warehouse

Online Machine Learning
Train a model on a combination of streaming and historical data

Benefits of stream processing

Lower Latency

Application responds quickly

Efficiency

Incremental computation

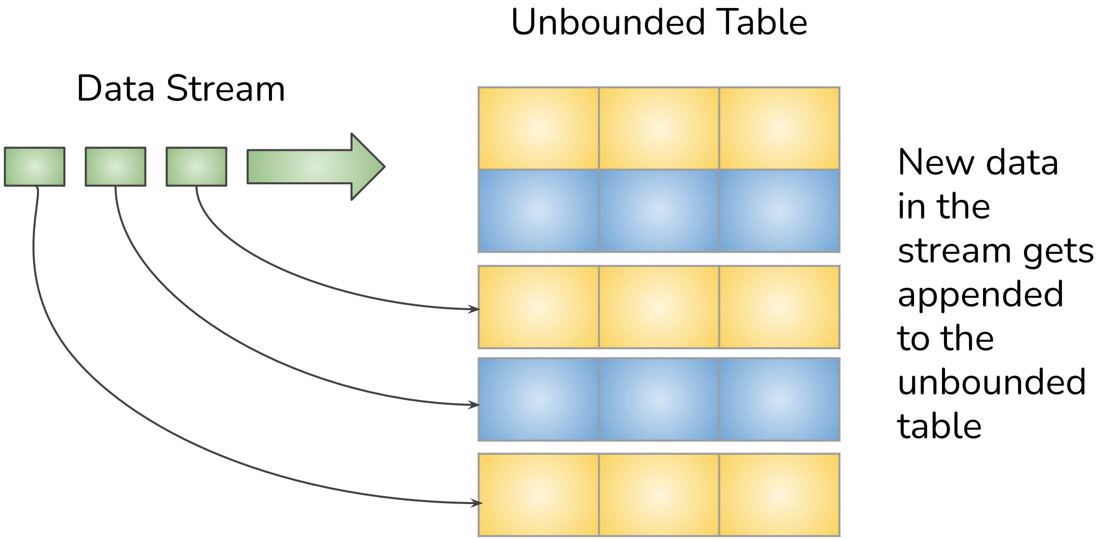
Challenges of Stream Processing

- Respond to events at low latency
- Ability to restart from failure with state management
- Handle out-of-order data
- Support different types of windows
- Determine how to update output sink (or calculate aggregates) as new data arrives
- Update business logic at runtime
- Handle load imbalance

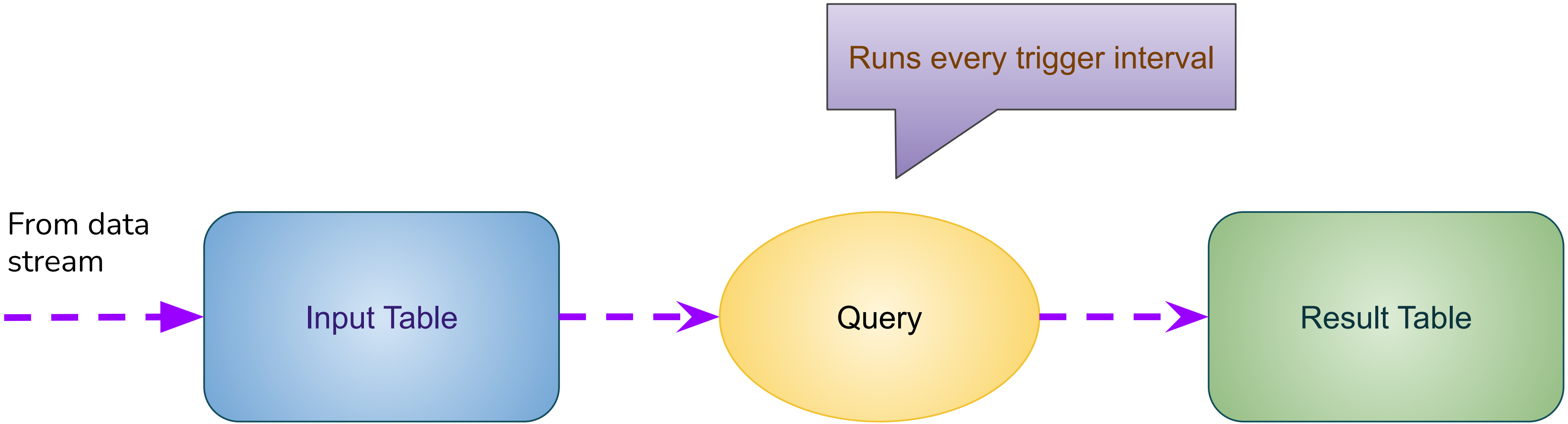
Spark Structured Streaming Concepts

Concepts of Structured Streaming

- Input data stream is considered as the "**Input Table**".
- Every **data** item that is arriving on the stream is like a new row being **appended** to the Input Table.
- A **query** on the input will generate the "**Result Table**".
- Every **trigger** interval, new rows get appended to the Input Table, which eventually **updates** the **Result Table**.



Concepts of Structured Streaming



Spark Streaming APIs

Spark Streaming

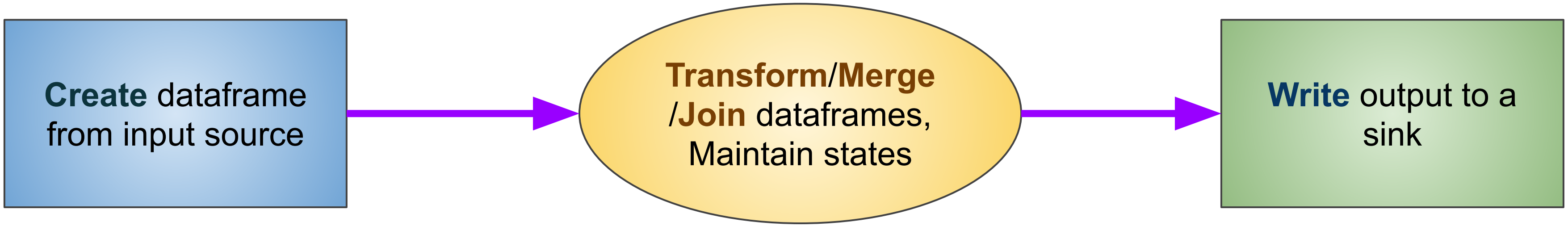
- supports only micro-batch processing
- hard to deal with late data
- needs custom logic to work with event time constructs
- uses DStreams which are internally RDDs. Hence no benefit of Tungsten and Catalyst optimisations

Spark Structured Streaming

- supports microbatch as well as continuous processing (~ 1 ms latency)
- in-built support to handle late data
- works with event time as well as processing time constructs
- built on top of DataFrames/DataSets and hence takes benefit of:
  - Catalyst optimisations
    - constant folding, predicate pushdown, projection pruning, pipelining operations, cost based optimisations, code generation
  - Tungsten optimisations
    - binary encoding, customized memory management, code generation, cache aware computation etc.

