CS4225/CS5425 Big Data Systems for Data Science

Spark II: Advanced Topics

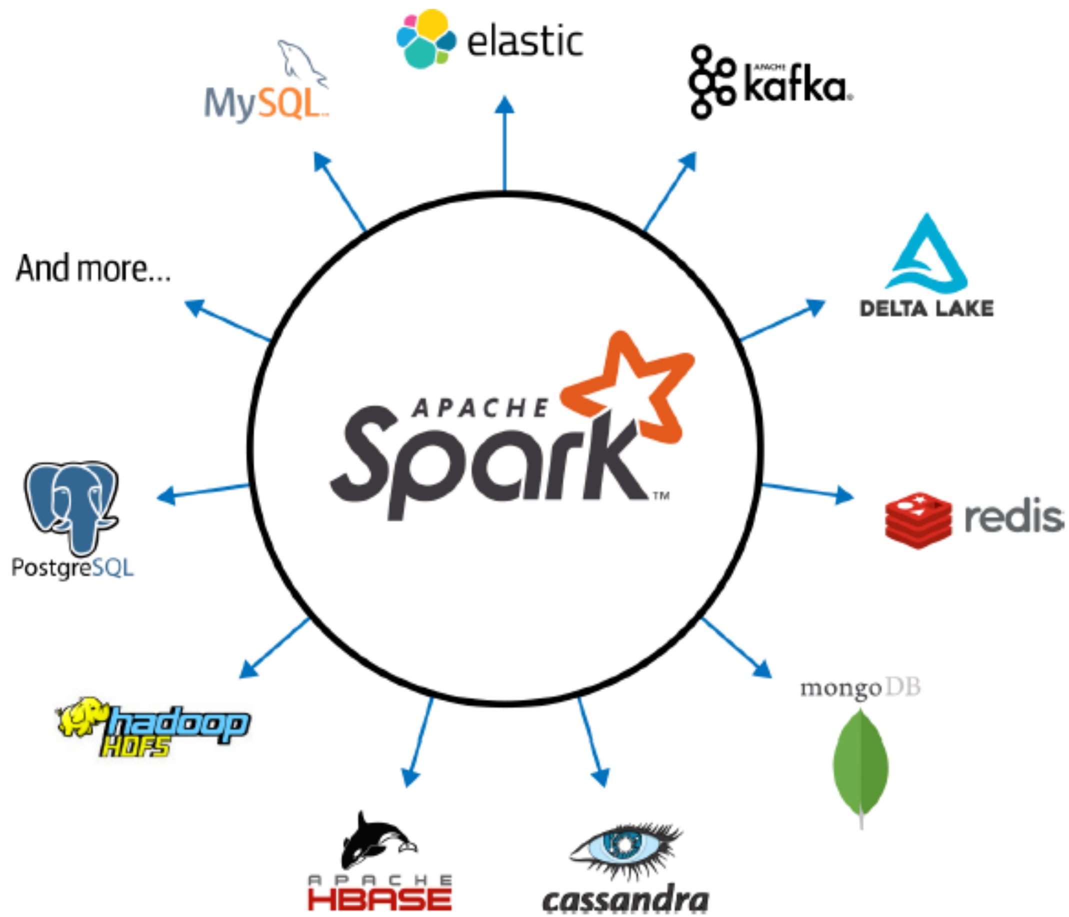
Ai Xin
School of Computing
National University of Singapore
aixin@comp.nus.edu.sg

Today's Plan

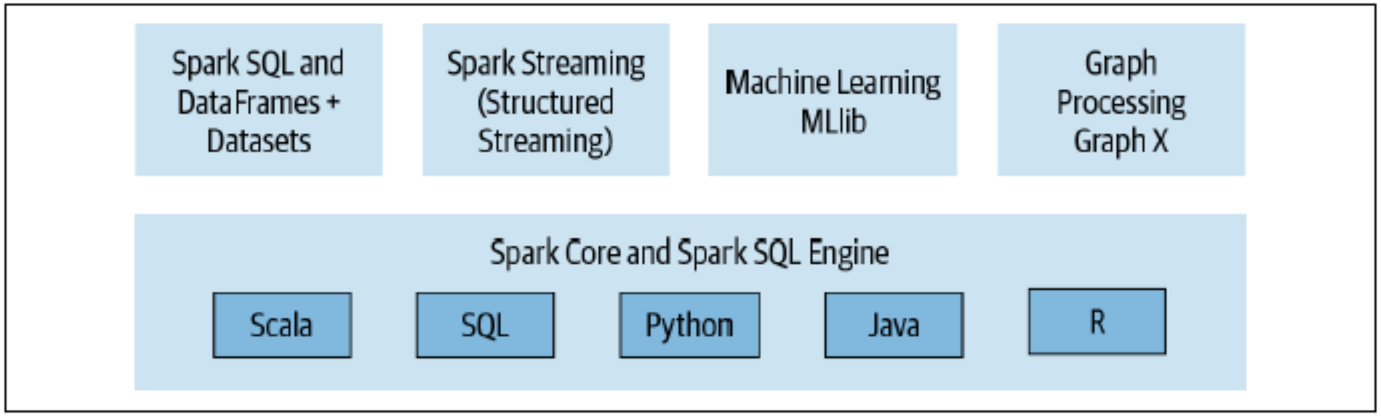
- Spark SQL and Catalyst Optimizer
- Machine Learning with Mllib
- Structured Streaming

