Apache Spark

Alessandro Margara

alessandro.margara@polimi.it

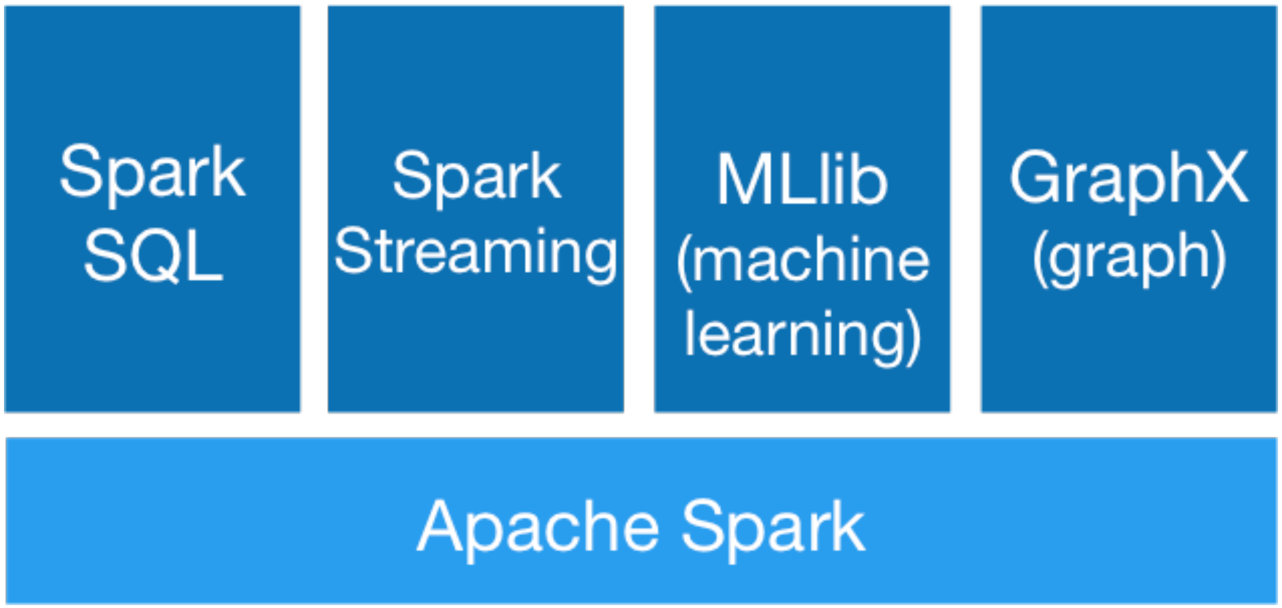
License

Slides adapted from the official Apache Spark documentation: https://spark.apache.org/docs/latest/index.html

Overview

- Unified analytics engine for large-scale data processing
  - As we will see, different types of applications: batch, streaming, structured, machine learning, graph processing, …
- Architecture similar to MapReduce …
  - Data stored on disk
  - Operators are scheduled on workers
- … with differences that enable for better performance
  - Data cached in memory
  - Jobs can consist of many “stages”
  - Support for iterative jobs

A unified analytics engine



Overview

- A Spark cluster consists of a *master* and one or more workers (*slaves*)
  - Typical configuration: one slave per host, using as many cores as available in the host
- The master
  - Accepts jobs from *driver* programs, and
  - Schedules processing tasks on available slaves

Overview

- We will write *driver programs* (in Scala) that submit jobs to the cluster
  - Each job consists of various *parallel operations*
- The main abstraction that Spark provides is the RDD (resilient distributed dataset)
  - Collection of elements
  - Partitioned across the nodes of the cluster
  - Can be processed in parallel
  - Fault tolerant

Initializing Spark

- A Spark program accesses the Spark cluster through a SparkContext object
- Contains relevant parameters
  - Name of the Spark application / job
  - Address of the master
  - …
- Only one context can be active per JVM
  - stop() closes a context and enables starting a new one

Initializing Spark

```
val conf = new SparkConf()
      .setAppName(appName)
      .setMaster(master)
val sc = new SparkContext(conf)

…

sc.stop()
```

Initializing spark

- When running in a real cluster
  - The application is packaged in a jar file
  - The jar is submitted to the cluster with a provided script
  - The address of the master is typically not hardcoded, …
  - … but extracted from a configuration file

- Local mode
  - Run the driver and the workers in the same JVM
  - For testing and debugging
  - Setting the master to local[n] requests Spark to use n cores to run the workers

RDDs: initialization

- RDDs can be created from
  - Existing collections in the driver program (a Scala Seq)
  - External datasets
    - Local filesystem
    - HDFS
    - Kafka
    - Several DBMSs
    - …

RDDs: initialization

- RDD from local collection

```
val data = Array(1, 2, 3, 4)
val myRDD = sc.parallelize(data)
```

- RDD from file

```
val myRDD = sc.textFile(“filename.txt”)
```

RDDs: operations

- RDDs support two types of operations
  - *Transformations* create a new RDD from an existing one
  - *Actions* return a value to the driver program after running a computation on the dataset

- All transformations are lazy
  - They do not compute the results when invoked
  - They remember the computation to be implemented, …
  - … and execute the computation when an action requires a result to be returned to the driver program

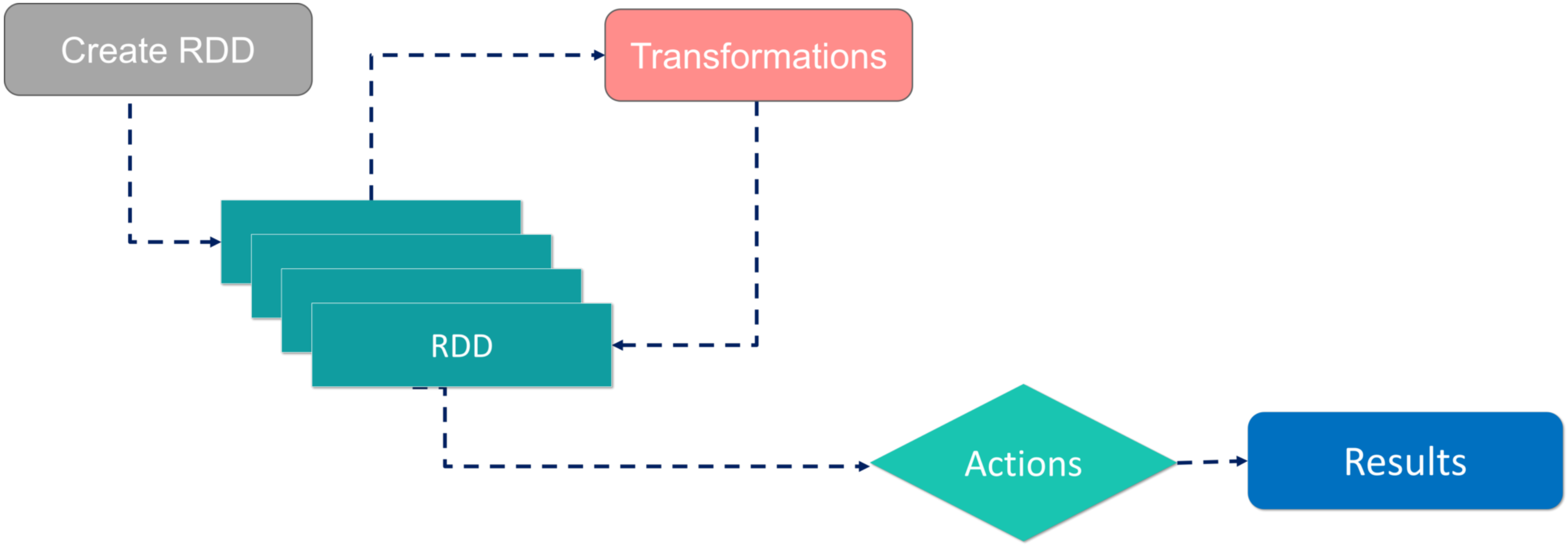
RDDs: operations

- By default, each transformed RDD is recomputed each time the driver runs (directly or indirectly) an action on it

- The programmer can also persist/cache an RDD in memory (in the workers) for faster access in subsequent queries

- Various storage levels available
  - In-memory objects
  - In-memory serialized objects
  - In-memory + on disk serialized objects
  - On disk serialized objects
  - …

RDD: operations



RDDs: simple example

```
/* Creates an RDD. The number of partitions is decided
based on the available workers */
val lines = sc.textFile("data.txt")


/* Transformation. Map applies a function to each and
every element of the original RDD. In this case, it
transforms each string (line) into a number (the length
of the line) */
val linesLen = lines.map(s => s.length)


/* Action. Reduce aggregates all the values in a single
element and returns the result to the driver. In this
case, it returns the sum of all the length of all the
lines) */
val totLen = linesLen.reduce((a, b) => a + b)
```

RDDs: simple example

```
val lines = sc.textFile("data.txt")
val linesLen = lines.map(s => s.length)
val totLen = linesLen.reduce((a, b) => a + b)
```

- `lines` and `linesLen` are not immediately computed
  - `lines` does not load any data from the file

- When the `reduce` action is invoked, it requests the value of `linesLen`, which requests the value of lines