Writing a Spark Structured Streaming Application

Parts of a structured streaming application



Input Sources

File
Read files arriving in a directory

Socket
Read UTF8 text data from a socket

Rate
Generates data at specified records per second

Kafka
Read from kafka topics

Sinks

Kafka
Stores the output to kafka topics

File
Write to output directory

Foreach
Custom/arbitrary operations on each output record of a streaming query

Sinks

Console
Print to the console

ForeachBatch
Custom/arbitrary operations on each output batch of a streaming query

Memory
Store output in memory as in-memory table

Triggers

Decides when data is output

- **Unspecified** (default)
  - look for the new data as soon as the previous group of data has been processed
- **Fixed interval** microbatches
  - Query executed in micro batch mode
  - Trigger.ProcessingTime(“5 seconds”)
- **One time** micro-batch
  - Run once and shut down
  - Trigger.Once()
- **Continuous** with fixed checkpoint interval
  - Run the query continuously and save checkpoint periodically
  - Trigger.Continuous(“1 second”)

Parts of Structured Streaming Program Step 1 of 3: Create dataframe from source

```
## Read the text data from socket connection

lines = spark\
        .readStream\
        .format('socket')\
        .option('host', host)\
        .option('port', port)\
        .load()
```

Parts of Structured Streaming Program
Step 2 of 3: Transform input dataframe

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

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

Parts of Structured Streaming Program
Step 3 of 3: Run a query

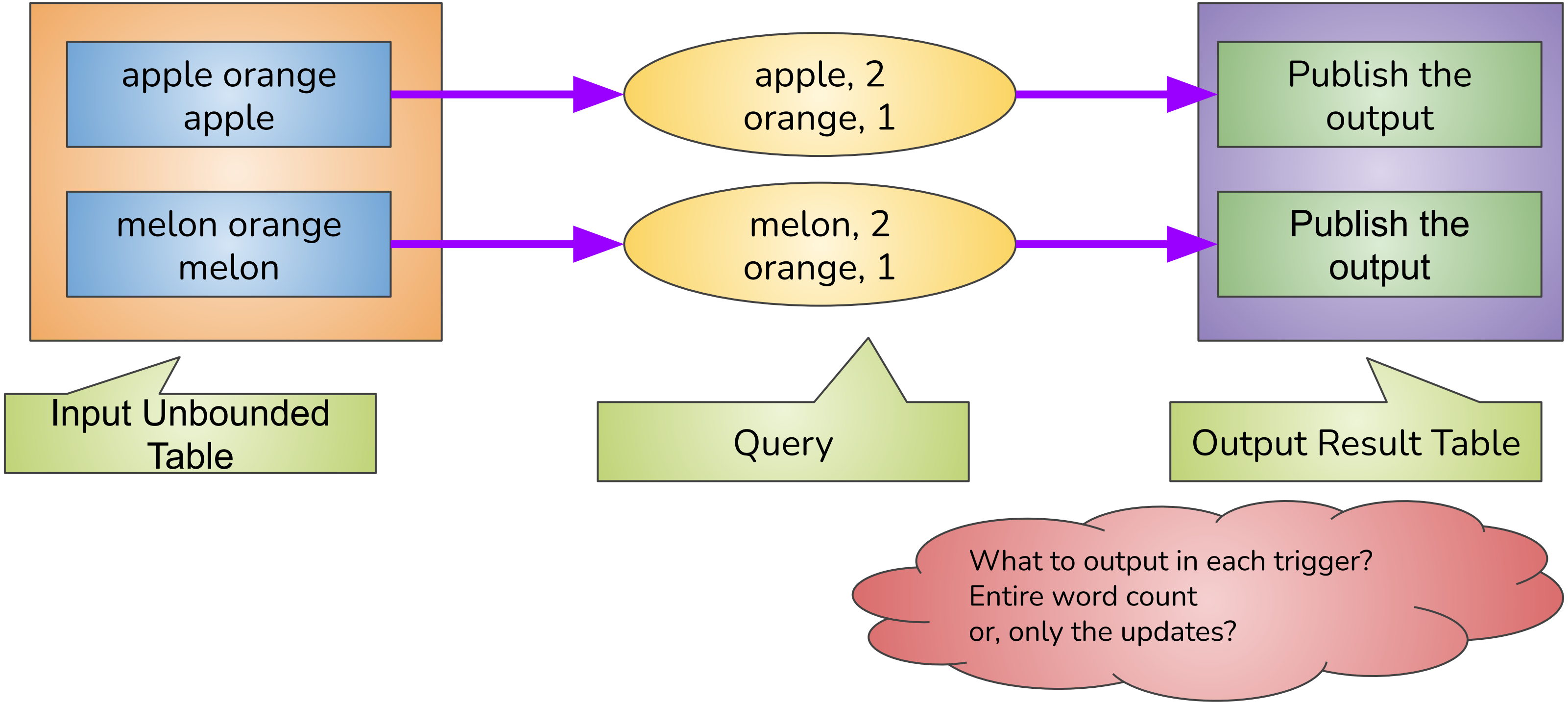
```
query = wordCounts\
        .writeStream\
        .outputMode('complete')\
        .format('console')\
        .start()

query.awaitTermination()
```

Demo
Network Word Count

Output Modes

Understanding Output Modes

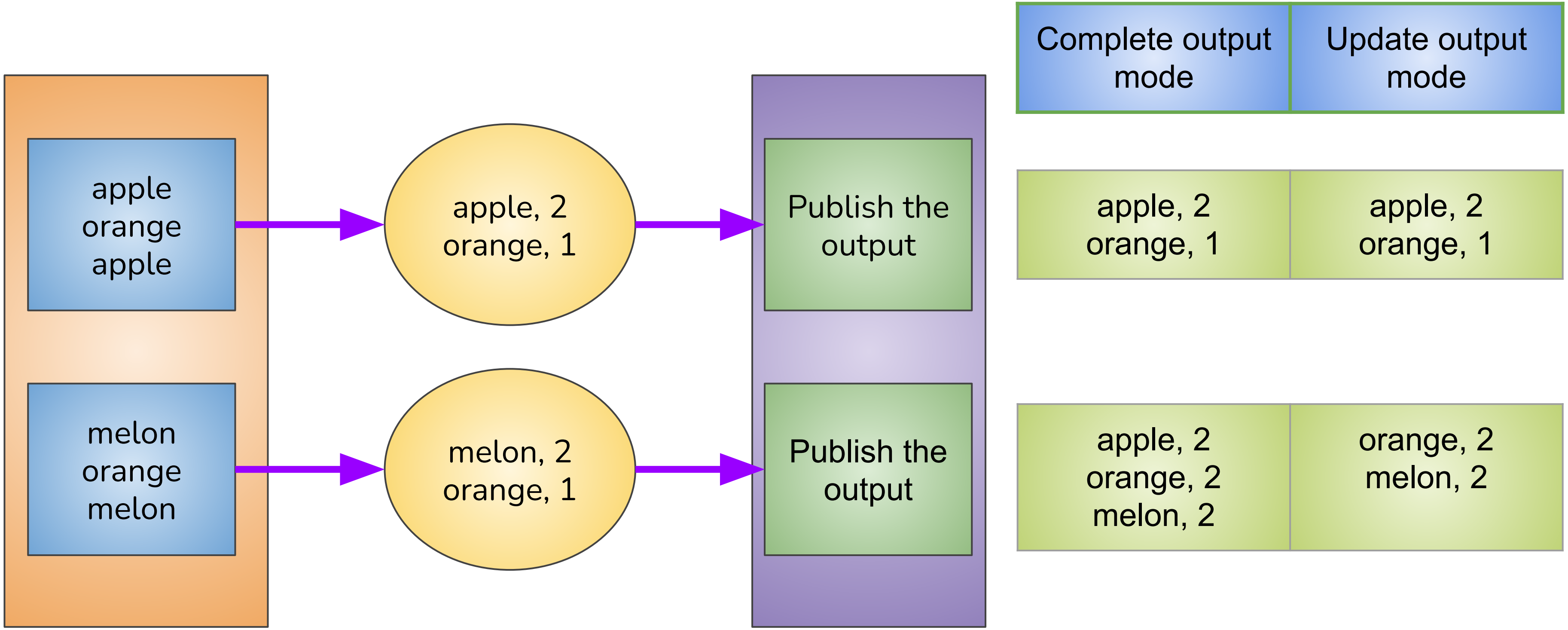


Output Modes

Output mode controls which part of the result table has to be produced in each trigger

Complete	Update	Append
• The entire updated Result Table will be written to the external storage	• Only the rows that were updated in the Result Table since the last trigger will be written to the external storage • If the query doesn't contain aggregations, it will be equivalent to Append mode.	• Only the new rows appended in the Result Table since the last trigger will be written to the external storage. • Applicable only on the queries where existing rows in the Result Table are not expected to change