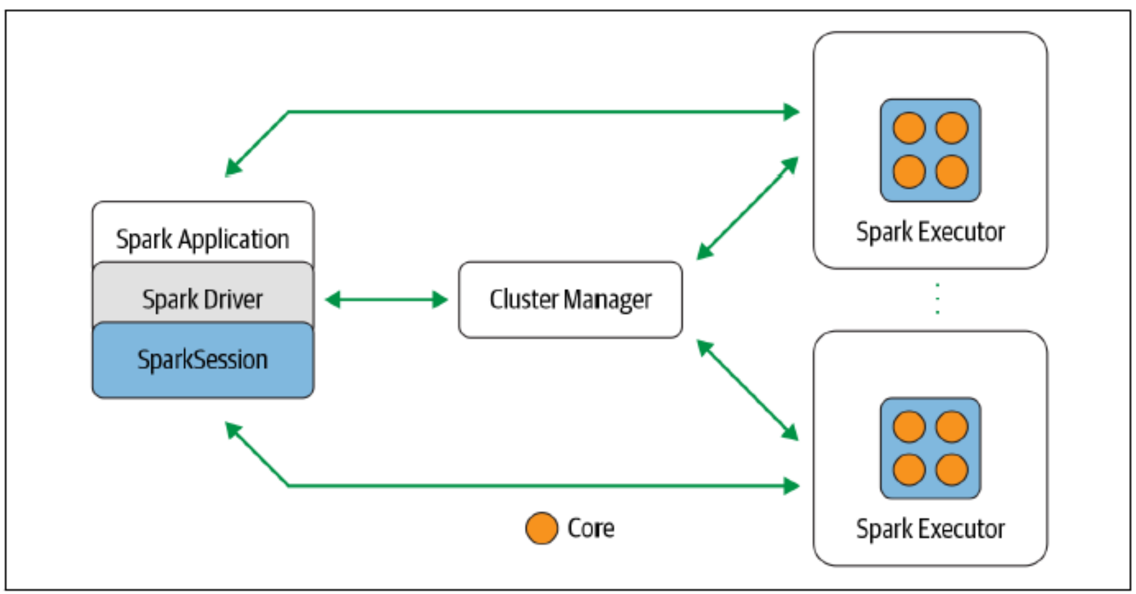
Spark Design Philosophy

- Speed
- Ease of use
- Modularity
- Extensibility



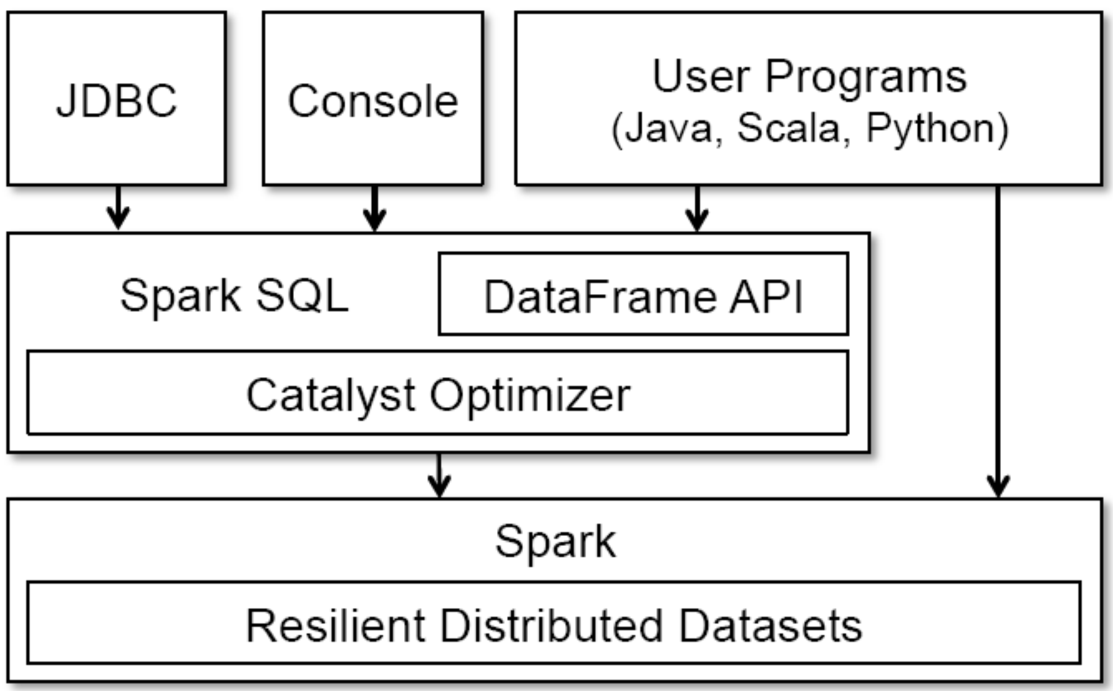
Spark: a unified stack for distributed execution





Spark SQL

- Unifies Spark components and permits abstraction to DataFrames/Datasets in Java, Scala, Python, and R
- Keep track of schema and support optimized relational operations



RDD vs. DataFrame

- RDD

```python
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average

agesRDD = (dataRDD
  .map(lambda x: (x[0], (x[1], 1)))
  .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
  .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

- DataFrame

```python
# Create a DataFrame
data_df = spark.createDataFrame([("Brooke", 20), ("Denny", 31), ("Jules", 30),

  ("TD", 35), ("Brooke", 25)], ["name", "age"])
# Group the same names together, aggregate their ages, and compute an average
avg_df = data_df.groupBy("name").agg(avg("age"))
# Show the results of the final execution
avg_df.show()
```