- These values are not persisted
  - Unless the programmer explicitly invokes `cache()` / `persist()`

RDDs: fault tolerance

- As said, by default RDDs are not persisted

- In the case an RDD is persisted (cached), the cache is fault-tolerant
  - If any partition of an RDD is lost, …
  - … it will automatically be recomputed using the transformations that originally created it

RDDs: some transformations

Transformation	Semantics
map(fun)	Applies fun to each and every element in the source RDD
filter(fun)	Returns a new RDD with all and only the elements e in the source RDD for which fun(e) == true
flatMap(fun)	As map, but fun can return zero, one, or more results for each element in the source RDD

RDDs: some transformations

Transformation	Semantics
union(otherRDD)	Returns the union of the source RDD and otherRDD
intersection(otherRDD)	Returns the intersection of the source RDD and otherRDD
distinct()	Returns a new RDD that contains the distinct elements in the source RDD

RDDs: some transformations

Transformation	Semantics
groupByKey()	When called on a RDD of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs
reduceByKey(fun)	When called on a RDD of (K, V) pairs, returns a new RDD of (K, V) pairs where the values for each key are aggregated using the given reduce function fun (of type (V, V) => V)
aggregateByKey(zero)(seqOp, combOp)	When called on a RDD of (K, V) pairs, returns a RDD of (K, U) pairs where the values of each key are aggregated using the given combine function and a neutral zero value

RDDs: some transformations

Transformation	Semantics
join(otherRDD)	When called on RDDs of type (K, V) and (K, W), returns a RDD of (K, (V, W)) pairs with all pairs of elements for each key: can be configured for outer joins
cogroup(otherRDD)	When called on RDDs of type (K, V) and (K, W), returns a RDD of (K, Iterable<V>, Iterable<W>) tuples
cartesian(otherRDD)	When called on RDDs of type T and U, returns a RDD of (T, U) pairs (all pairs of elements)

RDDs: some actions

Transformation	Semantics
reduce(fun)	Aggregate the elements of the source RDD using the function fun
collect()	Returns all the elements of the RDD as an array
count()	Returns the number of elements in the RDD
take(n)	Returns an array with the first n elements in the RDD
saveAsTextFile(path)	Writes the elements as a text file (or set of text files) in the local filesystem or HDFS

Shuffle operations

- Some operations *shuffle* the data = re-distribute data changing the way they are grouped across partitions
- Shuffle operations involve copying data across workers, making it a complex and costly operation
- Consider for example the classic word count example
  - Data is initially partitioned by document
  - It needs to be re-partitioned by word

Shuffle operations

- Shuffle is an expensive operation because it involves
  - Serialization/deserialization
  - Disk/network IO

- Internally, shuffle operations can consume memory to store intermediate results while reorganizing the data
  - Data structures kept in memory until they can't fit
  - Then they are spilled to disk

- After the data has been reorganized by key, each key is transmitted to the partition responsible for it

Spark Architecture

Spark architecture

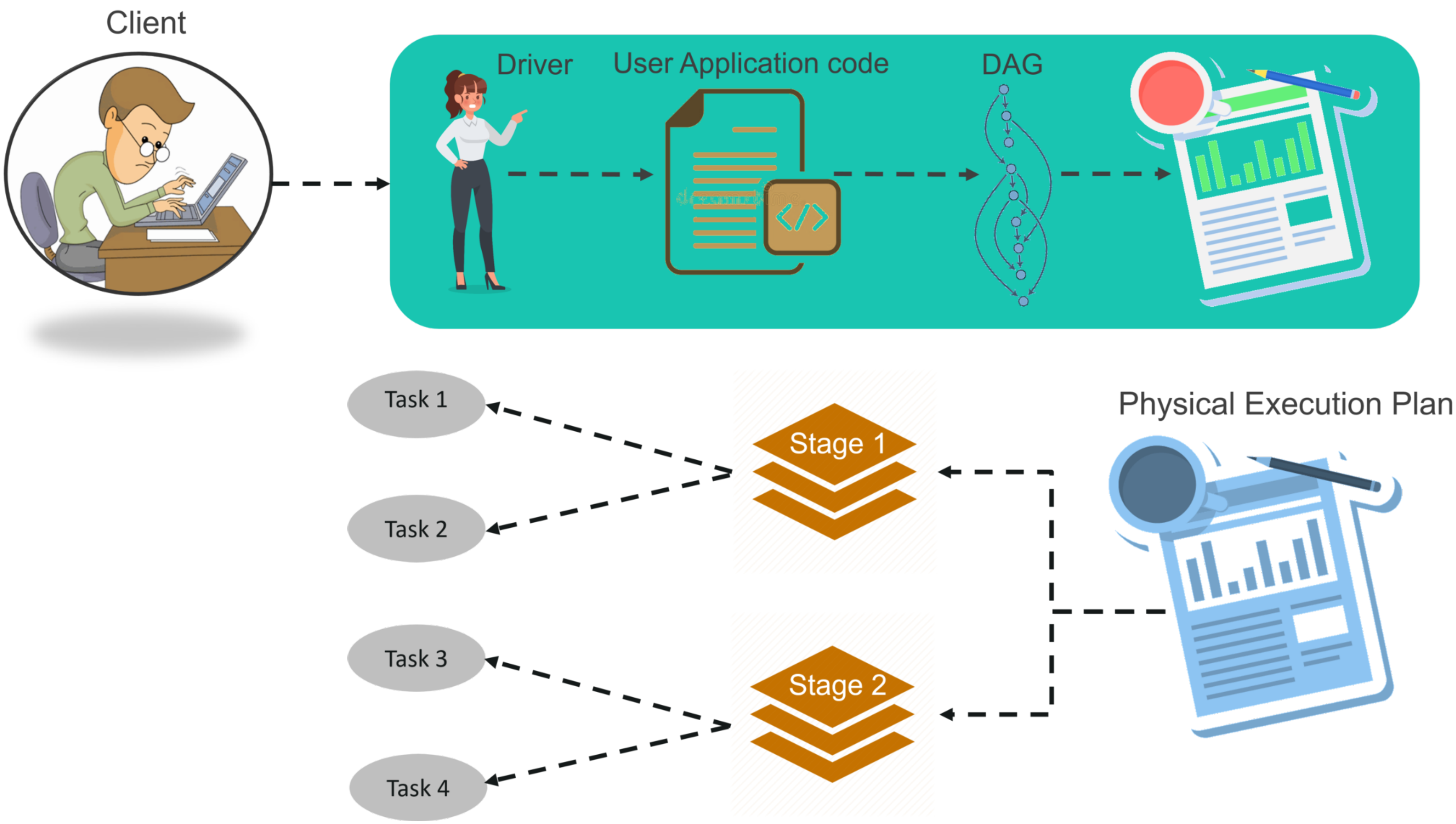
- The previous slides presented the key programming abstractions …
- … and showed how some operations can influence performance
  - Shuffle operations
- We now present the Spark architecture in more details

Spark architecture

- When a driver program submits a job to the Spark Context

1. The Spark context extracts a DAG of operators
2. The logical DAG is transformed into a physical execution plan
   - Multiple *stages*: each stage contains a sequence of operations with no intermediate data shuffle
3. Each stage is split into tasks (one for each partition)
4. Tasks are scheduled on the cluster
   - Where / close to the data they consume
   - Taking into account the dependencies between tasks