Understanding Output Modes



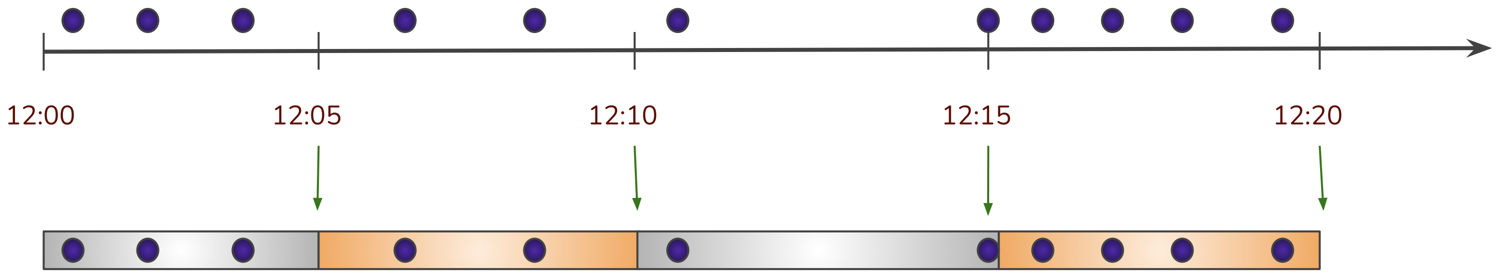
Demo
Revisiting Network Word Count
[Output Modes]

Windows in Stream Processing

Tumbling Windows

- Triggers at the interval specified (called as batch interval)
- The size of the window equals the size of the batch interval
- One message/event belongs to only one window (non-overlapping)

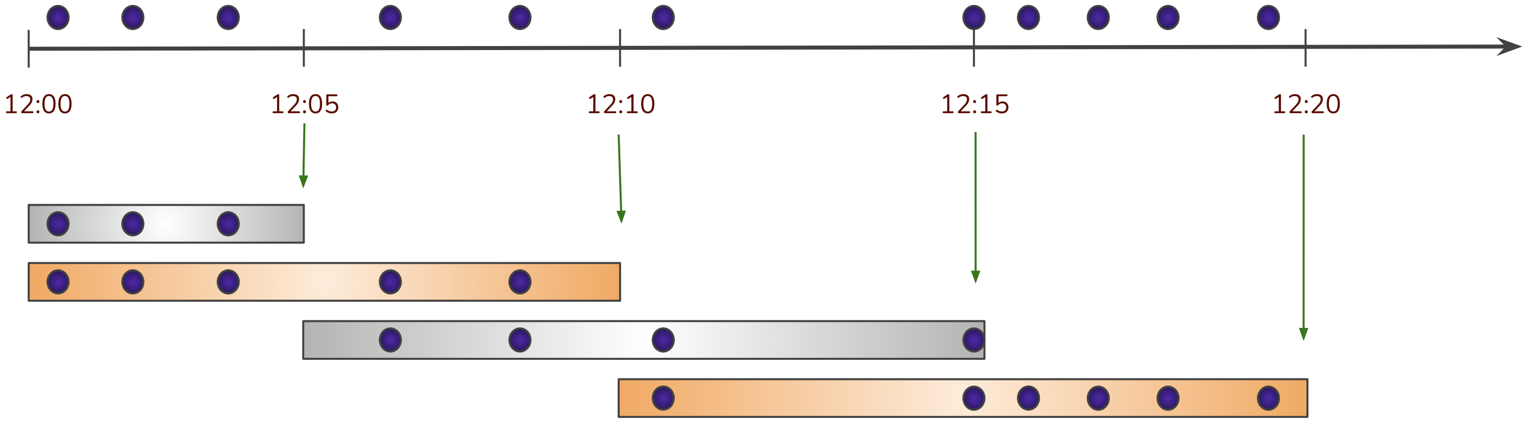
Batch Interval: 5 mins



Sliding (Hopping) Windows

- Has a sliding and a batch interval
- Triggers at the sliding interval. For tumbling windows, batch interval equals the sliding interval
- The size of the window equals the size of the batch interval
- One message/event may belong to multiple windows (overlapping)

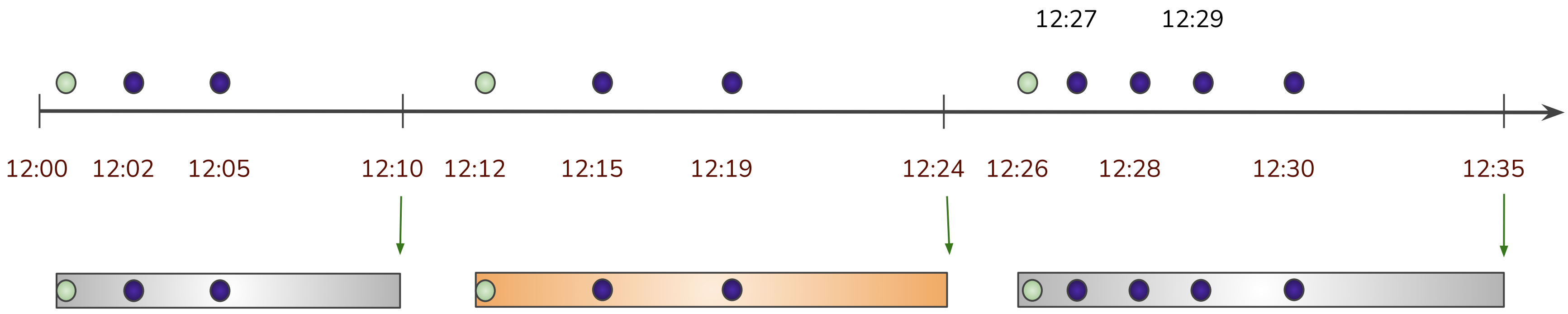
Sliding Interval: 5 mins
Batch Interval: 10 mins



Session Windows

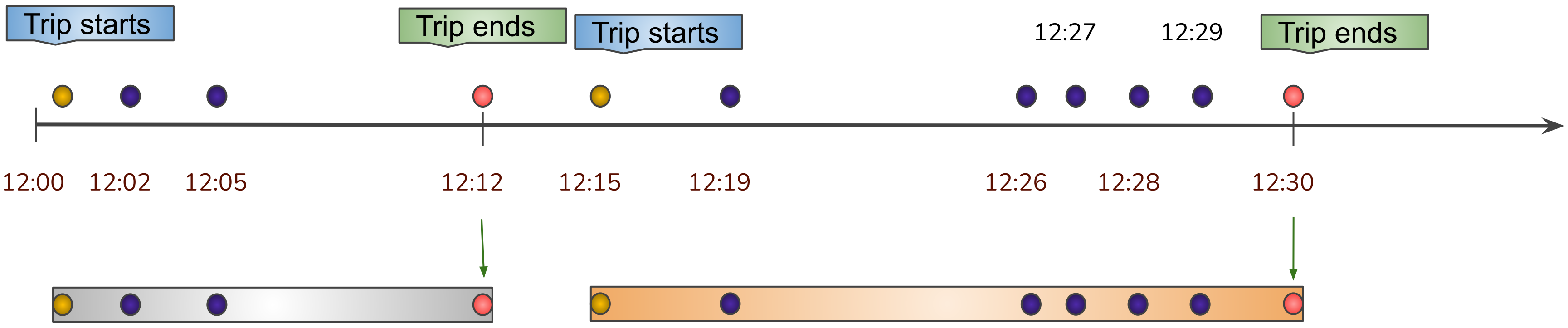
Sessions are the unpredicted periods of events happening, such that time gap between two events is less than the threshold.
Session windows are closed after the period of inactivity
Session windows can never be empty

Session timeout: 5 mins

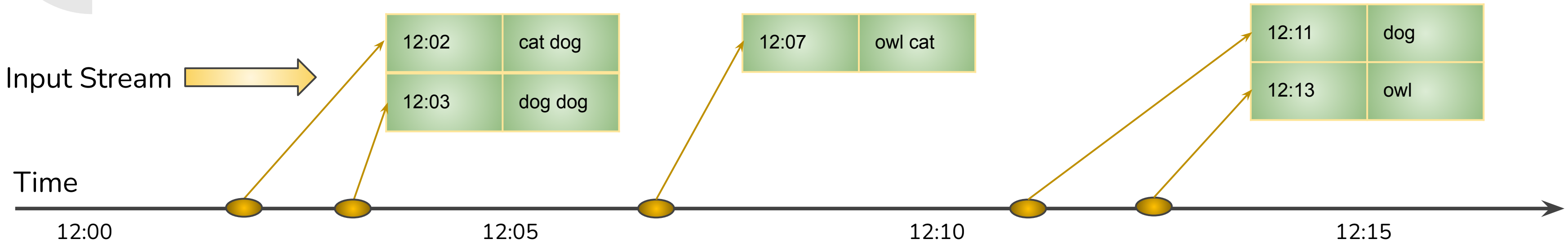


Custom sized Windows

Sometimes a pair of events decide the start and end of a window.
There is no information on the batch size or inactivity duration.
Eg: beginning of a trip starts a window and end of the trip closes the window. All the events coming between these two events go in the same window



