SINGAPORE DATA SCIENCE CONSORTIUM

NUS National University of Singapore

School of Computing

Stream Data Processing with Apache Spark

DataFrames & Structured Streaming

Dr. Salman Salloum

Research Fellow, NUS School of Computing

www.linkedin.com/in/ssalloum/

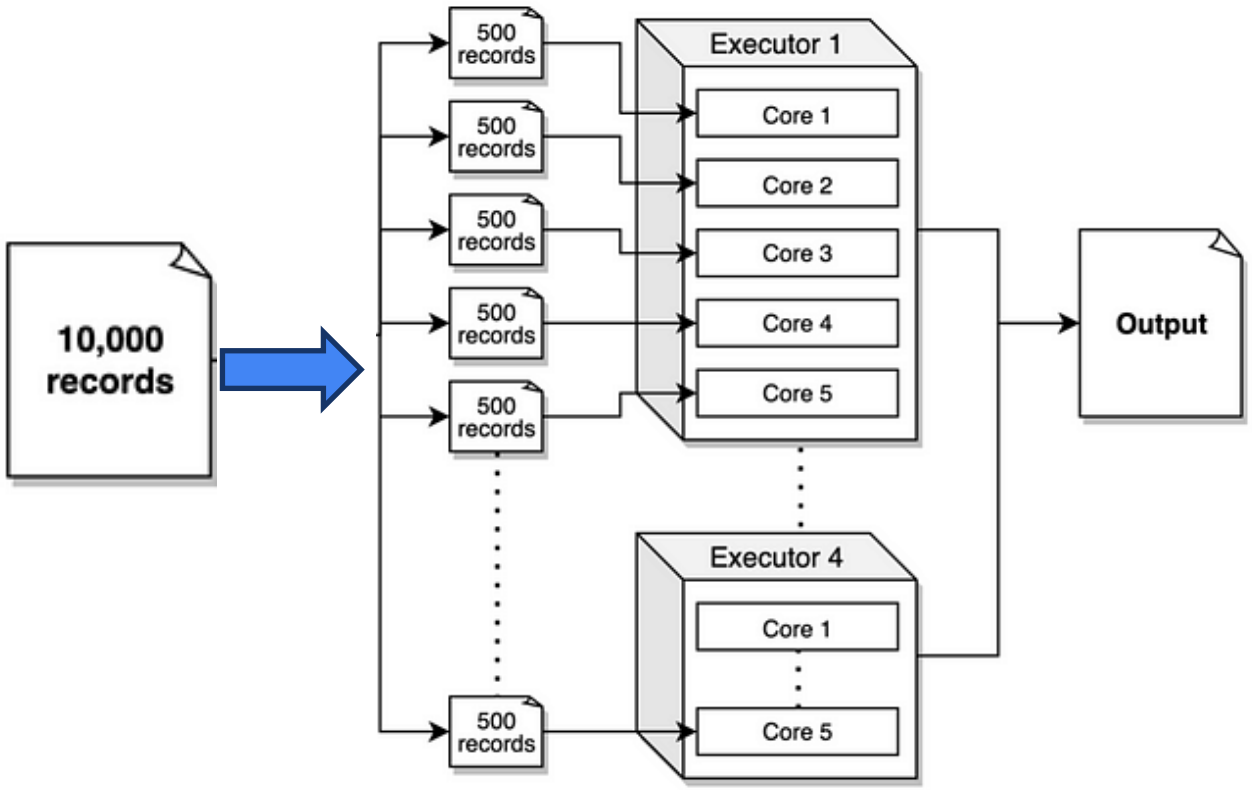
Thursday, 07 December 2023

Agenda

Lecture 1: • Overview of Apache Spark	9:30 - 10:30
Break 1	10:30 - 11: 00
Lecture 2: • Spark DataFrames • Spark Structured Streaming	11:00 - 12:30
Break 2	12:30 - 13:30
Lab 1: (Google Colab) • PySpark & DataFrames • Streaming DataFrames & Streaming Queries	13:30 - 15:30
Break 3	15:30 - 16:00
Lab 2: (Databricks) • Spark Clusters on Databricks • Operations on Streaming DataFrames: Aggregations	16:00 - 17:30

Overview of Apache Spark

Big Data Processing



[The Big Picture]

Nicole Lazar
Column Editor

Divide and Combine to Conquer Big Data

I've recently been reading more and hearing more talks about an approach called "Divide and Combine" (sometimes "Divide and Recombine"), which pops up in various guises and contexts. "Divide and conquer" algorithms in computer science are one aspect of this, but there is more to it than mere parallelization of computing.

The starting point is that it may be hard, or impossible, to analyze a full data set because of its size. In some extreme cases, it may not even be feasible to store the entire data set on disk, let alone manipulate it for statistical analysis. With clever application of a parallel approach, though (on the statistical side, rather than solely the computing), it is sometimes possible to break the data into manageable chunks (the "Divide" part), perform the analysis on

Meta-analysis. In a 2010 paper, Lin and Zeng compare meta-analysis based on complete subject data to that based on summary statistics from each study. Although not strictly "divide and combine" in the fashion I've described, this study is interesting because it examines the benefit that could be gained from acquiring the full data. As the authors note, it is potentially easier these days to actually obtain the complete data from each of the studies involved in a meta-analysis because of advances in technology, data sharing protocols, etc.

In some sense, this is an inverse divide-and-combine—think of the original data on all of the subjects across all of the studies in the meta-analysis as one large data set. Then the subjects are "divided" into the K studies in the meta-analysis and analyzed separately. Meta-analysis "combines" the results back. Of

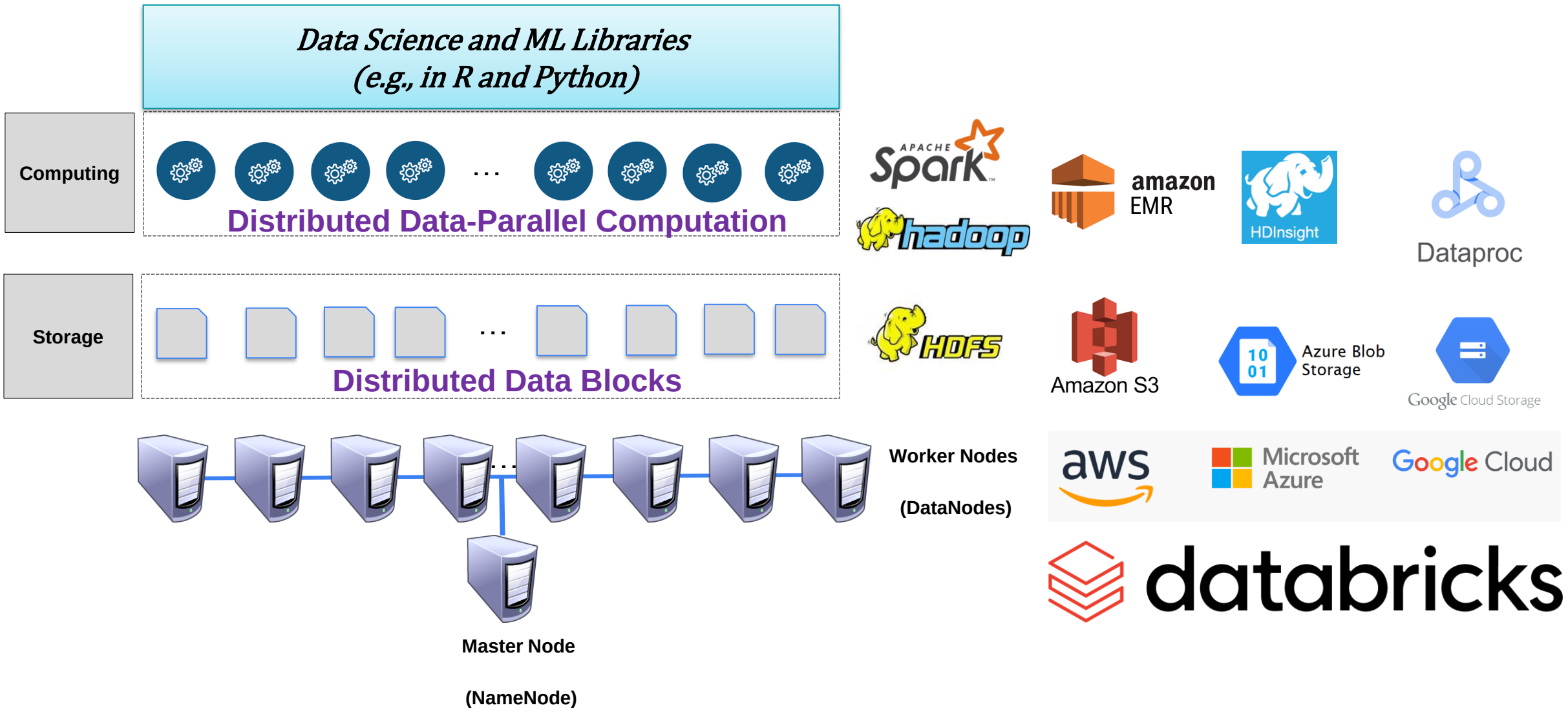
to each study; in the regression example, maybe each study has a different intercept.

Continuing with this example, the typical approach would be to obtain estimates of the regression coefficient of interest (the slope) from each study individually, and then combine those into one overall estimate, usually by a weighted average. The weights are given by the inverses of the estimated variances from each study. In this way, studies with more precision receive more weight in the combination.

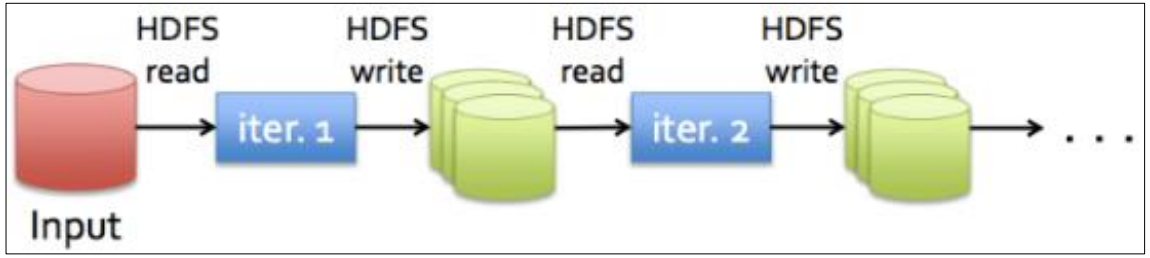
The fundamental result is that based on maximum likelihood estimation of the effects of interest, the large-sample distribution is the same whether one analyzes the complete (all subject) data, or performs meta-analysis from the summary statistics derived for each study. Hence, there is no loss of efficiency from using the summary

https://www.tandfonline.com/doi/abs/10.1080/09332480.2018.1438711

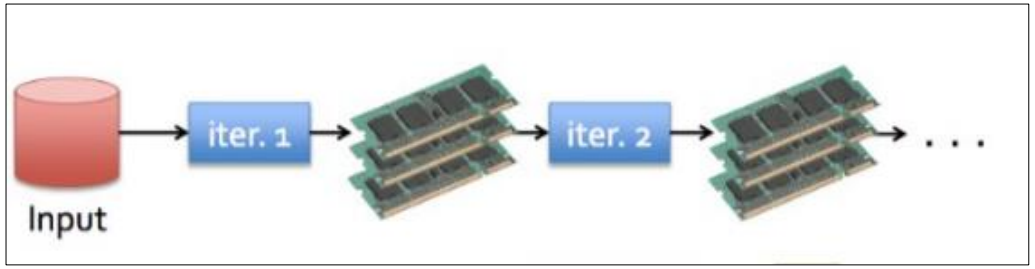
Big Data Platforms



Spark at UC Berkeley (AMPLab)



Iterative processing with Hadoop MapReduce



Iterative processing with Spark

Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.

- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it

https://spark.apache.org/research.html

Apache Spark

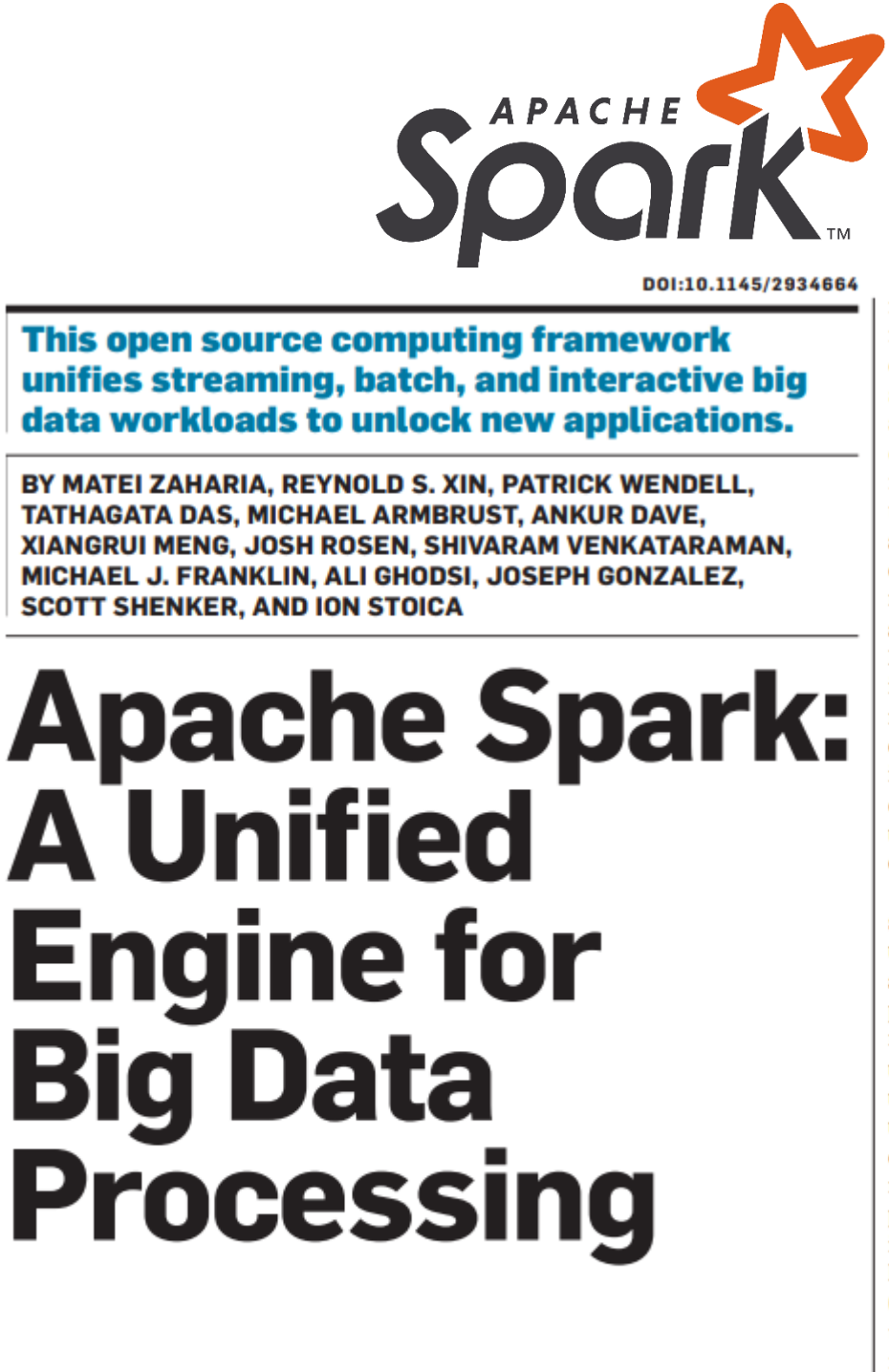
- 2013: The AMPLab contributed Spark to the Apache Software Foundation and launched Databricks
- 2014: Spark 1.0
- 2016: Spark 2.0
- 2020: Spark 3.0
- Sep 2023: Spark 3.5