Spark architecture



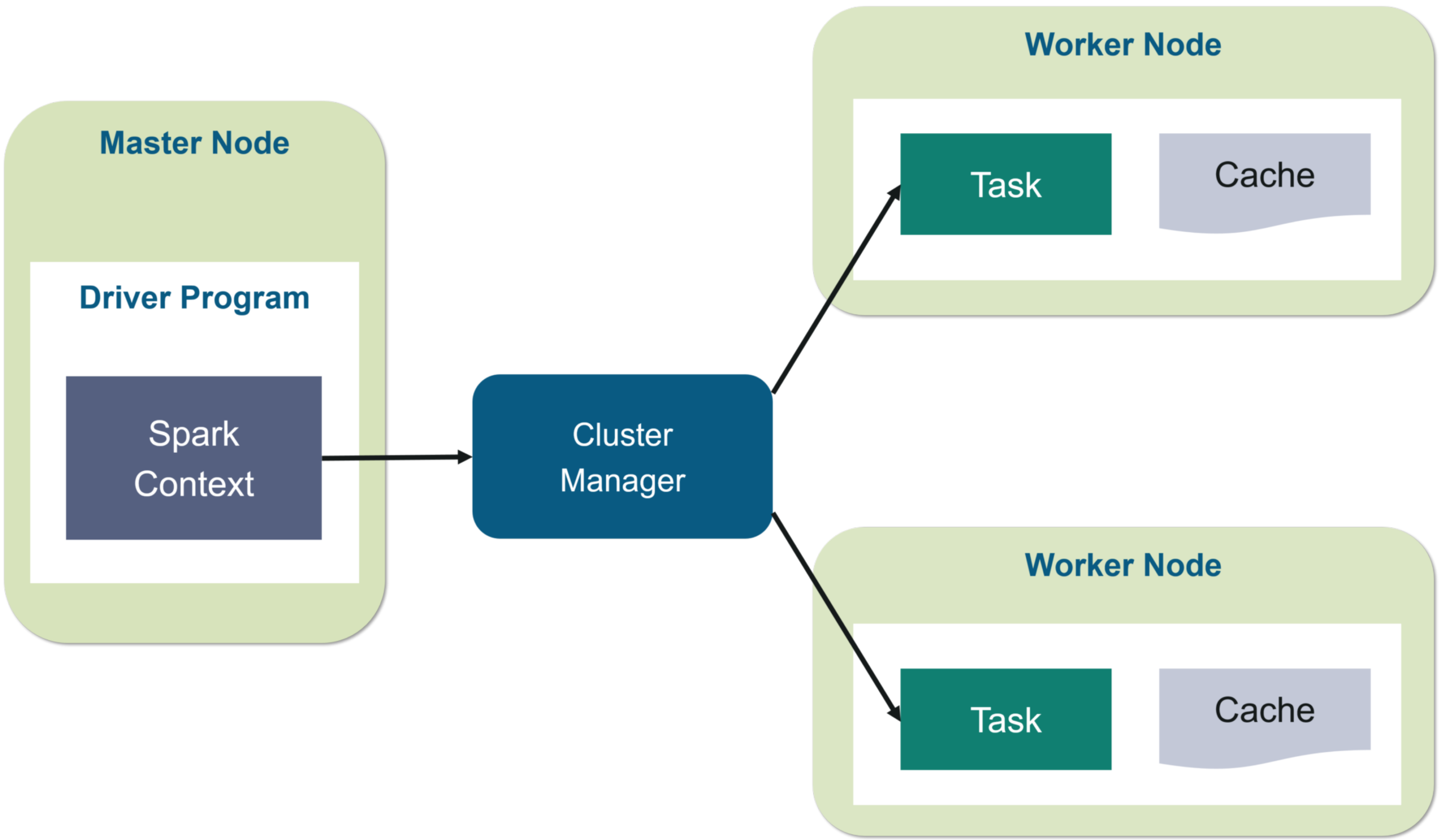
Spark architecture

- When persisted, RDDs are stored in the cache of worker nodes
  - Depending on the specified level, it is cached in memory, on disk, or both
  - Cached in serialized, un-serialized form, or both

- RDDs are partitioned and distributed across workers according to the specified key

- Tasks are scheduled where the data they consume is located

Spark architecture



Shared Variables

Shared variables

- After diving into some architectural details, we can go back to the programming primitives

- As we said, Spark operations consume data and produce new data, …
- … they do not operate on shared state

- Spark offers some limited support to shared variables, which can useful in some scenarios
  - Broadcast variables
  - Accumulators

Broadcast variables

- If a function passed to Spark accesses a variable …
- … it works on a separate copy of that variable in each process (when executed in cluster cluster)
  - Warning: this might not be discovered when testing in local mode!!!

- Broadcast variables enable to keep read-only variables cached in each machine
  - They are used to give to each node a copy of a dataset that everybody needs to read in an efficient manner

Broadcast variables

- Spark automatically broadcasts the common data within each stage

- Data is cached in serialized form and deserialized before running a task

Broadcast variables

- Broadcast variables are created from a variable v by calling `SparkContext.broadcast(v)`
  - The broadcast variable is a wrapper around v
  - The value can be accessed by calling the value method

```
val broadcastVar = sc.broadcast(v)
broadcastVar.value
```

- The original variable should not be modified after it is broadcast …
- … to ensure that all the nodes get the same value of the variable when it is delivered

Broadcast variables

- In some cases, using broadcast variables can be better than accessing an RDD
  - For example, if several operators need to access a static dictionary (that never changes)
- The broadcast variable is stored in the cache of each node in non-serialized form
  - If the dictionary is implemented as a hash map, it is possible to directly retrieve values by key in constant time

Broadcast variables

- In general, use broadcast variables when you have data that is
    - Not “too large”
    - Shared across multiple operators
        - Not partitioned
    - Read-only

Accumulators

- Accumulators are variables that can only be modified using associative and commutative operations
  - Example: counters, sums, …
  - Can be easily supported in parallel
- Accumulators can have an associated name
  - In this case they are displayed in the Web UI
  - They are useful to keep track of tasks

Accumulators

- Numeric accumulators can be created with
  - `SparkContext.longAccumulator()`
  - `SparkContext.doubleAccumulator()`
- Tasks can add using the `add()` method
- Only the driver can read the accumulator's value using the value method

Accumulators

```
val acc = sc.longAccumulator(“Name”)

sc.parallelize(Array(1, 2, 3, 4))
  .foreach(x => acc.add(x))

accum.value
```

Accumulators

- Developers can create custom accumulators by
  - Inheriting from the `AccumulatorV2` class
  - Overriding `reset()` to reset the accumulator to the initial empty / zero value
  - Overriding `add()` to add a new value
  - Overriding merge() to merge two values (partial results)

- To register an accumulator `acc` of a custom type `accType`

```
SparkContext.register(acc, “accName”)
```

Accumulators

- Accumulators do not change the lazy evaluation approach of Spark: they are only executed when a transformation is triggered directly or indirectly by an action

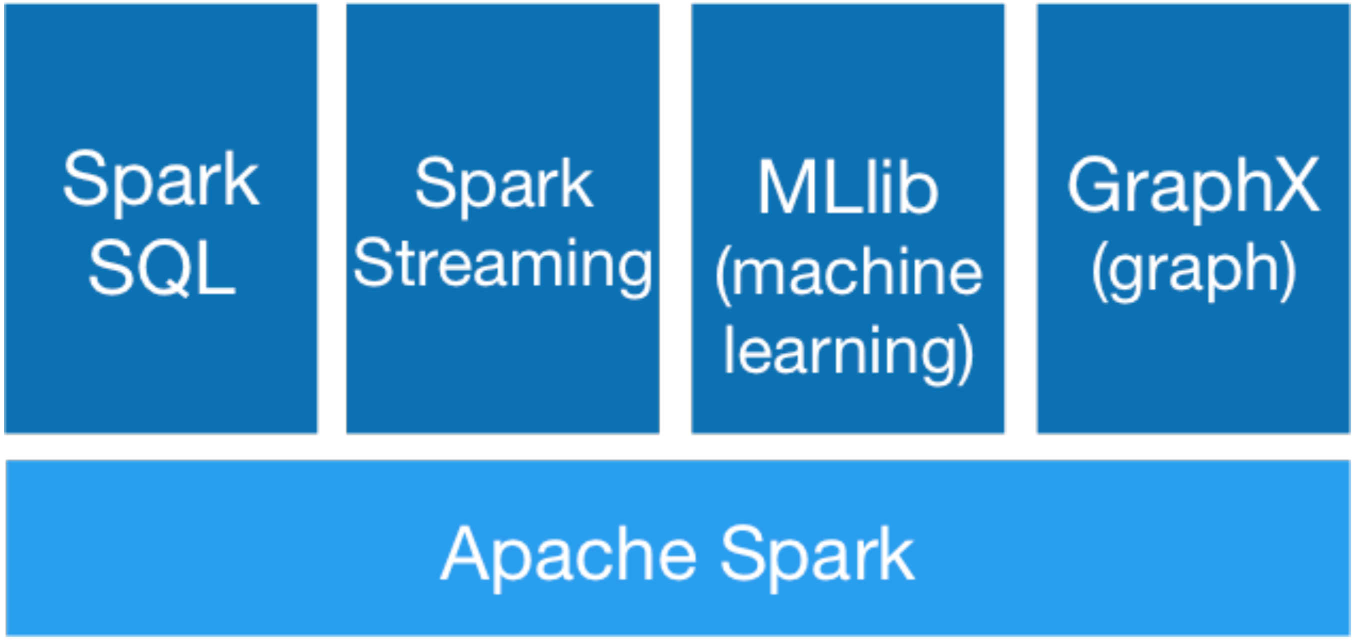
- Spark guarantees that accumulators inside actions are updated exactly once

- Warning: Spark does not guarantee that accumulators inside transformations are updated exactly once
  - They can be updated more than once if the task is re-executed

Spark SQL

SQL, Datasets, and DataFrames

Spark SQL


Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)
Apache Spark

Spark SQL

- Spark SQL is a Spark module for structured data processing
  - Built on top of the core Spark infrastructure
- The interface provides more information about the structure of the data, with multiple advantages
  - Higher-level, declarative API, derived from well known database concepts
  - More opportunities for the engine to optimize the computation

Spark SQL

- Spark SQL offers different API/languages
  - SQL
  - Dataset API
  - DataFrame

- The same Spark SQL engine is used, independently from the API/language adopted
  - It is possible / easy to switch between different APIs depending on the specific needs of the application

Spark SQL: SQL API

- The SQL API enables developers to execute standard SQL queries
  - From a programming language
  - From command-line
  - Over JDBC/ODBC
  - …

Spark SQL: Dataset and DataFrames

- Dataset is an interface that provides the level of abstraction of RDDs …
  - Transformations and actions over large collections

- … with the additional benefits of the Spark SQL optimized execution engine

Spark SQL: Dataset and DataFrames

- Datasets use a specialized encoder to serialize the objects, processing them, and transmitting over the network

- By knowing the format, Spark can perform many operations without deserializing the object!
  - Filtering, sorting, hashing, …