Examining sliding window



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

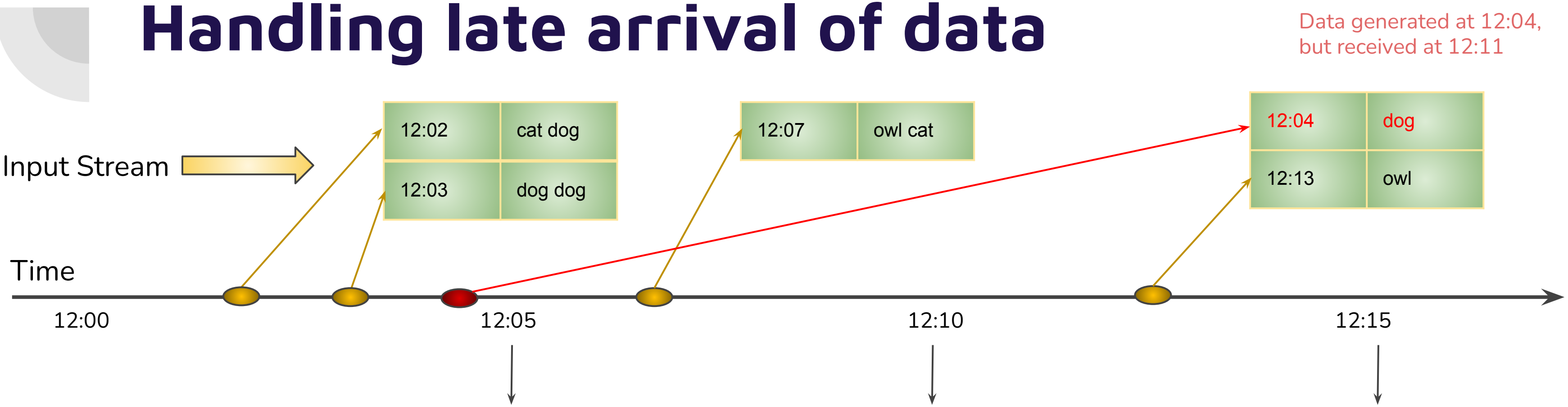
12:00 - 12:10	cat	2
12:00 - 12:10	dog	3
12:00 - 12:10	owl	1
12:05 - 12:15	cat	1
12:05 - 12:15	owl	1

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

Windowed Grouped Aggregation
Batch interval: 10 mins
Sliding interval: 5 mins

Results table updated every 5 mins

Handling late arrival of data



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
12:05 - 12:15	dog	1
12:10 - 12:20	owl	1

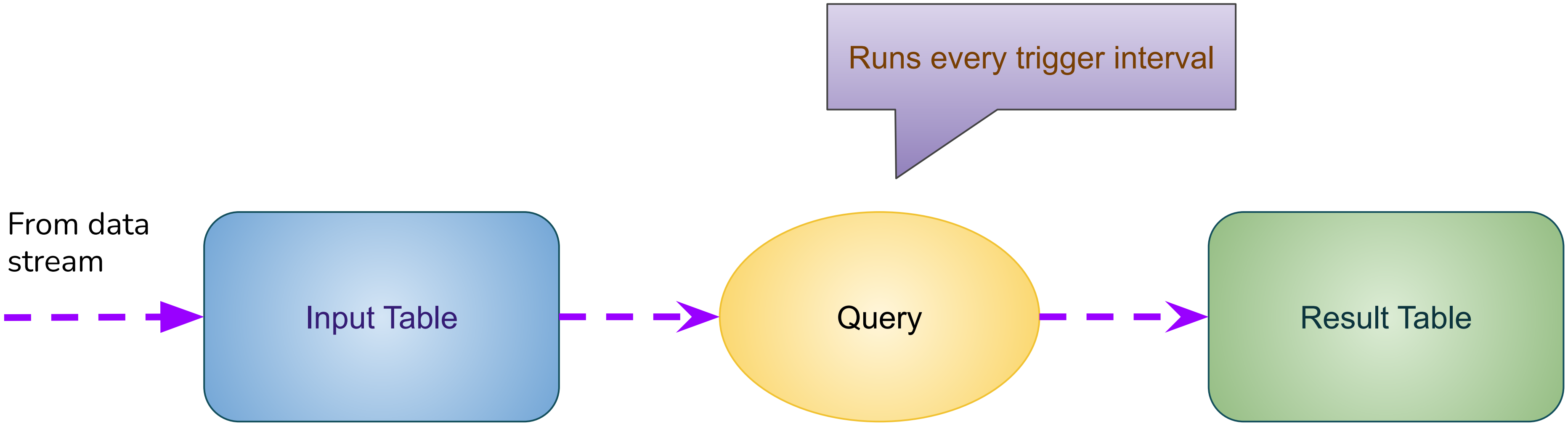
Windowed Grouped Aggregation
Batch interval: 10 mins
Sliding interval: 5 mins

Results table updated every 5 mins

Count corresponding to older window updated

Watermarking

Concepts of Structured Streaming



Watermarking

- Define the watermark of a query by specifying the event time column and the threshold on how late the data is expected to be in terms of event time.
- Watermark value:
    - **T = max event time seen by the engine - late threshold**
    - Eg: if max event time is 12:10 and late threshold is 10 minutes, then watermark value is 12:00
    - All states related to data before the watermark value gets dropped
- Late data within the threshold will be aggregated, but data later than the threshold will start getting dropped.
- Use **`withWatermark(ts_col, late_threshold)`** on the streaming dataframe.