SIGMOD Systems Award for Apache Spark

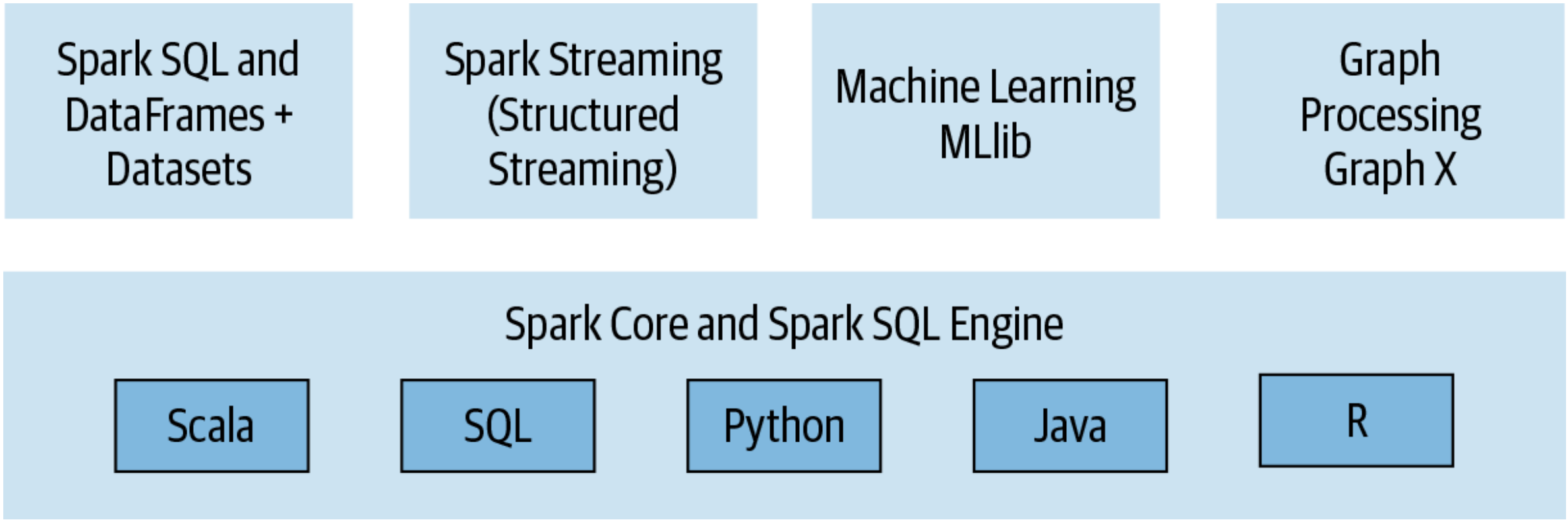
Apache Spark received the SIGMOD Systems Award this year, given by SIGMOD (the ACM's data management research organization) to impactful real-world and research systems:

> The 2022 ACM SIGMOD Systems Award goes to "Apache Spark", an innovative, widely-used, open-source, unified data processing system encompassing relational, streaming, and machine-learning workloads.

https://spark.apache.org/news/sigmod-system-award.html

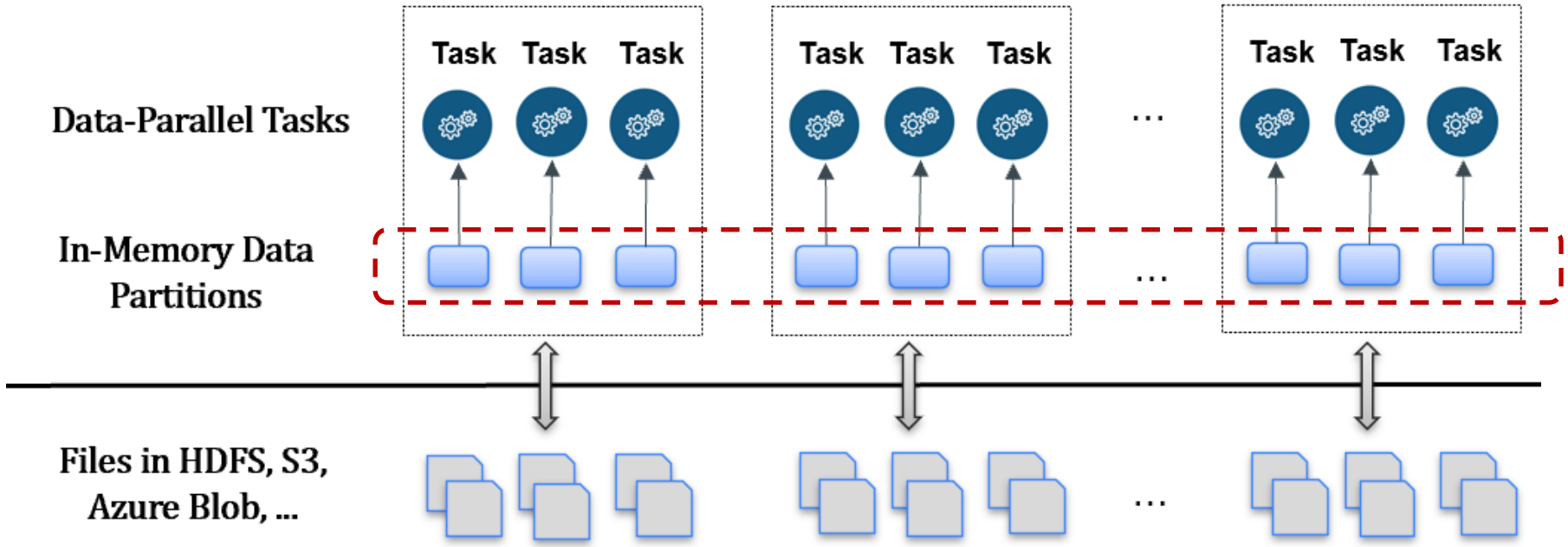


Apache Spark Components



Apache Spark is a unified computing engine designed for large-scale distributed data processing, on single-node machines, on-premises clusters, or in the cloud.

In-Memory Distributed Data Processing

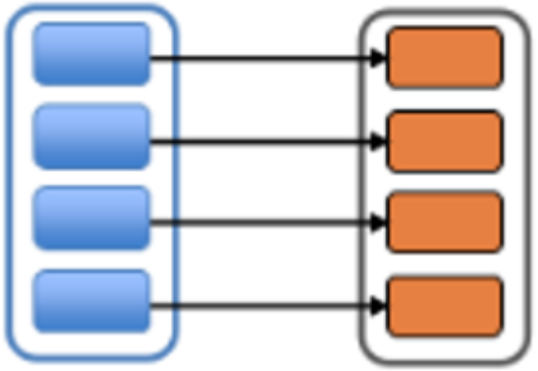


Resilient Distributed Dataset (RDD)

Transformations and Actions

- Transformations are simple ways of specifying a series of data manipulation.
- Two types of transformations: narrow transformations, and wide transformations.
- Spark computes transformations only in a lazy fashion.
- Actions trigger the computation in Spark.
- Actions can be used to view/collect/write the output data after a series of transformations.

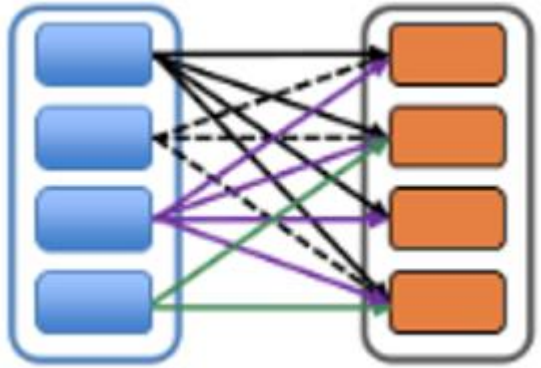
Narrow Dependencies



Each input partition contributes to only one output partition

```
map()
filter()
select()
mapPartition()
```

Wide Dependencies

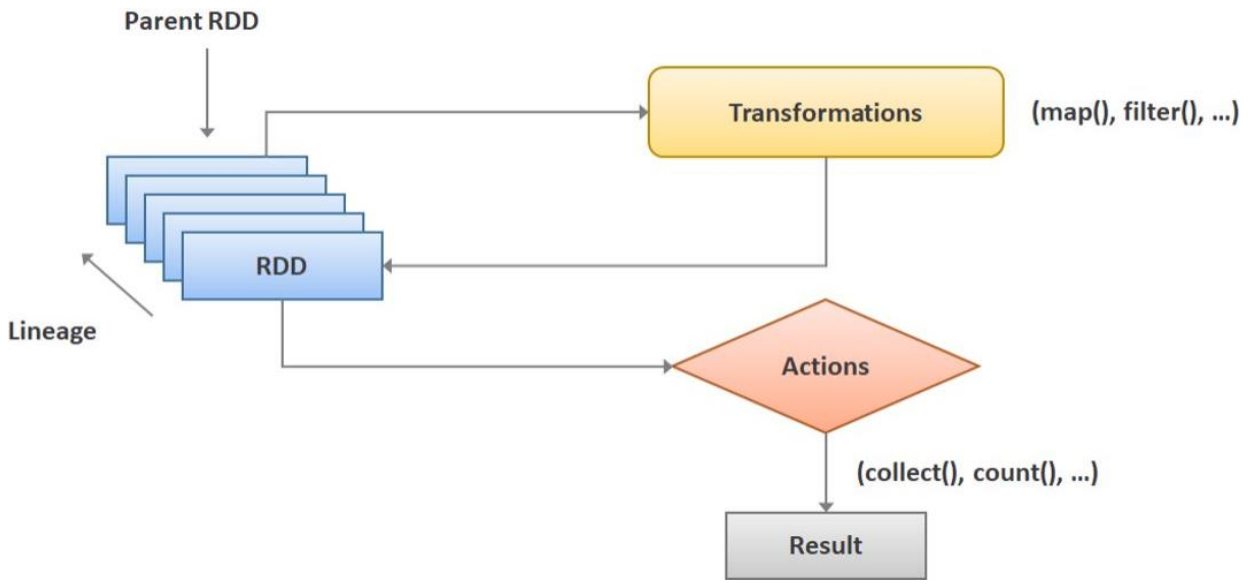


Each input partition may contribute to many output partitions

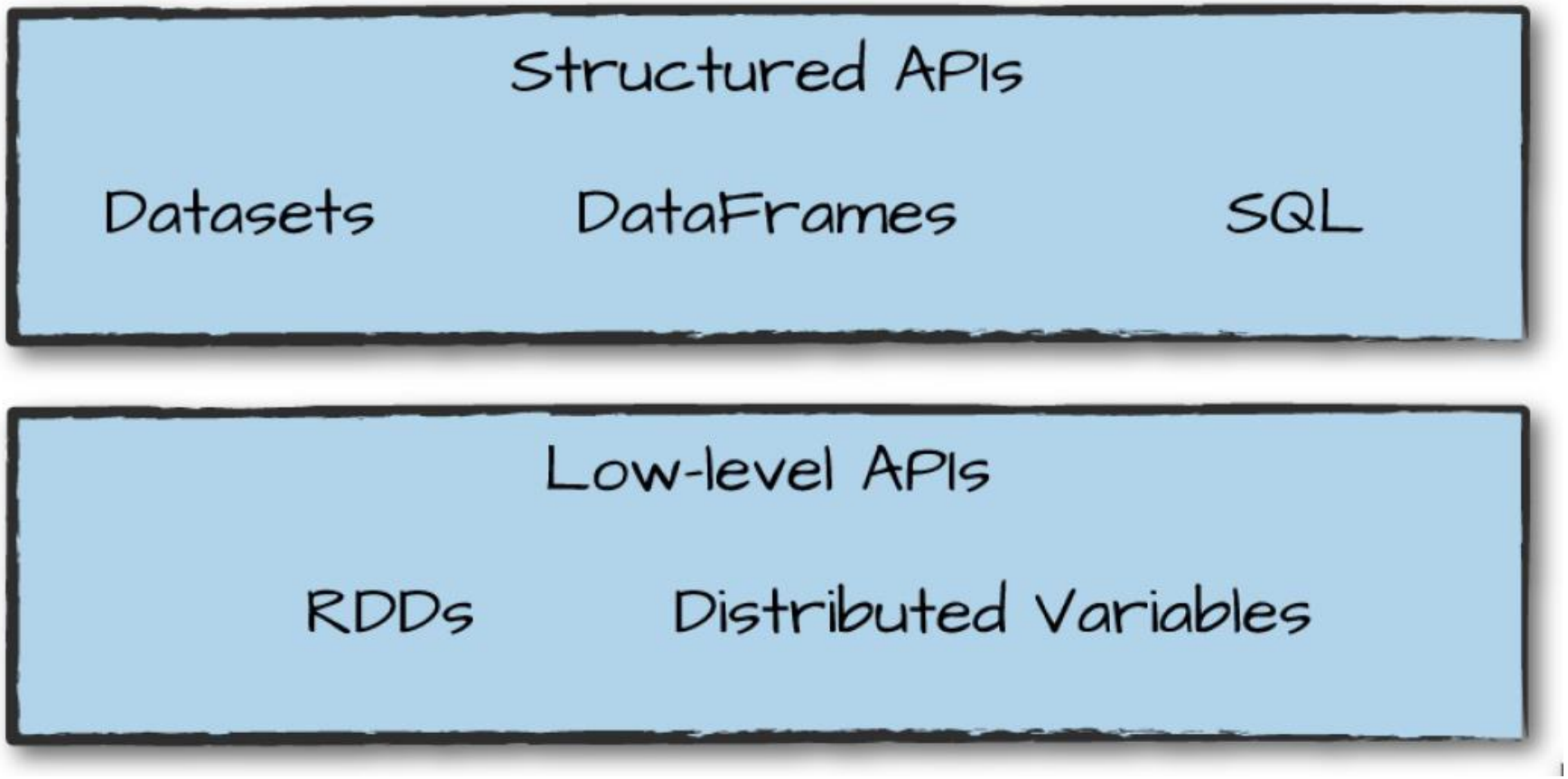
```
groupByKey()
join()
sort()
repartition()
```

Lazy Evaluation

- Lazy evaluation is Spark's strategy for delaying execution until an action is invoked.
- This allows spark to optimize the execution plan.
- Transformations are recorded in a lineage.
- Lineage and immutability provide fault- tolerance.