```
+------+--------+
|  name|avg(age)|
+------+--------+
|Brooke|    22.5|
| Jules|    30.0|
|    TD|    35.0|
| Denny|    31.0|
+------+--------+
```

RDD vs. DataFrame

- RDD
  - Instruct Spark how to compute the query
  - The intention is completely opaque to Spark
  - Spark also does not understand the structure of the data in RDDs (which is arbitrary Python objects) or the semantics of user functions (which contain arbitrary code)
- DataFrame
  - Tell Spark what to do, instead of How to do
  - The code is far more expressive as well as simpler
    - Using a domain specific language (DSL) similar to python pandas
    - Use high-level DSL operators to compose the query
  - Spark can inspect or parse this query and understand our intention, it can then optimize or arrange the operations for efficient execution

Spark SQL

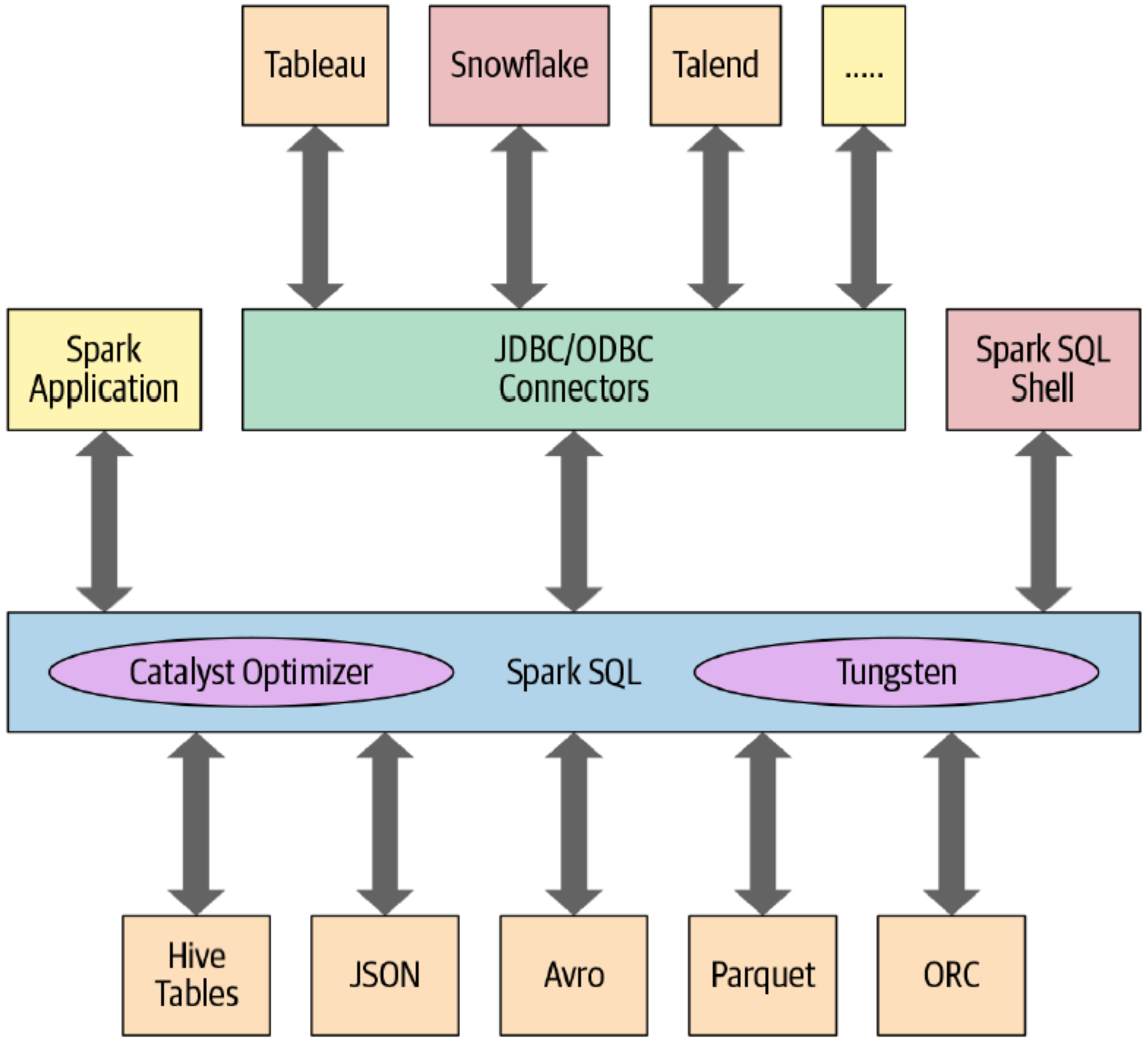
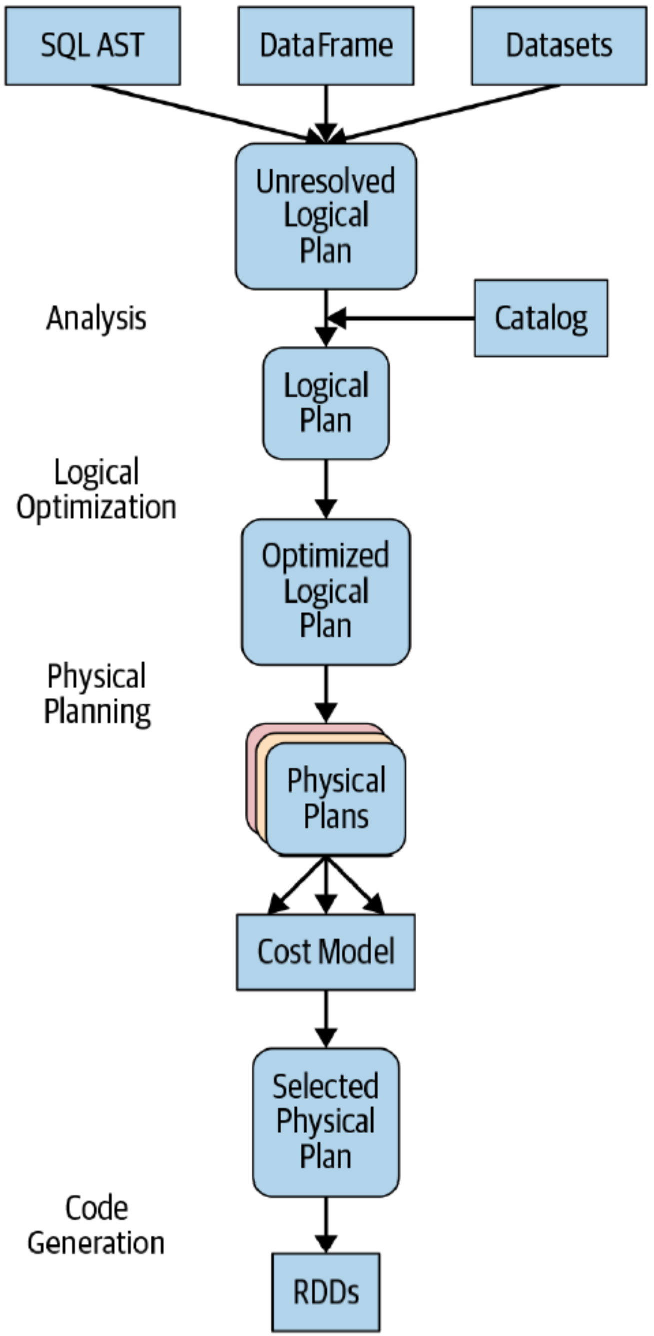


Tableau
Snowflake
Talend
.....
Spark Application
JDBC/ODBC Connectors
Spark SQL Shell
Catalyst Optimizer
Spark SQL
Tungsten
Hive Tables
JSON
Avro
Parquet
ORC

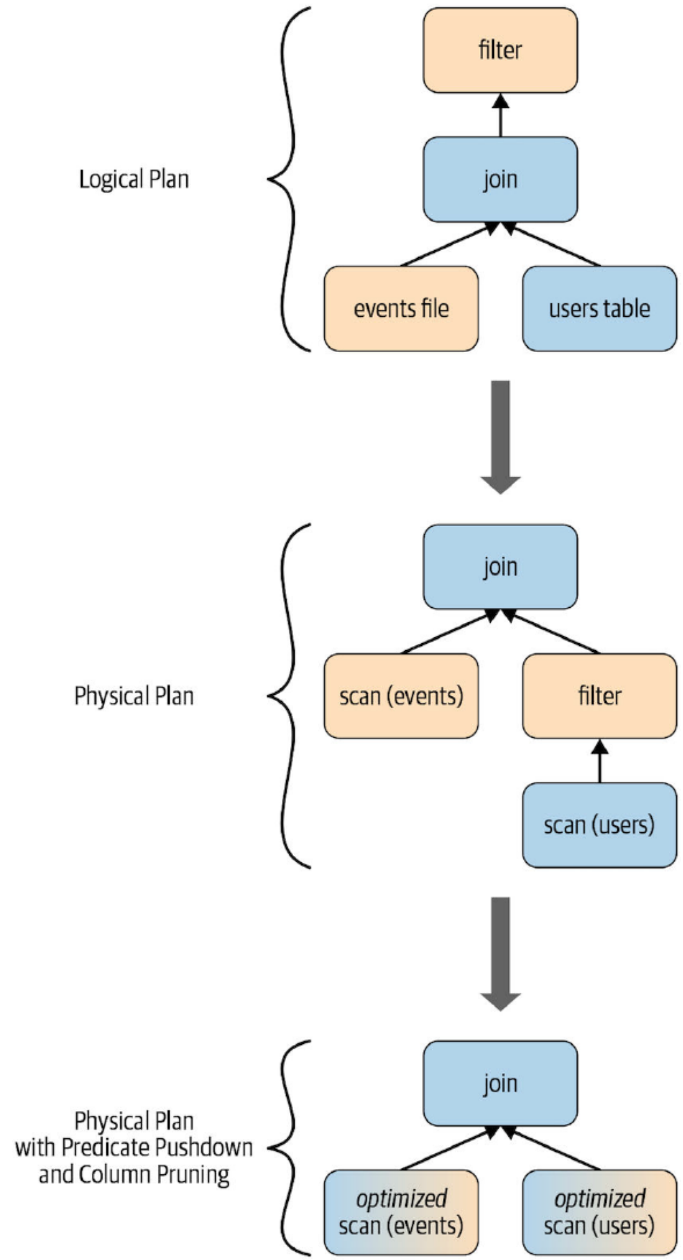
The Catalyst Optimizer

- Takes a computational query and converts it into an execution plan through four transformational phases:
  1. Analysis
  2. Logical optimization
  3. Physical planning
  4. Code generation

A Spark computation's four-phase journey

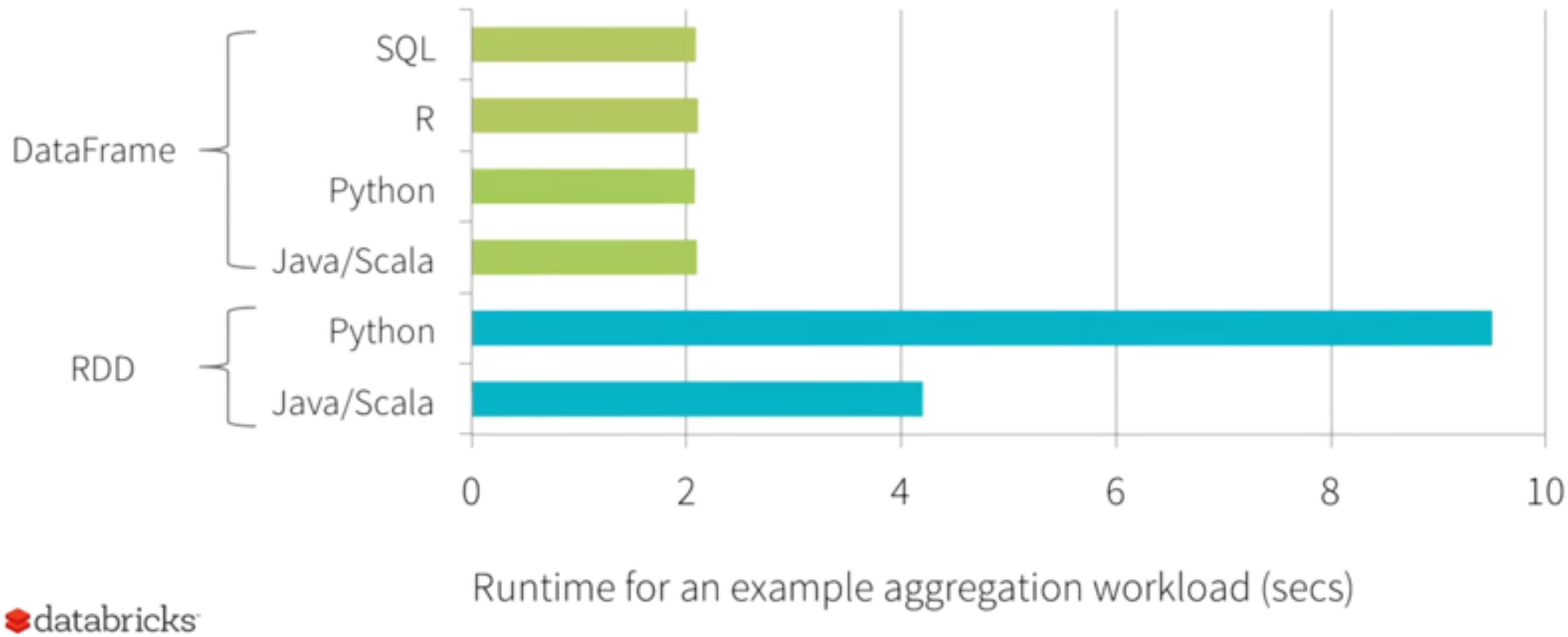


