EDEM
Escuela de Empresarios

Stream Processing

GFT

Roberto López

Curso 2020/2021 - Edición 2

Fecha 15/01/2021

EDEM
Escuela de Empresarios

GFT

Agenda

EDEM
Escuela de Empresarios

GFT

Agenda

1. **Overview**
   - What is Stream Processing?
   - Benefits
   - How Real time processing is done
   - Use Cases
2. **Concepts**
3. **Technologies**
4. **Ejemplo de Arquitectura completa**

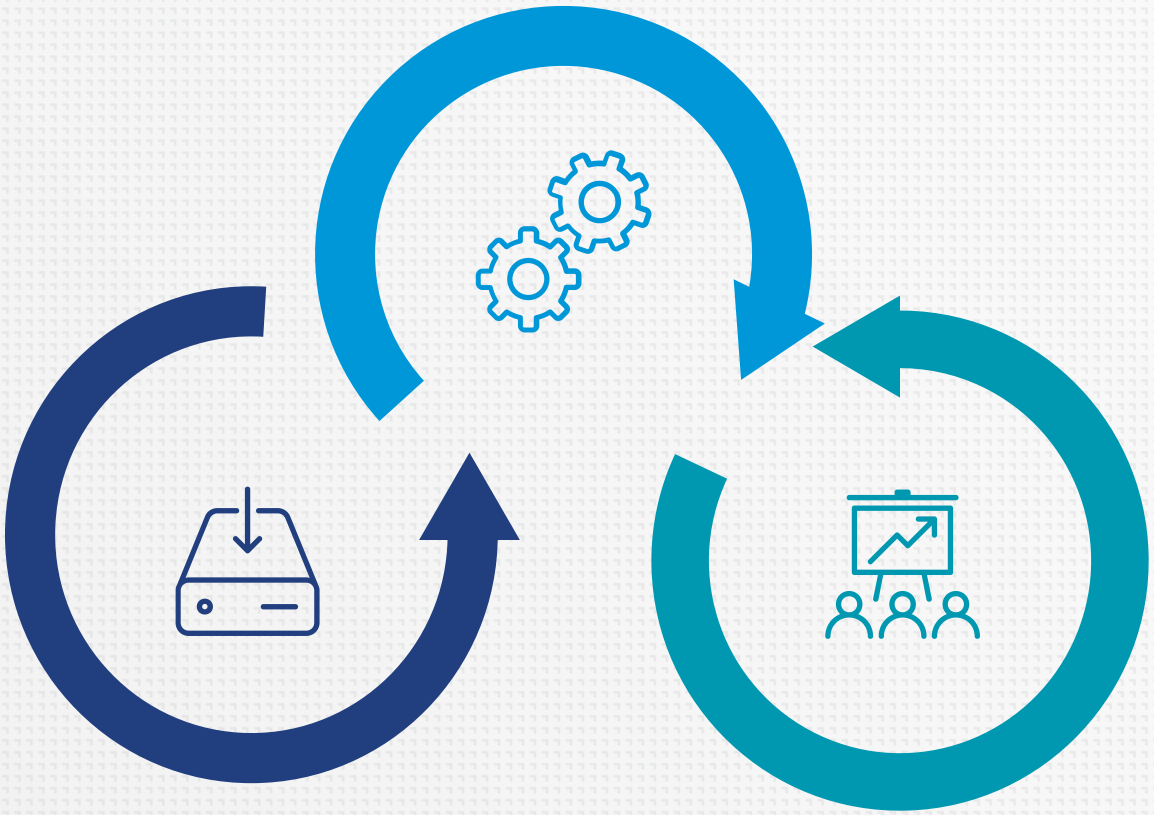
What is Stream processing

Stream processing allows to **ingest**, **process** and **get insights** on **real time** of continuous flows of data.

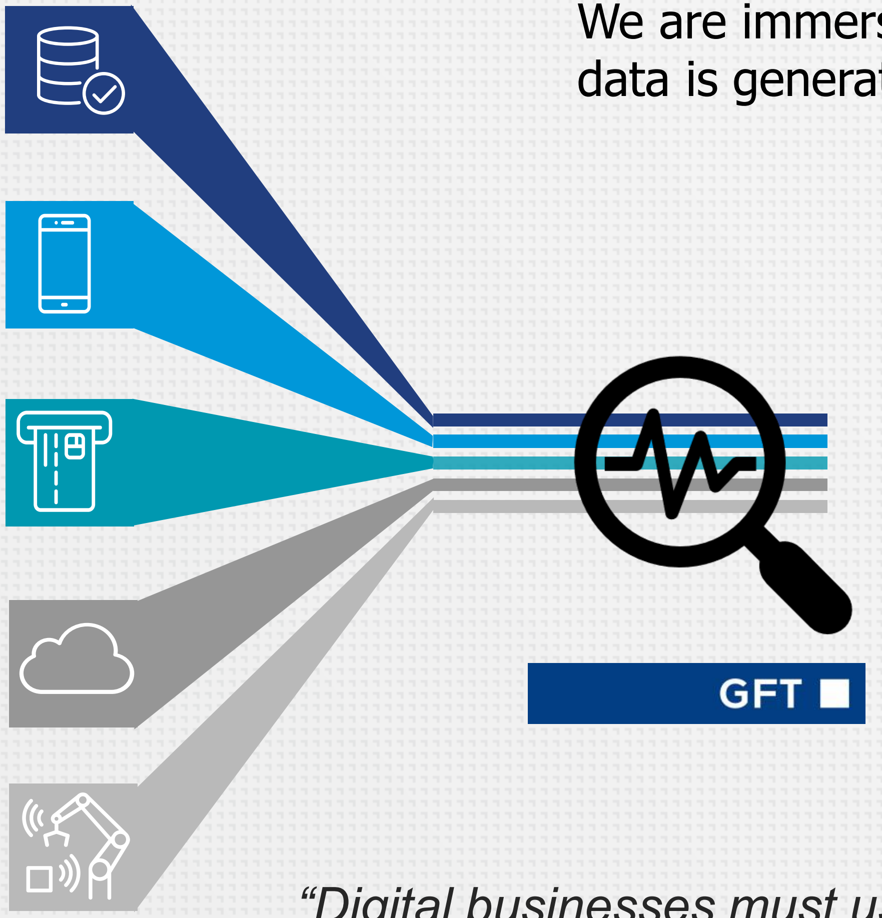
Ingest at the rate the data is generated nowadays

Process the data as it is received

Get insights quickly, increasing significantly the value of the data



Benefits of Stream Processing



We are immersed of **data era**, where the **volume** and **speed** that the data is generated has never seen before.

With **Fast Data** you can:

- **React** to the problem, when it is generated
- Get always **up to date insights**
- **Risk reduction**, by having timely and more accurate data
- **Reduce infrastructure costs**, by reducing the technology stack

"Digital businesses must use stream processing to meet their needs for continuous intelligence and real-time analytics." - **Gartner**

How Real Time processing is done?

- Note this include both two implementations:
  - True streaming
  - Micro-batch streaming

- Low-latency, approximate, and/or speculative results
  - These features are not a requirement for stream processing
  - but characteristics for some implementations

Use cases

BANKING

- **Real time risk management**
- **Real time accounting and reporting**
- Data analytics cross-company
- Smart Transactions categorization
- Ad-hoc financial products
- Real time anti fraud and anti money laundering

INSURANCE

- **Flexible pricing insurance**
- Car damage calculation
- Fraud detection on time

INDUSTRY

- Planning and Control
- Quick response to Stock changes
- Advanced logistics
- **Datalake for machinery data**

OTHERS

- **Realtime mobility planning**
- **Connected Cars**
- **Precision agriculture**
- Reputational Risk management with Social Mining

Agenda

1. **Overview**
2. **Concepts**
   - Delivery Guarantees
   - Time, Windowing and Watermarks
   - Late Elements, Triggers, Accumulation
   - Queryable State
3. **Technologies**
4. **Ejemplo de Arquitectura completa**

Delivery Guarantees

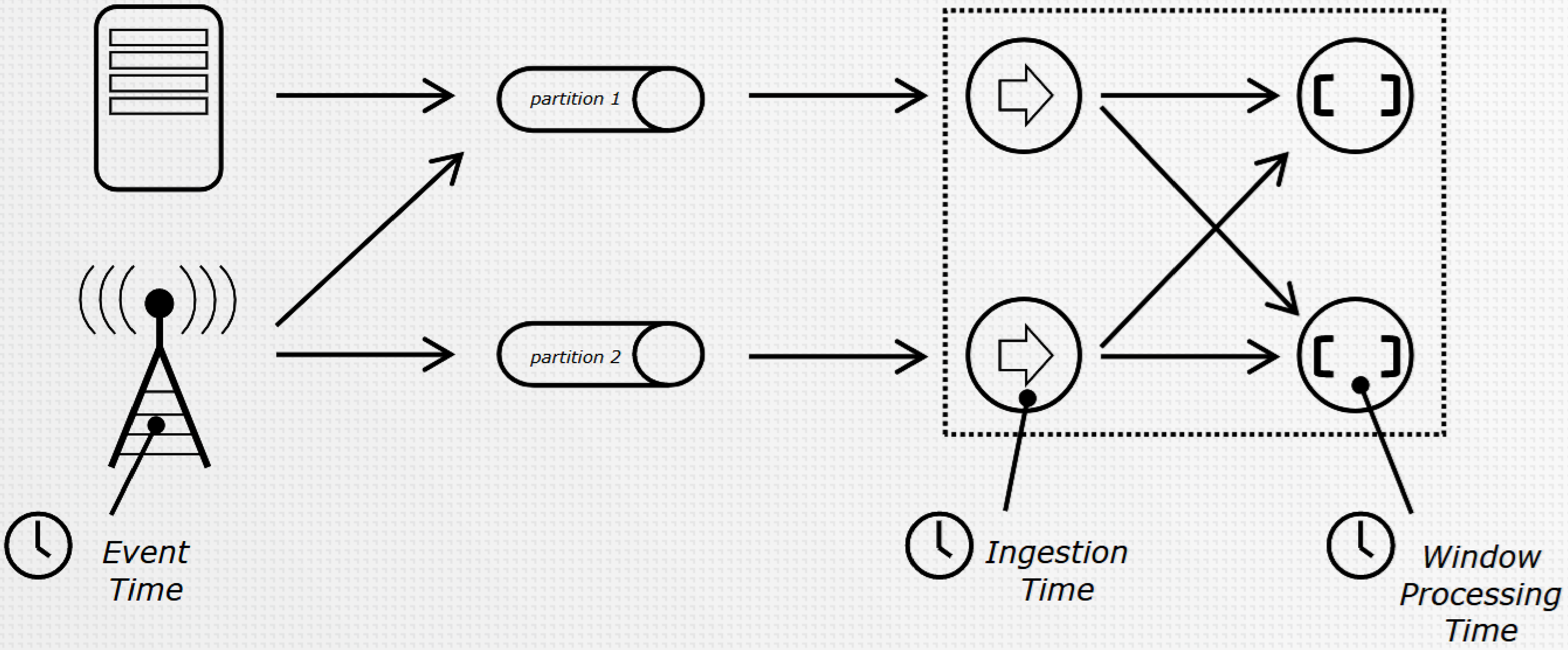
- **At most once - "fire and forget"**
  - **Messages may be lost but are never redelivered**
  - The message is sent, but the sender doesn’t care if it’s received or lost
  - If **data loss is not a concern**, which might be true for monitoring telemetry, for example, then this model imposes no additional overhead to ensure message delivery, such as requiring acknowledgments from consumers
  - It is the easiest and most performant behaviour to support
- **At least once**
  - **Messages are never lost but may be redelivered**
  - Retransmission of a message will occur until an acknowledgment is received
  - Since a delayed acknowledgment from the receiver could be in flight when the sender retransmits the message, the message may be received one or more times
  - This is the most practical model when message loss is not acceptable (e.g., for bank transactions) but **duplication can occur**