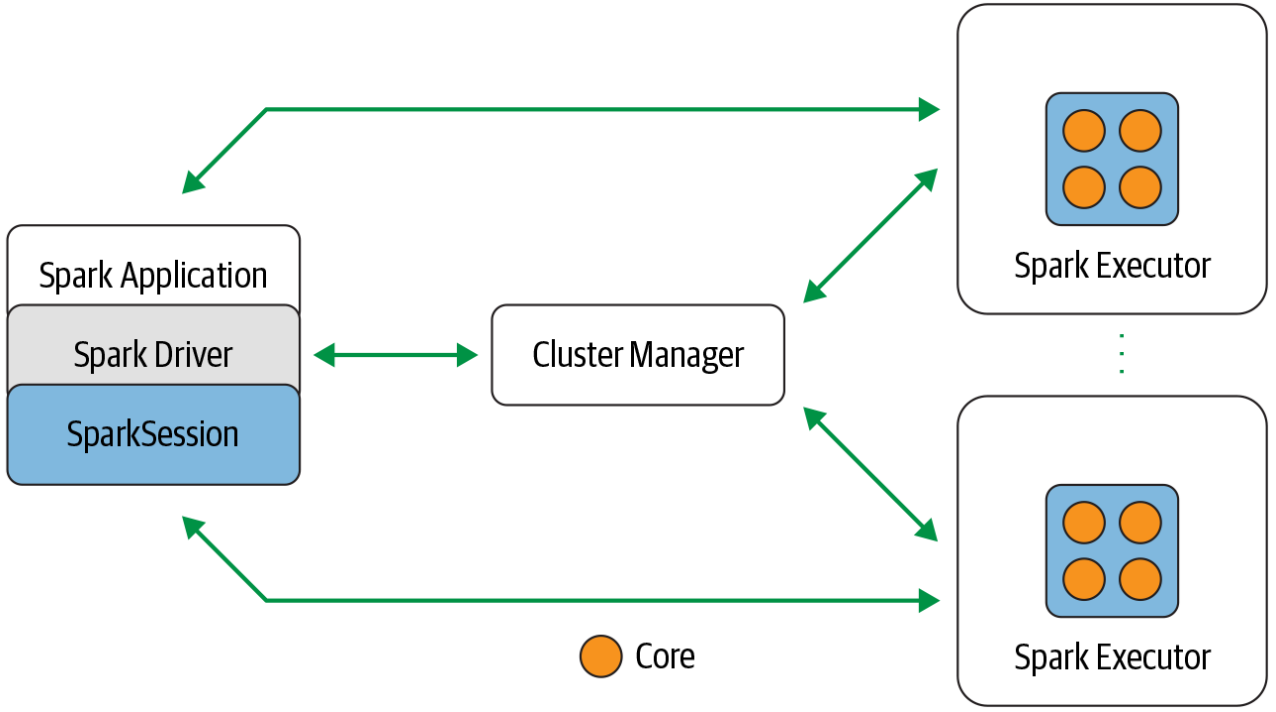


Spark APIs


Structured APIs
Datasets
DataFrames
SQL
Low-level APIs
RDDs
Distributed Variables

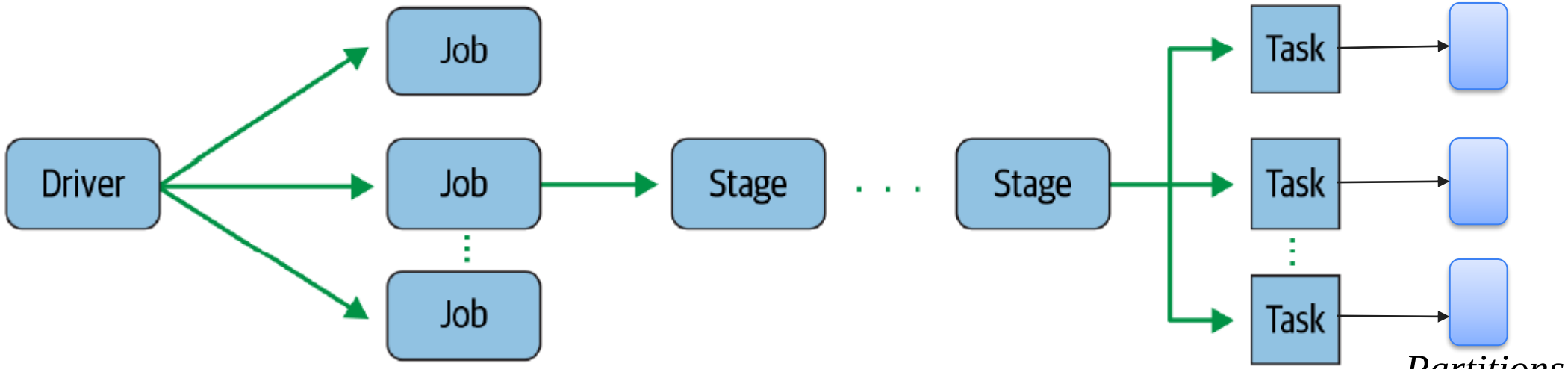
Spark's Distributed Architecture

- Spark Driver: orchestrating parallel operations on the Spark cluster.
- Spark Session: a unified entry point to all Spark functionality.
- Spark Executors: running the tasks assigned by the driver.
- Cluster Manager: allocating resources to Spark Applications.

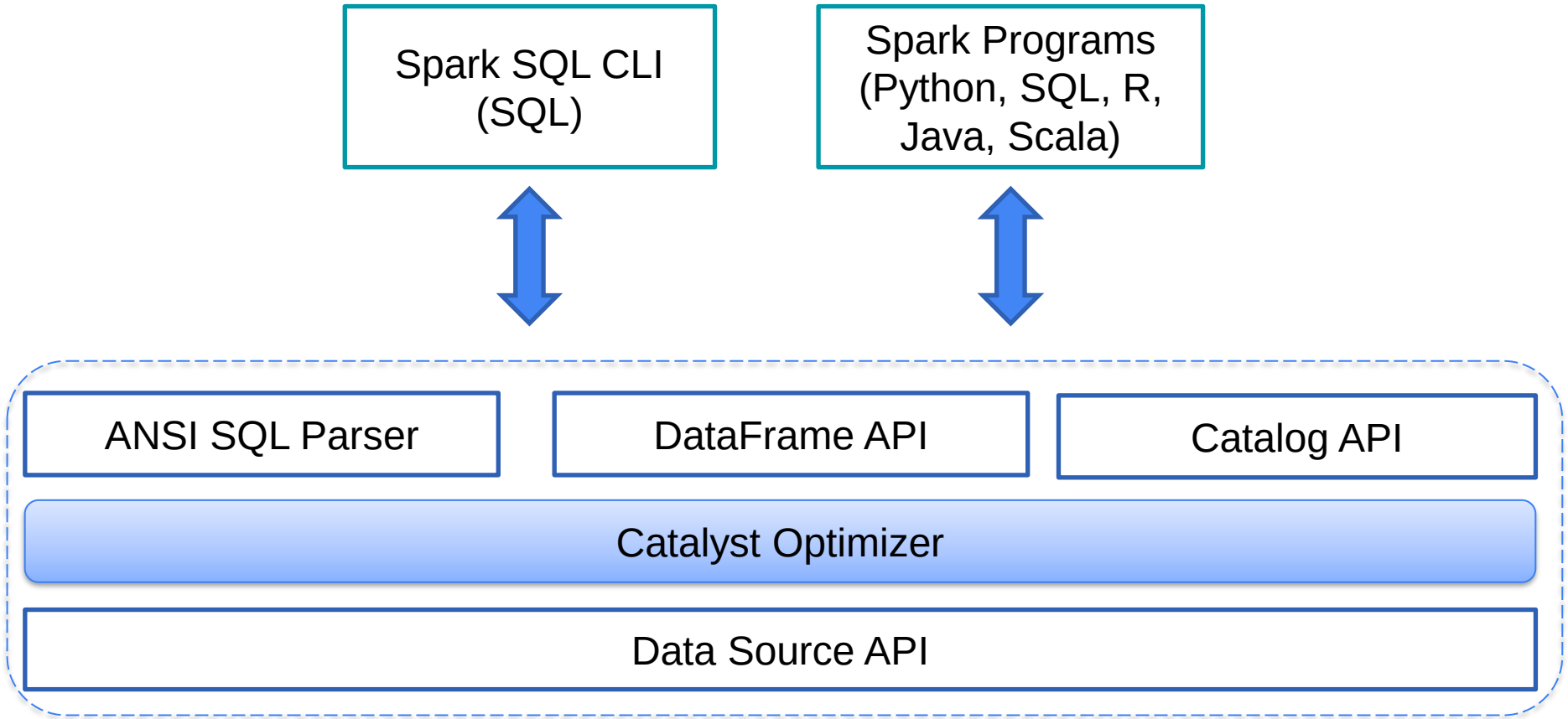


Anatomy of a Spark Application

- Each Spark Application is made up of one or more Spark Jobs.
- A Spark Job is a parallel computation that is triggered in response to a Spark action.
- Each job gets divided into smaller stages that depend on each other.
- Each Stage consists of a set of tasks
- Spark Task is a single unit of work or execution that will be sent to a Spark Executor.
- Each Spark Task is assigned a data partition

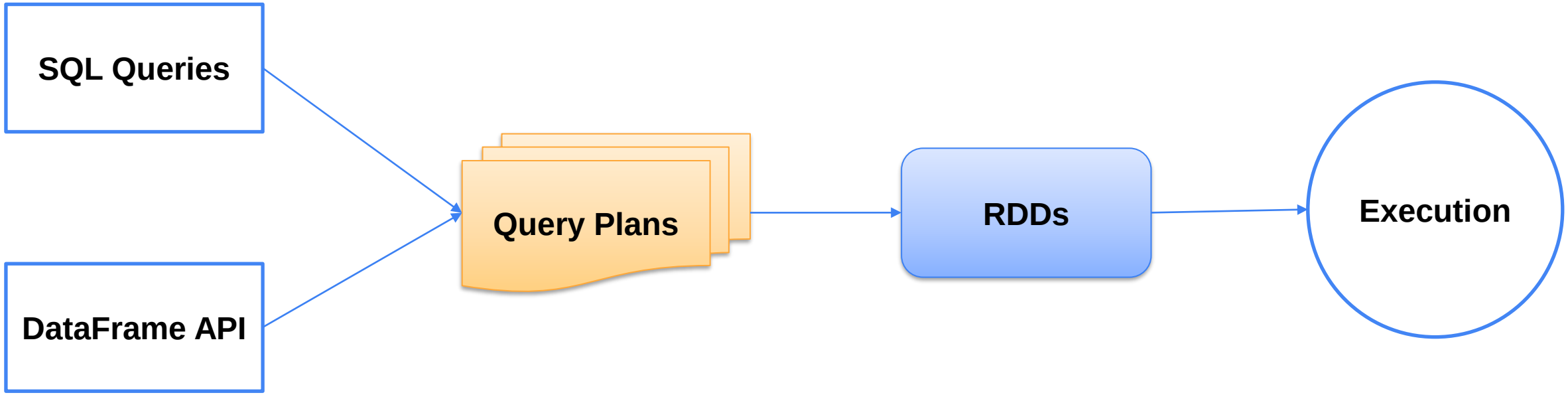


Spark SQL: Main Components



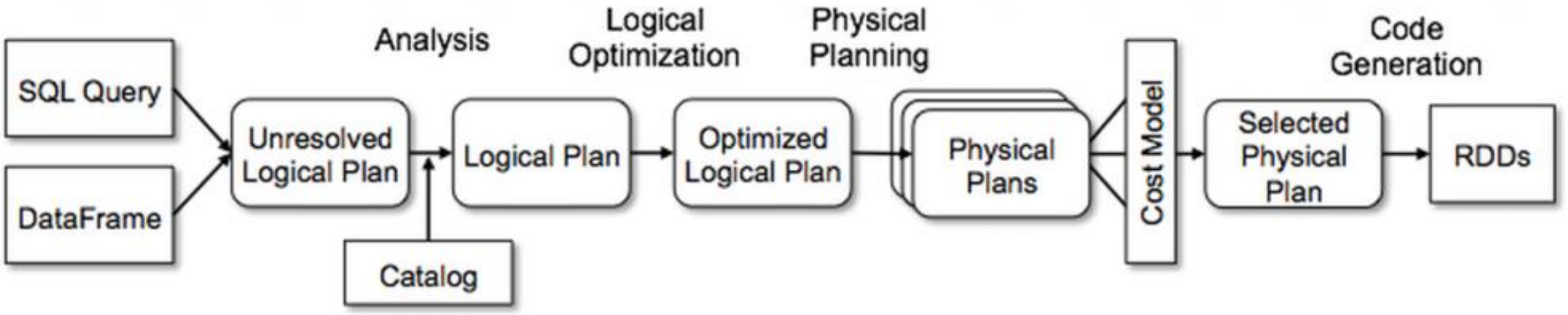
Spark SQL: Execution

- All queries/operations are optimized and executed in the same way



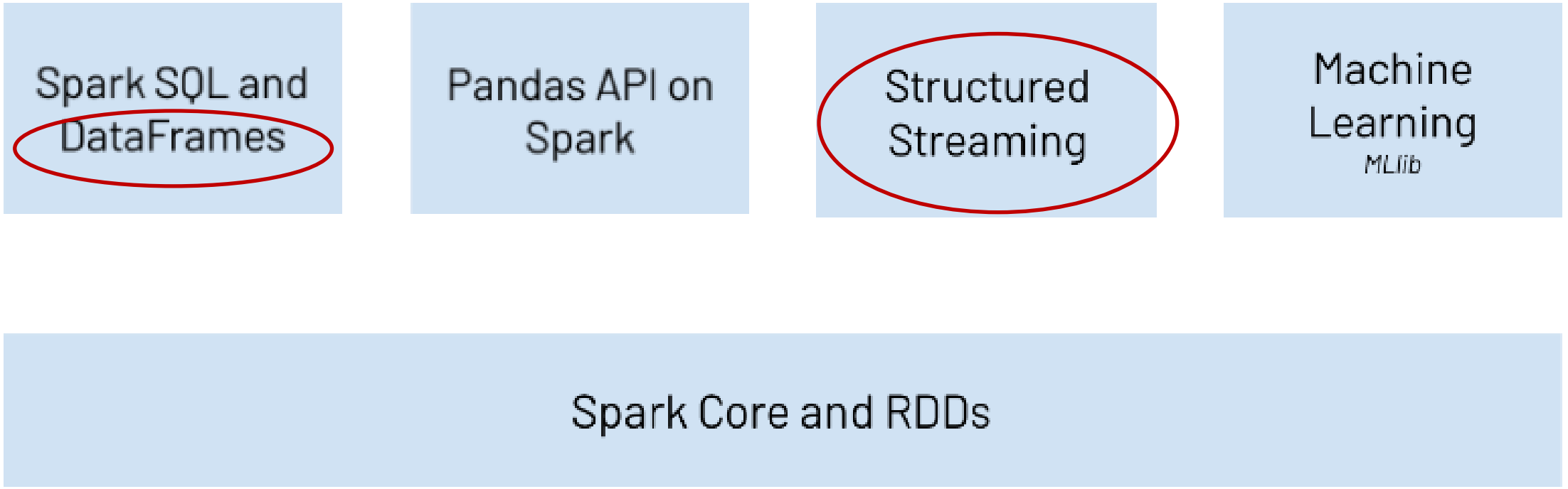
Spark SQL: The Catalyst Optimizer

- It converts queries/DataFrame operations into an execution plan



https://www.databricks.com/glossary/catalyst-optimizer

PySpark: The Python API for Apache Spark



https://spark.apache.org/docs/latest/api/python/index.html

PySpark: The Python API for Apache Spark

- Install as any Python package
- PySpark Shell

```
!pip install pyspark
```

SparkSession

```
[ ]  # Import SparkSession
     from pyspark.sql import SparkSession

     # Create a Spark Session
     spark = SparkSession.builder\
             .master("local[2]")\
             .appName("Hello Spark")\
             .config('spark.ui.port', '4050')\
             .getOrCreate()
```

https://spark.apache.org/downloads.html
https://spark.apache.org/docs/latest/quick-start.html