```scala
// In Scala
// Users DataFrame read from a Parquet table
val usersDF  = ...
// Events DataFrame read from a Parquet table
val eventsDF = ...
// Join two DataFrames
val joinedDF = users
  .join(events, users("id") === events("uid"))
  .filter(events("date") > "2015-01-01")
```



Benefit of Logical Plan

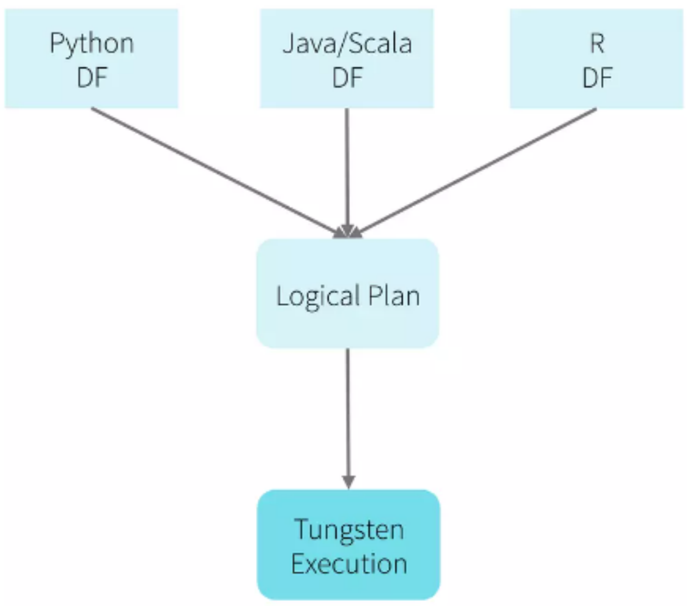
- Performance Parity Across Languages

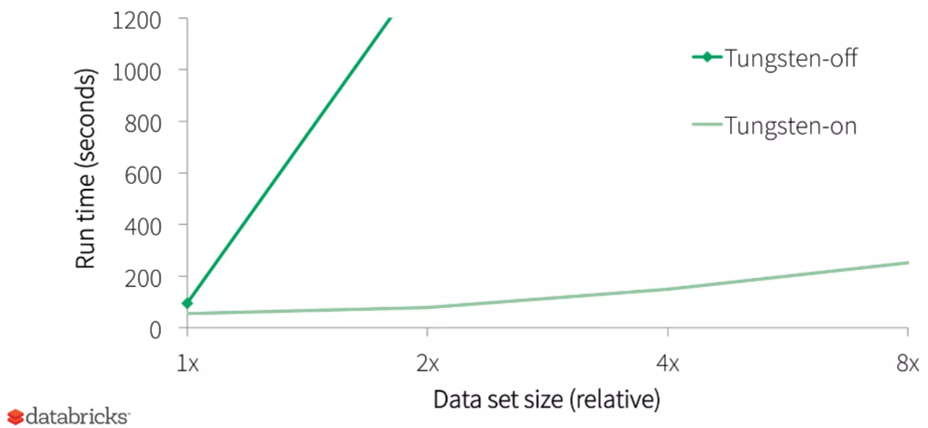


Source: https://youtu.be/VbSar607HM0

Project Tungsten

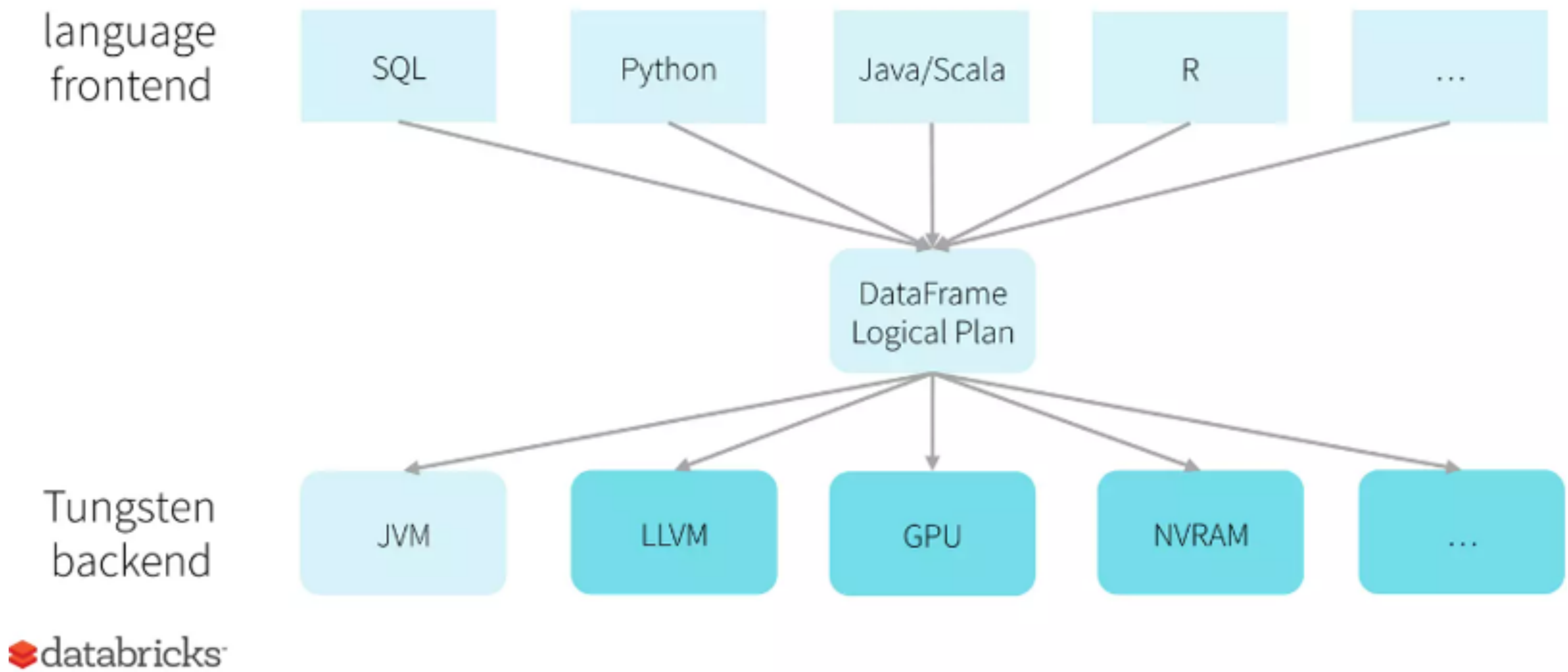
- Objectives:
  - Substantially improve the memory and CPU efficiency of Spark applications