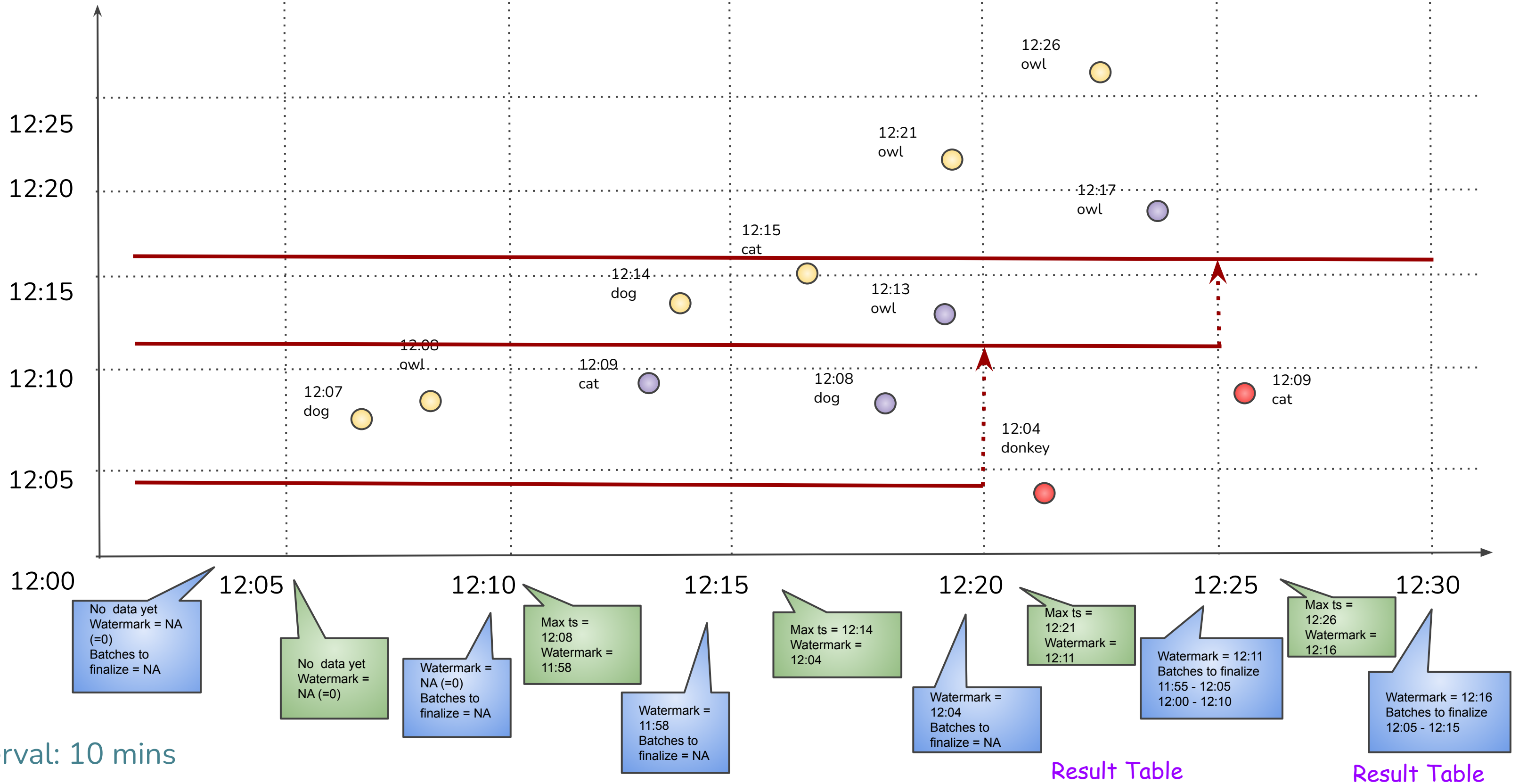
Watermarking with append output mode

Before processing:

- Based on watermark value, decide the batch to finalize
  - batch is finalized once we are sure that data contained in the interval is no longer going to change
  - and this can only happen when watermark becomes more than the end boundary of the batch

After trigger:

- Watermark is updated after the microbatch processing
- Watermark acts as a barrier, a threshold or a limit of how much old data can arrive and get processed.



Batch Interval: 10 mins

Trigger Interval: 5 mins

Watermark: 10 mins

Result Table

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

Result Table

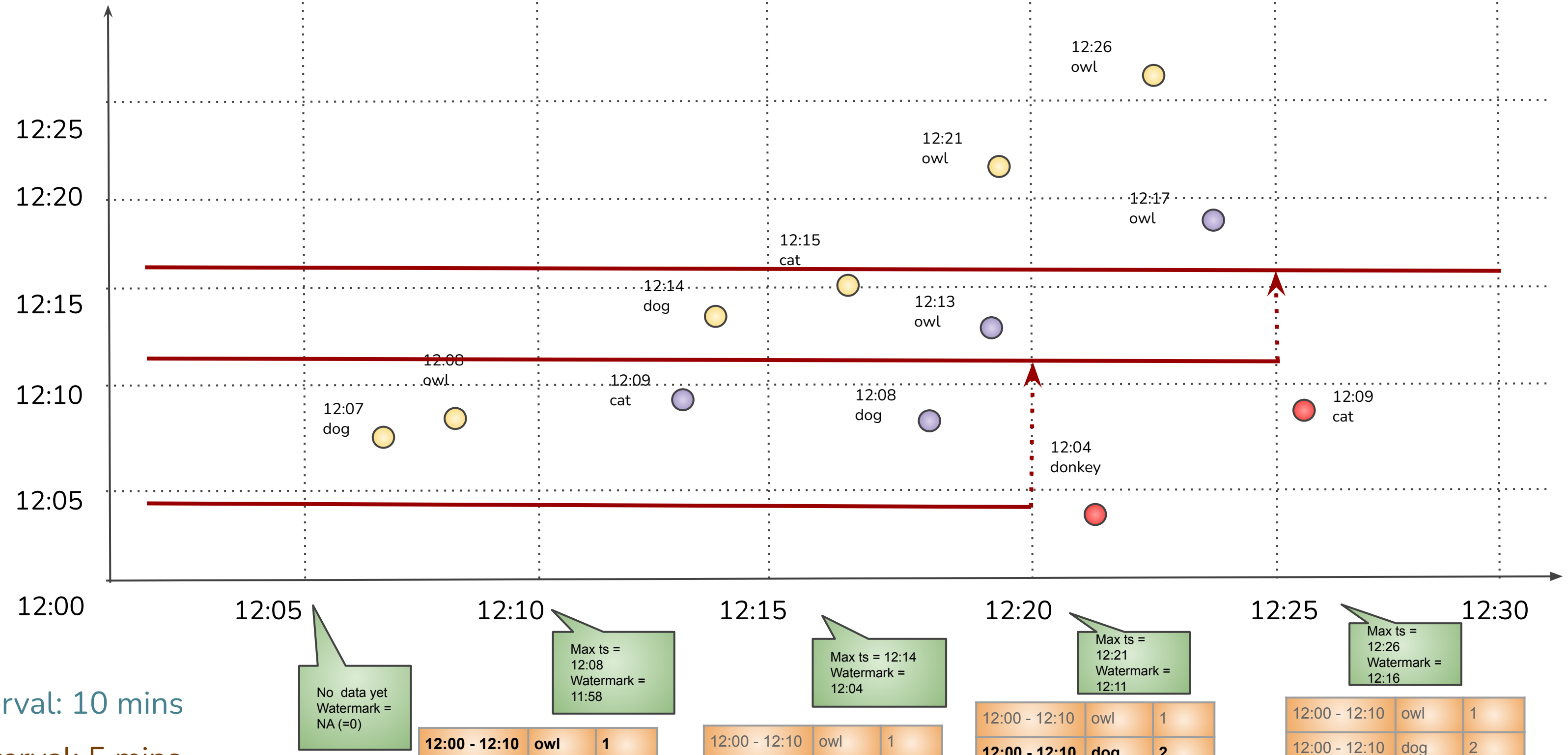
12:05 - 12:15	owl	2
12:05 - 12:15	cat	2
12:05 - 12:15	dog	3

Watermarking with append output mode

- We set the limit, but is calculated dynamically based on the event time seen by the system.
- If no new event arrive for sometime, watermark can remain unchanged
- Until a batch is finalized, all the data for that batch is kept in an internal state

Watermarking with update output mode

- Output is generated whenever there is input data
- Older batch can get modified, when data belonging to that batch arrives (but within the watermark threshold)
- After trigger, watermark is updated after the microbatch processing



Batch Interval: 10 mins

Trigger Interval: 5 mins

Watermark: 10 mins

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

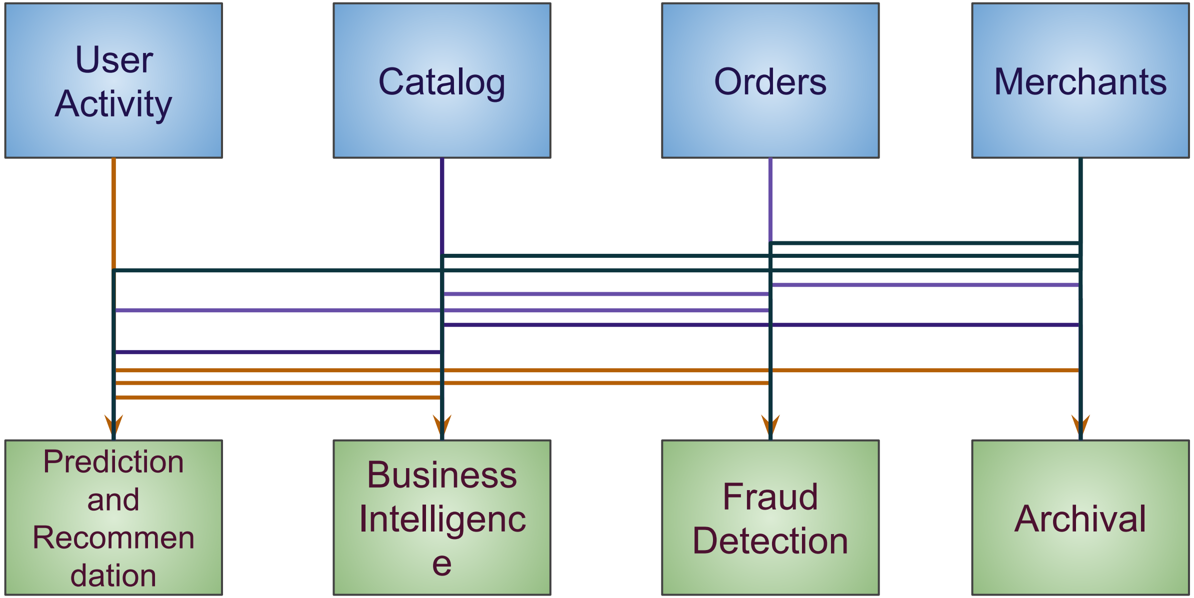
12:00 - 12:10	owl	1
12:00 - 12:10	dog	1
12:00 - 12:10	**cat**	**1**
12:05 - 12:15	owl	1
12:05 - 12:15	**dog**	**2**
12:05 - 12:15	**cat**	**1**
12:10 - 12:20	**dog**	**1**