Spark in the Cloud

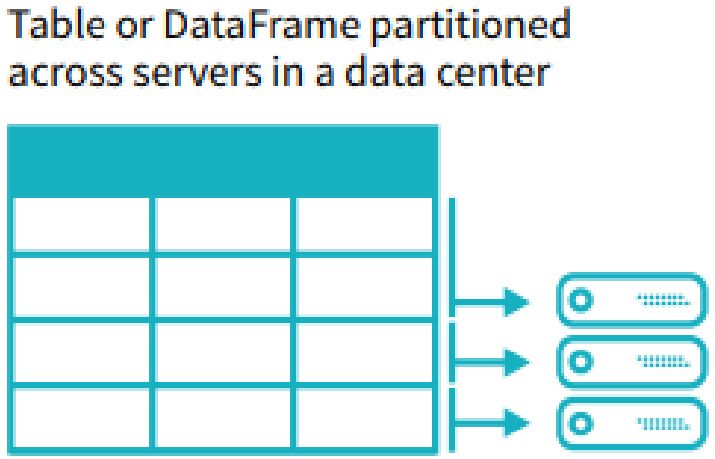
- Managed Spark environments

- Spark is a core computing engine in the Lakehouse Architecture

Spark DataFrames

DataFrames: Distributed Data Representation

- A Spark DataFrame is a distributed collection of data organized into named columns.
- It is conceptually equivalent to a table in a relational database or a data frame in R/Python.
- It is divided into chunks of data called *partitions.*
- A partition is a set of rows that sit on one machine.
- A DataFrame is like RDD, but it has a structure
- DataFrames benefit from the Catalyst optimizer



DataFrames: Schema

```
+-------+---------+-------+-----+
|Book_Id|Book_Name| Author|Price|
+-------+---------+-------+-----+
|      1|      PHP| Sravan|  250|
|      2|      SQL|Chandra|  300|
|      3|   Python| Harsha|  250|
|      4|        R| Rohith| 1200|
|      5|   Hadoop| Manasa|  700|
+-------+---------+-------+-----+
```

```
root
 |-- Book_Id: long (nullable = true)
 |-- Book_Name: string (nullable = true)
 |-- Author: string (nullable = true)
 |-- Price: long (nullable = true)
```

- A Schema defines the column names and types of a DataFrame
- Infer schema (e.g., JSON, CSV)
- Extract schema (e.g., Parquet)
- Define schema programmatically
    - Schema is a StructType: it can be saved as JSON and loaded later
- It is advisable to save DataFrames in structured formats that keeps the schema (e.g., parquet) for future use

DataFrames: Transformations and Actions

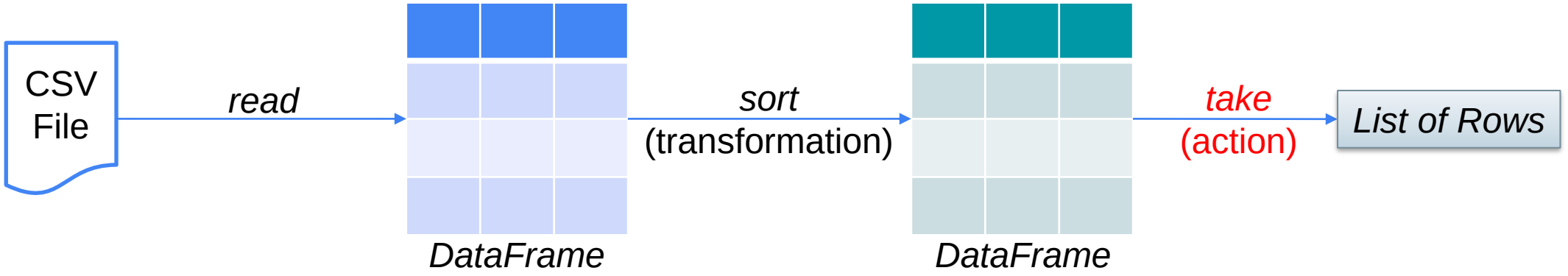
- DataFrame transformations are methods that return a new DataFrame and are lazily evaluated.
- DataFrame actions are methods that trigger computation.
- An action is needed to trigger the execution of any DataFrame transformation.

```
df.select("id", "result")
  .where("result > 70")
  .orderBy("result")
```

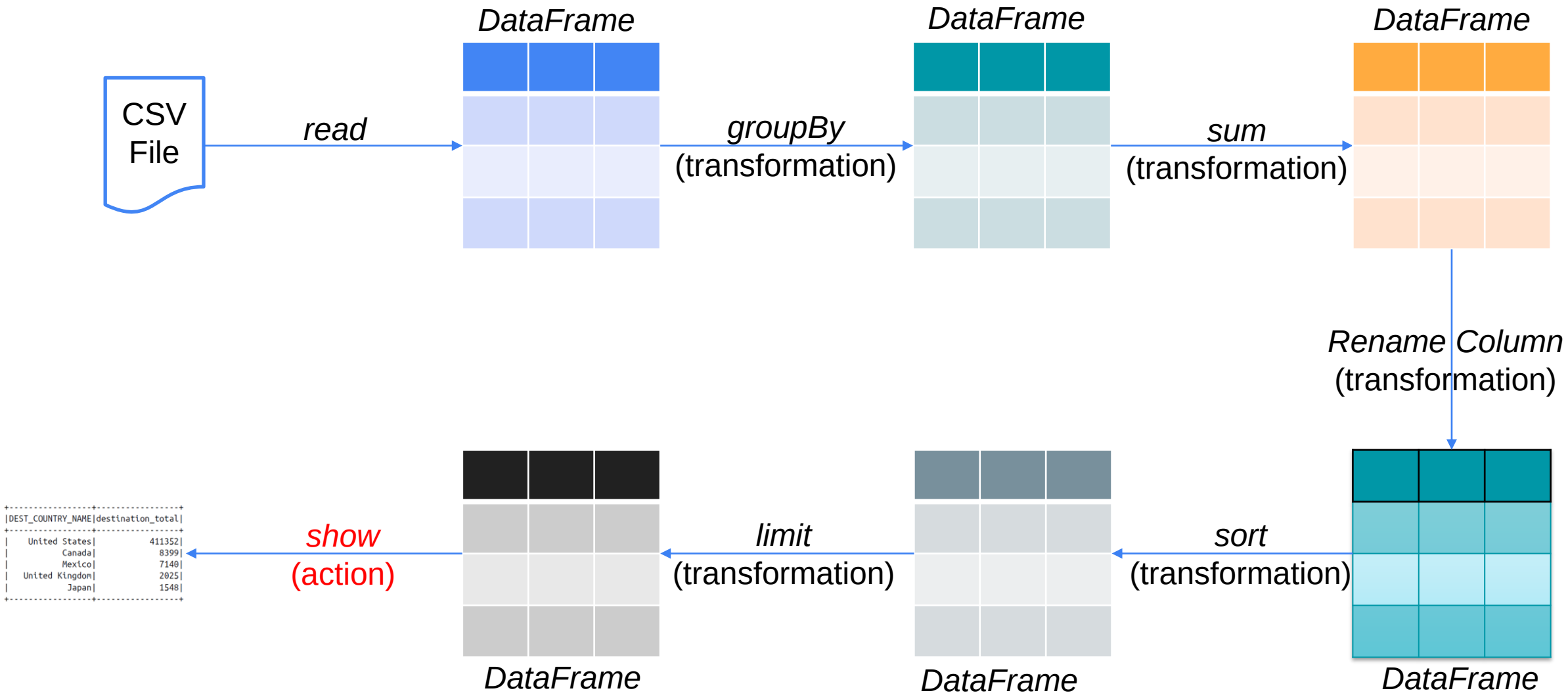
```
df.count()
df.collect()
df.show()
```

```
df.select("id", "result")
  .where("result > 70")
  .orderBy("result")
  .show()
```

DataFrames: Transformations and Actions

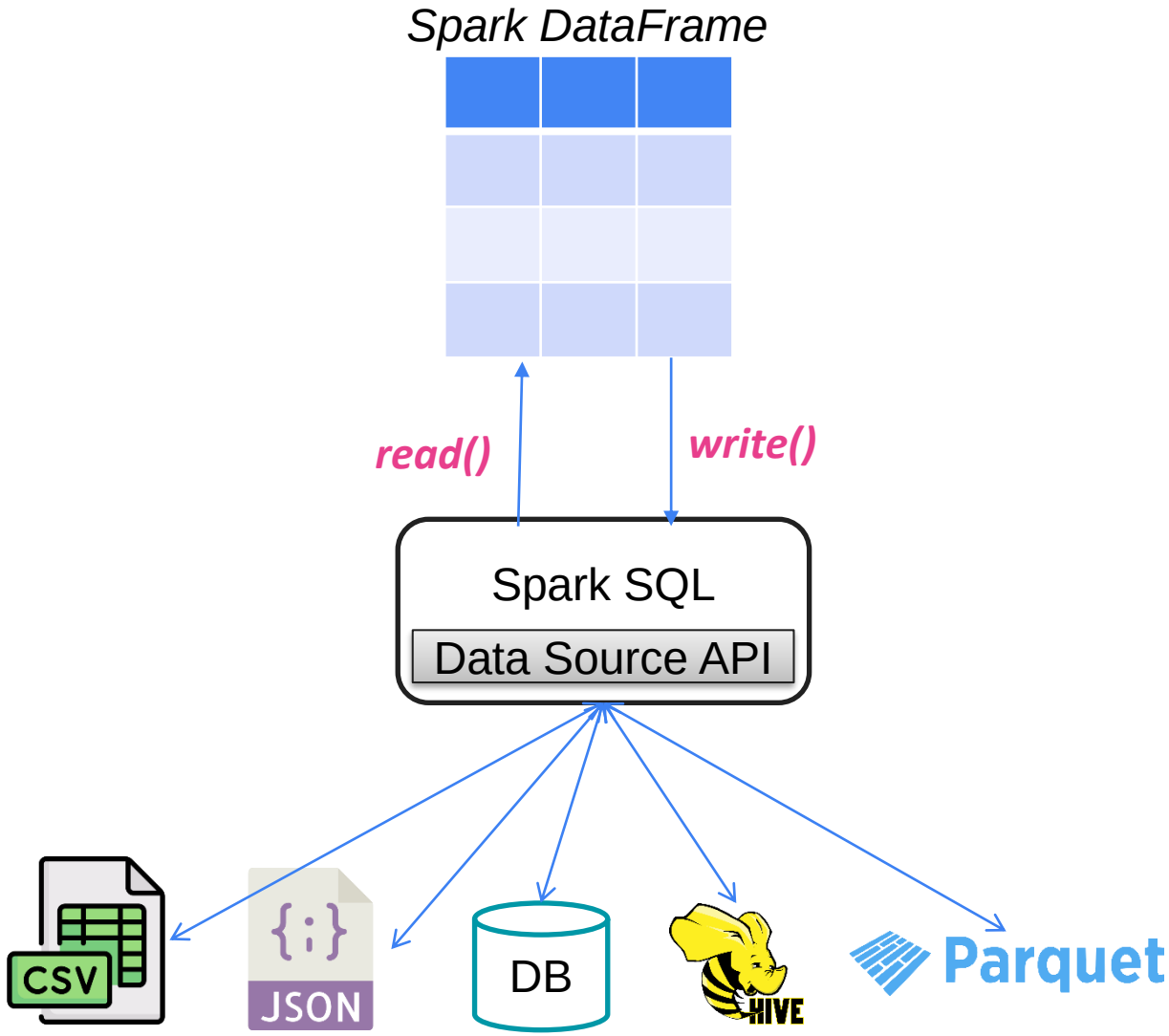


DataFrames: Transformations and Actions



Data Sources

- Spark SQL supports many data sources
- The default data source in Spark is Parquet (a columnar data file format)
- Image data source
  - Each image is represented as BinaryType with additional columns
- Binary file data source
  - Each binary file is represented as a BinaryType with additional columns



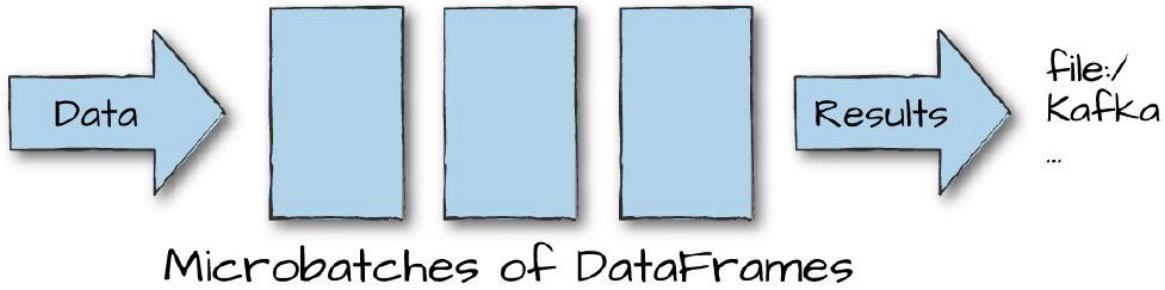
Spark SQL: PySpark APIs

- pyspark.sql.SparkSession
- pyspark.sql.DataFrame
- pyspark.sql.DataFrameReader
- pyspark.sql.DataFrameWriter
- pyspark.sql.Column
- pyspark.sql.Row
- pyspark.sql.GroupedData
- pyspark.sql.Catalog
- pyspark.sql.types
- pyspark.sql.functions

Spark Structured Streaming

(Fundamental Concepts)

Structured Streaming



- Scalable and fault-tolerant distributed stream processing engine built on the Spark SQL engine.
- Streaming computation can be expressed the same way as batch computation on static data (with the DataFrame API).
- Spark SQL engine will take care of running it incrementally and continuously and updating the result as streaming data continues to arrive.
- This system ensures end-to-end exactly-once fault-tolerance guarantees through checkpointing and Write-Ahead Logs.

Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark

Michael Armbrust†, Tathagata Das†, Joseph Torres†, Burak Yavuz†, Shixiong Zhu†, Reynold Xin†, Ali Ghodsi†, Ion Stoica†, Matei Zaharia†‡
†Databricks Inc., ‡Stanford University

Abstract

With the ubiquity of real-time data, organizations need streaming systems that are scalable, easy to use, and easy to integrate into business applications. Structured Streaming is a new high-level streaming API in Apache Spark based on our experience with Spark Streaming. Structured Streaming differs from other recent streaming APIs, such as Google Dataflow, in two main ways. First, it is a purely *declarative* API based on automatically incrementalizing a static relational query (expressed using SQL or DataFrames), in contrast to APIs that ask the user to build a DAG of physical operators. Second, Structured Streaming aims to support *end-to-end* real-time applications that integrate streaming with batch and interactive analysis. We found that this integration was often a key challenge in practice. Structured Streaming achieves high performance via Spark SQL's code generation engine and can outperform Apache Flink by up to 2× and Apache Kafka Streams by 90×. It also offers rich operational features such as rollbacks, code updates, and mixed streaming/batch execution. We describe the system's design and use cases from several hundred production deployments on Databricks, the largest of which process over 1 PB of data per month.