Delivery Guarantees

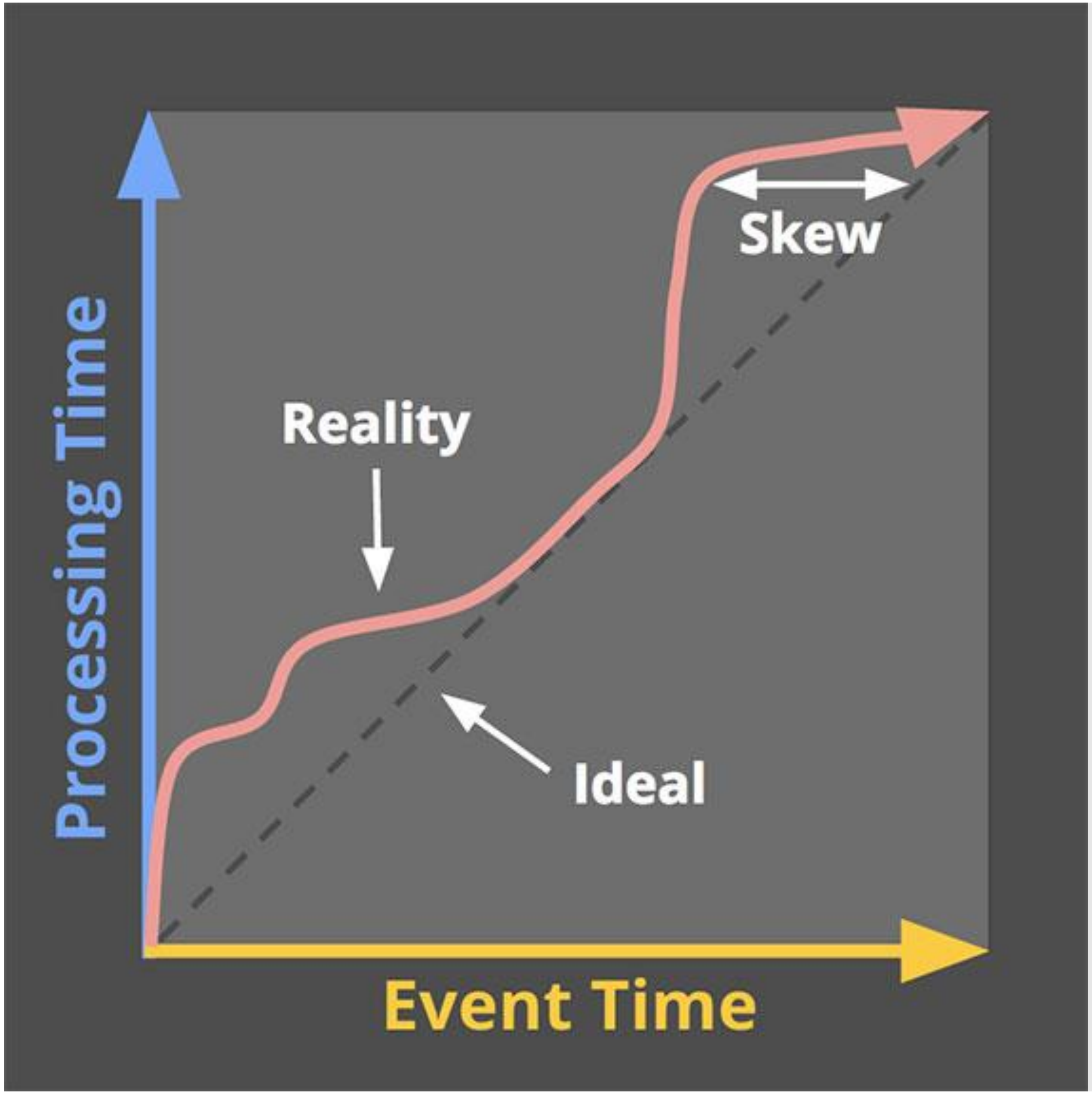
- Exactly once
  - **Messages are never lost and never redelivered**
  - It ensures that a message is received **once and only once, and is never lost and never repeated**
  - This is the ideal scenario, because it is the easiest to reason about when considering the evolution of system state
  - It is also impossible to implement in the general case, but it can be successfully implemented for specific cases (at least to a high percentage of reliability)

  - Kafka provides this feature since v0.11.0.0, June 2017
    - Durability guarantees for publishing a message – **commit messages**
    - Guarantees when consuming a message – **commit offsets**
    - Other systems provide "exactly once" semantics, but they do not consider consumers or producers can fail!

Time



Source: https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/event_time.html



Skew

- **Skew is not only non-zero but often large**
- Causes:
  - **Shared resource limitations**, such as network congestion, network partitions, or shared CPU in a non-dedicated environment.
  - **Software causes**, such as distributed system logic, contention, etc.
  - **Features of the data** themselves, including key distribution, variance in throughput, or variance in disorder

E.g. a plane full of people taking their phones out of airplane mode after having used them offline for the entire flight

Event Time

- **Event time is the time that each individual event occurred on its producing device.**
- **Typically embedded** within the records at source system side. So it can be extracted from the record.
- **Deterministic**. Event time gives correct results even on out-of-order events, late events, or on replays of data from backups or persistent logs. The progress of time depends on the data, not on any wall clocks.
- Event time processing often incurs a **certain latency**, due to its nature of waiting a certain time for late events and out-of-order events.

Processing Time

- **Processing time refers to the system time of the machine that is executing the respective operation.**

- All time-based operations (like time windows) will use the system clock of the machines that run the respective operator.

- It is the **simplest notion of time and requires no coordination** between streams and machines. So it provides the **best performance** and the **lowest latency**.

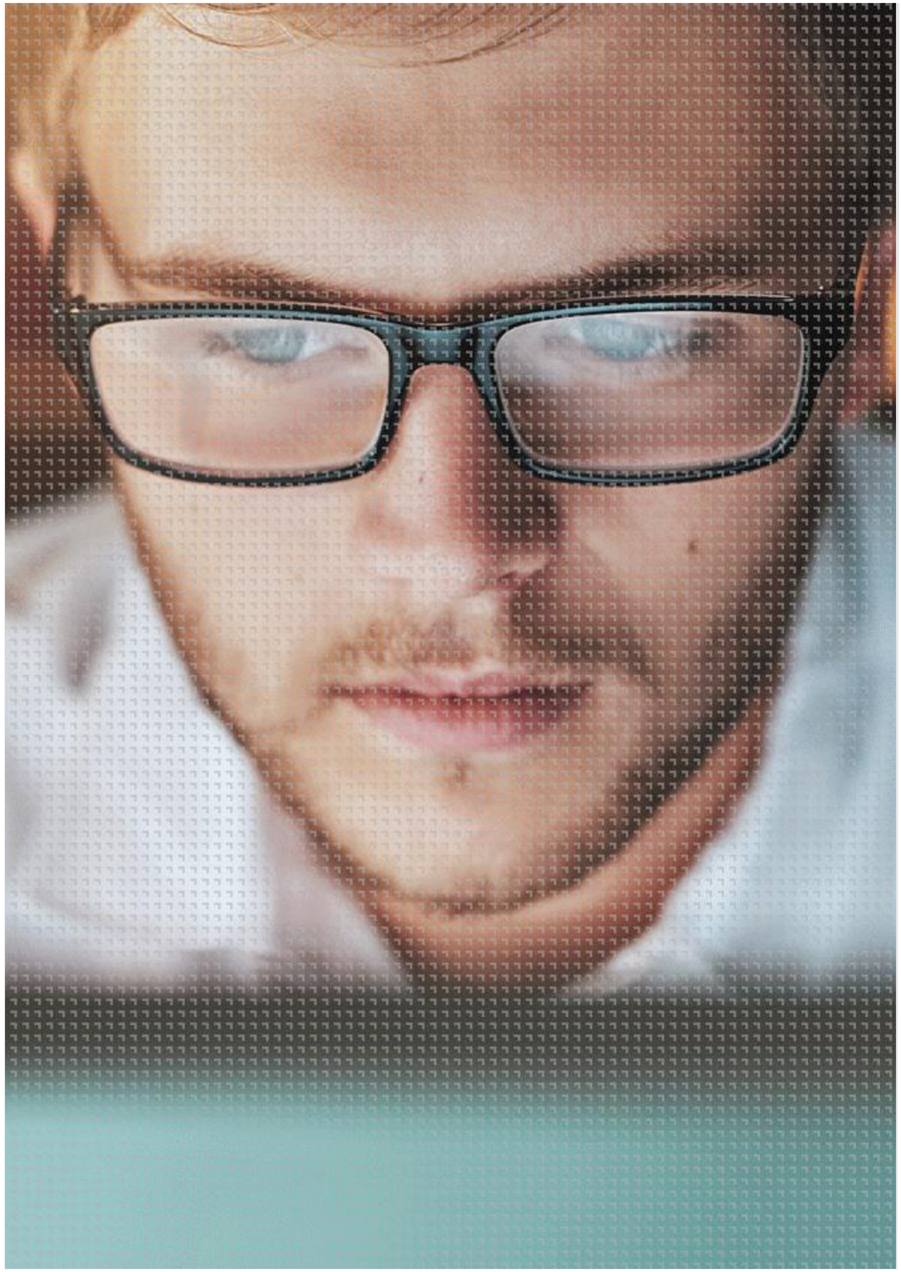
- In distributed and asynchronous environments it **does not provide determinism**.
  - It is susceptible to the speed at which records arrive in the system (for example from the message queue), and to the speed at which the records flow between operators inside the system.

Ingestion Time

- **Ingestion time is the time that events enter the Stream Processing system.**

- At the source operator **each record gets the source's current time as a timestamp**.

- Ingestion time sits conceptually in between event time and processing time.
  - Compared to **processing time**, it is **more expensive**, but gives **more predictable** results.
    - Ingestion time uses stable timestamps (assigned once at the source) so different window operations over the records will refer to the same timestamp, whereas in processing time each window operator may assign the record to a different window (based on the local system clock and any transport delay).
  - Compared to **event time**, it is **less expensive**, but give **less deterministic** results.
    - Ingestion time determines the order of the events and no chance to late date, whereas in Event Time events may arrive out-of-order.



Exercise 1 – Windows type

Imagine that you work on a e-commerce Company. And you have a web, where customers buy your products. Yo need to do the following:

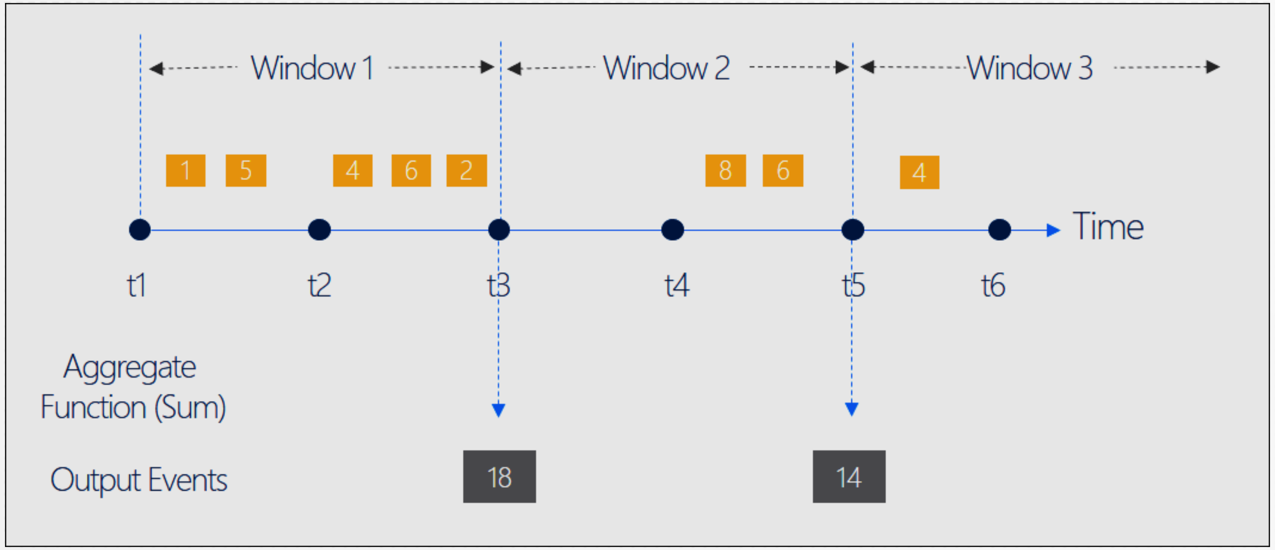
- Every 4 hours you want to pack together all buyed products in order to be sent by the main logistic warehouse to the distribution channel.
- In order to identify your advertising campaigns you want to see at any time last 5 minutes number of buyed products for each type.