  - Push performance closer to the limits of modern hardware
- How?
  - Memory Management and Binary Processing
  - Cache-aware computation
  - Code generation



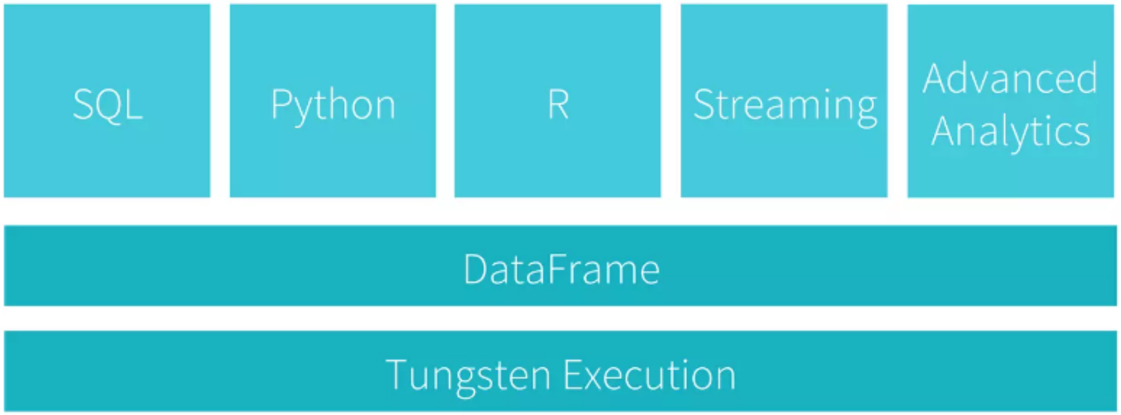


Source: https://youtu.be/VbSar607HM0

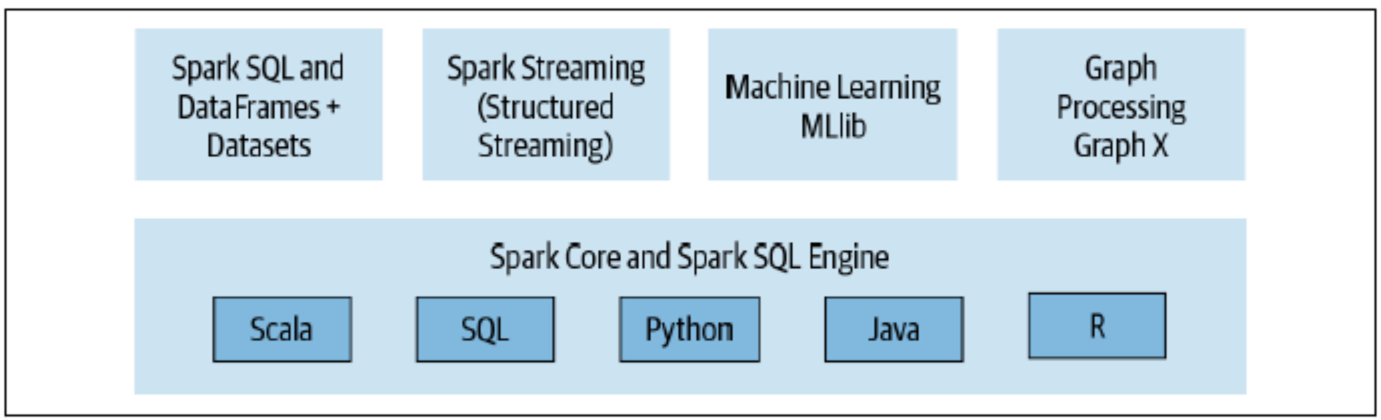
Unified API, One Engine, Automatically Optimized



Source: https://youtu.be/VbSar607HM0



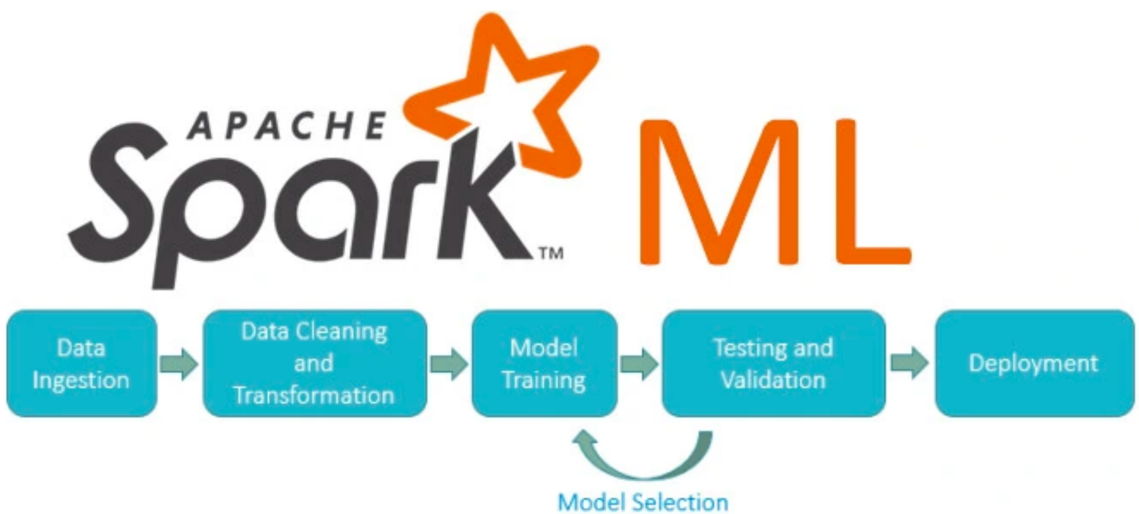
databricks



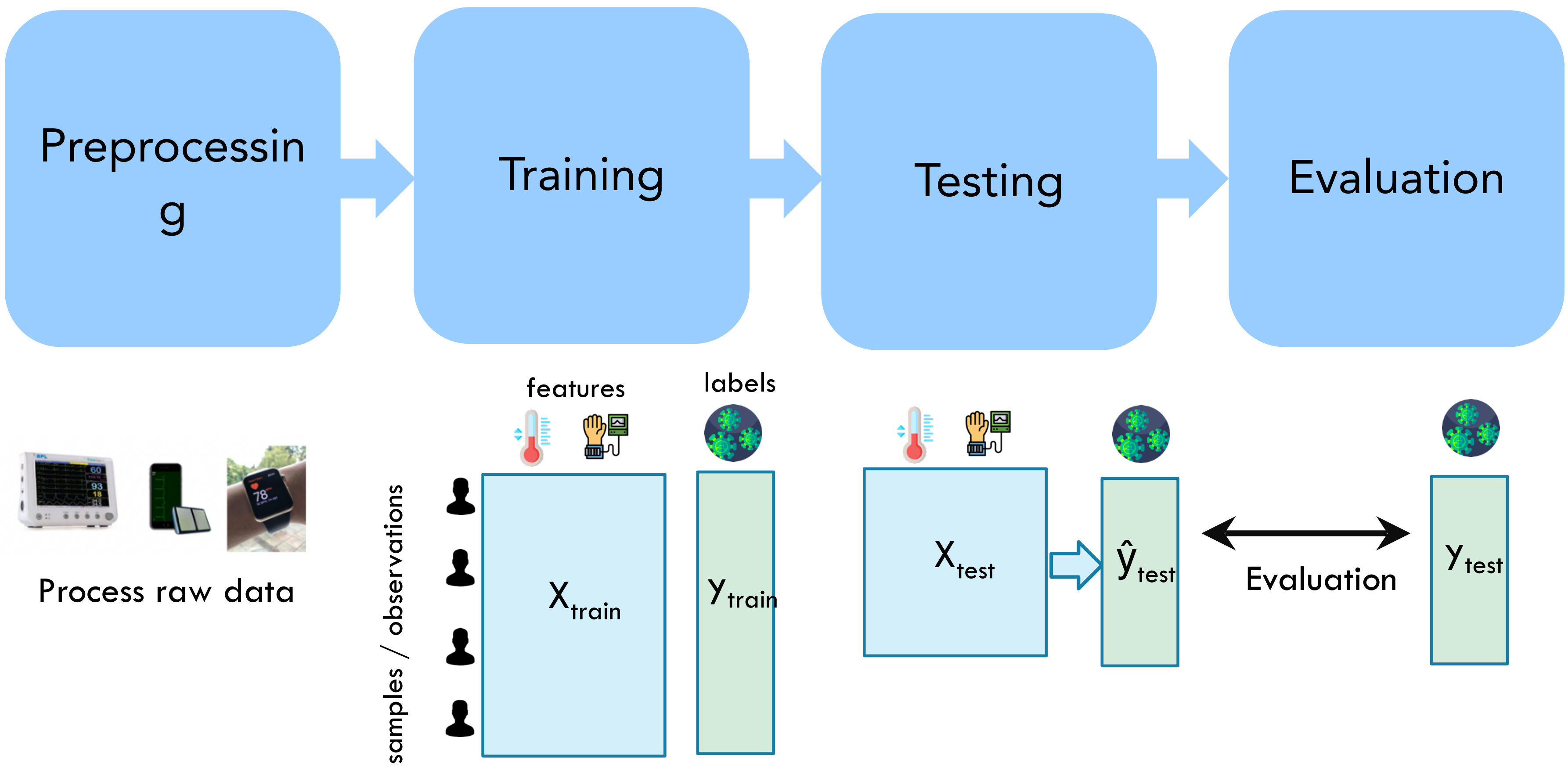
Source: https://youtu.be/VbSar607HM0

Today’s Plan

- Spark SQL
- **Machine Learning with MLlib**
- Structured Streaming



Typical Machine Learning Pipeline



Spark MLLib: Simple Logistic Regression Model