Spark SQL: SparkSession

- The Spark SQL module exposes its functionalities through the SparkSession class
  - Similar to SparkContext for the core Spark API
  - Obtained using a builder

```
val spark = SparkSession
    .builder()
    .appName("App name")
    .master("local")
    .config("option name", "option value")
    .getOrCreate()
```

Spark SQL: Datasets creation

- Datasets enable the engine to optimize the execution using encorders
  - Differently from a serializer, encoders are code generated dynamically that performs many operations without de-serializing the objects

- In Scala, encorder are available for:
  - Most common types (using implicits)
  - Case classes

- DataFrames can be converted into Datasets by providing a class
  - More on DataFrames, later …

Spark SQL: Datasets creation

```
// Dataset from case classes
case class Person(name: String, age: Long)
val ds1 = Seq(Person(“Ale”, 24)).toDS()

// Dataset from “common” types (integer here)
val ds2 = Seq(1, 2, 3).toDS()

// Dataset from DataFrame
val dataFrame = spark.read.csv(“some_path”)
val ds3 = dataFrame.as[Person]
```

Spark SQL: DataFrames

- A DataFrame is a Dataset organized into named columns
- Conceptually equivalent to a relational database
- In Scala, a DataFrame is a Dataset of Row
  - `DataFrame` is an alias of `Dataset[Row]`

Spark SQL: DataFrame creation

- Application can create DataFrames using the methods in SparkSession.read
  - Different sources

```
spark.read.json("file")
spark.read.csv("file")
spark.read.textFile("file)
spark.read.jdbc(…)
```

Spark SQL: DataFrame creation

- DataFrames can also be created from RDDs
  - By inferring the schema using reflection
  - By programmatically defining the schema

- For example, reflection is used with case classes
  - `People` is a case class
  - `peopleRDD` is `RDD[People]`
  - You can obtain a *`DataFrame`* using `peopleRDD.toDF()`
  - You can refer to fields (columns) by name, which is inferred by the name of the fields in `People`

Spark SQL: DataFrame creation

- Instead, it is possible to programmatically define the schema of a DataFrame when it is not known upfront
  - For instance, if it is loaded from a file

```
val mySchema = StructType(Array(
   StructField(“name”, StringType, true),
   StructField(“surname”, StringType, true),
   StructField(“age”, IntegerType, true),
))

val df = spark.createDataFrame(myRDD, mySchema)
```

Spark SQL: DataFrame

- It is possible to print the content of a DataFrame on the standard output

```
df.show()
```

- Textual representation of the table, …
- … very useful for testing and debugging
  - Display the results of transformation on a small dataset

Spark SQL: DataFrame operations

- DataFrames include additional information on their content (e.g., names of columns)
  - Enables more declarative processing
  - Implicit conversions and simplified column access (using the $ notation) improve readability

```
import spark.implicits._

df.filter($“age” > 18)
  .select($“name”, $“salary” + 10)
```

Spark SQL: DataFrame operations

- SparkSQL also accepts SQL queries as strings

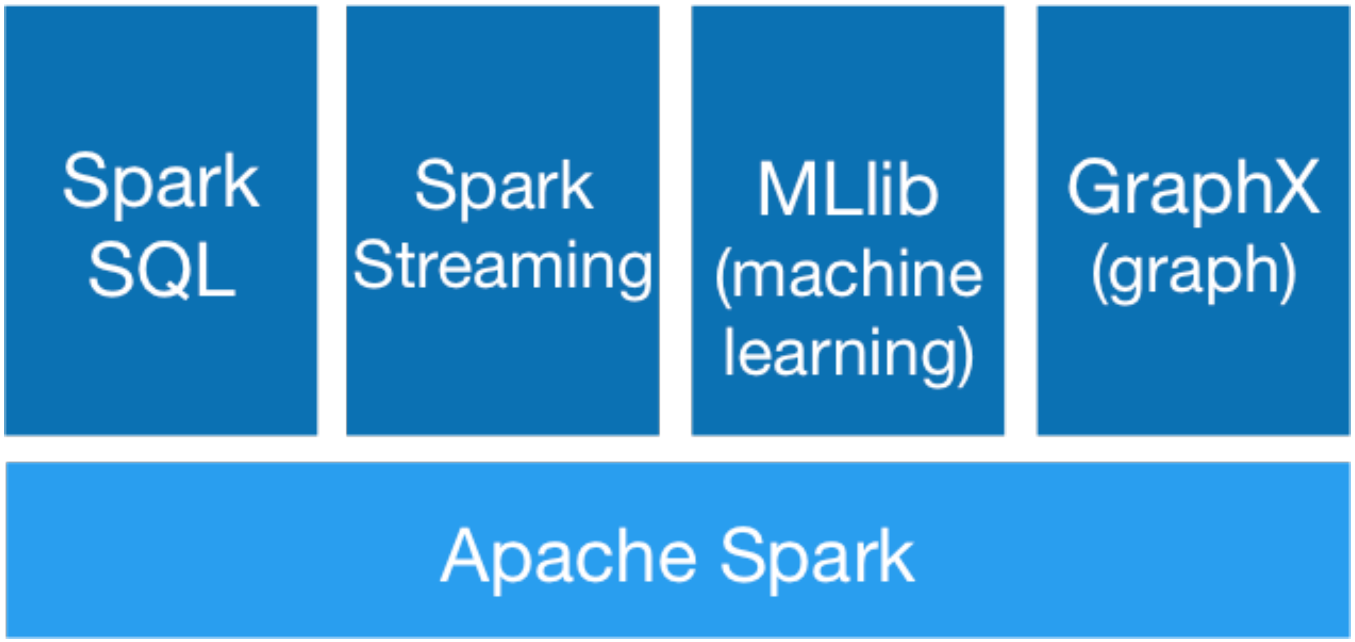
```
employeeDataFrame
  .createOrReplaceTempView(“employee”)

val resultDF = spark.sql
(“   SELECT *
     FROM  employee
     WHERE salary < 100  ”)
```

Spark SQL: DataFrame operations

- Spark SQL supports common aggregations
  - count(), countDistinct(), avg(), sum(), min(), max()
- It enables developers to write and use their own custom aggregations
  - Extending the UserDefinedAggregateFunction class
  - Overriding methods to
    - Initialize the aggregate value
    - Update the aggregate value
    - Merge partial results

Spark Streaming

Spark SQL



Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)
Apache Spark

Spark Streaming

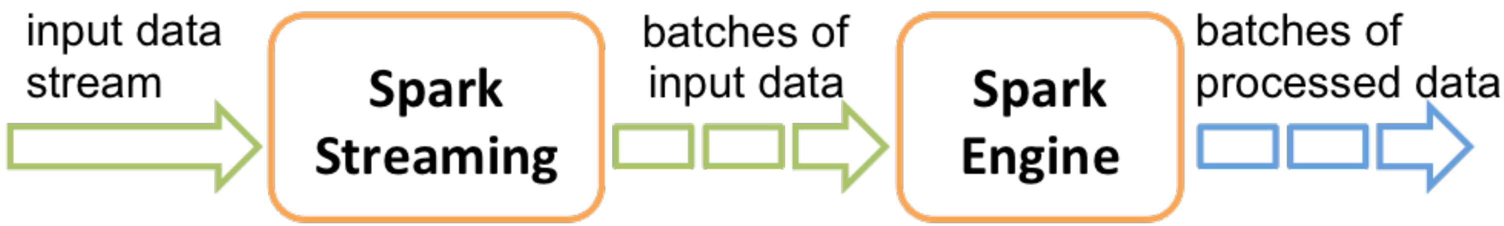
- Spark Streaming is an extension of the core Spark API to process streaming data
- Other processing engines (e.g., Apache Storm, Apache Flink) adopt a streaming architecture
  - Operators are instantiated and deployed
  - Streams of data flow from operator to operator
  - Pro: very low delay
  - Cons: dynamic adaptation (e.g., dynamic scalability) more difficult
    - Since operators are pre-deployed

Spark Streaming

- Spark Streaming adopts a different “micro-batch” approach:

1. It splits the input streams into small batches, …
2. … which are processed by the Spark engine …
3. … to generate the final stream of results in batches

Spark Streaming



- Pro: dynamic adaptation is easier (scheduling decisions can change over time)
- Cons: higher processing delay

Spark Streaming API

- Spark Streaming's main abstraction is the *discretized stream* (*DStream*)
- Internally, a DStream is represented as a sequence of RDDs
- DStreams provide operations to transform the RDDs in the sequence
  - Also provides stateful operations that preserve internal state across invocations

Spark Streaming Example

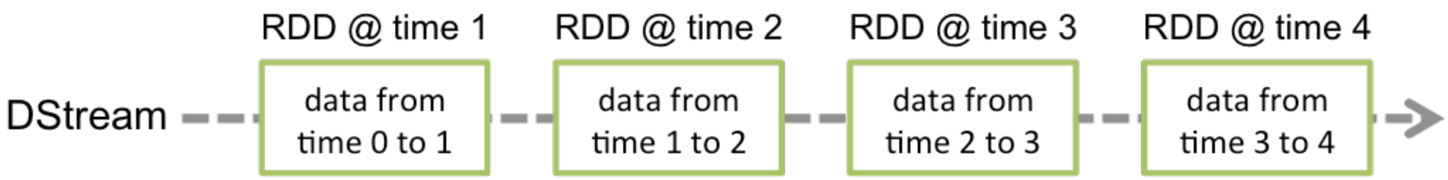
- As an example, consider again the word count application
- For the sake of simplicity, we read streaming data from a TCP socket
    - Production environments typically adopt data sources that can be replayed in the case of failures (e.g., Apache Kafka queues)
- When applied in a streaming context, the count is performed separately on each and every RDD