ACM Reference Format:
M. Armbrust et al.. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3183713.3190664

with Spark Streaming [37], one of the earliest stream processing systems to provide a high-level, functional API. We found that two challenges frequently came up with users. First, streaming systems often ask users to think in terms of complex physical execution concepts, such as at-least-once delivery, state storage and triggering modes, that are unique to streaming. Second, many systems focus *only* on streaming computation, but in real use cases, streaming is often part of a larger business application that also includes batch analytics, joins with static data, and interactive queries. Integrating streaming systems with these other workloads (e.g., maintaining transactionality) requires significant engineering effort.

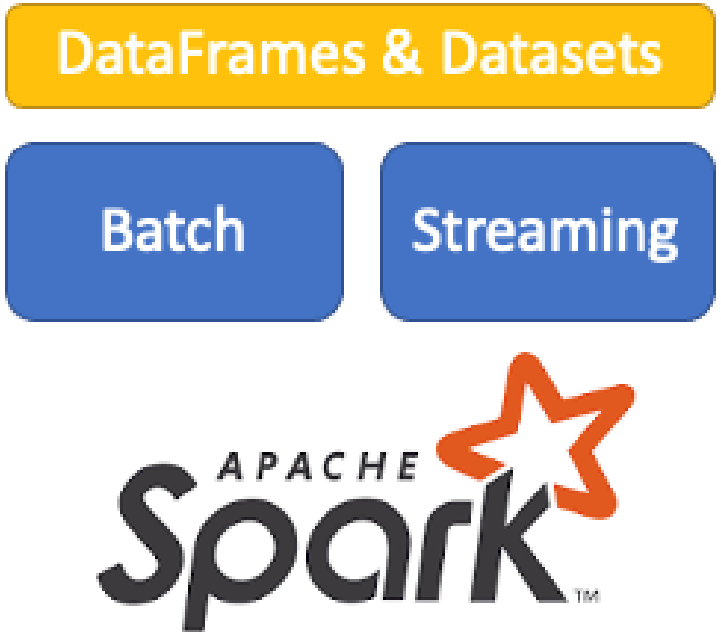
Motivated by these challenges, we describe Structured Streaming, a new high-level API for stream processing that was developed in Apache Spark starting in 2016. Structured Streaming builds on many ideas in recent stream processing systems, such as separating processing time from event time and triggers in Google Dataflow [2], using a relational execution engine for performance [12], and offering a language-integrated API [17, 37], but aims to make them simpler to use and integrated with the rest of Apache Spark. Specifically, Structured Streaming differs from other widely used open source streaming APIs in two ways:

- **Incremental query model:** Structured Streaming automatically incrementalizes queries on static datasets expressed through Spark's SQL and DataFrame APIs [8], meaning that users typically only need to understand Spark's batch APIs to write a

https://www.databricks.com/sites/default/files/2023-08/structured-streaming-a-declarative-api-for-real-time-applications-in-apache-spark.pdf

Unified Batch and Streaming APIs

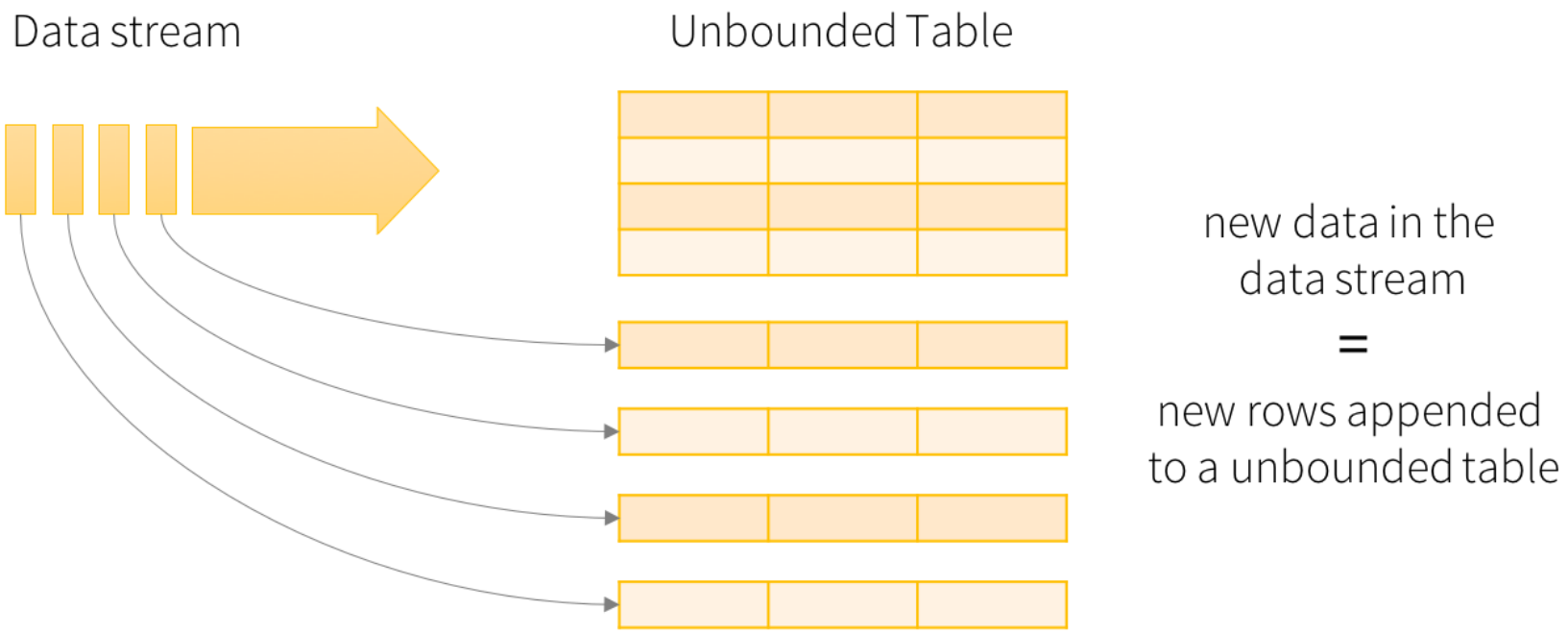
- **Build streaming applications and pipelines with the same Spark DataFrame API**
- **Programming Guide:** https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
- **PySpark API Reference:** https://spark.apache.org/docs/latest/api/python/reference/pyspark.ss/index.html



https://spark.apache.org/streaming/

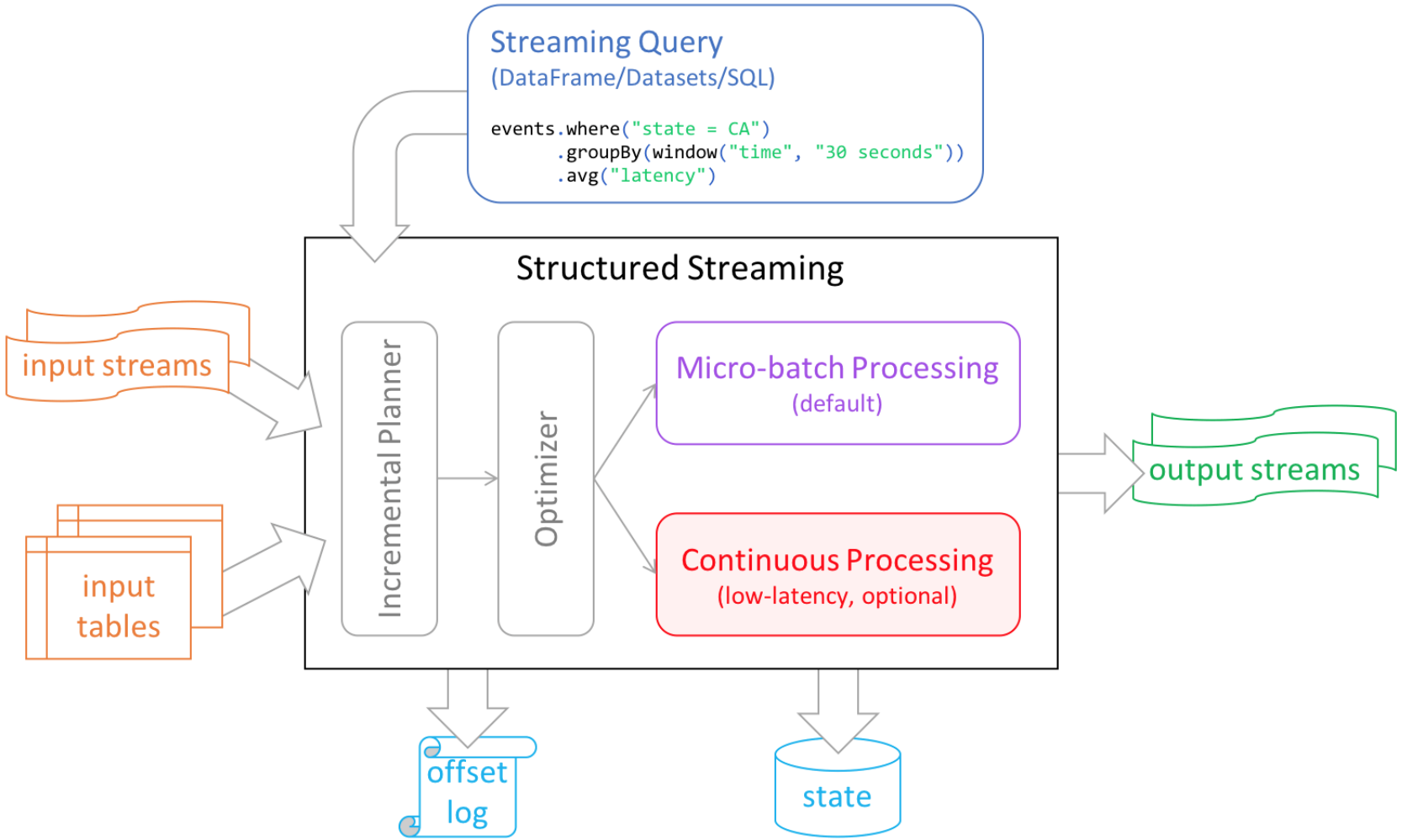
Streaming DataFrames

- **Data stream as an unbounded, continuously appended table**
- **Lazy creation and execution (like static DataFrames)**



https://spark.apache.org/docs/latest/img/structured-streaming-stream-as-a-table.png