```
from pyspark.ml.classification import LogisticRegression

training = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

lr = LogisticRegression(maxIter=10)

lrModel = lr.fit(training)

print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
```

Pipelines

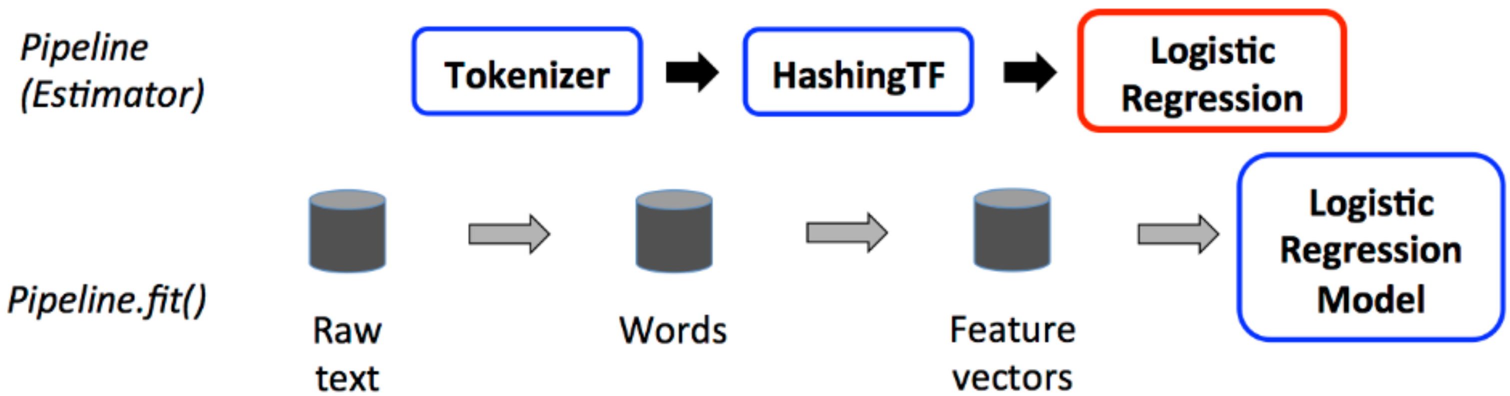
Idea: building complex pipeline out of simple building blocks
(Note: scikit-learn pipelines are basically the same as Spark MLLib ones)

Pipelines

Idea: building complex pipeline out of simple building blocks: e.g. normalization, feature transformation, model fitting.

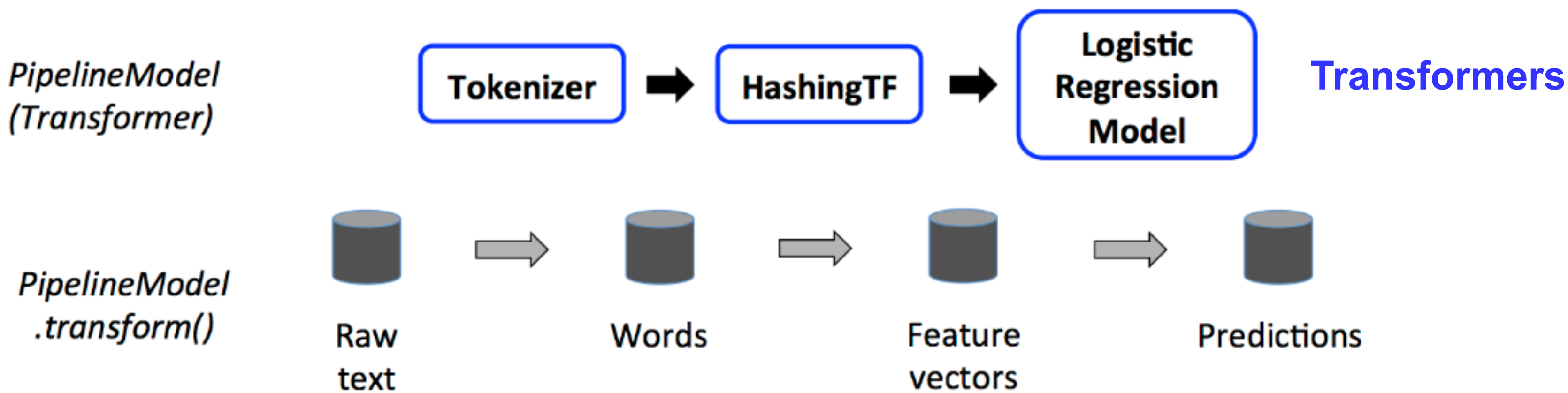
Why?

- Better code reuse: without pipelines, we would repeat a lot of code, e.g. between the training and test pipelines, cross-validation, model variants, etc.
- Easier to perform cross validation, and hyperparameter tuning.



Building Blocks: Transformers

- **Transformers** are for mapping DataFrames to DataFrames
  - Examples: one-hot encoding, tokenization
  - Specifically, a Transformer object has a transform() method, which performs its transformation
- Generally, these transformers output a new DataFrame which ***append*** their result to the original DataFrame.
  - Similarly, a fitted model (e.g. logistic regression) is a Transformer that transforms a DataFrame into one with the predictions appended.



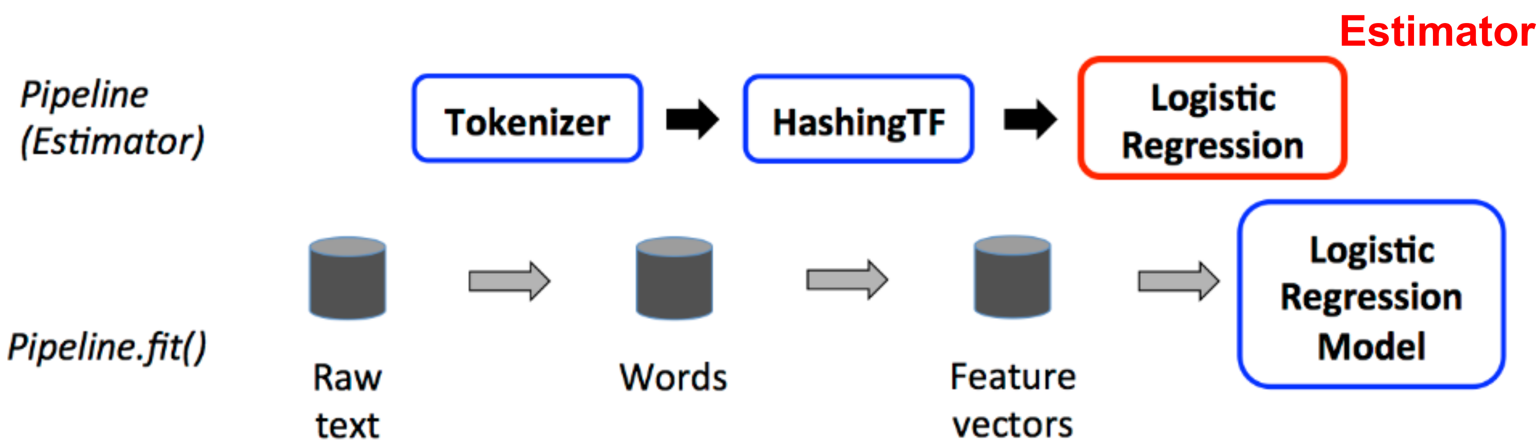
Building Blocks: Estimator

- **Estimator** is an algorithm which takes in data, and outputs a fitted model. For example, a learning algorithm (the LogisticRegression object) can be fit to data, producing the trained logistic regression model.
- They have a fit() method, which returns a Transformer.