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	owl	1
12:10 - 12:20	**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	owl	1
12:10 - 12:20	cat	1
12:10 - 12:20	**dog**	**2**

A Quick Introduction to Kafka

Need for Kafka



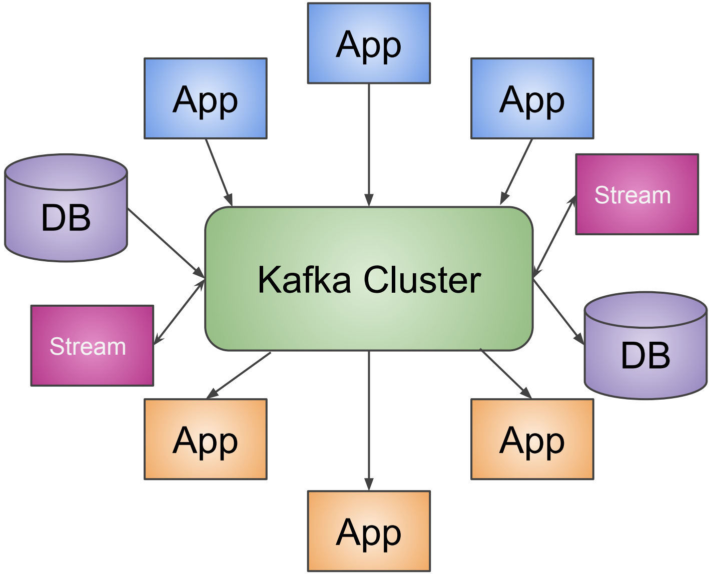
- Too many integrations
- Each application to handle connection and failure in the source, leading to huge code duplication
- Difficult to add, remove and track the source and targets

What is Kafka

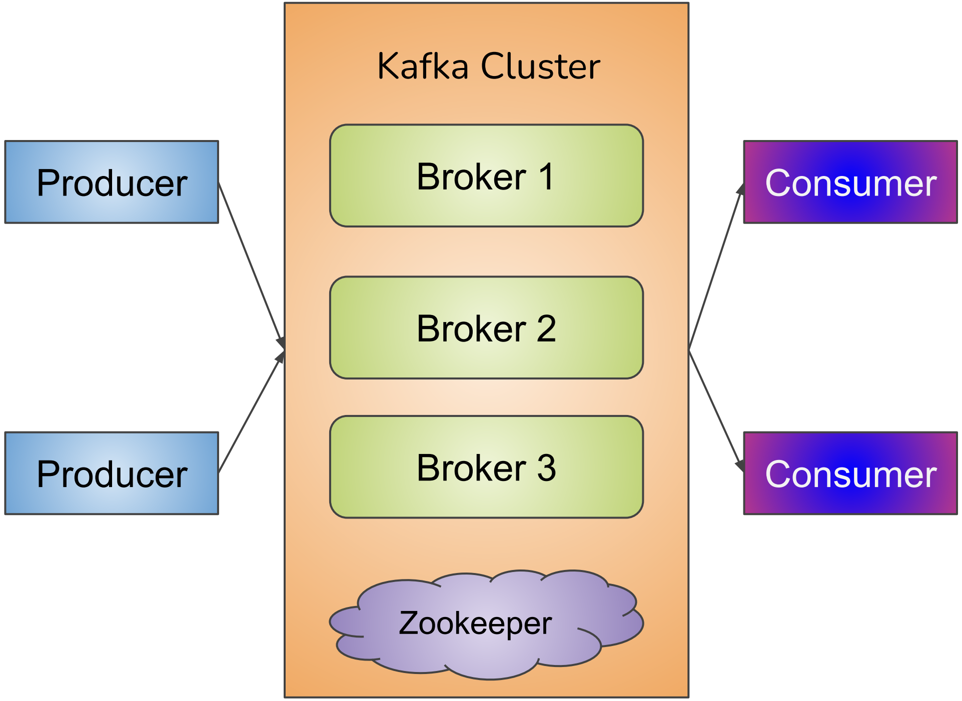
- Apache Kafka is a message intermediation system.
- It is a messaging system based on the publisher/subscriber model in which several producers and subscribers can read and write.

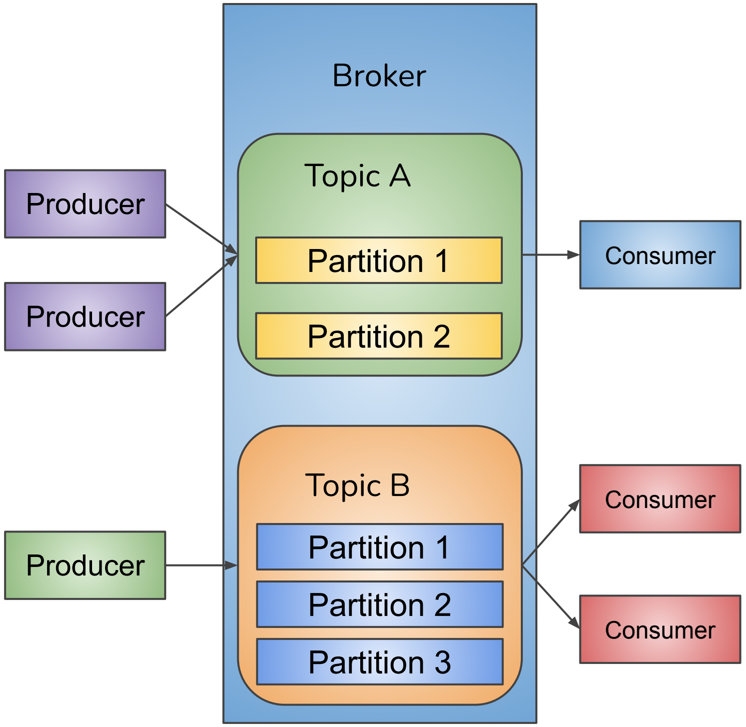
- It is horizontally scalable, fault-tolerant, fast, and runs in production in thousands of companies.
- It can connect to wide variety of systems for data ingress/egress.

- Used for building real-time data pipelines and streaming apps.