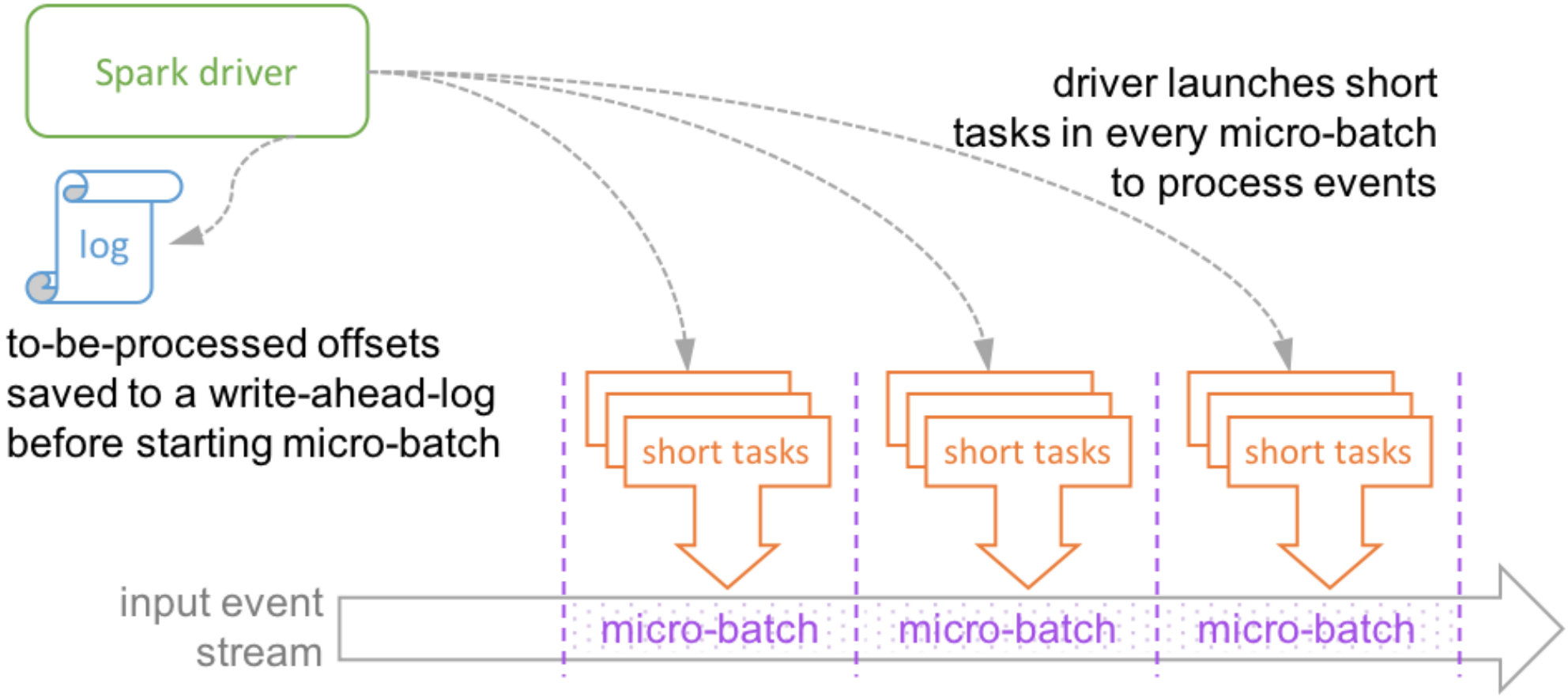
Components of Structured Streaming



Structured Streaming processing modes: Micro-batch and Continuous Processing

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

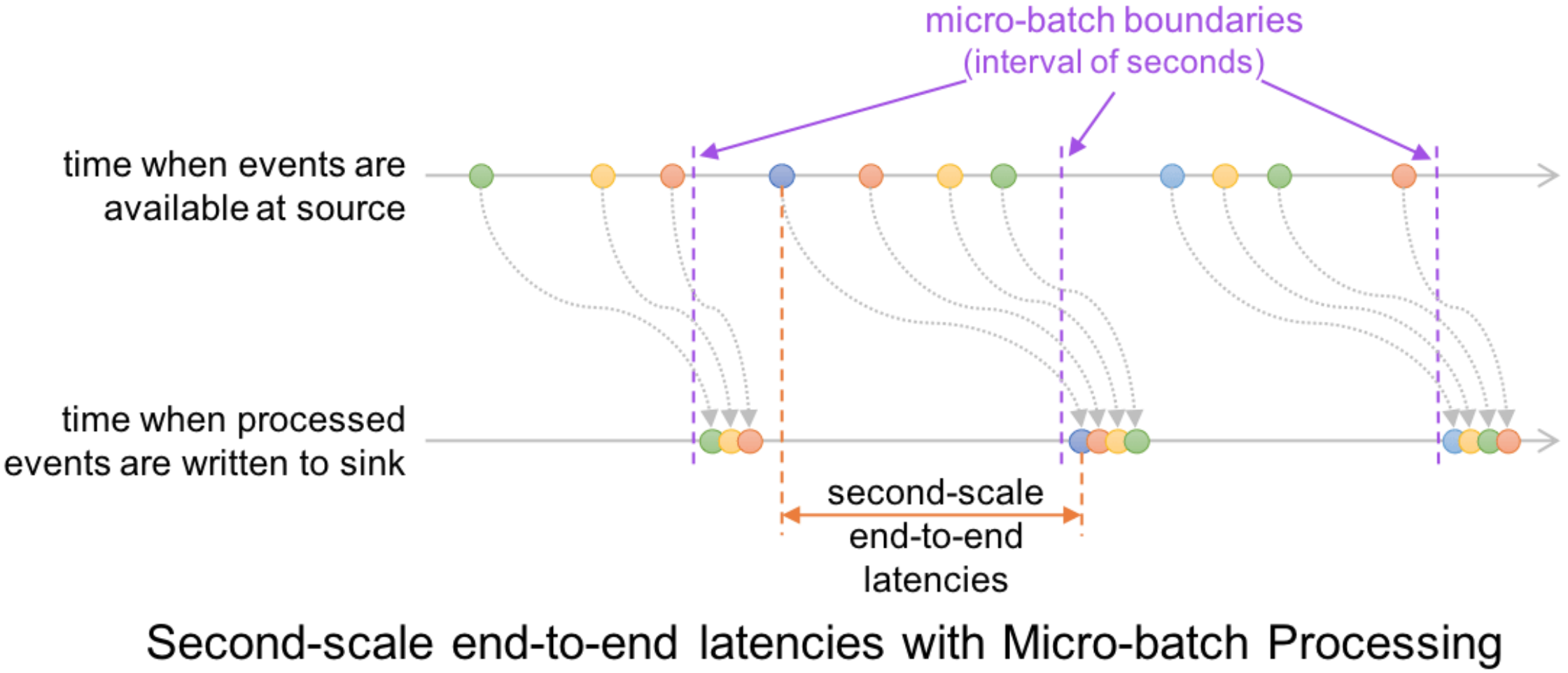
Micro-Batch Processing: exactly-once guarantees



Micro-batch Processing uses periodic tasks to process events

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

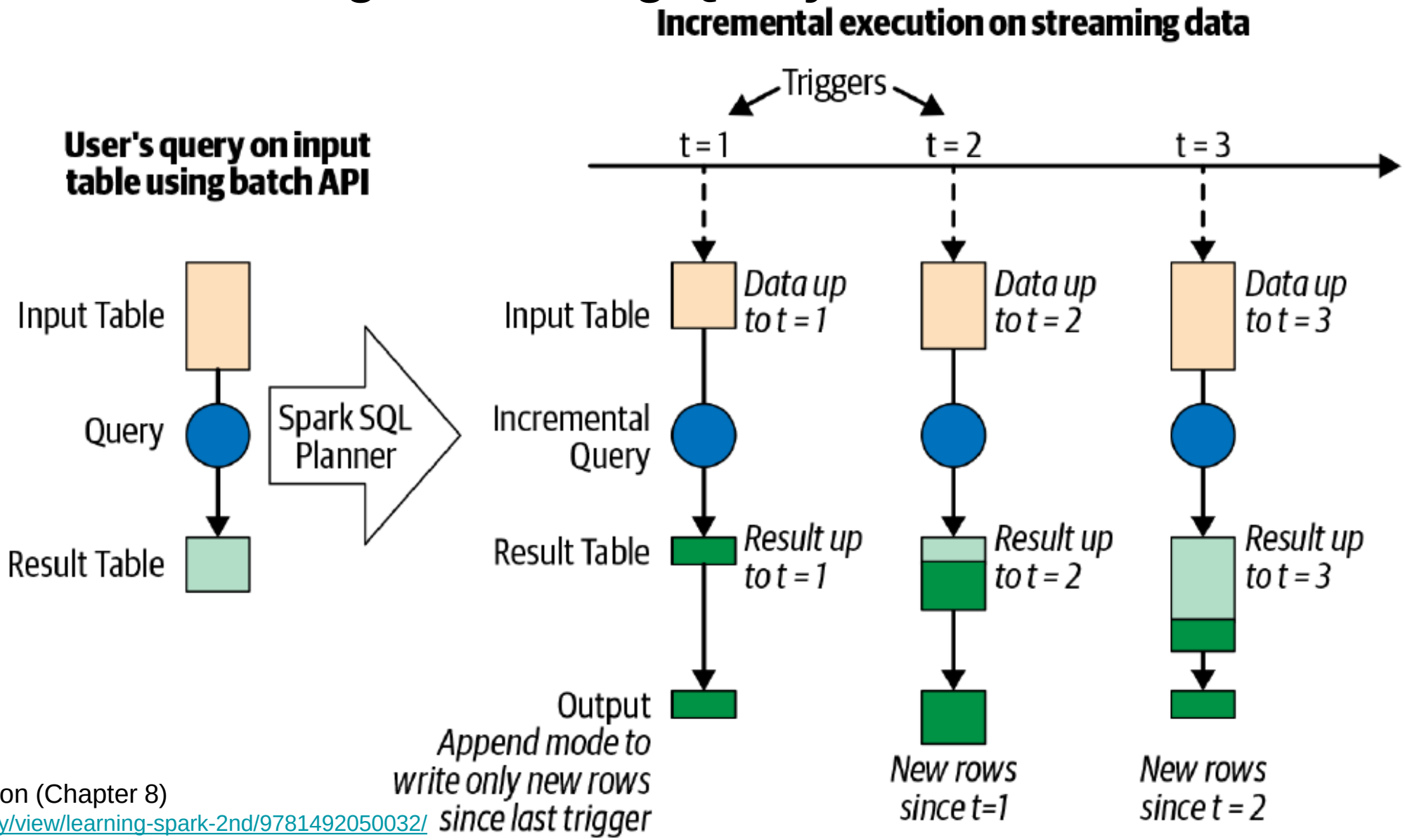
Micro-Batch Processing: second-scale latency



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

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

Micro-Batch Processing: Streaming Query Execution



Learning Spark, 2nd Edition (Chapter 8)
https://www.oreilly.com/library/view/learning-spark-2nd/9781492050032/

Micro-Batch Processing: Streaming Query Execution

- **The DataFrame operations are converted into a logical plan**
- **Spark SQL analyzes and optimizes this logical plan**
- **Spark SQL starts a background thread that continuously executes an optimized execution plan for each new batch of data (micro-batch)**
- **This loop continues until the query is terminated (explicitly, failure, or trigger settings)**

Triggers

- Triggers define when to check for new input data and update its result
  - Default: run the query as fast as possible
  - processingTime: run a micro-batch query periodically (e.g., “5 seconds”)
  - Once: process only one batch of data in a streaming query then terminates the query
  - availableNow: process all available data in multiple batches then terminates the query
  - Continuous: run a continuous query with a given checkpoint interval

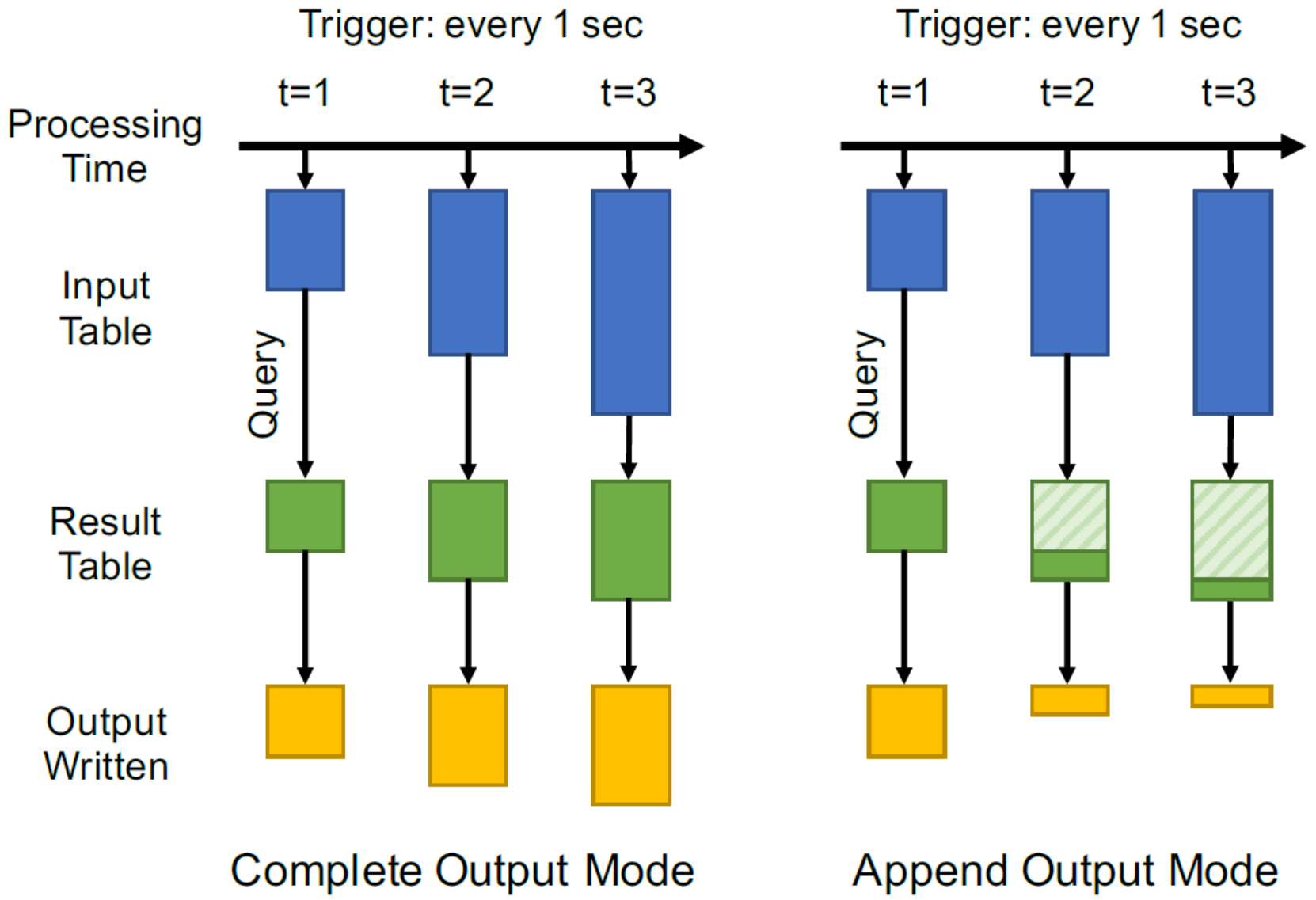
https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#triggers

Output Modes

- **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.**
- **Update Mode: Only the rows that were updated in the Result Table since the last trigger will be written to the external storage.**

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#output-modes

Output Modes

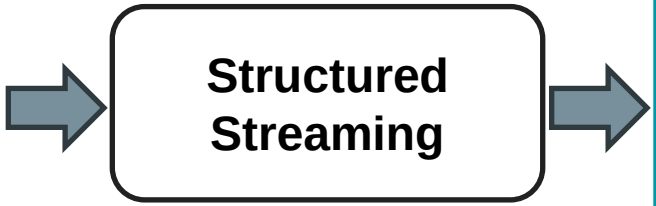


https://www.databricks.com/sites/default/files/2023-08/structured-streaming-a-declarative-api-for-real-time-applications-in-apache-spark.pdf

Streaming Data Sources and Sinks

Input Streaming Sources

- **File source**
- **Kafka source**
- Socket source
- Rate source
- Rate Per Micro-Batch source
- **Table**



Output Streaming Sinks

- **File sink**
- **Kafka sink**
- **Foreach sink**
- **ForeachBatch sink**
- Console sink
- Memory sink
- **Table**

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#input-sources

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#output-sinks

Spark Structured Streaming

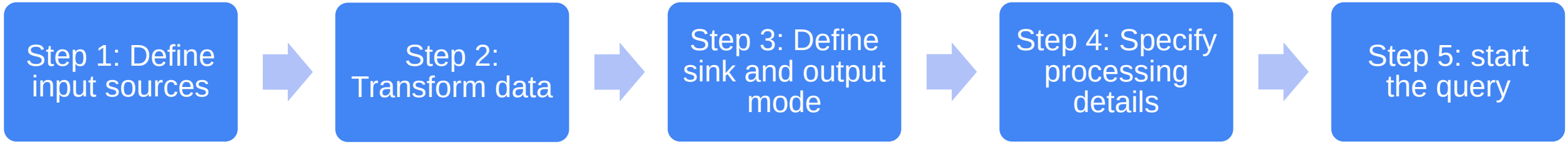
(PySpark API)

pyspark.sql.streaming

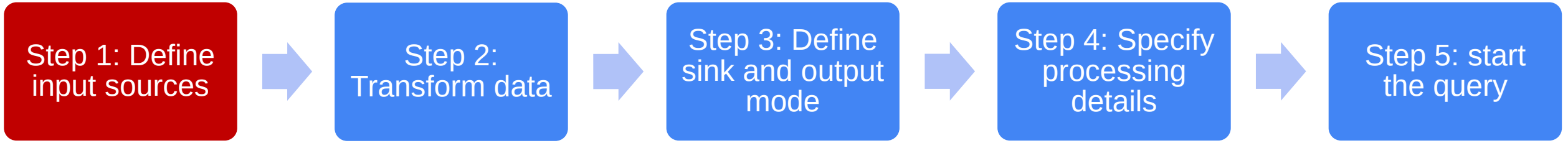
- **DataStreamReader**
- **DataStreamWriter**
- **StreamingQuery**
- **StreamingQueryManager**
- **StreamingQueryListener**

https://spark.apache.org/docs/latest/api/python/reference/pyspark.ss/index.html

Defining a Streaming Query



Defining a Streaming Query

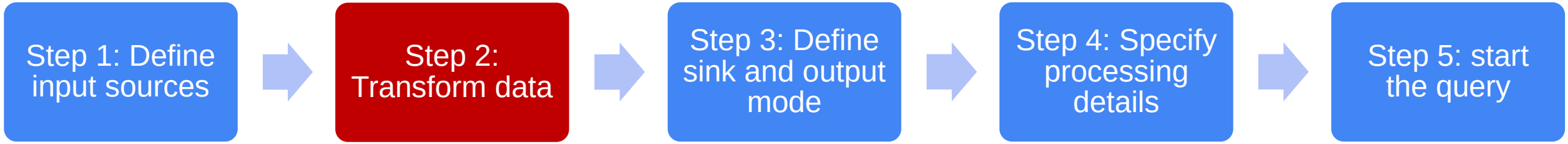


- *SparkSession.readStream()* ➔ **DataStreamReader**
- **This does not immediately start reading the streaming data (like *SparkSession.read()*)**

pyspark.sql.streaming.DataStreamReader

DataStreamReader.format
DataStreamReader.schema
DataStreamReader.table
DataStreamReader.option
DataStreamReader.options
DataStreamReader.csv
DataStreamReader.json
DataStreamReader.orc
DataStreamReader.parquet
DataStreamReader.text
DataStreamReader.load