```
from pyspark.ml.classification import LogisticRegression

training = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

lr = LogisticRegression(maxIter=10)

lrModel = lr.fit(training)

print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
```

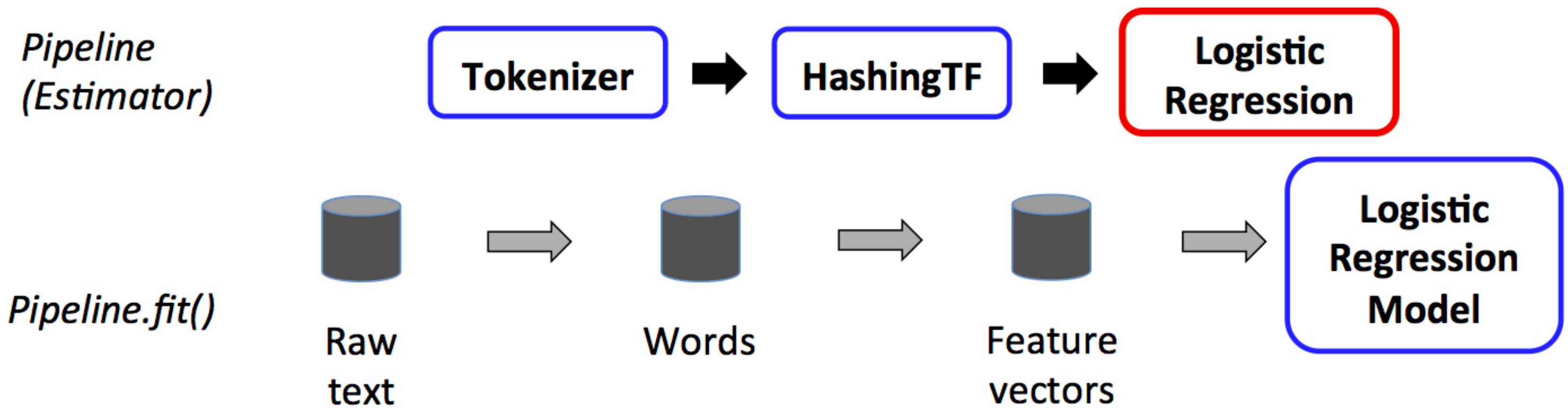
Building Blocks: Estimator

- **Estimator** is an algorithm which takes in data, and outputs a fitted model. For example, a learning algorithm (the LogisticRegression object) can be fit to data, producing the trained logistic regression model.
- They have a fit() method, which returns a Transformer.



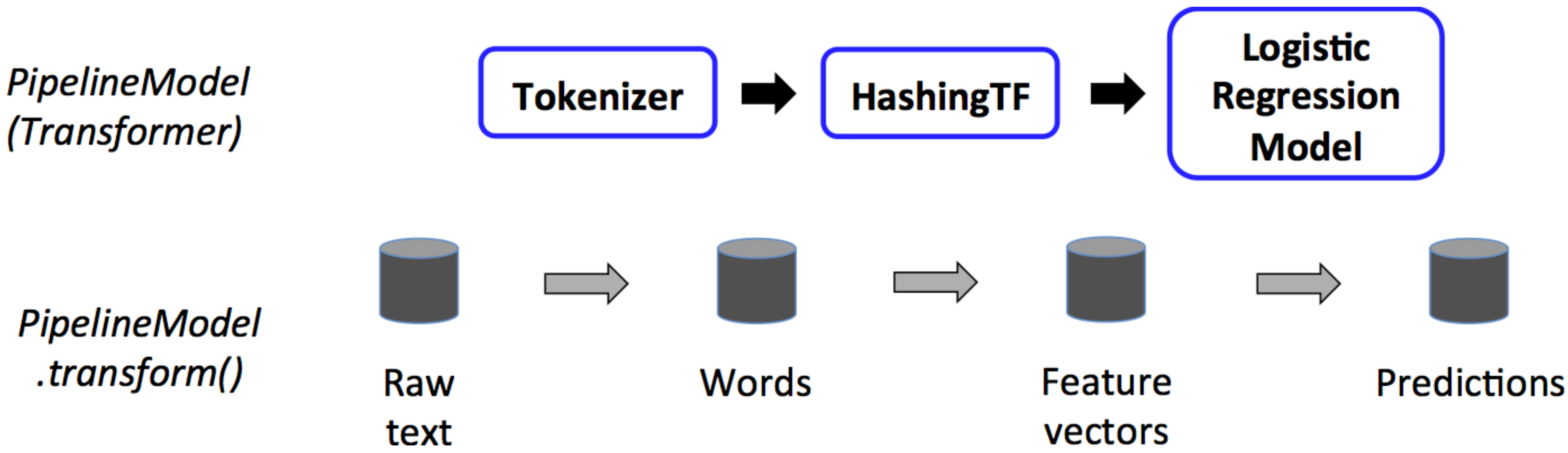
Pipeline: Training Time

- A pipeline chains together multiple Transformers and Estimators to form an ML workflow.
- Pipeline is an Estimator. When Pipeline.fit() is called:
  - Starting from the beginning of the pipeline:
  - For Transformers, it calls transform()
  - For Estimators, it calls fit() to fit the data, then transform() on the fitted model



Pipeline: Test Time

- The output of Pipeline.fit() is the estimated pipeline model (of type PipelineModel).
  - It is a transformer, and consists of a series of Transformers.
  - When its transform() is called, each stage's transform() method is called.



Demo_3: Machine Learning Pipeline

```python
# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])
```

```python
# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

```python
# Fit the pipeline to training documents.
model = pipeline.fit(training)
```

```python
# Make predictions on test documents and print columns of interest.
pred_test = model.transform(test)
pred_test.show()
```

(3) Spark Jobs

pred_test: pyspark.sql.dataframe.DataFrame = [id: long, text: string ... 6 more fields]

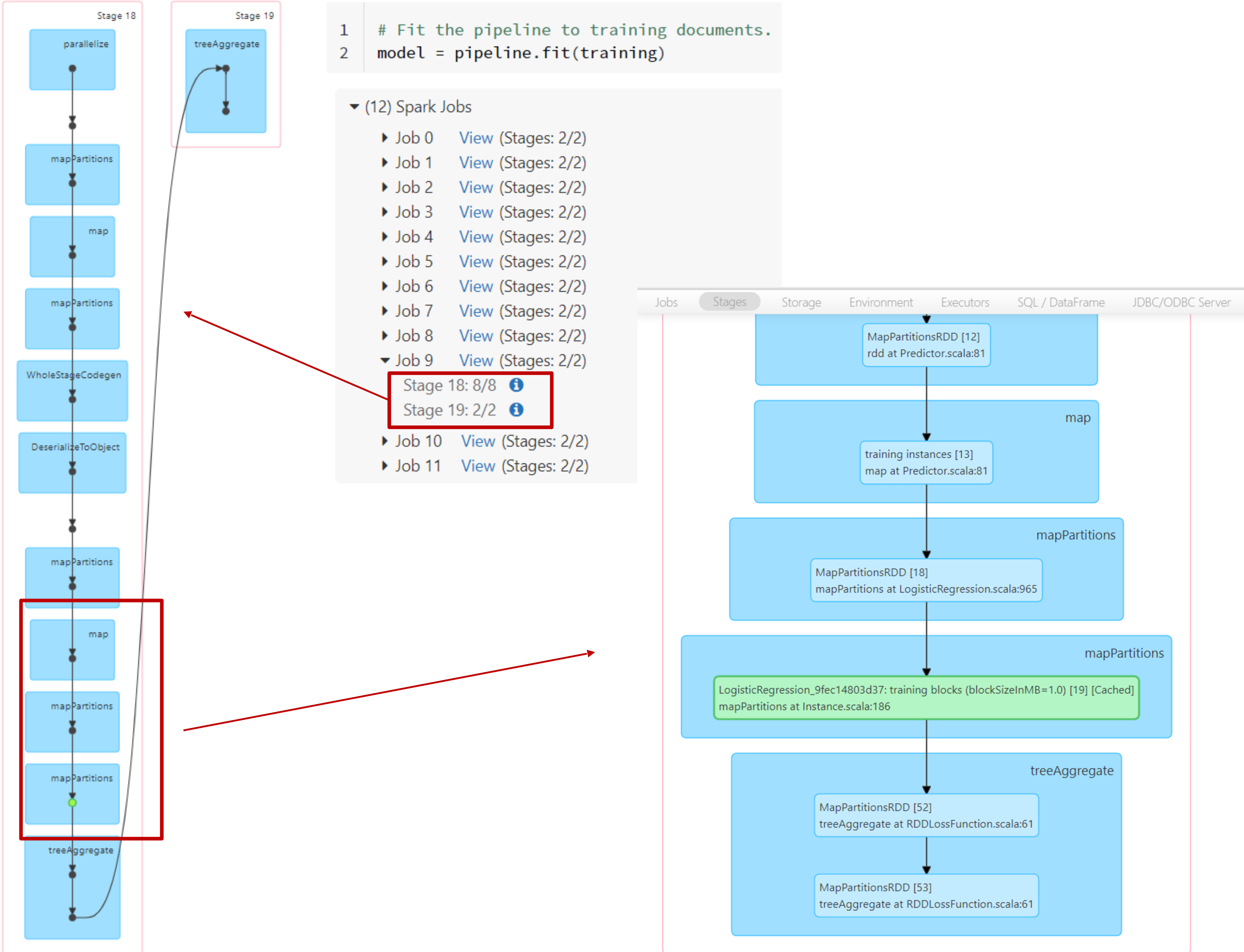
```
+---+-----------------+-----+--------------------+--------------------+--------------------+--------------------+----------+
| id|             text|label|               words|            features|       rawPrediction|         probability|prediction|
+---+-----------------+-----+--------------------+--------------------+--------------------+--------------------+----------+
|  4|      spark i j k|  1.0|    [spark, i, j, k]|(262144,[19036,68...|[0.52882855227968...|[0.62920984896684...|       0.0|
|  5|            l m n|  0.0|           [l, m, n]|(262144,[1303,526...|[4.16914139534005...|[0.98477000676230...|       0.0|
|  6|spark hadoop spark|  1.0|[spark, hadoop, s...|(262144,[173558,1...|[-1.8649814141188...|[0.13412348342566...|       1.0|
|  7|    apache hadoop|  0.0|    [apache, hadoop]|(262144,[68303,19...|[5.41564427200184...|[0.99557321143985...|       0.0|
+---+-----------------+-----+--------------------+--------------------+--------------------+--------------------+----------+
```

```python
# compute accuracy on the test set
predictionAndLabels = pred_test.select("prediction", "label")
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print("Test set accuracy = " + str(evaluator.evaluate(predictionAndLabels)))
```