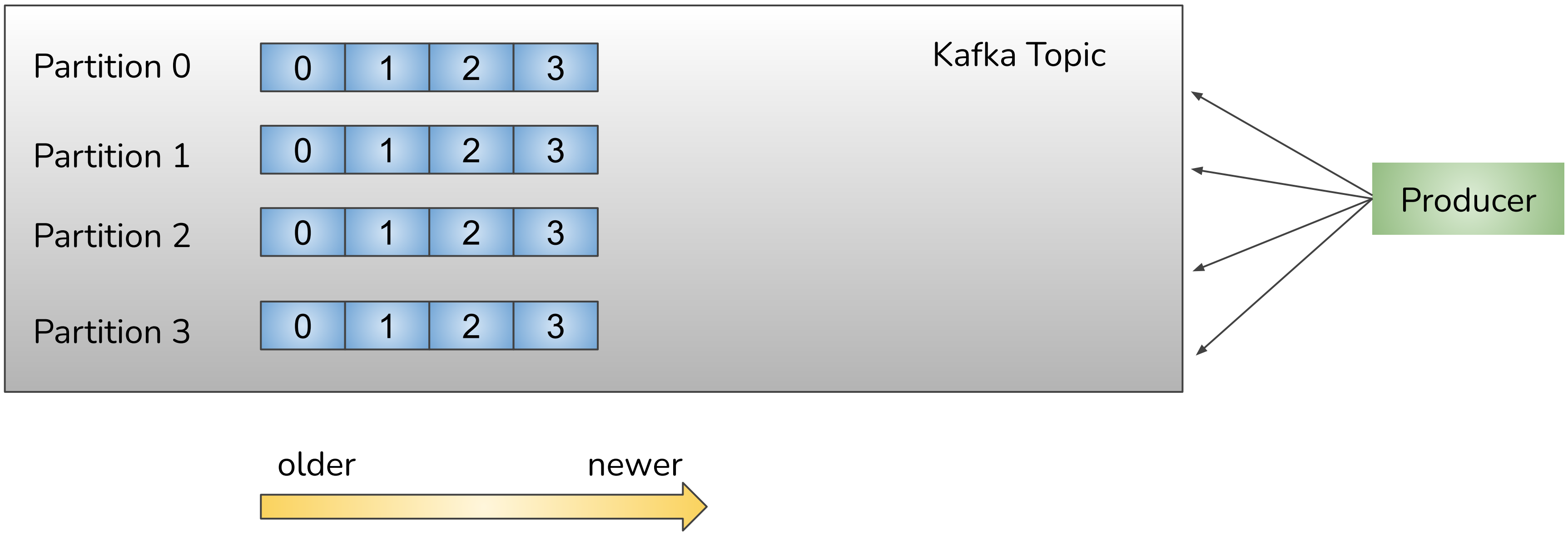


Design of Kafka





Offsets in Kafka



Demo
Watermarking

Demo
Joining Kafka Topics

Spark Structured Streaming in Production

Structured Streaming in Production

- Dropping Duplicates
- Using supervisor
- Monitoring
- Recovering from checkpoint

Dropping Duplicates in a Microbatch

- Use a column or a combination of column(s) to drop the duplicate values.

```
streamingDf
  .dropDuplicates("guid", "eventTime")
```

- Can be used with watermark
- Define a watermark on a event time column and deduplicate using both the unique and the event time columns.

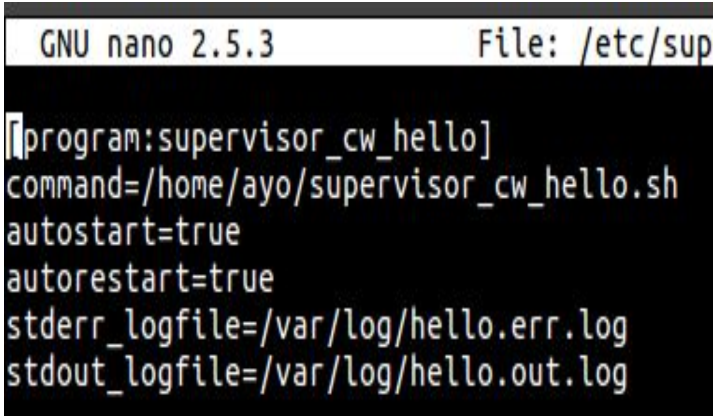
```
streamingDf
  .withWatermark("eventTime", "10 seconds")
  .dropDuplicates("guid", "eventTime")
```

- Old state data will be removed from past records that are not expected to get any duplicates any more.
- Bounds the amount of the state the query has to maintain.

Using supervisor

Supervisor is a client/server system that allows its users to control a number of processes on UNIX-like operating systems.

Supervisor Documentation



Monitoring Streaming Queries

- `streamingQuery.lastProgress()` returns a `StreamingQueryProgress` object
  - information about the progress made in the last trigger of the stream
- `streamingQuery.recentProgress` which returns an array of last few progresses.
- `streamingQuery.status()` returns a `StreamingQueryStatus` object.
  - It gives information about what the query is immediately doing like
    - is a trigger active
    - is data being processed

Monitoring Streaming Queries

- Can push to metric collection system like graphite and then visualize the metrics via grafana or other dashboards

```
spark.conf.set("spark.sql.streaming.metricsEnabled", "true")
```

- Support for graphite has been since long and support for **prometheus** has been added in **Spark 3.0**
- **Native Support of Prometheus in Spark 3.0**

Recovering from checkpoint

- In case of a failure or intentional shutdown, you can **recover** the previous progress and state of a previous query, and continue where it left off.
- You can configure a query with a checkpoint location, and the query will **save all the progress information** (i.e. range of offsets processed in each trigger) and the **running aggregates** to the checkpoint location.
- This checkpoint location has to be a path in an **HDFS compatible** file system, and can be set as an **option** in the **DataStreamWriter** when starting a query.
- `outputStream.option("checkpointLocation", "path/to/HDFS/dir")`
- On changing the type of aggregation, recovery using checkpoint is not possible.

Other considerations

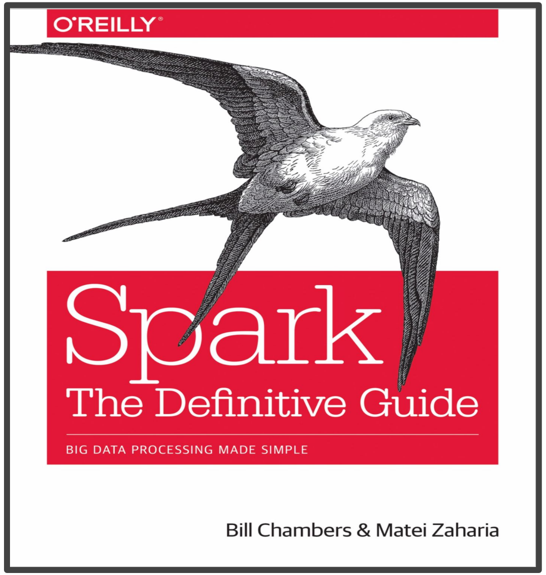
- Once a spark streaming has been setup, following changes to it are not allowed:
  - Config values for:
    - spark.sql.shuffle.partitions
    - spark.sql.streaming.stateStore.providerClass
    - spark.sql.streaming.multipleWatermarkPolicy
  - The aggregation key if state is maintained
- So, either remove the checkpoint for making these changes or implement a custom state store
- When running streaming with dynamic allocation, shuffle files are not deleted and hence can continue to pile up.
- Use maxOffsetsPerTrigger to limit number of kafka messages per trigger.

Addressing challenges of stream processing with Apache Spark

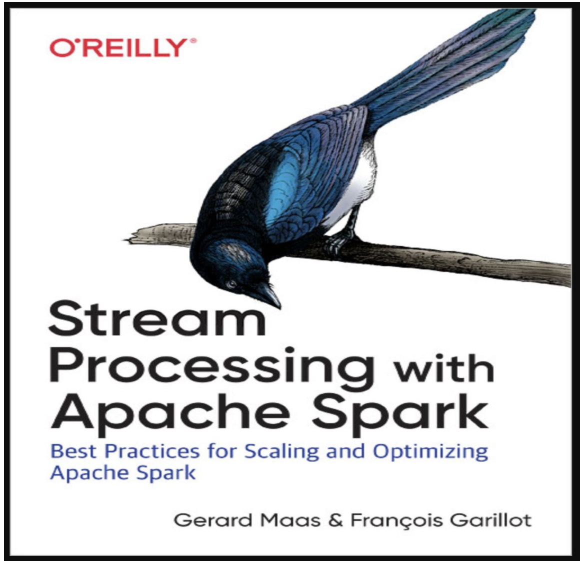
- Consistent API for Stream Processing and Static data using DataFrames
- Handling Event time and late data
    - Supports notion of processing time as well as event time
    - Handles window-based aggregations with support of watermarking
- Fault tolerance semantics
    - Use of checkpoint
- Continuous Processing supports low latency processing as low as 1 ms