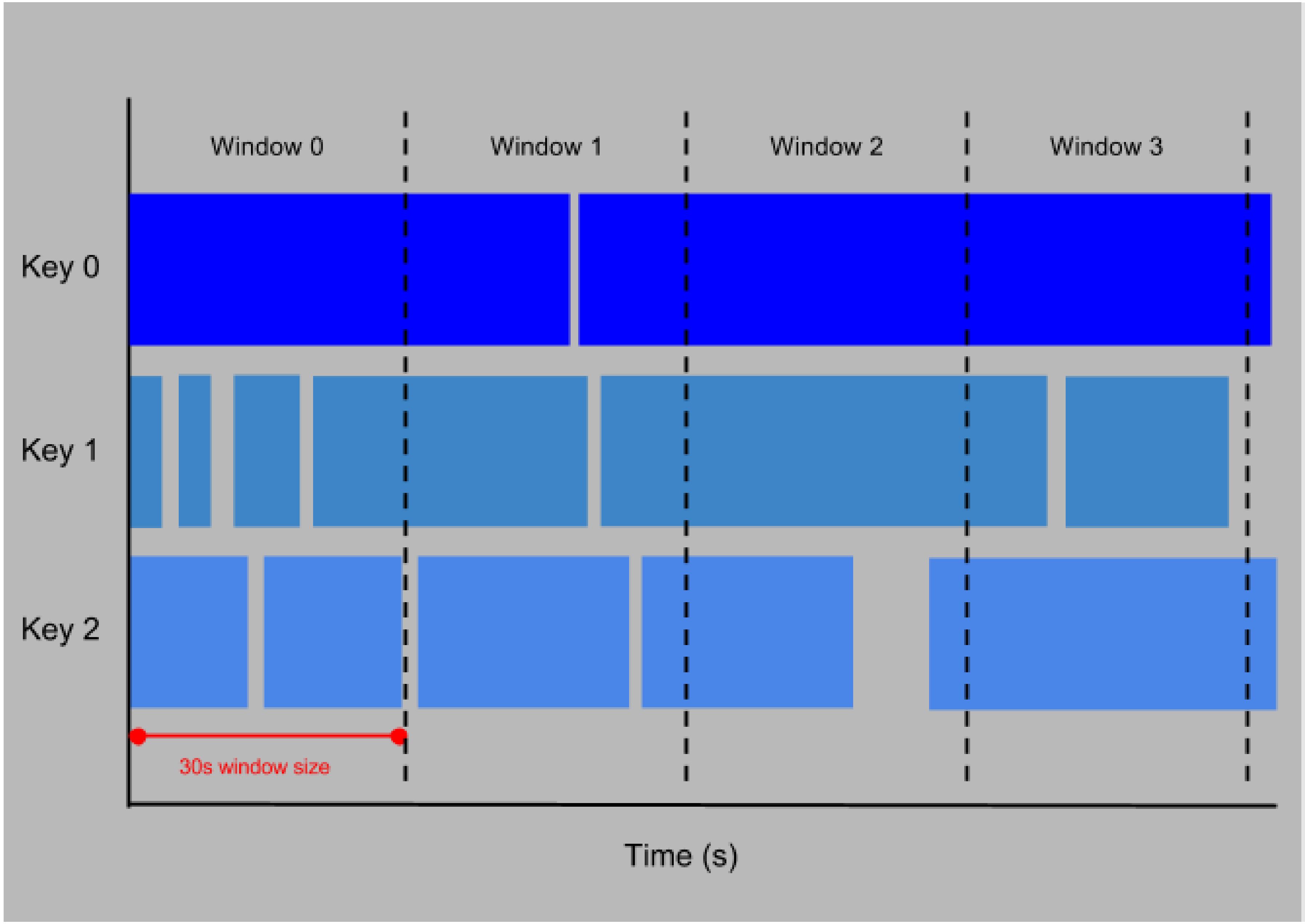
How would you implement this?

Windowing

- Stream Processing makes real-time computations possible. However, **computation requiring aggregations need to break the stream into bounded groups of events**
  - Otherwise processing the whole unbounded data stream would be required!
- Windowing is the ability to perform some **set-based computation** (aggregation) or other operations **over subsets of events that fall within some period of time.**

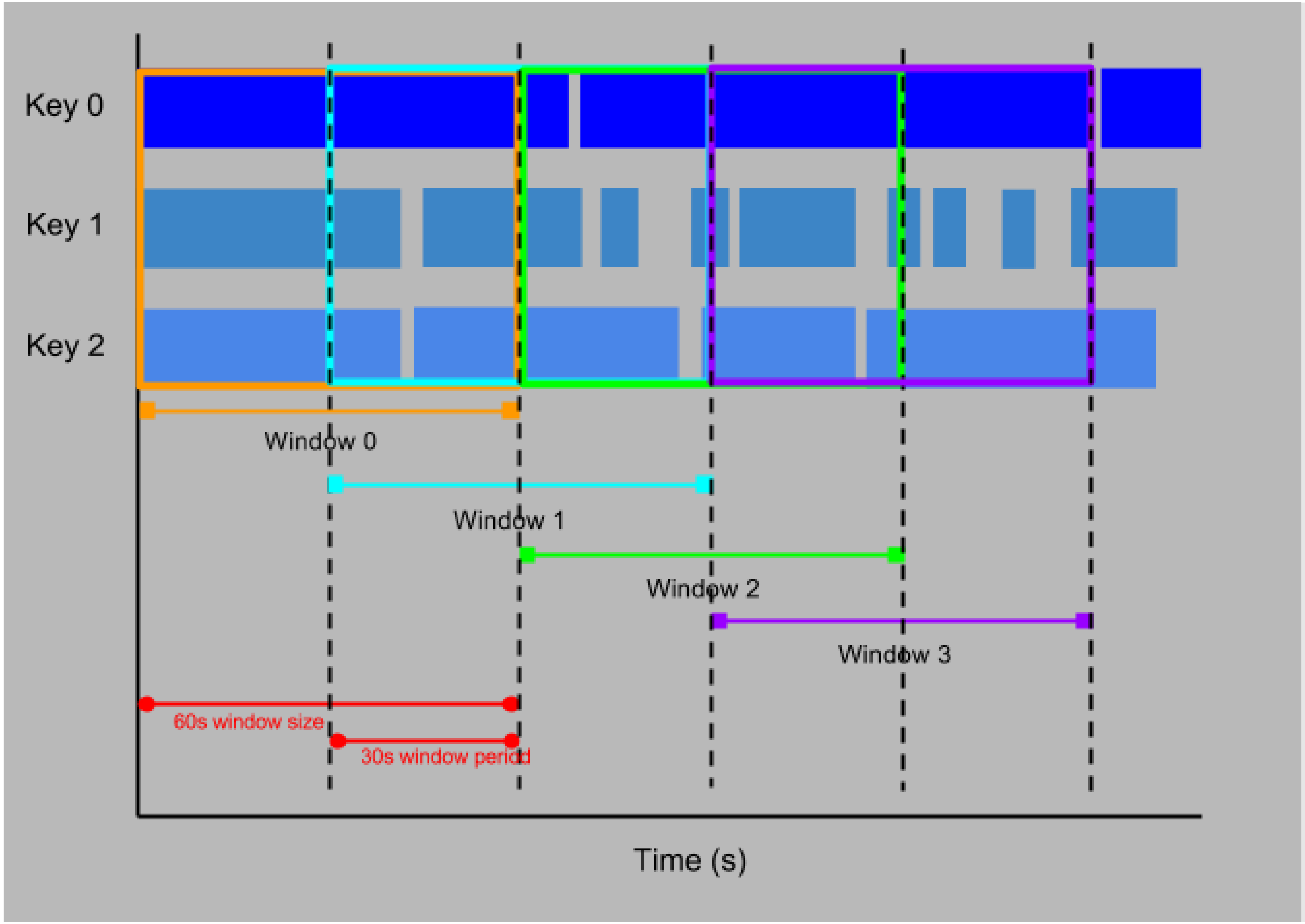
- **Window types:**
  - Fixed
  - Sliding
  - Session





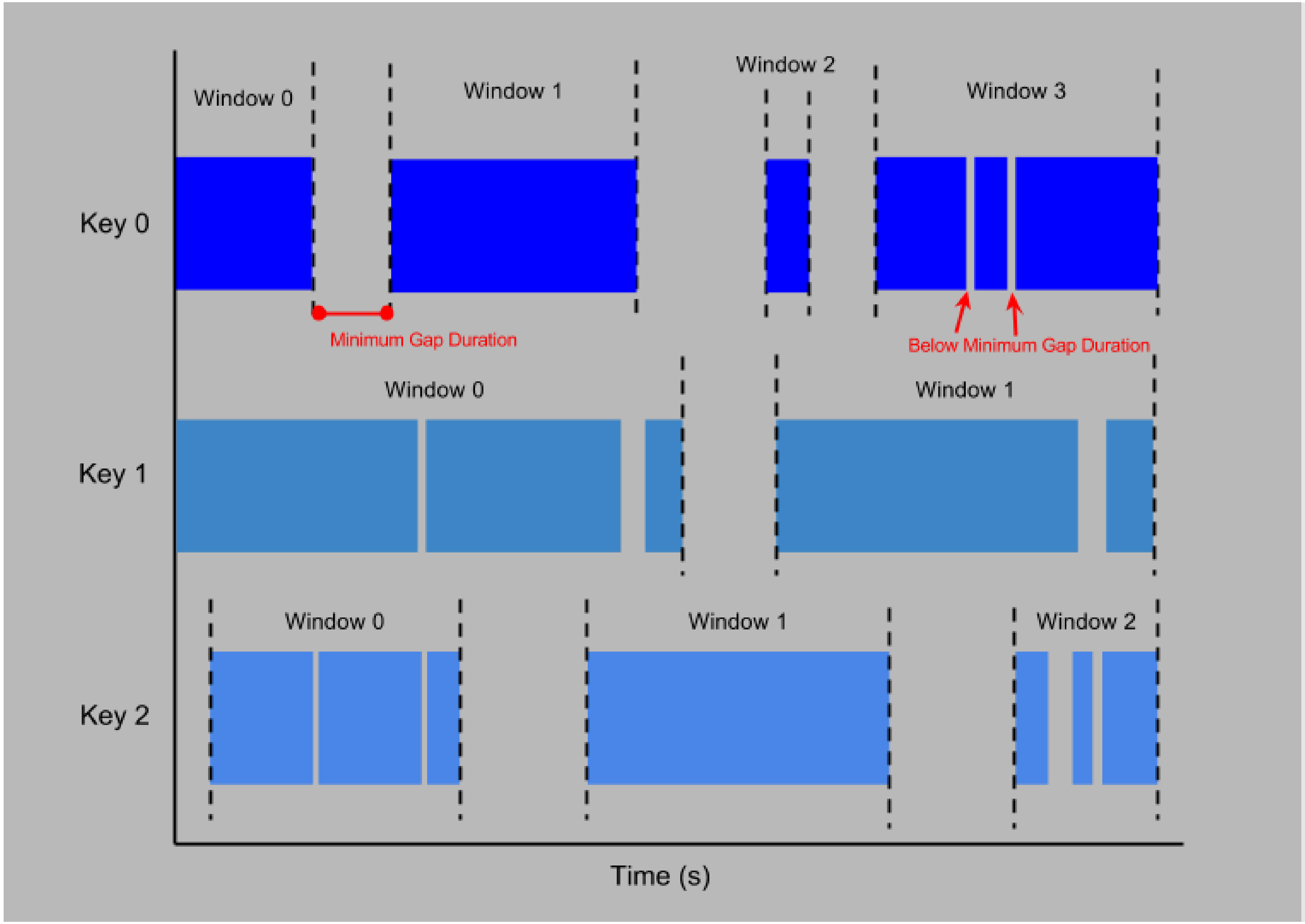
Tumbling Window

- **Periodic and Non-overlapping**
- Each element belongs to exactly one window



Sliding Window

- **Periodic and Overlapping**
- Each element can belong to multiple windows
- Window defined by Period and Size



Session Window

- **Dynamic and Non-overlapping**
- Just for key-value streams
- Windows have various sizes and defined basing on data
  - Typically windows around areas of concentration – minimum gap duration
- Useful for data that is irregularly distributed with respect to time

Windowing Challenges

- Powerful semantics rarely come for free and event time windows are no exception

- **Buffering**
  - Extended window lifetimes require buffering of data (state) → **State**
  - How do we limit size of this state? → **Watermarks**

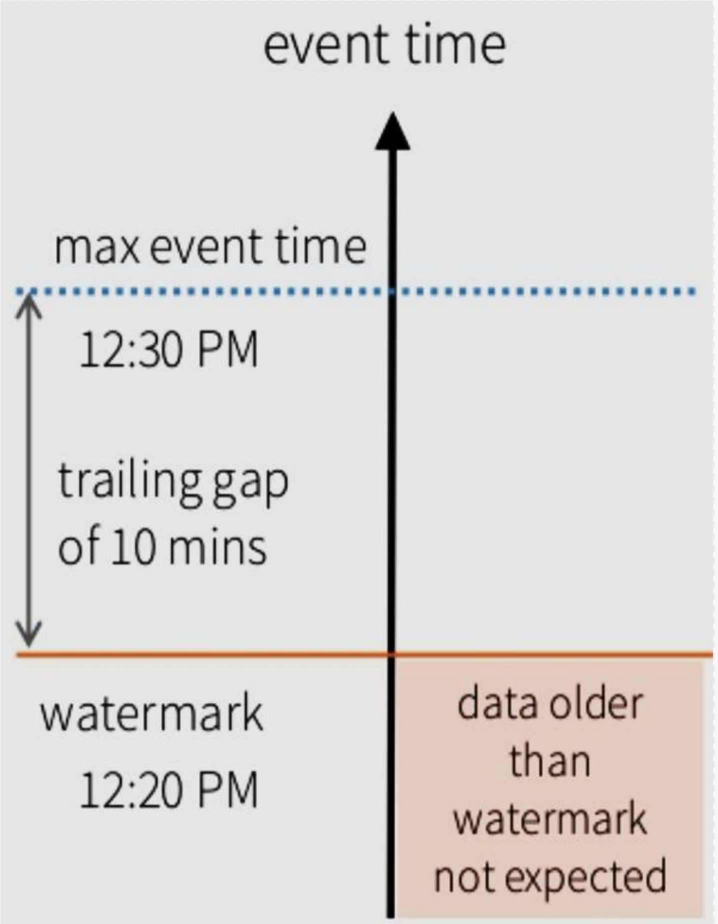
- **Completeness** How do we know when the results for the window are ready to materialize?
  - Heuristic → **Allowed lateness**
  - Additionally, give responsibility to the pipeline builder to decide:
    1. When they want results for windows to be materialized → **Trigger**
    2. How those results should be refined over time (multiple firings) → **Accumulation mode**