Spark Streaming Example

```
val conf = new SparkConf()
               .setMaster("local[2]")
               .setAppName("StreamingWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

val counts = ssc.socketTextStream("localhost", 2345)
                .flatmap(_.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _)
counts.print()

ssc.start()
ssc.awaitTermination()
```

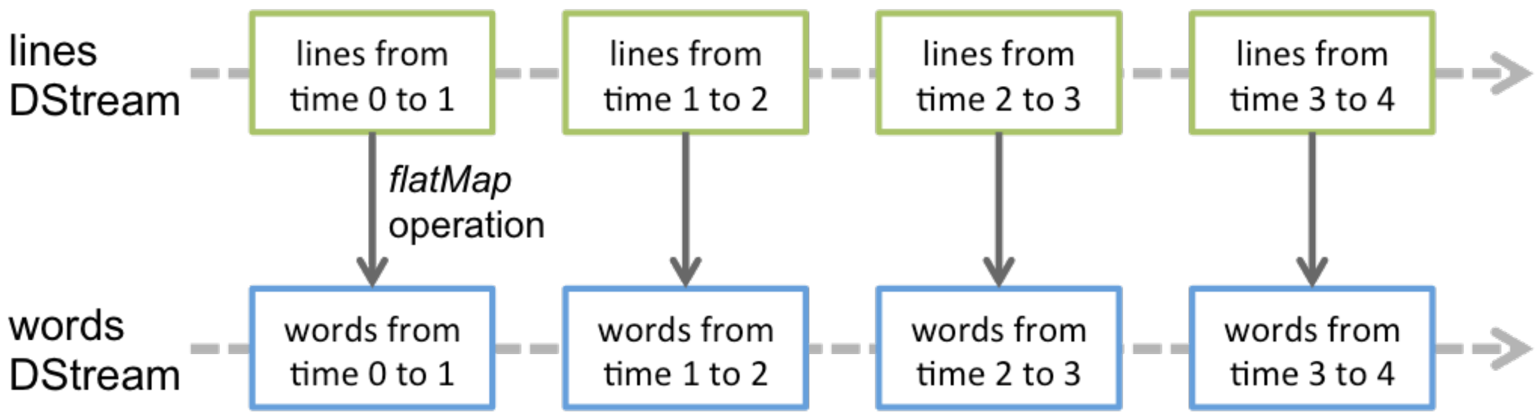
Spark Streaming API

- A DStream is a sequence of RDDs
  - Each RDD contains data from a certain interval
  - More on time, later



Spark Streaming API

- Any operation applied on a DStream translates to operations on the underlying RDDs



Receivers

- Input DStreams, received from external sources such as a socket, are associated with a *Receiver* object
  - It received the data from a source and stores it in the memory of Spark

- A receiver occupies one thread
  - When running in local mode, we need to allocate a number of threads n that is larger than the number of input streams / receivers (“local[n]” as the master URL)
  - When running on a cluster, the number of cores allocated to the Spark Streaming applications must be more than the number of receivers

Receivers: reliability

- Some sources allow the data to be acknowledged
  - E.g., Apache Kafka
- As a consequence, there are two types of receivers
  - Reliable receivers acknowledge the data to the sources once it has been received and stored in Spark with replication
  - Unreliable receivers do not send acknowledgements to a source

Fault tolerance semantics

- RDDs are stored on durable storage (replicated file system)
- In the case of failure, since transformations are deterministic, the data in transformed RDDs can always be recomputed by re-executing the transformations over the original RDDs
- With Spark Streaming there is an additional problem
  - Data is received from external sources

Fault tolerance semantics

- In Spark Streaming there are two types of data that the system needs to recover in the case of failure

1. Data received and replicated: it survives the failure of a node as a copy exists in other nodes

2. Data received but not yet replicated: the only way to recover this data is to get it again from the source, if possible

Fault tolerance semantics

- In general, there are three possible types of guarantees when processing streaming data records

1. At most once: each record is either processed once or not processed at all
2. At least once: each record is processed one or more times (duplicates are possible in the case of failures)
3. Exactly once: each record is processed once and only once

Fault tolerance semantics

- To ensure at least once semantics, sources can send again all the data that was not acknowledged by the receiver
  - This can lead to duplicates in the case the data was received but the acknowledge lost during a failure
- To ensure exactly once semantics, receiving and acknowledging data must be atomic
  - Implemented using some transactional mechanism
  - The data is acknowledged only when it is saved and replicated
  - Data has associated sequential numbers to enable discarding duplicates

Fault tolerance semantics

- The same holds for receivers, in the case of external processes

- Receivers must also implement some transactional mechanism
  - Ensure that the data is acknowledged only when it is saved on durable state
  - Use sequential numbers to discard duplicates

Fault tolerance semantics

- In summary, the actual fault tolerance guarantees depend on the specific data sources and sinks
- For instance, exactly once semantics is guaranteed if the input streams and the results are stored on Kafka queues
  - This is a common scenario in modern (micro service) distributed architectures

DStreams transformations

- DStreams support many of the transformations available on normal RDDs
  - map, flatMap, fliter, union, reduce, …

- As we have seen in the streaming word count example, these transformations are applied separately on each and every RDD in the Dstream

DStreams transformations

- Another class of interesting transformations are *stateful* operations
  - When processing an element, they preserve some state that can be subsequently accessed while processing further elements

- We will see three examples of stateful operations
  - updateStateByKey
  - mapWithState
  - windows

DStreams transformations

- updateStateByKey creates a *state* DStream
  - This is used to maintain a key-value store
  - The value is updated by applying a given function on the previous state of the key and the new state of the key

DStreams transformations

// Definition

```
def updateFunction
(newValues: Seq[Int], oldValue: Option[Int]):
Option[Int] = {
      val newValue = ... //
      Some(newCount)
}
```

// Application

```
val stateRDD =
someRDD.updateStateByKey[Int](updateFunction _)
```

DStreams transformations

- Spark also enables state to be updated and used as part of a transformation
  - Example: mapWithState
- We can use this to change the semantics of the streaming word count application
  - The count is preserved and updated across RDDs

DStreams transformations

```
val initialRDD =
sc.sparkContext.emptyRDD[Tuple2[String, Int]]

val stateMapFunction =
(word: String, count:
Option[Int], state: State[Int]) => {
  val sum = count.getOrElse(0) +
            state.getOption.getOrElse(0)
  val output = (word, sum)
  state.update(sum)
  output
}
```

DStreams transformations

```
val counts = textLines
  .map(_.toLowerCase)
  .flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .mapWithState(StateSpec
     .function(stateMapFunction)
     .initialState(initialRDD)
  )
```

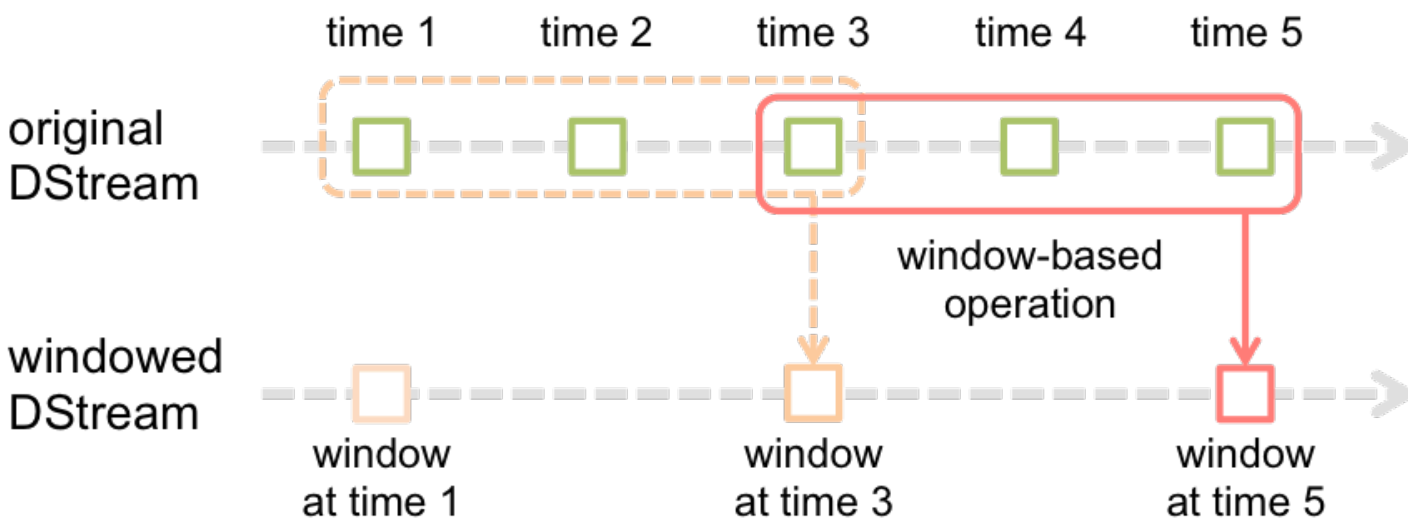
Windows

- Spark Streaming provides windowed computations, to apply transformations over a sliding window of data