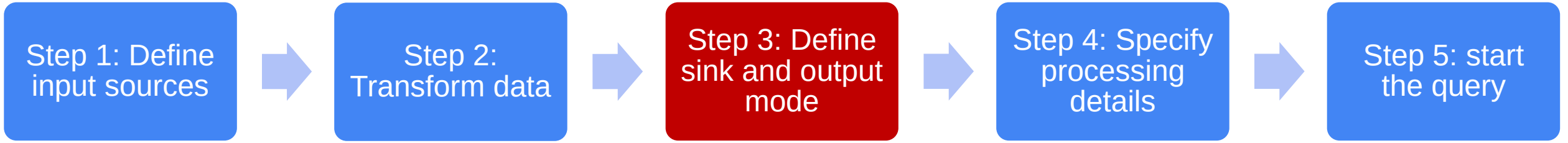
Defining a Streaming Query



- **Stateless transformations:** do not require any information from previous rows to process the next row, e.g., select(), filter(), map(), …
- **Stateful transformations:** require maintaining state to combine data across multiple rows like those operations involving grouping, joining, or aggregating, e.g., count()
- **Only the DataFrame transformations that can be executed incrementally**

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#unsupported-operations

Defining a Streaming Query

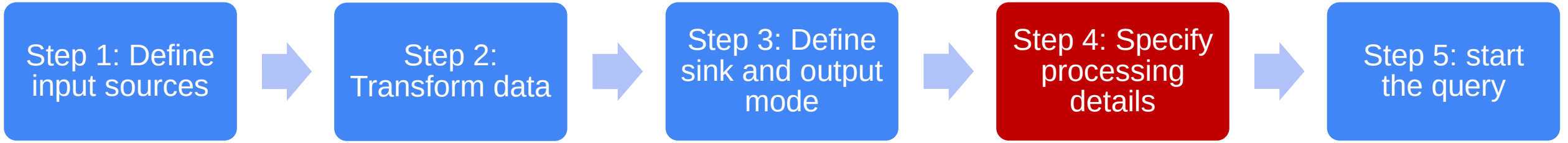


- *DataFrame.writeStream()* ➔ **DataStreamWriter**
- **Where to write the output?**
  - **Output streaming sinks (e.g., file, Kafka)**
- **How to write the output?**
  - **Output modes: append, complete, update**

pyspark.sql.streaming.DataStreamWriter

DataStreamWriter.queryName
DataStreamWriter.format
DataStreamWriter.outputMode
DataStreamWriter.trigger
DataStreamWriter.start
DataStreamWriter.option
DataStreamWriter.options
DataStreamWriter.foreach
DataStreamWriter.foreachBatch
DataStreamWriter.partitionBy
DataStreamWriter.toTable

Defining a Streaming Query

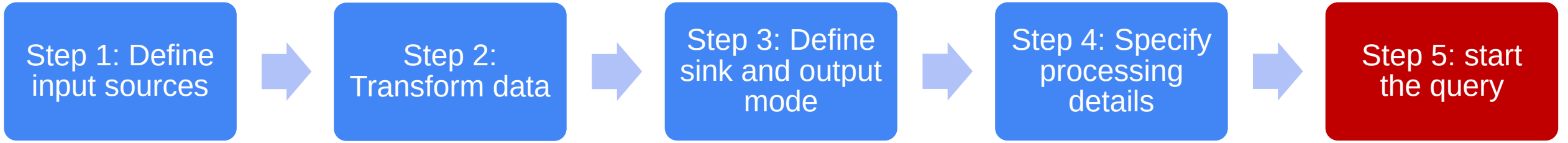


- **How to process the data?**
  - **Trigger: Default (asap), interval, once,…**
- **How to recover from failure?**
  - **Checkpoint location (***for failure recovery with exactly-once guarantees***)**

pyspark.sql.streaming.DataStreamWriter

DataStreamWriter.queryName
DataStreamWriter.format
DataStreamWriter.outputMode
DataStreamWriter.trigger
DataStreamWriter.start
DataStreamWriter.option
DataStreamWriter.options
DataStreamWriter.foreach
DataStreamWriter.foreachBatch
DataStreamWriter.partitionBy
DataStreamWriter.toTable

Defining a Streaming Query



- *DataStreamWriter.start()* ➔ **StreamingQuery**
- **A background thread continuously reads new data from the streaming source, processes it, and writes it to the streaming sink.**
- **start() is a nonblocking method (it will return as soon as the query has started in the background).**

pyspark.sql.streaming.DataStreamWriter

DataStreamWriter.queryName
DataStreamWriter.format
DataStreamWriter.outputMode
DataStreamWriter.trigger
DataStreamWriter.start
DataStreamWriter.option
DataStreamWriter.options
DataStreamWriter.foreach
DataStreamWriter.foreachBatch
DataStreamWriter.partitionBy
DataStreamWriter.toTable

Managing and Monitoring a Streaming Query

- The StreamingQuery object created when a query is started can be used to monitor and manage the query.
- StreamingQuery.status(): what the background query thread is doing at this moment?
- StreamingQuery.lastProgress(): Information on the last computed micro-batch

pyspark.sql.streaming.StreamingQuery

StreamingQuery.awaitTermination
StreamingQuery.processAllAvailable
StreamingQuery.stop
StreamingQuery.id
StreamingQuery.isActive
StreamingQuery.lastProgress
StreamingQuery.name
StreamingQuery.recentProgress
StreamingQuery.runId
StreamingQuery.status

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#managing-streaming-queries

Managing and Monitoring a Streaming Query

- Manage queries in a SparkSession
- Multiple queries can be started in a single SparkSession
- They will all be running concurrently sharing the cluster resources.
- SparkSession.streams() ➔ get the StreamingQueryManager that can be used to manage the currently active queries.

pyspark.sql.streaming.StreamingQueryManager

StreamingQueryManager.active
StreamingQueryManager.addListener
StreamingQueryManager.awaitAnyTermination
StreamingQueryManager.get
StreamingQueryManager.removeListener
StreamingQueryManager.resetTerminated

Recovering from Failures with exactly-once Guarantees

- **End-to-end exactly-once guarantees:** the output is as if each input record was processed exactly once
- **Structured Streaming is designed to reliably track the exact progress of the processing**
  - **Replayable streaming sources**
  - **Deterministic computations, checkpointing and write-ahead logs**
  - **Idempotent streaming sink**
- **All the progress information (i.e., range of offsets processed in each trigger) and the running aggregates (e.g., counts) are saved to the configured checkpoint location**
- **Recovering from failures by restarting and/or reprocessing**
- **The checkpoint location must be the same across restarts**

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#fault-tolerance-semantics

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#recovery-semantics-after-changes-in-a-streaming-query

Spark Configurations for Structured Streaming

- **Structured streaming doesn’t allow schema inference without explicitly enabling it:**
  - **spark.sql.streaming.schemaInference**
- **The number of progress updates retained for each stream**
  - **spark.sql.streaming.numRecentProgressUpdates**
- **Metrics are by default not enabled for Structured Streaming queries due to their high volume of reported data.**
  - **spark.sql.streaming.metricsEnabled**

https://spark.apache.org/docs/latest/configuration.html

Other Functions for Structured Streaming

- **pyspark.sql.DataFrame.isStreaming**
- **pyspark.sql.DataFrame.withWatermark**
- **pyspark.sql.functions.window**
- **pyspark.sql.DataFrame.observe**

Event-Time Processing

- **Analyzing data with respect to the event time, not processing time**
- **Event time**
  - **The time at which the event occurred**
  - **It is embedded in the data itself**
  - **Important to use (it provides a more robust way of comparing events against one another)**
  - **Event data can be late or out-of-order**
- **Processing time**
  - **The time that it was processed or reached the stream processing system**
  - **A property of the streaming system (not an external system)**
  - **Less important than event time**
  - **Can’t ever be out-of-order**
- **The order of events in the processing system does not guarantee an ordering in event time**

Spark Structured Streaming

(Operations on Streaming DataFrames)

Streaming Data Transformations

- **Stateless transformations: no dependence on previous input data**
  - **Projection and selection operations (e.g., select(), explode(), map(), filter(), where())**
- **Stateful transformations: combine data across multiple rows**
  - **grouping, joining, or aggregating operations (e.g., DataFrame.groupBy().count())**
  - **State is maintained in the memory of the Spark executors**
  - **It is also checkpointed to the configured checkpoint location for fault tolerance**

Types of Stateful Operations

- **Managed stateful operations: automatically identify and clean up old state**
  - **streaming aggregations**
  - **stream-stream joins**
  - **stream deduplication**
- **Unmanaged stateful operations: users define custom state cleanup logic**
  - **MapGroupsWithState**
  - **FlatMapGroupsWithState**

Stateful Streaming Aggregations: not based on time

- **Aggregate data by keys (e.g., streaming word count)**
- **Direct aggregation operations are not allowed (**e.g., *streamingDataFrame.count()***)**
- **Global aggregations: across all the data**

```
runningCount = sensorReadings.groupBy().count()
```

- **Grouped aggregations: within each group or key**

```
baselineValues = sensorReadings.groupBy("sensorId").mean("value")
```

- **All built-in and User-defined aggregation functions**

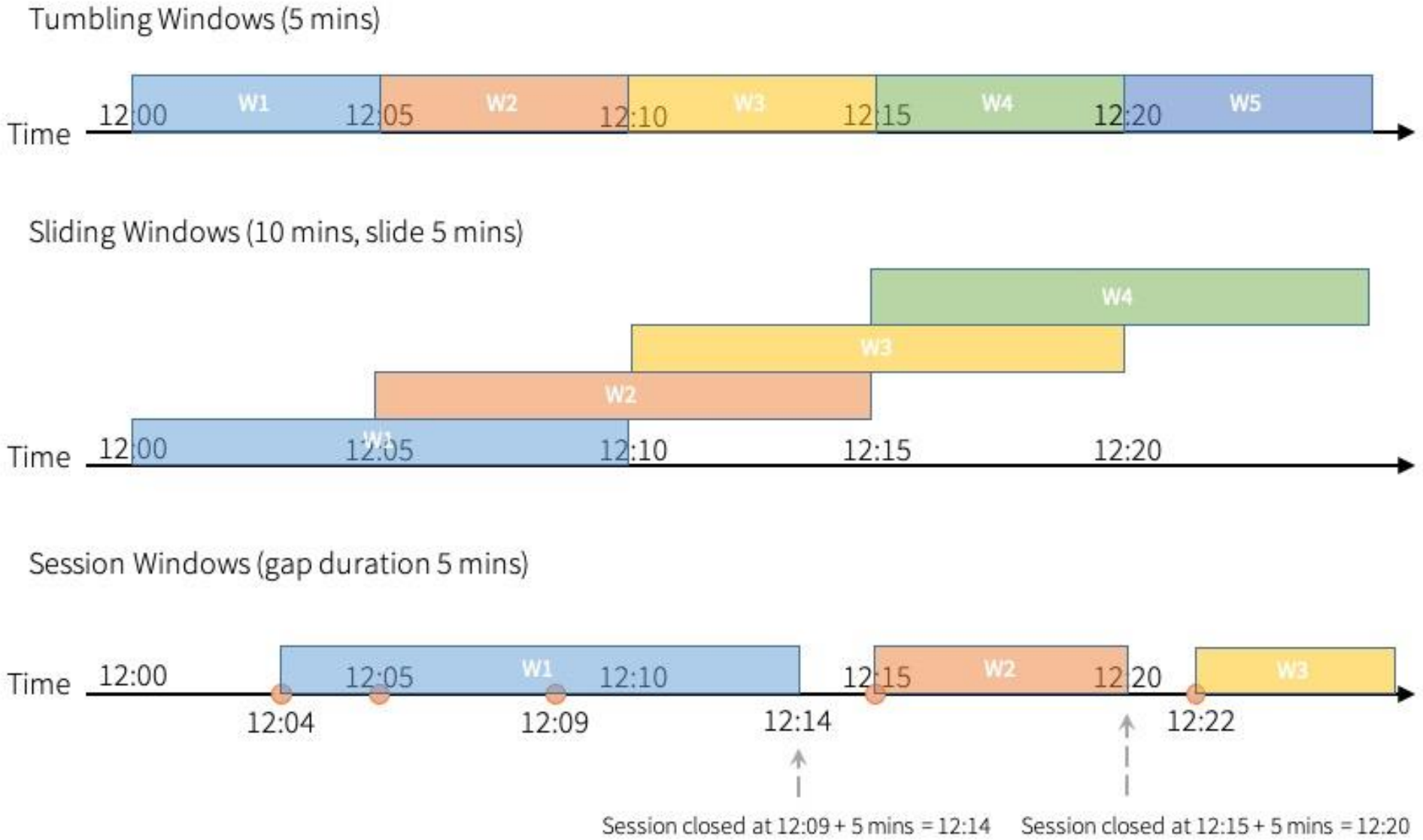
```
multipleAggs = (sensorReadings
  .groupBy("sensorId")
  .agg(count("*"), mean("value").alias("baselineValue"),
    collect_set("errorCode").alias("allErrorCodes")))
```

- **Multiple aggregations can be applied together**
- **Streaming aggregates are maintained as a distributed state**
- **Output modes: update or complete**

Stateful Streaming Aggregations: based on event time

- **Aggregate data by time (e.g., count records received every hour).**
- **Event-time windows**
  - **Dynamically computed grouping columns**
  - **Based on when the data was generated at the sensor and not based on when the data was received, as any transit delay would skew the results.**
  - **Aggregate values are maintained for each window**
- **Late and out-of-order events are assigned to the appropriate windows based on event time**
- **Output modes: all**