Watermarks

- **Watermark is essentially a timestamp based on event-time.**
  - It a moving threshold that helps to understand how far we are from event-time.
  - When the results for the window are ready to materialize?
  - **When all events for a window have already arrived?**

- **Watermarking to Limit State while Handling Late Data**
  - Drop “old” state store

- A mechanism for “**allowed lateness**”
- Applies for **windows defined over event time**
  - Otherwise (processing, ingestion time) lateness does not make sense

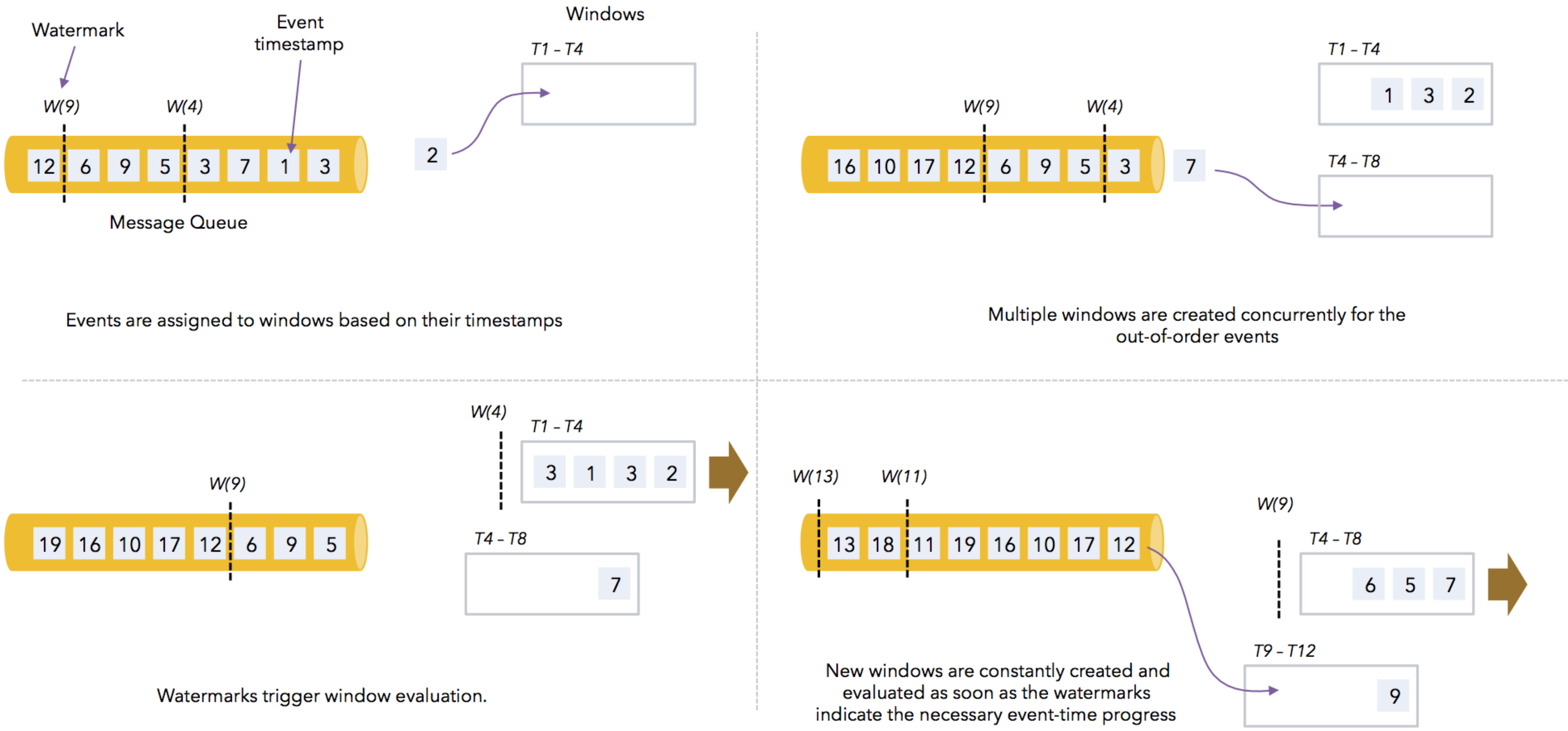


Source: http://vishnuviswanath.com/flink_eventtime.html

Watermarks

```
eventsDF = …

windowedCountsDF = eventsDF
    .withWatermark("eventTime", "10 minutes")
    .groupBy("deviceId", window("eventTime", "10 min", "5 min"))
    .count()
```

- **A watermark with a value of time X makes the statement: “all input data with event times less than X have been observed.”**
- As events arrive, the watermark is moved behind them
- Depends on the data source → set on the Reader



Watermark Caveats

- **Allowed lateness to avoid missing late events.**
- **Too less**
  - Increases late data
- **Too much**
  - Increases latency and buffering requirements
  - Mitigation with **speculative results** → early firings → multiple results for each window!

Source: https://data-artisans.com/blog/how-apache-flink-enables-new-streaming-applications-part-1

Late Events

- **Certain elements may violate the watermark condition**
  - After the Watermark(t) has occurred, more elements with timestamp t' <= t will occur.

- **Strategies**

  - Late elements are **dropped** when the watermark is past the end of the window

  - Maximum **allowed lateness** for window operators.
    - Elements that arrive after the watermark has passed the end of the window but before it passes the end of the window plus the allowed lateness, are still added to the window
    - A late but not dropped element may cause the **window to fire again**
    - Keep the state of windows until their allowed lateness expires

Triggers

- **Determines when a window is ready to be processed by the window function**

- **Trigger types**
  - Time-based
    - Event time – eg. after watermark
    - Processing time
  - Data-driven triggers – eg. Certain number of data elements
  - Composite triggers
- **Strategies**
  - Fire: trigger the computation
  - Purge: clear the elements in the window
  - Fire & Purge

Source: https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/windows.html#triggers

Accumulation

- For some configurations window results can be computed multiple times (multiple triggers)
  - Early firings → speculative results
  - On-time
  - Late firings

- **Accumulation determines how results emitted from a window relate to previously emitted results for the same window**

- How to handle the new values?
  - **Discard** old value, use only the new window result
  - **Accumulate** new value with the old value (e.g. add results to a counter)
  - **Retract & accumulate**: specify the old value to retract, replace with new value

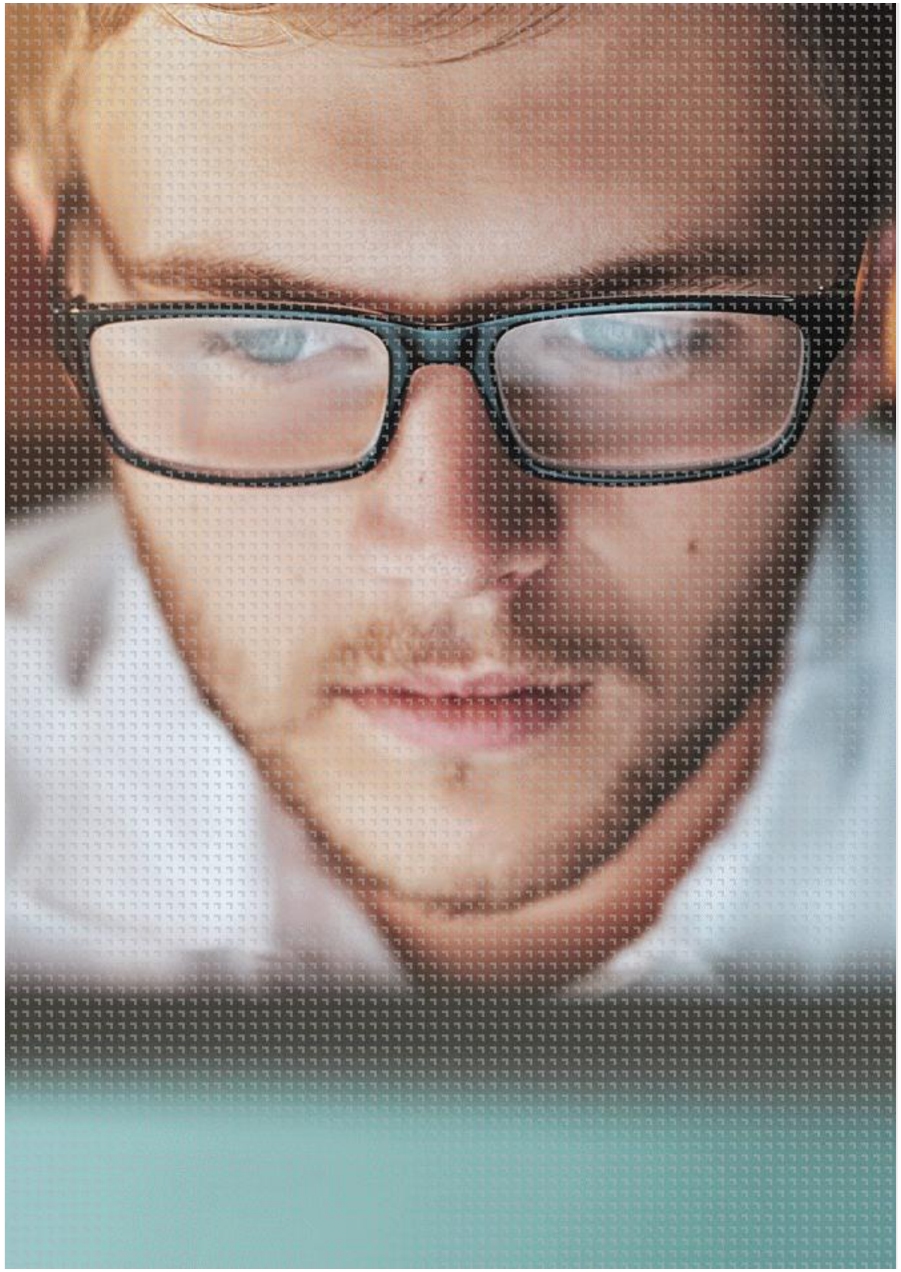
Accumulation

- Ex: trigger fires every time three elements arrive
- **Discarding Mode**
- **Accumulating Mode**
- **Retract & accumulate**

First trigger firing: [A → 5, B → 8]
Second trigger firing: [C → 15, B → 10]
Third trigger firing: [A → 2]

First trigger firing: [A → 5, B → 8]
Second trigger firing: [A → 5, B → 18, C → 15]
Third trigger firing: [A → 7, B → 18, C → 15]

First trigger firing: [A → 5, B → 8]
Second trigger firing: [A → -5, B → -8, A → 5, B → 18, C → 15]
Third trigger firing: [A → -5, B → -18, C → -15, A → 7, B → 18, C → 15]



Exercise 2 - Tumbling window

Purpose of this exercise is to run an Spark Streaming program to see how windowing works.

- Go to Exercises doc -> http://bit.ly/EDEMAPExercises
- Perform Exercise 2 on Streaming Section

EDEM
Escuela de Empresarios

GFT

Agenda

1. **Overview**
2. **Concepts**
3. **Technologies**
   - Reference Architecture
   - Spark Streaming
4. **Ejemplo de Arquitectura completa**

There is a big ecosystem

Mature Frameworks

Spark Streaming

New players

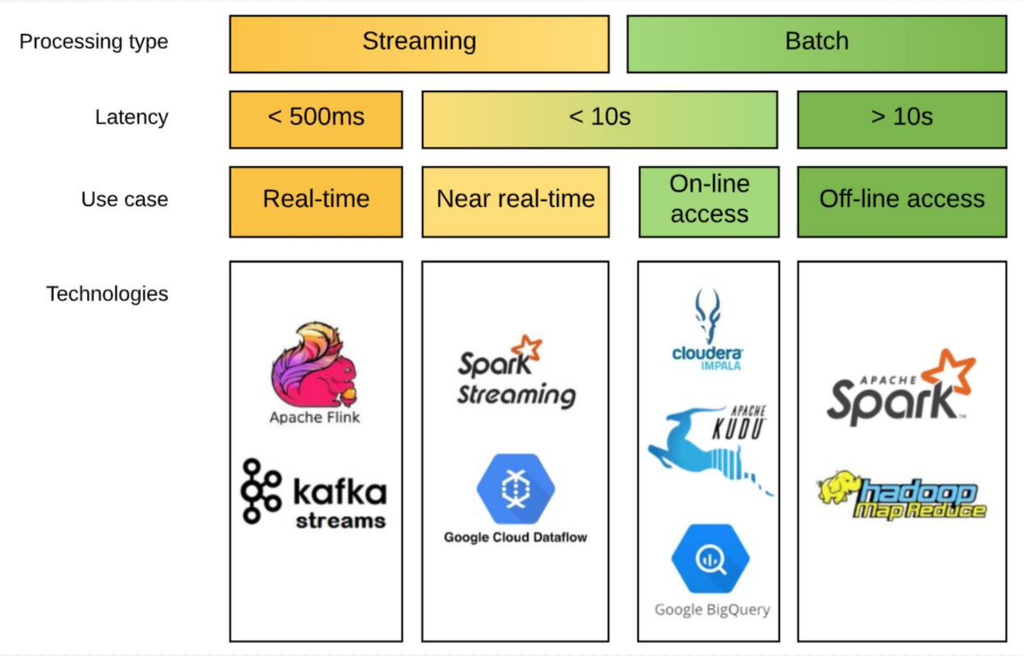
hazelcast JET

PULSAR

redis

What technology should we use?