- A window is defined in terms of two parameters
    - Window length: the duration of the window
    - Sliding interval: the interval (rate) at which the window operation is performed

- Note: these two parameters must be multiples of the batch interval of the source DStream

Windows

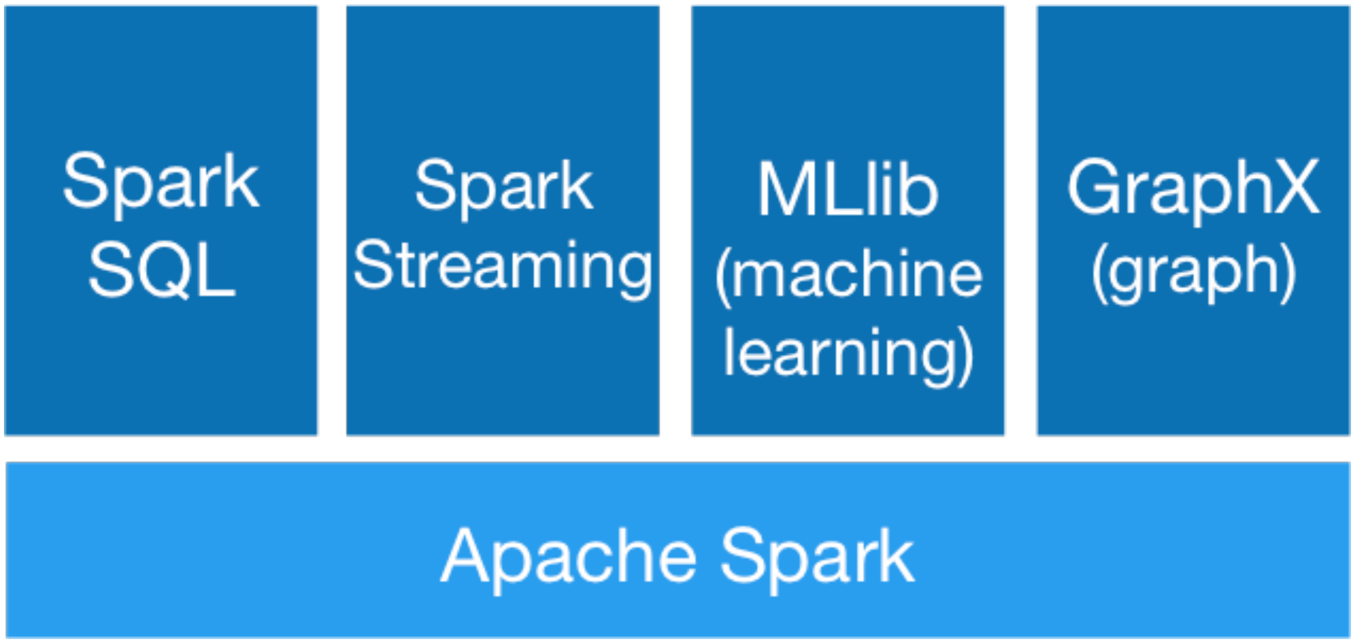

time 1
time 2
time 3
time 4
time 5
original DStream
window-based operation
windowed DStream
window at time 1
window at time 3
window at time 5

Windows

- Spark Streaming offers several operations to define windows and perform computations over windows
  - countByWindow
  - reduceByWindow
  - countByKeyAndWindow
  - reduceByKeyAndWindow
  - …

Structured Streaming

Spark SQL


Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)
Apache Spark

Structured Streaming

- Alternative API and programming model w.r.t. Spark Streaming

- Build on the Spark SQL engine

- Core ideas
    - Express streaming computations in the same way as batch computations on static data
    - The engine takes care of continuous and incremental execution to update the final results as new data arrives

Structured Streaming

- Internally, Structured Streaming queries are processed using the Spark micro-batch approach
  - Same latency as Spark Streaming (hundreds of milliseconds)
  - Same fault tolerance semantics (end-to-end exactly once semantics if sources and sinks enable so)

- Spark 2.3 introduced a new Continuous Processing mode
  - Latency in the order of milliseconds

Programming model

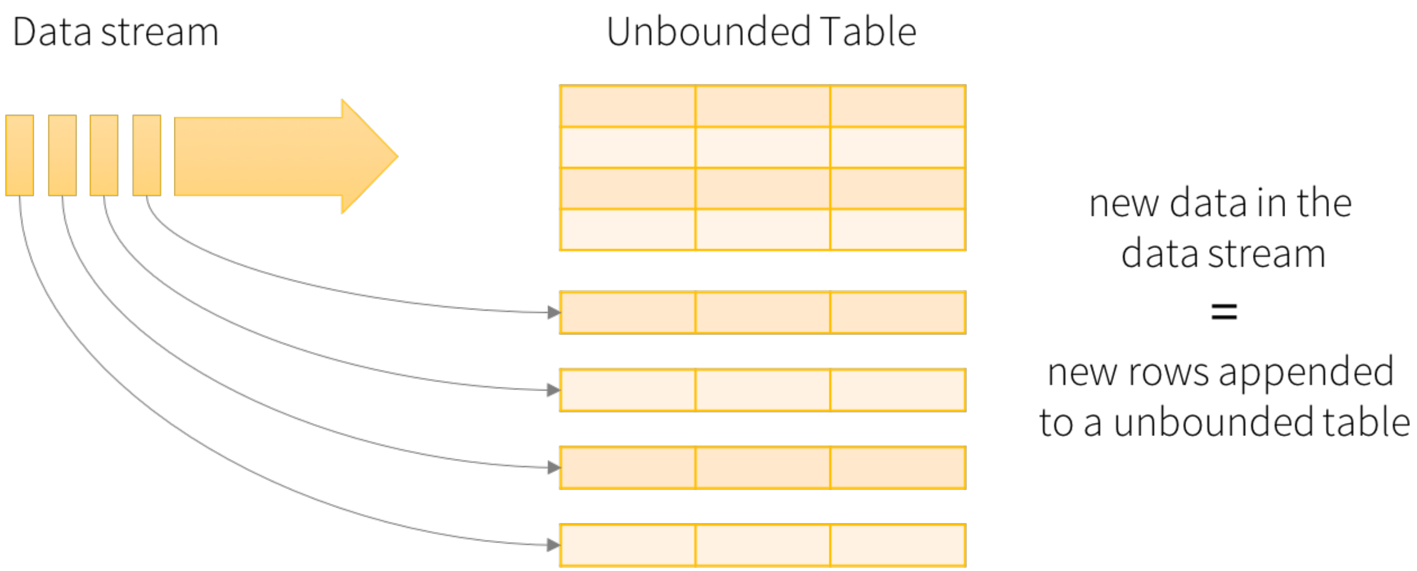
- The core concepts of this programming model are becoming a standard in stream processing
  - SQL / Table API in Flink
  - Kafka Streams / KSQL
  - …
- Spark Streaming instead, although widely adopted, exploits a programming model that is very engine-specific
  - Micro-batch approach to enable streaming computations on a pure batch-/scheduling-oriented engine

Programming model

- Key ideas
  - Consider a live stream as a table that is being continuously appended
  - Express streaming computations as standard batch-like queries on static tables
  - Spark runs the computations *incrementally* on the *unbounded* input table

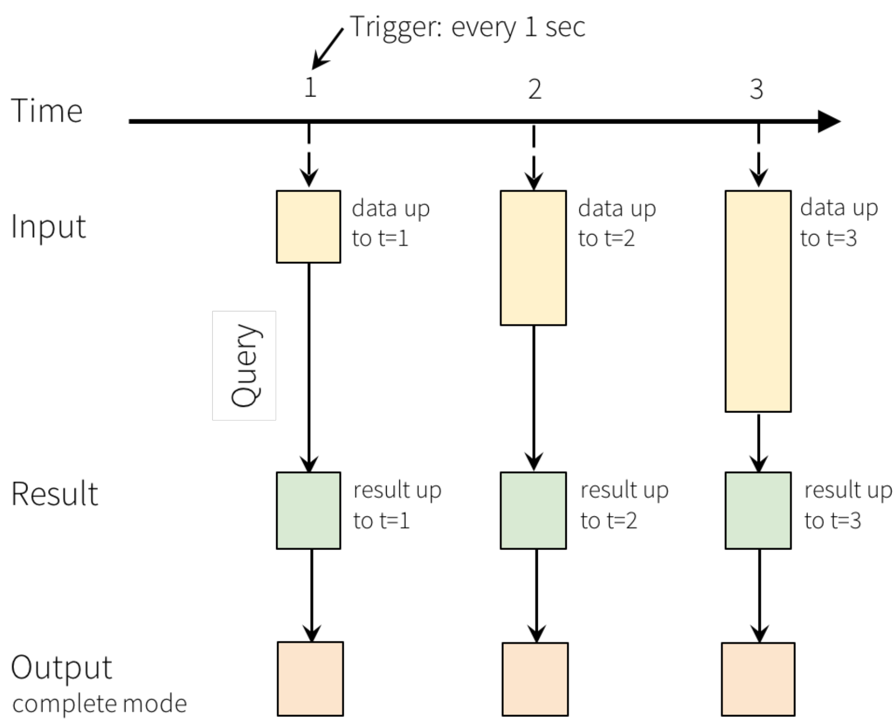
- See the blog post / lecture “Turning the databased inside out with Apache Samza” by M. Kleppmann