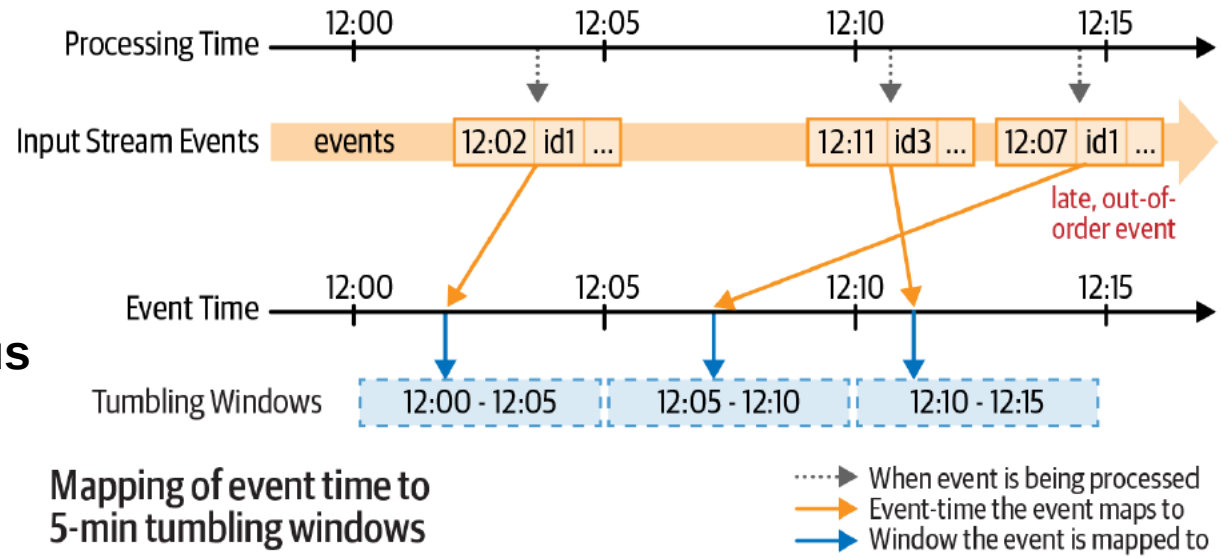
Event-Time Windows



https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#types-of-time-windows

Window-Based Aggregations: Tumbling Windows

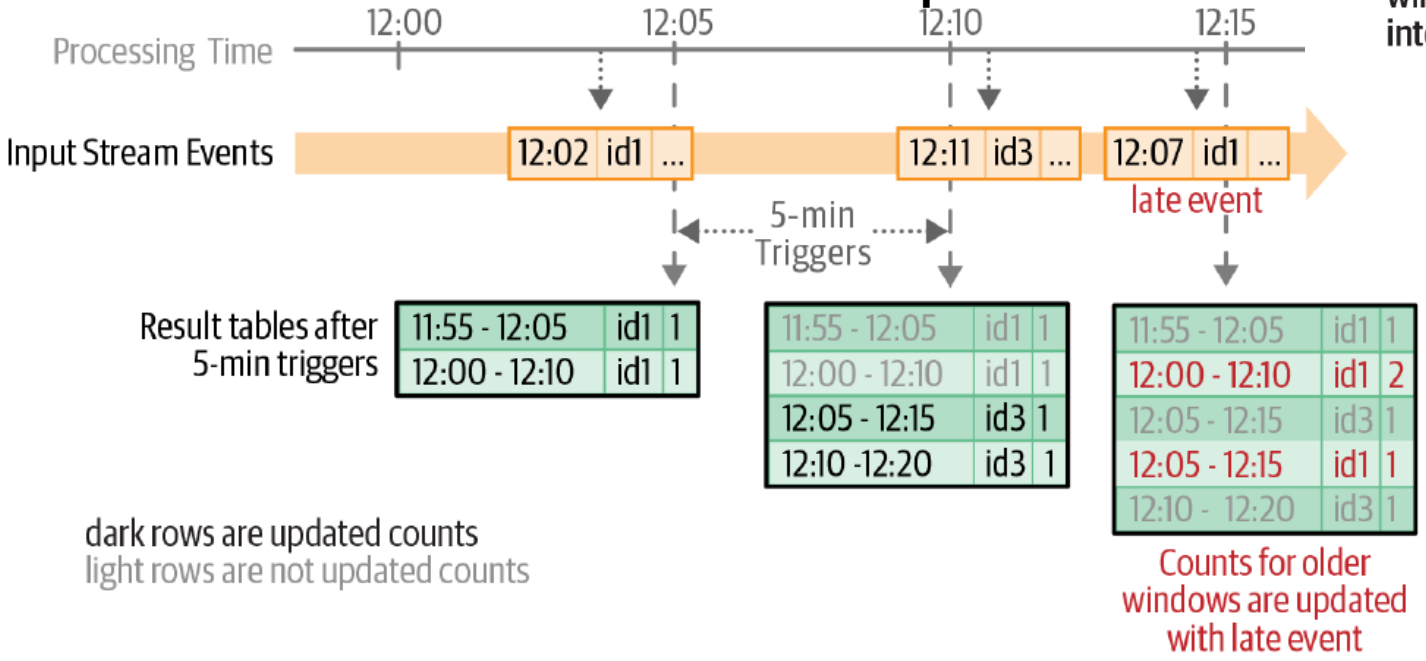
- *pyspark.sql.functions.window*
- **Event time column, window duration**
- **Fixed-sized, non-overlapping and contiguous time intervals.**
- **An event can only be bound to a single window**

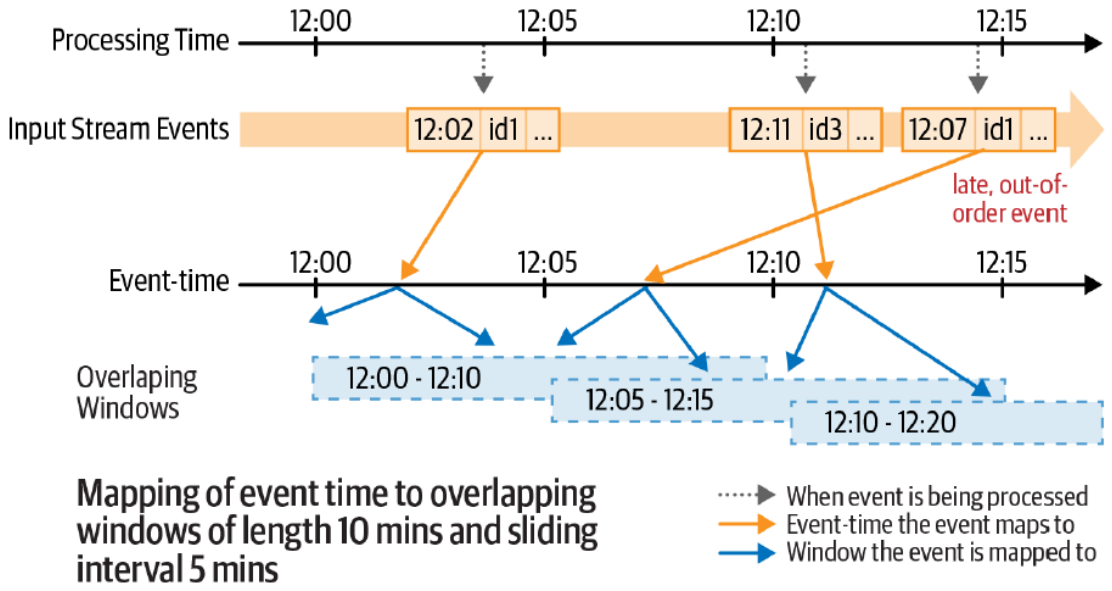


Mapping of event time to 5-min tumbling windows

Window-Based Aggregations: Sliding Windows

- *pyspark.sql.functions.window*
- **Event time column, window duration, slide duration**
- **Overlapping windows (if slide smaller than window)**
- **An event can be bound to the multiple windows**





Mapping of event time to overlapping windows of length 10 mins and sliding interval 5 mins

Window-Based Aggregations: Session Windows

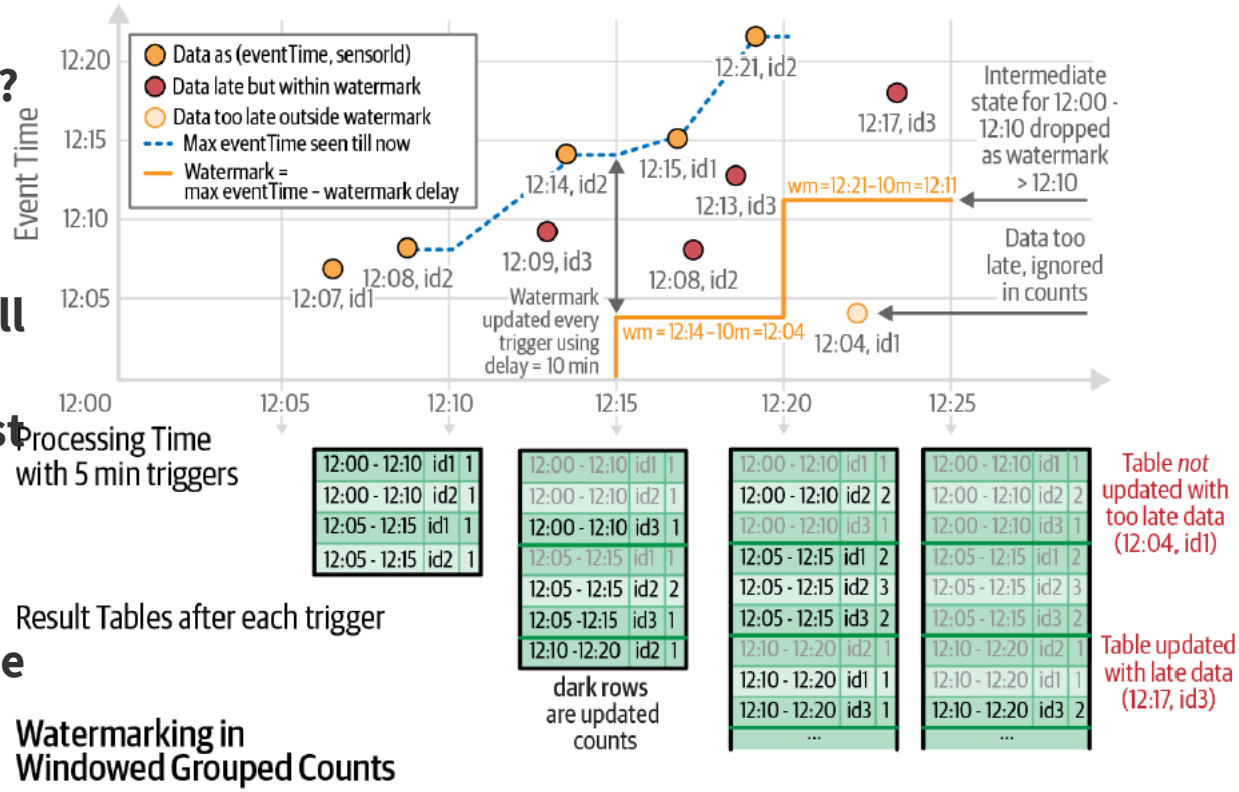
- **A dynamic size of the window length, depending on the inputs**
- **A session window starts with an input and expands itself if following input has been received within gap duration.**
- **For static gap duration, a session window closes when there's no input received within gap duration after receiving the latest input.**
- **With dynamic gap duration, the closing of a session window does not depend on the latest input anymore. A session window's range is the union of all events' ranges which are determined by event start time and evaluated gap duration during the query execution.**

https://www.databricks.com/blog/2021/10/12/native-support-of-session-window-in-spark-structured-streaming.html

Window-Based Aggregations: Handling Late Data - Watermarking

- **How late do we expect to see the data? When to close the aggregate window and produce the result?**
- *pyspark.sql.DataFrame.withWatermark*
- **Delay Threshold: defines how long the engine will wait for late data to arrive (relative to the latest record that has been processed)**
- **Larger values allow data to arrive later, but increase state size (i.e., memory usage), and vice versa.**

```
(sensorReadings
  .withWatermark("eventTime", "10 minutes")
  .groupBy("sensorId", window("eventTime", "10 minutes", "5 minutes"))
  .mean("value"))
```



https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#handling-late-data-and-watermarking

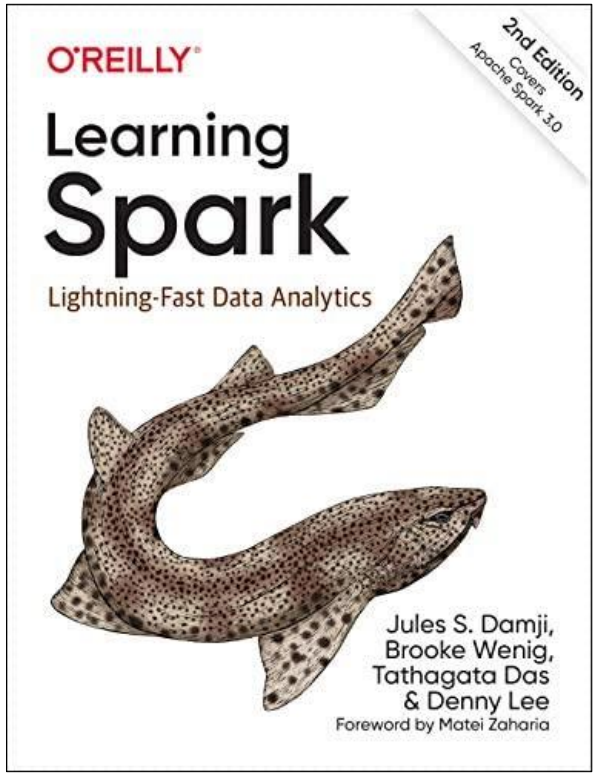
https://www.databricks.com/blog/2022/08/22/feature-deep-dive-watermarking-apache-spark-structured-streaming.html

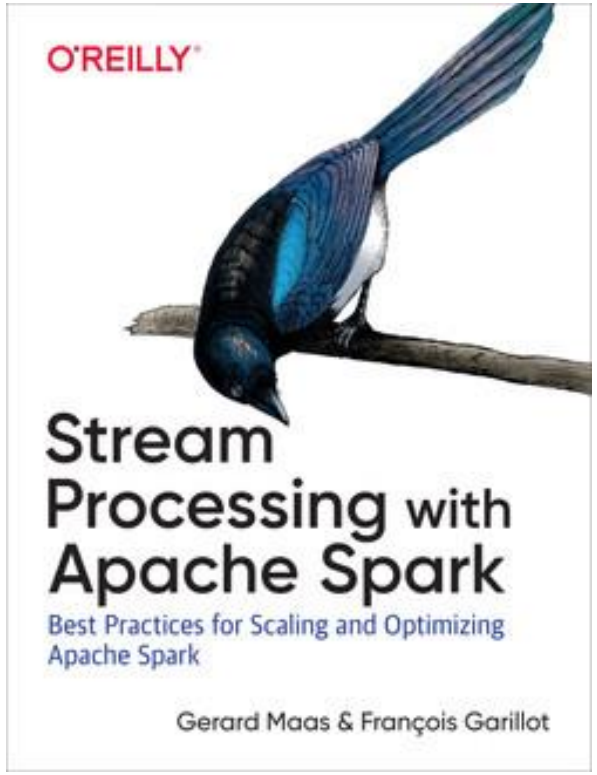
Summary & Resources

Summary

- Apache Spark: a unified computing engine designed for large-scale distributed data processing
- Spark SQL: a foundational component of Apache Spark
- Spark DataFrame: distributed collections of data, organized into rows and columns
- Spark DataFrame API: structured and high-level API (part of Spark SQL)
- Structured Streaming: a scalable and fault-tolerant stream processing engine built on Spark SQL
- Streaming DataFrame: an unbounded, continuously appended Spark DataFrame
- Micro-Batch Processing: processes data streams as a series of small batch jobs
- Building streaming applications and pipelines with the same and familiar Spark DataFrame APIs

Books


O'REILLY
2nd Edition
Covers Apache Spark 3.0
Learning
Spark
Lightning-Fast Data Analytics
Jules S. Damji,
Brooke Wenig,
Tathagata Das
& Denny Lee
Foreword by Matei Zaharia


O'REILLY
Stream
Processing with
Apache Spark
Best Practices for Scaling and Optimizing
Apache Spark
Gerard Maas & François Garillot

Labs

https://github.com/ssalloum/SDSC-Spark5

Thank You

Salman Salloum
www.linkedin.com/in/ssalloum/