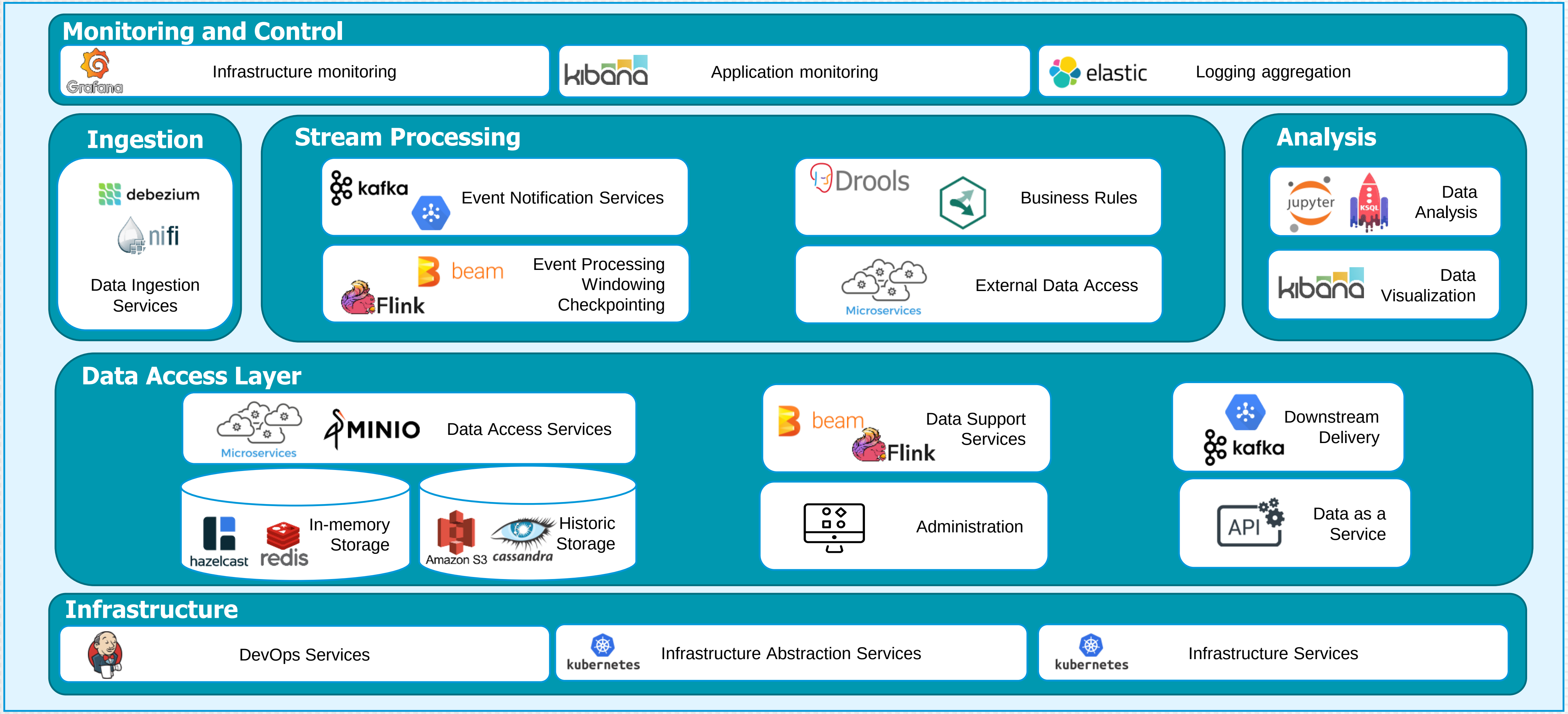
- If you are already **using the Spark** ecosystem, go ahead with **Spark Streaming**.
- If you are just **starting from scratch**, it is worth starting with **Beam** if you want multiplatform or **Flink** if you want to run on Kubernetes.
- If you already **have Kafka in your architecture** and you just require some simple ETL for your stream, **Kafka Streams** is a good selection.
- **Akka Streams and Gearpump** are more suitable for building general **microservices** over streaming data.



- If you are already using a **YARN** cluster, you may benefit from tools that can already run on it.
- Nowadays running over **Kubernetes** is something that almost all support and is key.

Source: https://agema.gft.com/confluence/display/DATA/Real-time+and+Streaming+Assessment#Real-timeandStreamingAssessment-Conclusions
https://confluence.gft.com/display/DATA/2019/01/09/How+to+choose+a+Big+Data+processing+engine+having+into+account+latency+requirements

Streaming Reference Architecture



Let’s Focus

Spark Streaming

What is Apache Spark

- Apache Spark is a distributed computing platform designed to be **fast** and **general purpose**
- Spark extends the **map-reduce** model to efficiently support more types of **in-memory** computations, including **interactive queries** and **stream processing**

What is Apache Spark

- From the **general purpose** Spark is designed to cover a wide range of workloads that previously required separate distributed systems:
  - Batch applications
  - Iterative algorithms
  - Streaming

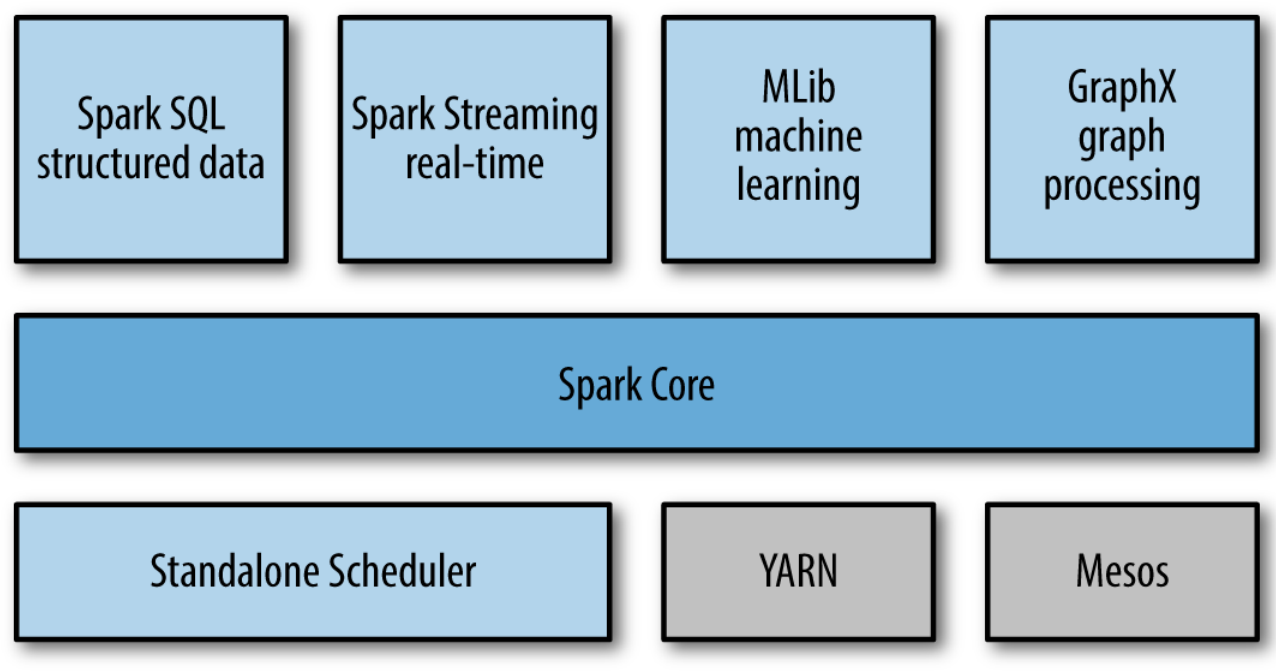
What is Apache Spark

- Spark is designed to be highly accessible, **offering simple APIs**
  - Python, Java, Scala, R, SQL

- Spark can **run in Hadoop clusters** and access any Hadoop data source

Unified Stack

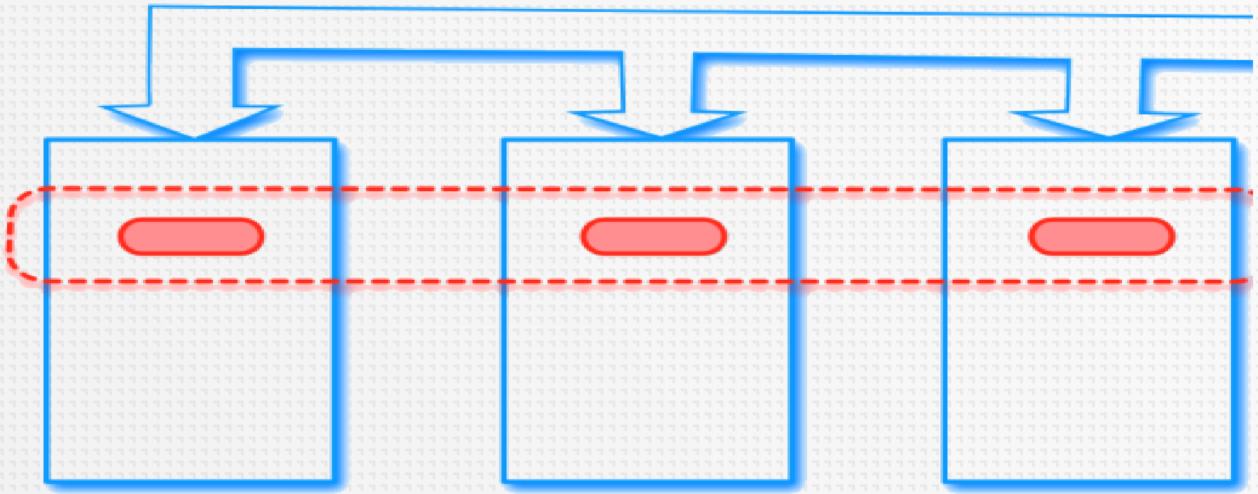
- **Core**:
  - Scheduling, distributing and monitoring computational tasks across nodes
- **High-level components:**
  - Specialized for various applications



- **SQL:** Allows querying data via SQL, Hive and Data Frames
- **Stream processing:** Processing of data live streams (events, logs, …)
- **Machine Learning:** Parallel computation of machine learning algorithms: classification, regression, clustering, etc.