Programming model



Data stream as an unbounded table

Programming model



Programming Model for Structured Streaming

Programming model

- A result table / output can be defined in different modes
  - Depending on the need of the sink

1. Complete mode: returns the entire result table

2. Append mode: returns only the new rows appended to the result table since the last trigger

3. Update mode: returns only the rows that were updated in the result table since the last trigger
   - If the computation does not contain aggregations, this is equivalent to Append mode

Programming model: example

- Let us consider again the classic word count example
- To interact with the Spark SQL engine we first need a SparkSession

```
val spark = SparkSession
   .builder
   .appName("StructuredStreamingWordCount")
   .getOrCreate()
```

Programming model: example

```
val lines = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()

val words = lines
  .as[String]
  .flatMap(_.split(" "))

val wordCounts = words
  .groupBy("value")
  .count()
```

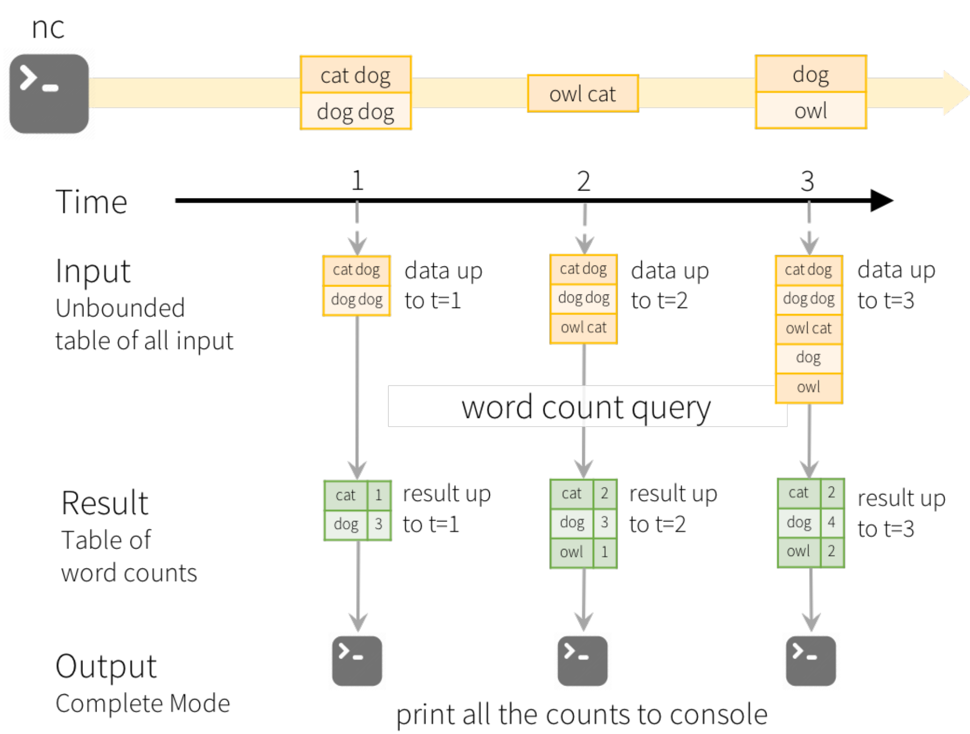
- `lines` represents an unbounded table containing streaming text data
  - One “value” column
  - Each line becomes a row in the table
- We convert the DataFrame into a Dataset of String
- `wordCounts` is again a DataFrame containing the count for each word

Programming model: example

- We can output the results using a query
  - We show the incremental computation model and the results with different modes

```
val query = wordCounts.
   writeStream
   .outputMode("complete")
   .format("console")
   .start()

query.awaitTermination()
```

Programming model: example



Model of the Quick Example

Incremental execution

- The engine does not materialize the entire table
- Instead, it *incrementally* updates the results upon receiving a new element from the input source
  - It only keeps the minimum intermediate state required to update the result
- This is possible because Structured Streaming implements a set of standard operators with well known semantics

Event-time and late data

- Some operators rely on time (e.g., windows)

- But what is the meaning of time when running Spark in a distributed environment?
  - Different nodes in the cluster have different clocks
  - Sources and sinks have yet other internal clocks
  - Network communication introduces delays when moving date into/out of the cluster and between the nodes of the cluster

Event-time and late data

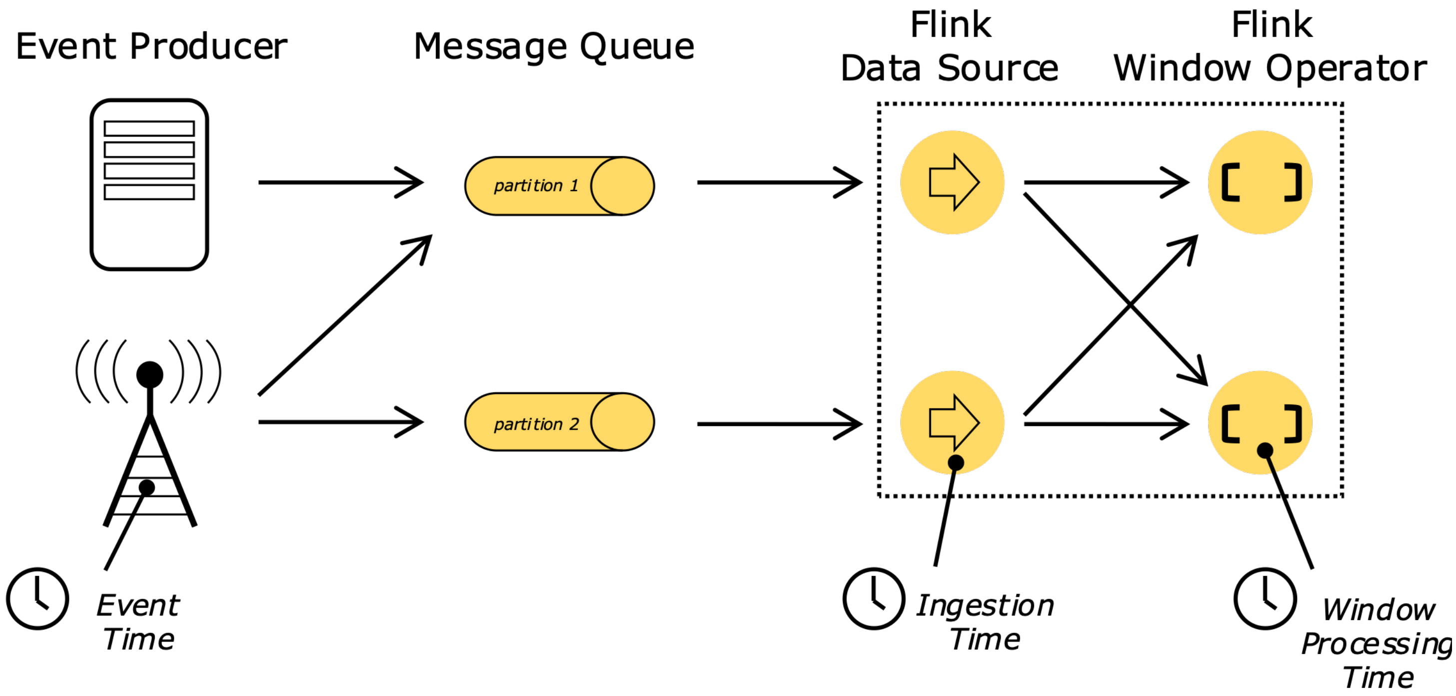
- We can identify three “definitions” of time in stream processing

1. Event time: is the time attached to a data element by its source

2. Ingestion time: is the time when a data element first enters the processing cluster

3. Processing time: is the wall clock time of the processing node

Event-time and late data



Event-time and late data

- In most applications, event time is the most significant for the users
  - It is deterministic: in the case of replay, event time does not change and leads to the same results
  - It is set by the application
  - Does not depend on runtime concerns (e.g., load of the processing nodes)

Event-time and late data

- However, event time is also the most complex to deal with
- In theory, we do not know if old messages are still coming from some source
  - We should wait forever for new messages!
- In practice, we can rely on sources to send information about time
  - *Watermarks*: a watermark with time t indicates that no further messages older than t will be received from that source