References

The Definitive Guide



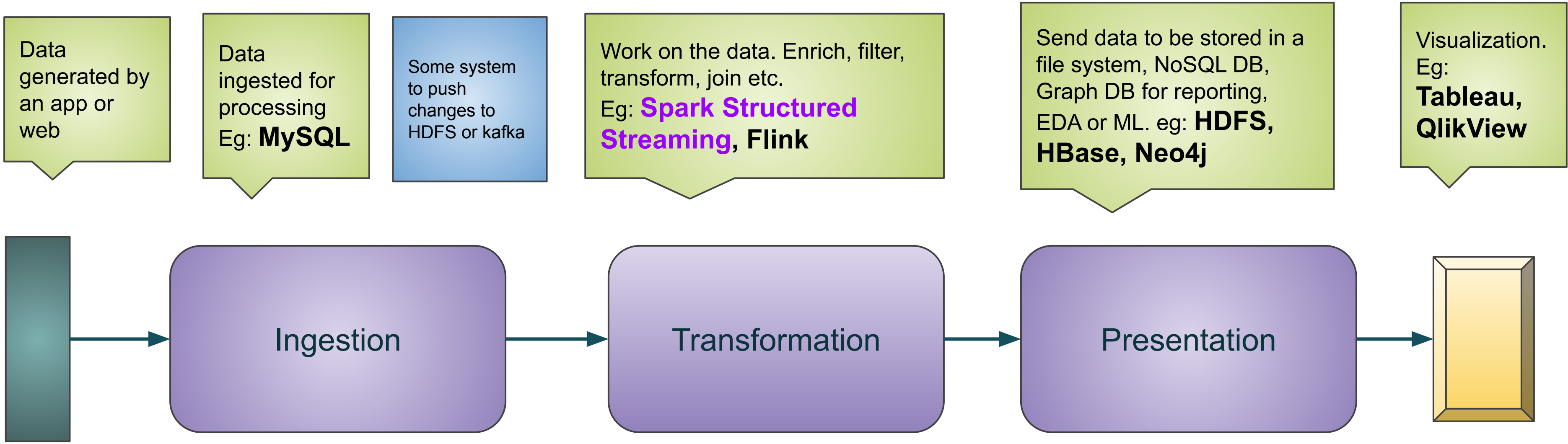
Specialized for Streaming



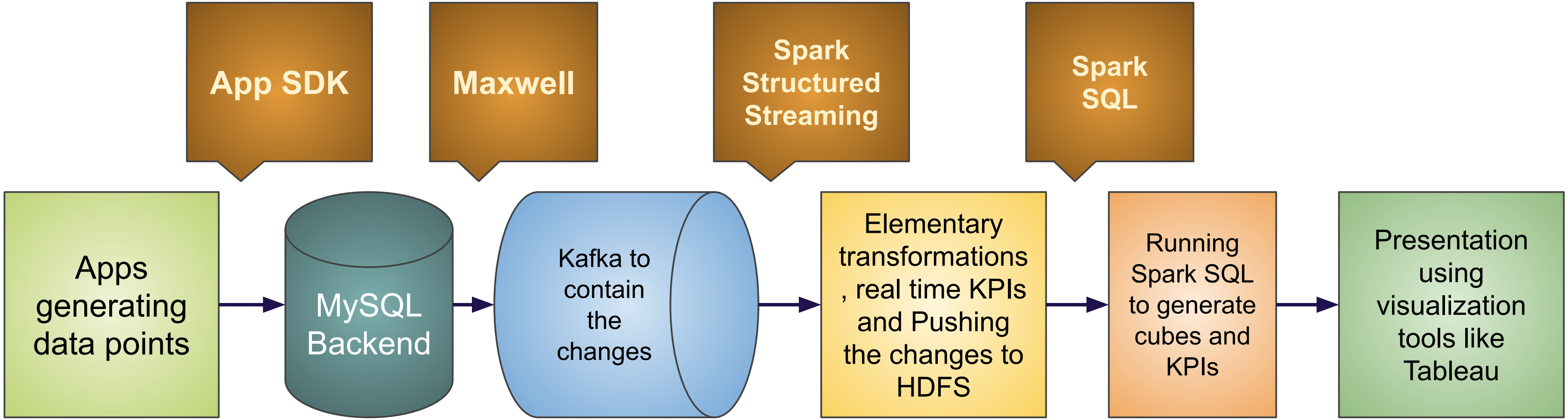
Documentation

System Designs involving **Spark Structured Streaming**

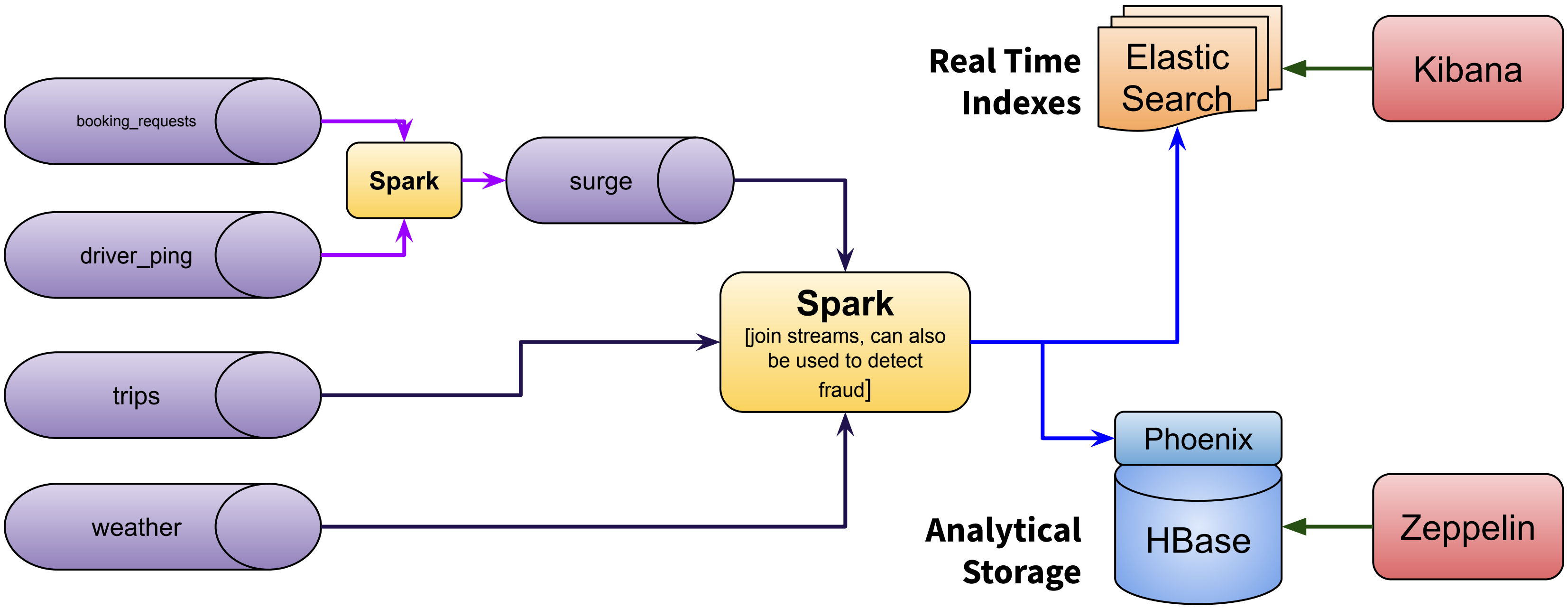
A standard flow



Use Case 1:
Near Real Time Data Lake



Use Case 2:
Real time taxi surge calculation



Road Ahead: Path to Expertise

What to learn next: Arbitrary State Management

- Spark maintains and updates state for most of our use cases
- Use cases for custom state management
    - Create window based on count of a query or a given action like start and end of a trip
    - Maintain a user session based on the duration between their activities
- Available APIs for custom state management
    - **mapGroupsWithState**
        - Operate on each group of data and generate at most a single row for each group
    - **flatMapGroupsWithState**
        - Operate on each group of data and generate one or more rows for each group

What to learn next: Arbitrary State Management

- Three class definitions:
    - Input
    - State
    - Output
- A function to update state based on:
    - Key
    - Iterator of values
    - Previous state
- A timeout parameter

What to learn next: Custom State Store

- Out of the box, Apache Spark has only one implementation of state store providers.
- It's **HDFSBackedStateStoreProvider** which stores all of the data in memory, what is a very memory consuming approach.
- To avoid OutOfMemory errors, custom state store providers can be created.
- Github: **Custom State Stores**

What to learn next: Monitoring Streaming Queries

Native Support of Prometheus in Spark 3.0

What to learn next: Spark Streaming on Kubernetes

Spark Documentation

Databricks Tech Talk

Congratulations!

&

All the Best :-)