RDD Definition

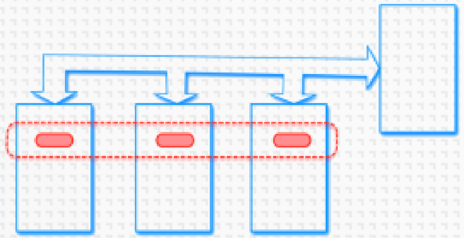
- An RDD in Spark is simply an **immutable distributed collection** of objects
- Each RDD is split into **multiple partitions**, which may be computed on different **nodes** of the cluster.
- RDDs can contain any type of objects:
  - Scala
  - Python
  - Java
  - R
  - User defined classes



EDEM
Escuela de Empresarios

GFT

RDD Creation



- RDDs can be created in **three ways**:

1. By **loading** from an external dataset in HDFS, S3, local file, etc…

```
lines = sc.textFile("README.md")
```

2. **Parallelizing** a collection of objects already existing in the memory of the driver / workers

```
input = sc.parallelize([1, 2, 3, 4])
```

3. As a result of a transformation **from another RDD**

```
rdd2 = rdd1.filter(col("balance") > 1000)
```

RDD Operations

- Once created, RDDs offer two types of operations:
  - ***Transformations*** construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate.

```
rdd2 = rdd1.filter(col("balance") > 1000)
```

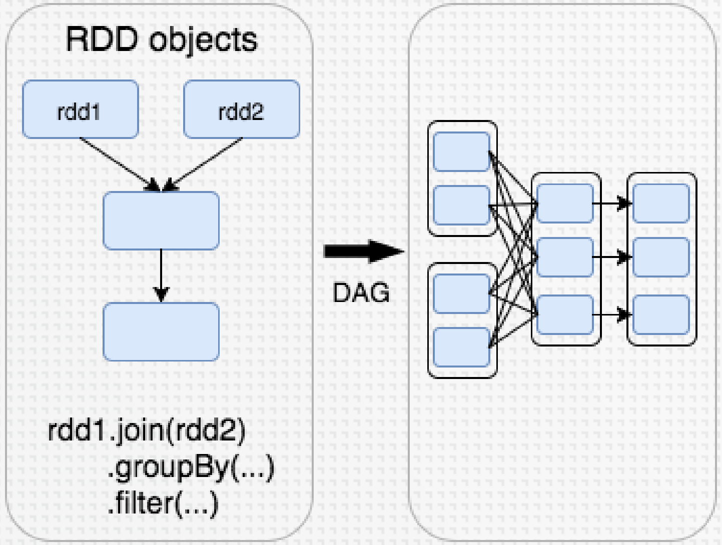
  - ***Actions*** compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g. HDFS).

```
lines.first()
```

RDD Lazy operation principle

- Spark **computes RDDs in a lazy way**. It sees the **whole chain** of transformations and computes just the data needed for its result the first time they are used in an action.
- Spark's **RDDs** are by **default recomputed** each time you run an action on them.
- If you would like to reuse an RDD in **multiple actions**, you can ask Spark to ***persist*** it using `RDD.persist()`.
- Example:

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
lineLengths.persist()  // If "lineLenghts" is to be reused
totalLength = lineLengths.reduce(lambda a, b : a + b)
```



Spark Streaming

Spark Streaming

Discretized Streams

- Based on RDDs
- Micro-batching
- Non-structured

- **Lack of features other Stream Processing systems have!**
  - Event-time, watermarking, late data, …

Structured Streaming

- Spark >= v2.0
- Infinite Dataframes/Datasets
- Structured → Catalyst optimizer
- Repeated queries → Stateful
- Watermarking
- Output models: complete, append, update

Creating an Streaming Context

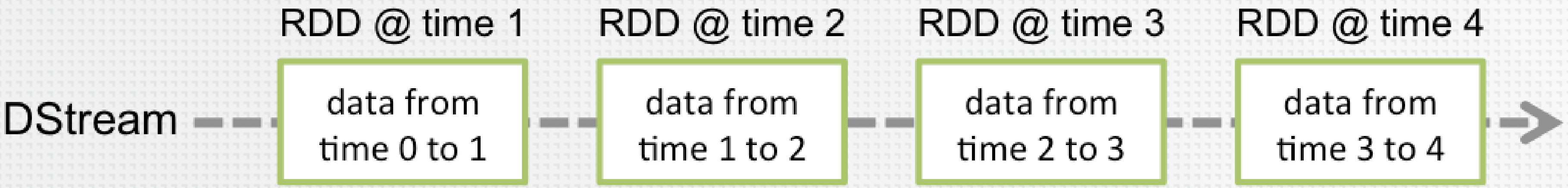
```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext(master, appName)
ssc = StreamingContext(sc, 1)
```

- The **appName** parameter is a name for your application to show on the cluster UI.
- **master** is a Spark, Mesos or YARN cluster URL, or a special “local[*]” string to run in local mode.

Source: https://spark.apache.org/docs/2.4.7/streaming-programming-guide.html

Spark Streaming DStream

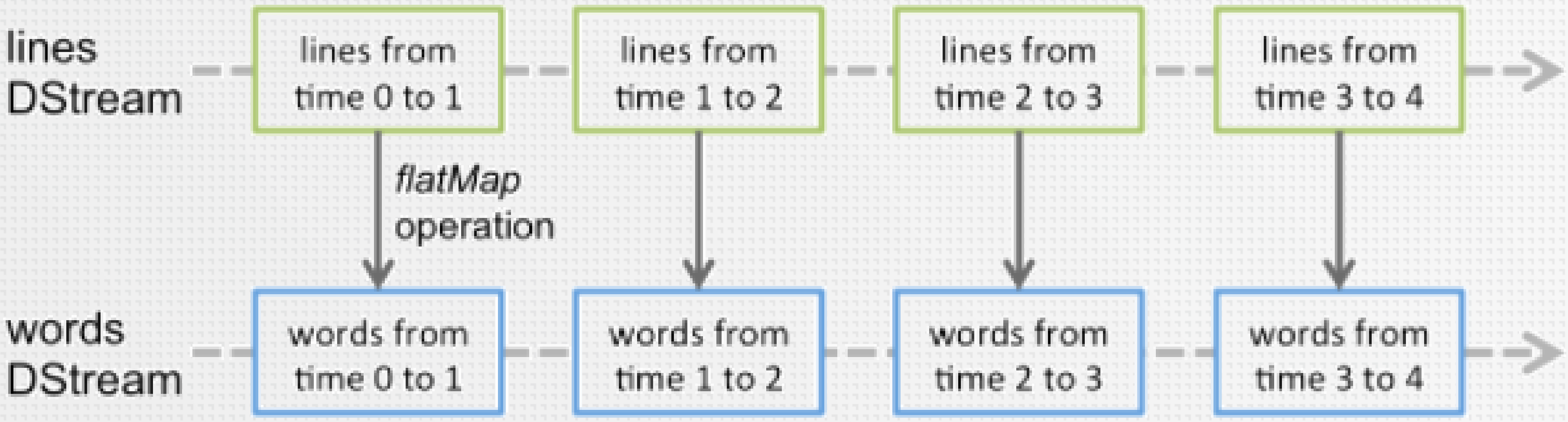
- Continuous series of RDDs → **microbatching**
  - The **batch interval** must be set based on the latency requirements and available cluster resources



Steps after defining a context

Spark Streaming

- Define the input sources by creating input **DStreams**.
- Define the streaming computations by applying **transformation** and **output** operations to DStreams.
- Start receiving data and processing it using **streamingContext.start()**.
- Wait for the processing to be stopped (manually or due to any error) using **streamingContext.awaitTermination()**.
- The processing can be manually stopped using **streamingContext.stop()**.



Things to consider

- Once a context has been **started**, **no new** streaming **computations** can be set up or added to it.
- Once a context has been stopped, it cannot be restarted.
- Only **one StreamingContext** can be active in a **JVM** at the same time.
- stop() on StreamingContext also stops the SparkContext. To stop only the StreamingContext, set the optional parameter of stop() called stopSparkContext to false.
- A SparkContext can be re-used to create multiple StreamingContexts, as long as the previous StreamingContext is stopped (without stopping the SparkContext) before the next StreamingContext is created.

Input Sources

Spark Streaming provides two categories of built-in streaming source:

- ***Basic sources***: Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
- ***Advanced sources***: Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies.

Points to remark

- When running a Spark Streaming program locally, do **not use "local" or "local[1]"** as the master URL. Either of these means that only one thread will be used for running tasks locally. Locally, always **use "local[n]"** as the master URL, where n > number of receivers to run
- Extending the logic to running on a cluster, the number of cores allocated to the Spark Streaming application must be more than the number of receivers.

Basic Sources

- **Socket** Input

```scala
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
```

- For reading **Files**:
    - Data can be read from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.).
    - File streams do not require running a receiver so there is no need to allocate any cores for receiving file data.

```scala
streamingContext.textFileStream(dataDirectory)
```

Advanced Sources

Spark Streaming

- This category of sources require interfacing with external non-Spark libraries, some of them with complex dependencies:
  - Kafka
  - Flume
  - Kinesis