Event-time and late data

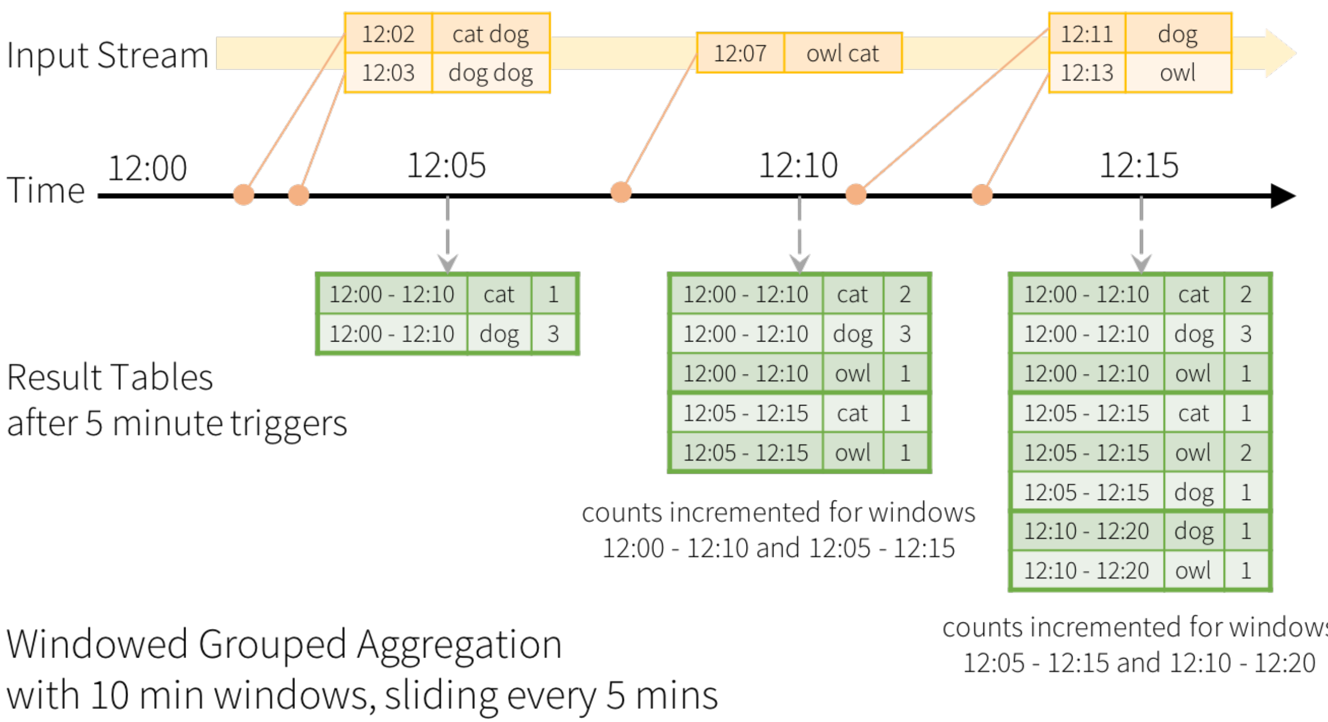
- When receiving a watermark t from all input sources, we are sure that the results up to time t are stable
  - We can safely output them

- Structured Streaming takes a different approach
  - Output results are provided immediately
  - They are changed in the case of late arrival
  - To avoid keeping old state forever, watermarks are used to limit the

Windows

- Aggregations over event-time windows are easy to express
  - Conceptually similar to grouped aggregations
  - Aggregate values are maintained for each window
  - Input data elements can fall into multiple (partially overlapping) time windows
    - They contribute to the value of all the time windows they are part of

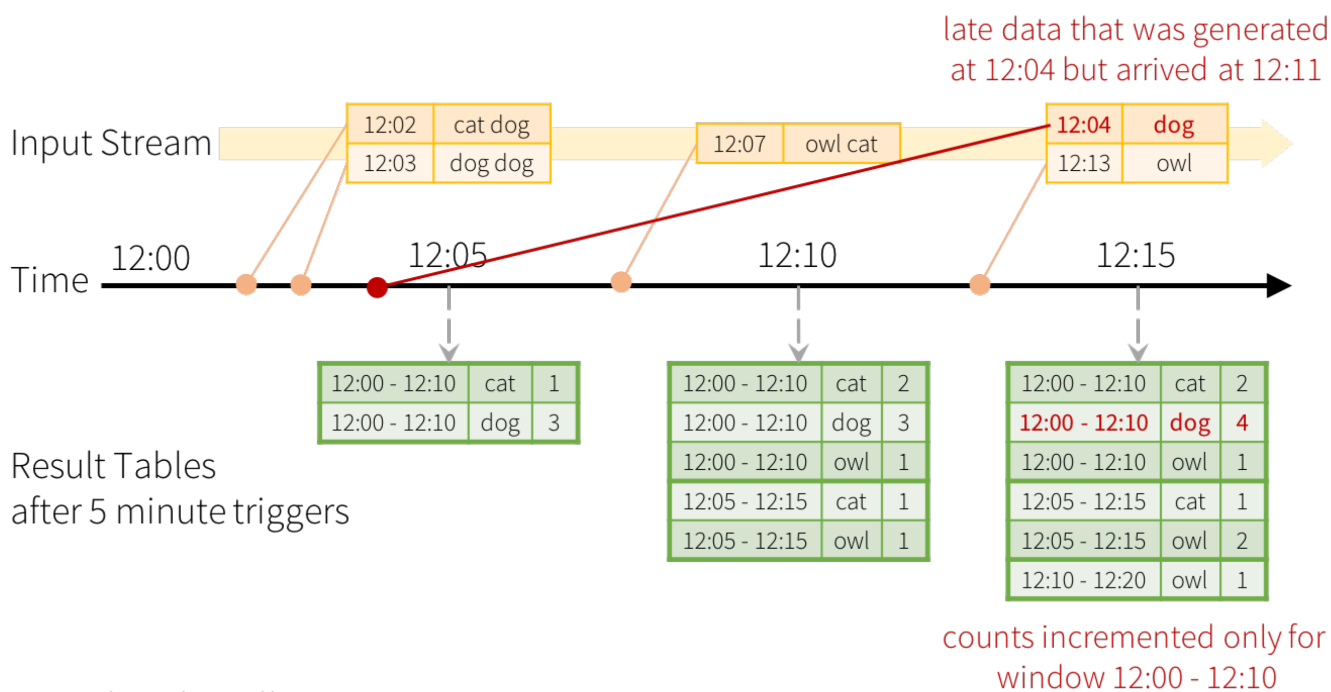
Windows


Input Stream
12:02 cat dog
12:03 dog dog
12:07 owl cat
12:11 dog
12:13 owl
Time
12:00
12:05
12:10
12:15
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
Result Tables
after 5 minute triggers
counts incremented for windows
12:00 - 12:10 and 12:05 - 12:15
counts incremented for windows
12:05 - 12:15 and 12:10 - 12:20
Windowed Grouped Aggregation
with 10 min windows, sliding every 5 mins

Windows and late data

- Window-based grouping is a good example to illustrate how Spark handles late data
- Spark maintain the intermediate state for partial aggregates for a long period of time, such that late data can update aggregates of old windows correctly
- Garbage collection of old intermediate state is handled through watermarking
  - After a threshold, late data is simply discarded

Windows and late data



Late data handling in
Windowed Grouped Aggregation

Joins

- Structured Streaming enables joining streaming datasets with static datasets as well as other streaming datasets

- The result of a streaming join is generated incrementally, similar to the results of streaming aggregations

Streaming joins: problem

- The problem with joins in streaming data is that a new element in a stream can be potentially joined with any element in another stream
  - We need to store the entire stream, …
  - … which grows without bounds over time

- Solutions
  - Specify temporal constraints (time ranges) in the join conditions
    - leftTime BETWEEN rightTime AND rightTime + INTERVAL 30 MINUTES

Streaming join: solutions

- Define explicit time constraints
  1. Time ranges in the join conditions
     E.g.
     JOIN ON leftTime
     BETWEEN rightTime AND rightTime + INTERVAL 30 MINUTES
  2. Join on event-time windows
     E.g.
     JOIN ON leftTimeWindow = rightTimeWindow

- Define watermarks on both input tables
  - The engine knows how late the data is and apply discard policies
  - Similar to streaming aggregations

Checkpointing

- We already discussed how Spark handles failures
  - End-to-end exactly once semantics
  - By replaying old streaming elements
    - From the receiver
    - From the external sources (e.g., Kafka)

- Problem: in the case of long-running queries, this requires to replay the *entire* stream
  - To restore the set of results
    - E.g., for aggregations over time-based windows
  - To restore the intermediate state for incremental computation

Checkpointing

- Spark offers the possibility to perform periodic *checkpointing*
  - Store a *snapshot* of the distributed state of the cluster with respect to a query

- Upon failure
  - The state of the cluster is restored from the last valid snapshot
  - The computation restarts from the streaming elements that were not part of the snapshot
    - Replay from receivers or from sources

Checkpointing

- Developers can specify a checkpoint location for each query
  - The checkpoint location must be in a HDFS-compatible file system

```
val query = streamData
    .writeStream
    .outputMode("complete")
    .option("checkpointLocation", "path/to/HDFS/dir")
    .format("memory")
    .start()
```

Continuous mode

- Since Spark 2.3, structured streaming also supports a continuous processing mode
  - Similar to other platforms such as Apache Storm and Apache Flink
- Operators are deployed and not scheduled
- They directly exchange messages over the network, …
- … rather than accessing the cache of the nodes they work on
  - Memory or disk
- While still experimental, this processing mode enables low delay processing
  - Order of milliseconds

Questions?