```
stream = KafkaUtils.createDirectStream(ssc, topics, kafkaParams)
```

Transformation on DStreams

Spark Streaming

map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.	rdd.map(lambda x: (x,1))
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.	lines.flatMap(lambda line: line.split(" "))
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.	df.filter(col("balance") > 1000)
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.	df.repartition(10)
union(*otherStream*)	Return a new DStream that contains the union of the elements in two DStreams with same Structure.	unionDF = df.union(df2)
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.	rdd.count()

Transformation on DStreams

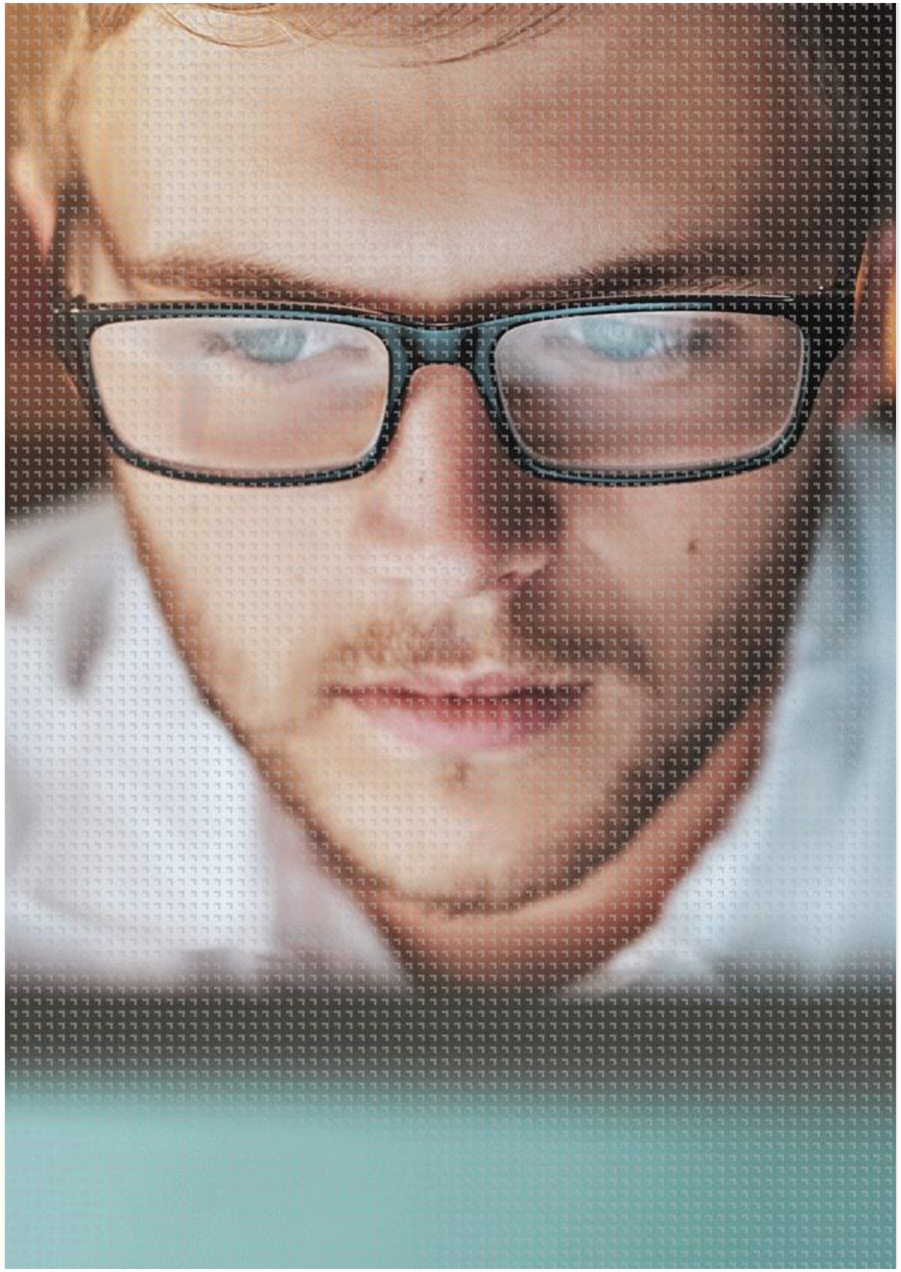
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one).	cSum = x.reduce(lambda accum, n: accum + n)
countByValue()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.	countValue=rdd.countByValue()
reduceByKey(*func*, [*numTasks*])	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.	rdd2=rdd.reduceByKey(lambda accum,b: accum+b)
join(*otherStream*, [*numTasks*])	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.	join=rdd.join(rdd2)

Transformation on DStreams

Spark Streaming

cogroup(*otherStream*, [*numTasks*])	When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.	x = sc.parallelize([("a", 1), ("b", 4)]) y = sc.parallelize([("a", 2)]) **x.cogroup(y)** [('a', ([1], [2])), ('b', ([4], []))]

transform(*func*)	Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.	spamInfoRDD = sc.pickleFile(...)
		# RDD containing spam information
		
		# join data stream with spam information to do data cleaning
		cleanedDStream = **wordCounts.transform**(lambda rdd: rdd.join(spamInfoRDD).filter(...))



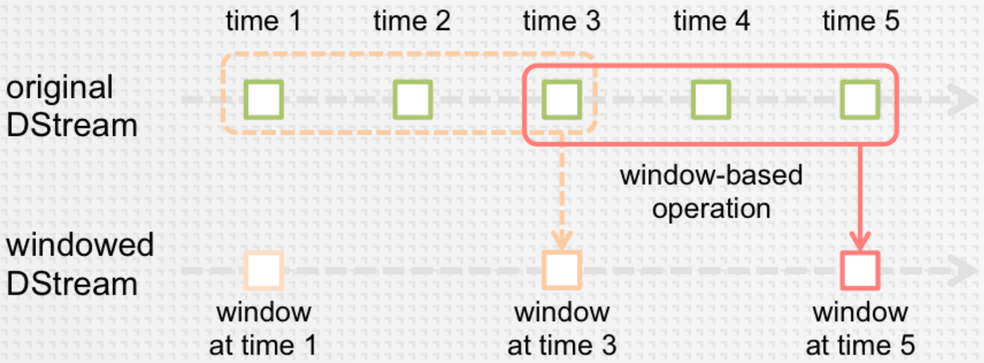
Exercise 3 - Windowing exercise Spark

Purpose of this exercise is to program a Spark Streaming program to see how it is implemented.

Exercise to be done is Exercise 3 on git → https://github.com/rlopezherrero/GFT-EDEM-MasterData/tree/master/AlmacenamientoProcesamiento/streaming

Spark Streaming DStream

- **Window Operations**
  - **window length** - the duration of the window (3 in the figure)
  - **sliding interval** - the interval/period at which the window operation is performed (2 in the figure)



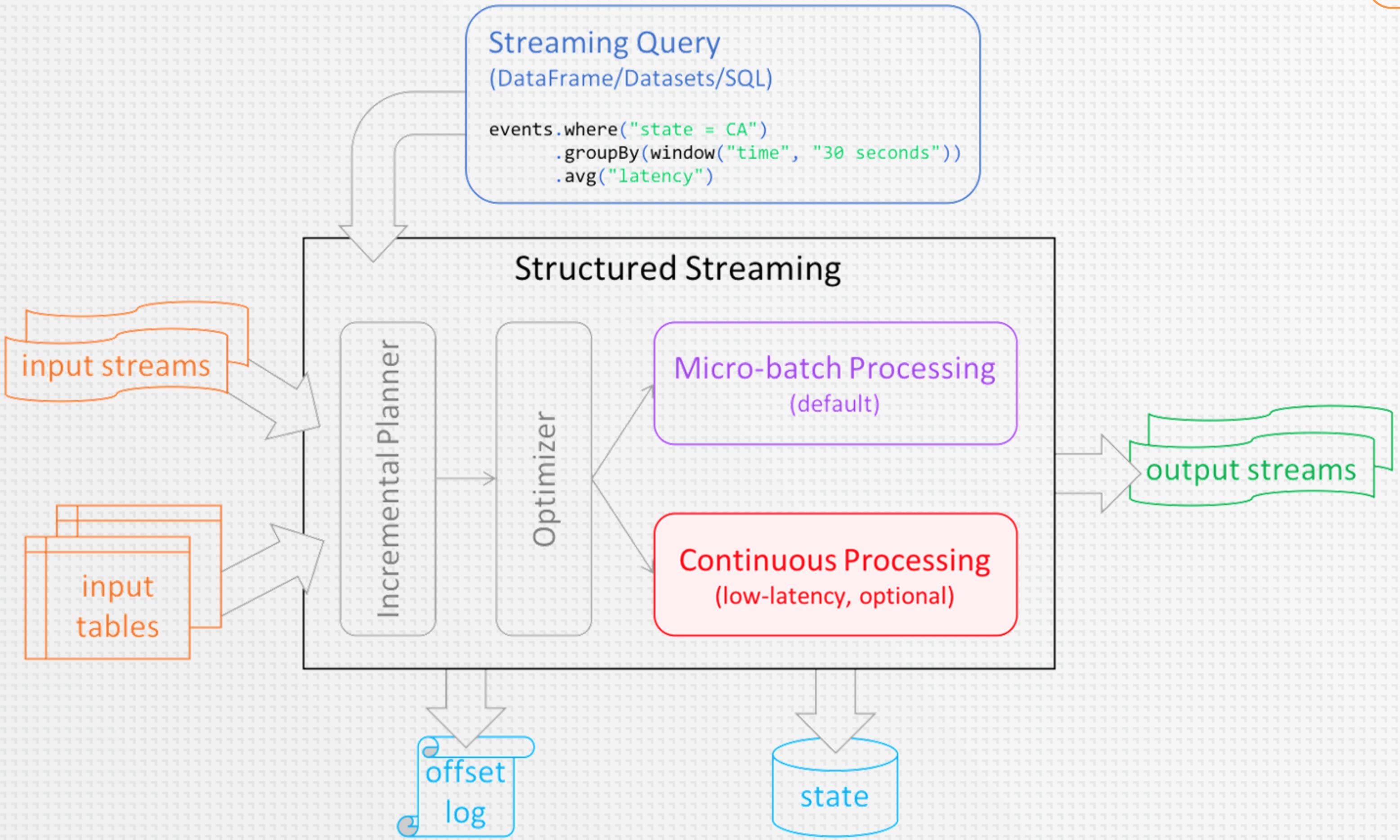
Window Operations on DStreams

window(*windowLength*, *slideInterval*)	Return a new DStream which is computed based on windowed batches of the source DStream.	stream.window(60,5)
countByWindow(*windowLength*, *slideInterval*)	Return a sliding window count of elements in the stream.	stream.countByWindow(60,5)
reduceByWindow(*func*, *windowLength*, *slideInterval*)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*.	stream.reduceByWindow(lambda x, y: x + y, 60, 5)
reduceByKeyAndWindow(*func*, *windowLength*, *slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function *func* over batches in a sliding window.	stream.reduceByKeyAndWindow(lambda x, y: x + y, 60, 5)
countByValueAndWindow(*windowLength*, *slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window.	stream.countByValueAndWindow(60,5)

Output Operations on DStreams

pprint()	Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging.	stream.pprint()
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as text files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.	stream.saveAsTextFiles(“words”,” ”txt”)
foreachRDD(*func*)	The most generic output operator that applies a function, *func*, to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database. Note that the function *func* is executed in the driver process running the streaming application	stream.foreachRDD(lambda rdd: rdd.foreach(sendRecord))

Spark Structured Streaming

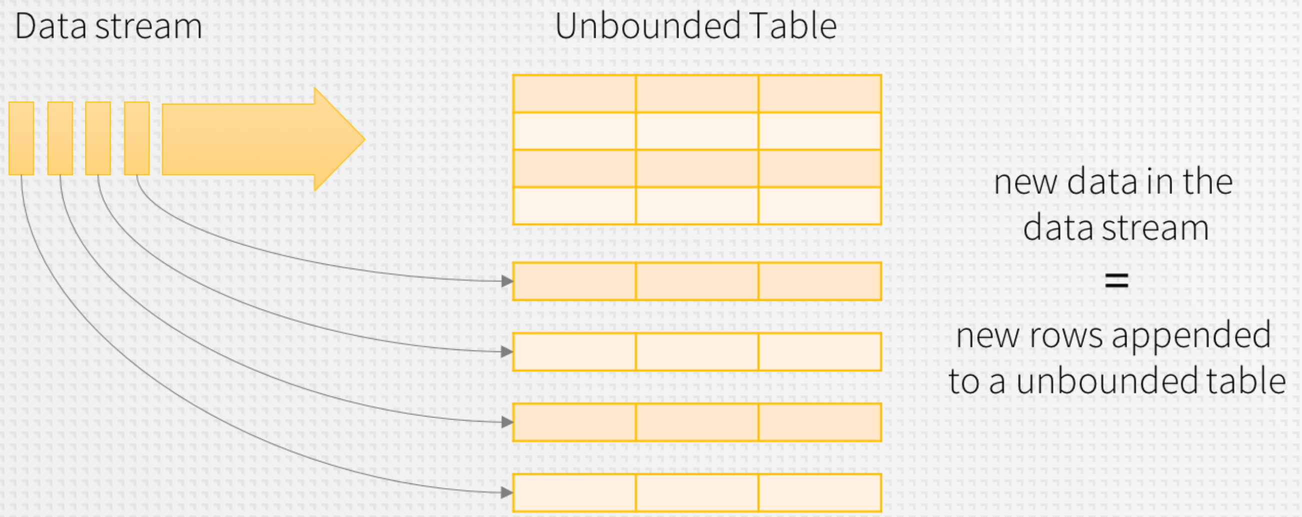


Structured Streaming processing modes: Micro-batch and Continuous Processing

Source: https://databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3-0.html

Spark Structured Streaming

- **Structured Streaming** treat live data as a table that is continuously appended
  - **Input table :** Every data item that is arriving on the stream is like a new row being appended to the Input Table.
  - **Result 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



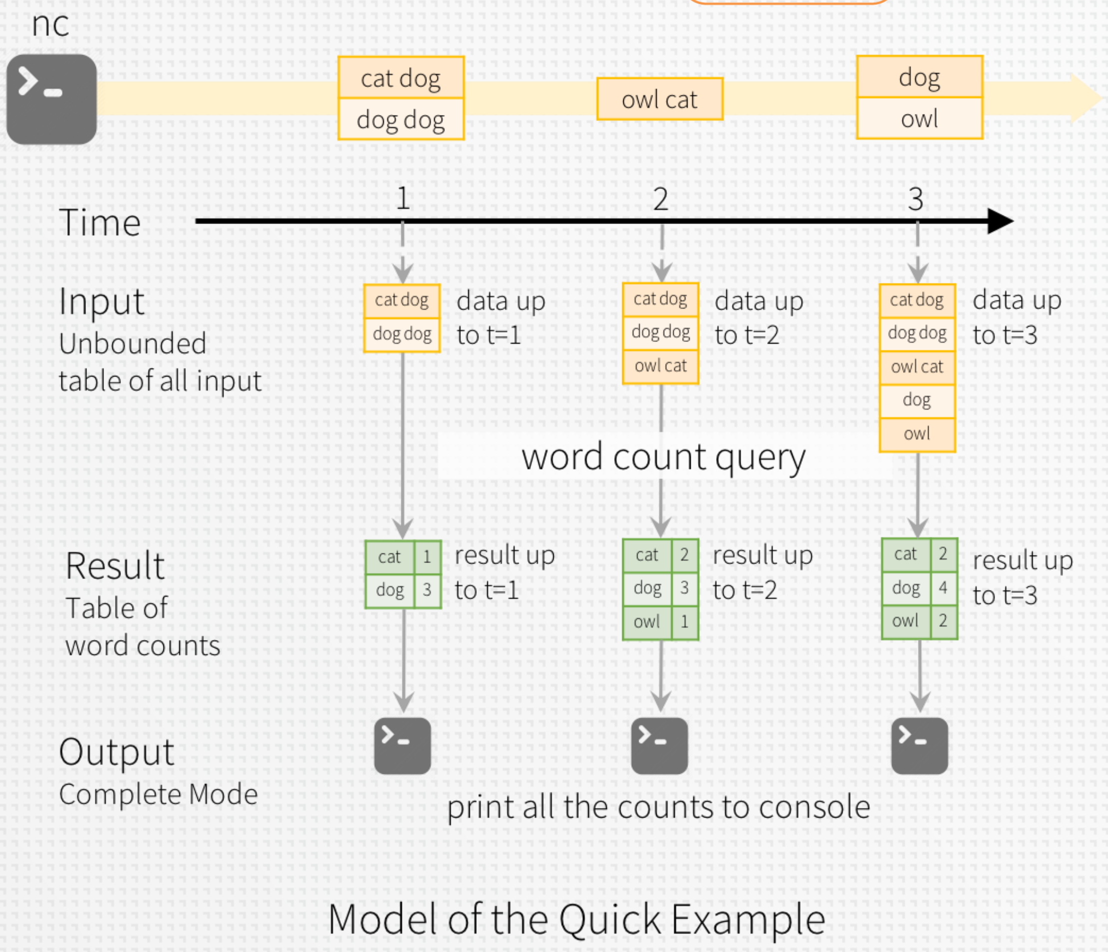
Data stream as an unbounded table

Source: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Spark Structured Streaming

Output strategies:

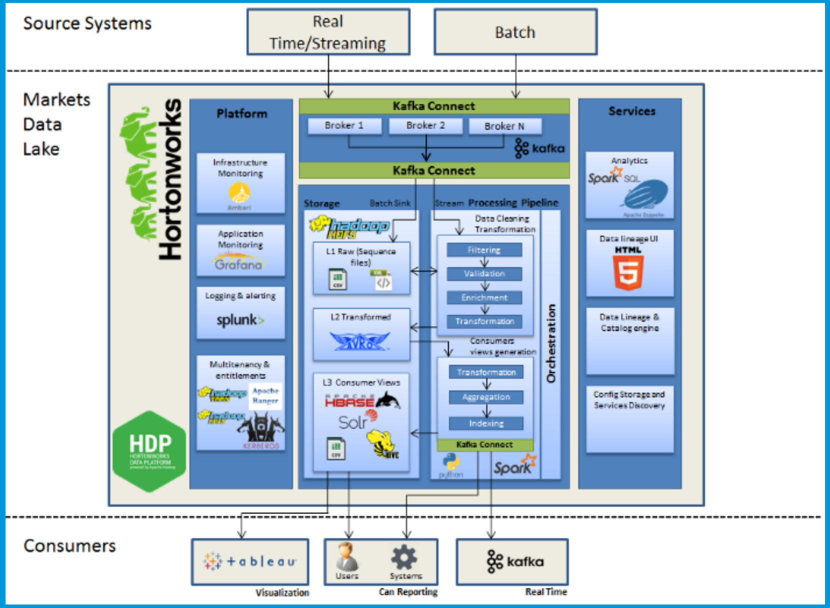
- **Complete Mode** - The entire updated Result Table will be written to the external storage.
- **Append Mode** - Only the new rows appended in the Result Table since the last trigger will be written to the external storage. This is applicable only on the queries where existing rows in the Result Table are not expected to change.
- **Update Mode** - 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.



Source: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Markets Data Lake

Success Story



The challenge

Design an implement an Strategic Data Lake using big data technologies covering all trading markets (FX, EQ, CM, MM, etc)

- Layered data storage having 3 levels: L1 for Raw data, L2 for transformed/enriched data, L3 for visualization/extraction data
- Multitenacy for different users and applications
- Authentication and authorization (entitlements) mechanism
- Ingestion of both real-time and batch data

The engagement

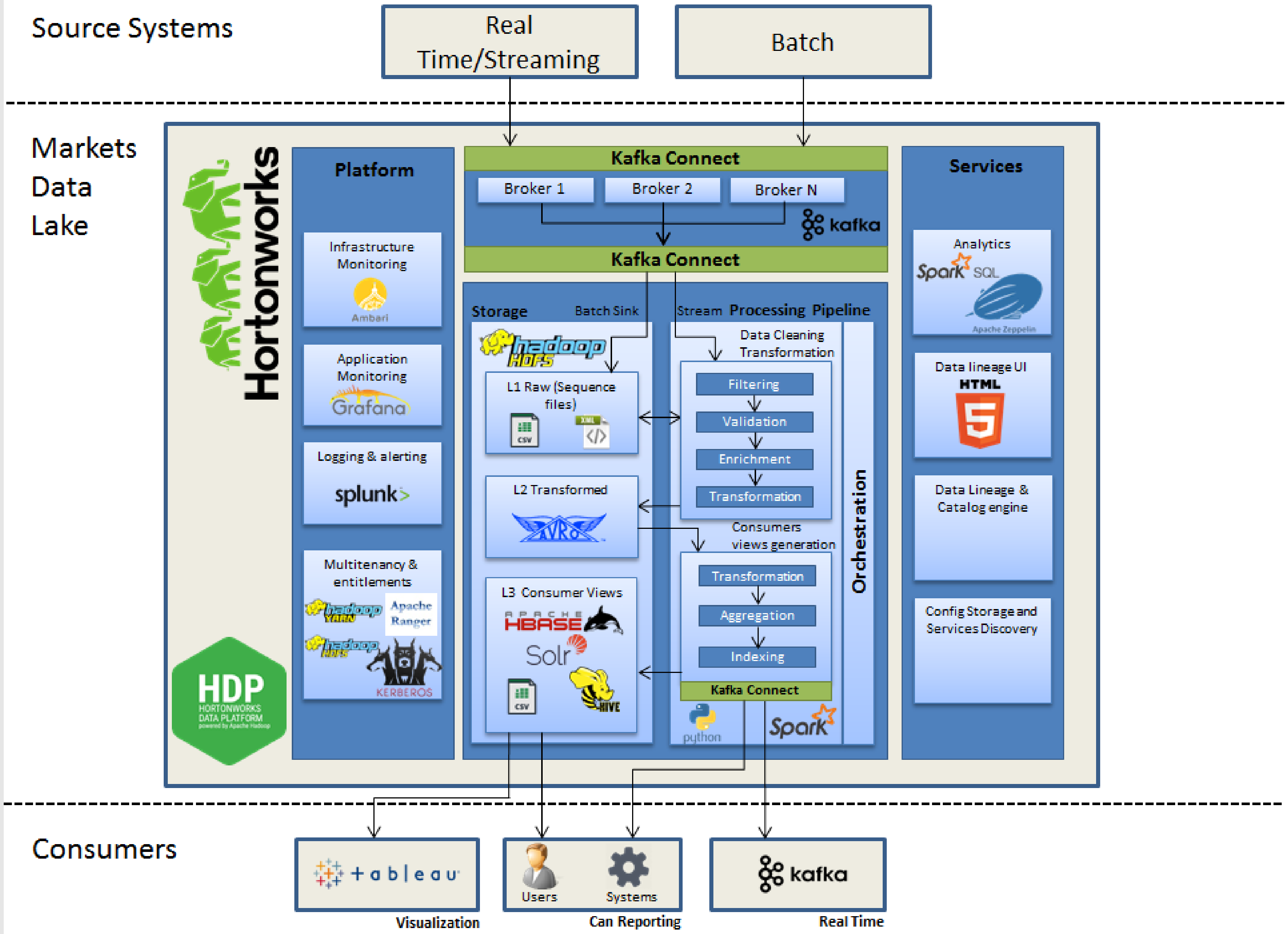
Design and implement a solution based on Kafka, Spark, Grafana, Splunk and HBase

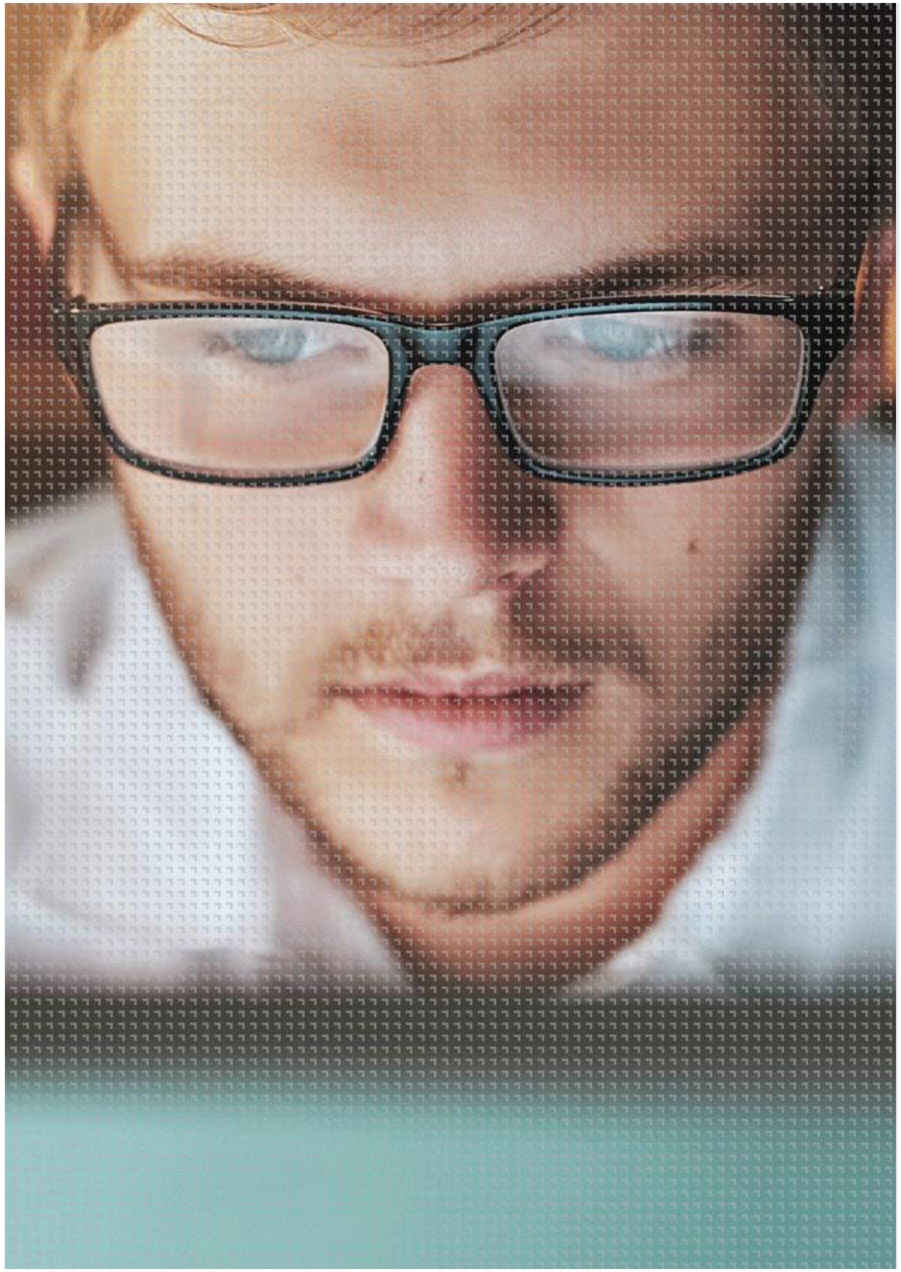
- Kafka is used as the message broker
- Kafka Connect to ingest data from any source (SFTP, JMS, etc)
- HDFS is used for the data storage in a 3-layers architecture
- Spark as the engine for the transformation pipelines
- HBase is the repository for the data catalog as well as to track data lineage
- Platform metrics collection and monitoring with Grafana
- Splunk for application logging and alerting

The benefit

Unique data repository where Quants teams (data scientists) perform market abuse insights

- Fully horizontally scalable solution
- Self-service ingestion, landing, and enrichment
- Extraction APIs
- Integration of Apache Zeppelin notebook for ad-hoc analysis


Source Systems
Real Time/Streaming
Batch
Markets Data Lake
Hortonworks
Platform
Infrastructure Monitoring
Ambari
Application Monitoring
Grafana
Logging & alerting
splunk>
Multitenancy & entitlements
hadoop YARN
Apache Ranger
hadoop HDFS
KERBEROS
HDP
HORTONWORKS DATA PLATFORM
Kafka Connect
Broker 1
Broker 2
Broker N
kafka
Kafka Connect
Storage
Batch Sink
hadoop HDFS
L1 Raw (Sequence files)
CSV
XML
L2 Transformed
AVRO
L3 Consumer Views
APACHE HBASE
Solr
CSV
HIVE
Stream Processing Pipeline
Data Cleaning Transformation
Filtering
Validation
Enrichment
Transformation
Consumers views generation
Transformation
Aggregation
Indexing
Kafka Connect
python
Spark
Orchestration
Services
Analytics
Spark SQL
Apache Zeppelin
Data lineage UI
HTML
Data Lineage & Catalog engine
Config Storage and Services Discovery
Consumers
tableau
Visualization
Users
Systems
Can Reporting
kafka
Real Time



Exercise 4 & 5 - Consuming from Kafka

Purpose of this exercise is to play with Streaming data stored on Kafka using Spark Streaming.

Exercise to be done is Exercise 4 & 5 on git →https://github.com/rlopezherrero/GFT-EDEM-MasterData/tree/master/AlmacenamientoProcesamiento/streaming

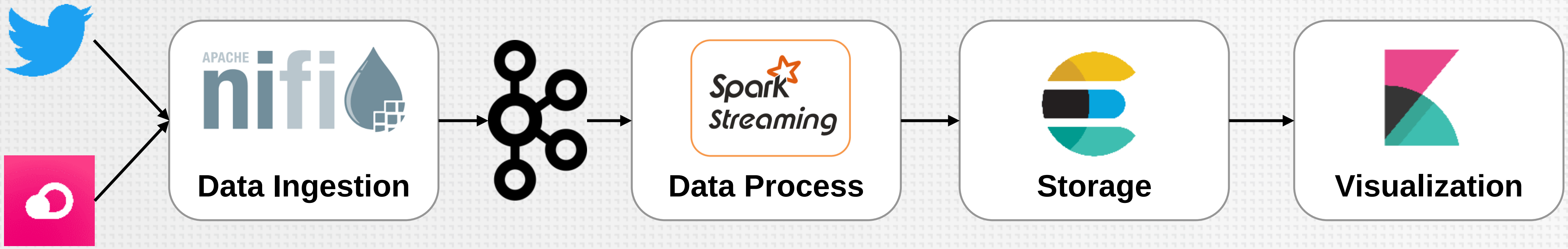
Agenda

1. Overview
2. Concepts
3. Technologies
4. Ejemplo de Arquitectura completa

Arquitectura Completa Streaming

Use case

- **Real-time processing and analytics** of trading data and news
- Data processing will include basic enrichment
- Data sources:
  - Trading data: **IEX Cloud**
  - News: **Twitter**



Instructions on -> https://github.com/rlopezherrero/GFT-EDEM-MasterData/tree/master/AlmacenamientoProcesamiento/streaming