(1) Spark Jobs

predictionAndLabels: pyspark.sql.dataframe.DataFrame = [prediction: double, label: double]

```
Test set accuracy = 0.75
```



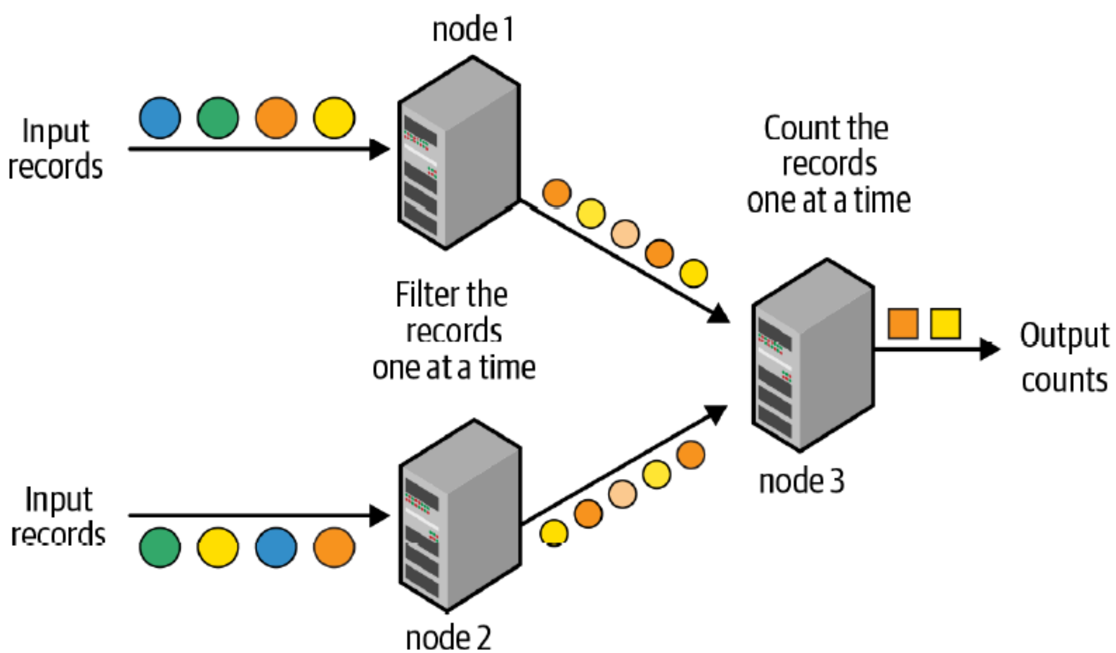
Stage 18
Stage 19
parallelize
mapPartitions
map
mapPartitions
WholeStageCodegen
DeserializeToObject
mapPartitions
map
mapPartitions
mapPartitions
treeAggregate
treeAggregate
1 # Fit the pipeline to training documents.
2 model = pipeline.fit(training)
(12) Spark Jobs
Job 0 View (Stages: 2/2)
Job 1 View (Stages: 2/2)
Job 2 View (Stages: 2/2)
Job 3 View (Stages: 2/2)
Job 4 View (Stages: 2/2)
Job 5 View (Stages: 2/2)
Job 6 View (Stages: 2/2)
Job 7 View (Stages: 2/2)
Job 8 View (Stages: 2/2)
Job 9 View (Stages: 2/2)
Stage 18: 8/8
Stage 19: 2/2
Job 10 View (Stages: 2/2)
Job 11 View (Stages: 2/2)
Jobs
Stages
Storage
Environment
Executors
SQL / DataFrame
JDBC/ODBC Server
MapPartitionsRDD [12]
rdd at Predictor.scala:81
map
training instances [13]
map at Predictor.scala:81
mapPartitions
MapPartitionsRDD [18]
mapPartitions at LogisticRegression.scala:965
mapPartitions
LogisticRegression_9fec14803d37: training blocks (blockSizeInMB=1.0) [19] [Cached]
mapPartitions at Instance.scala:186
treeAggregate
MapPartitionsRDD [52]
treeAggregate at RDDLossFunction.scala:61
MapPartitionsRDD [53]
treeAggregate at RDDLossFunction.scala:61

Today's Plan

- Spark SQL
- Machine Learning with MLlib
- Structured Streaming

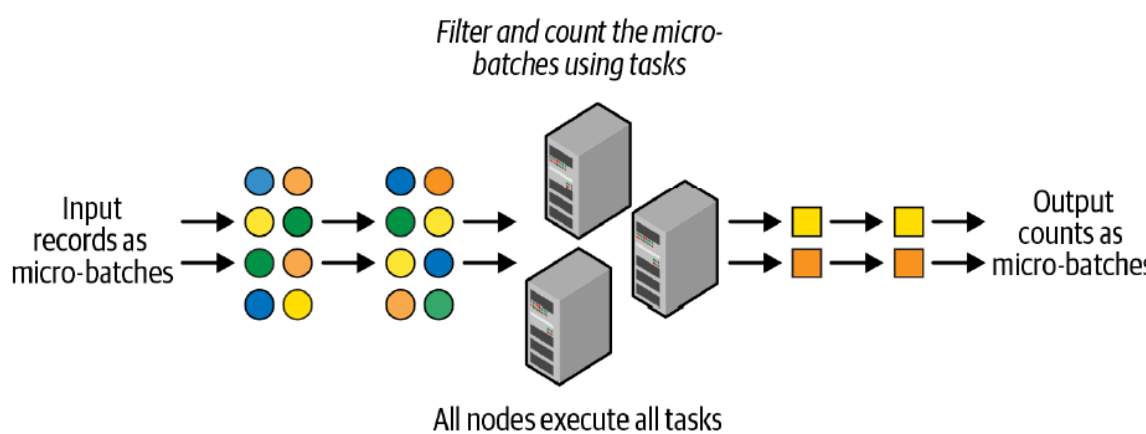
Traditional Model

- Traditional record-at-a-time processing model
  - can achieve very low latencies (e.g. milliseconds)
  - not very efficient at recovering from
    - node failures
    - straggler nodes: nodes that are slower than others



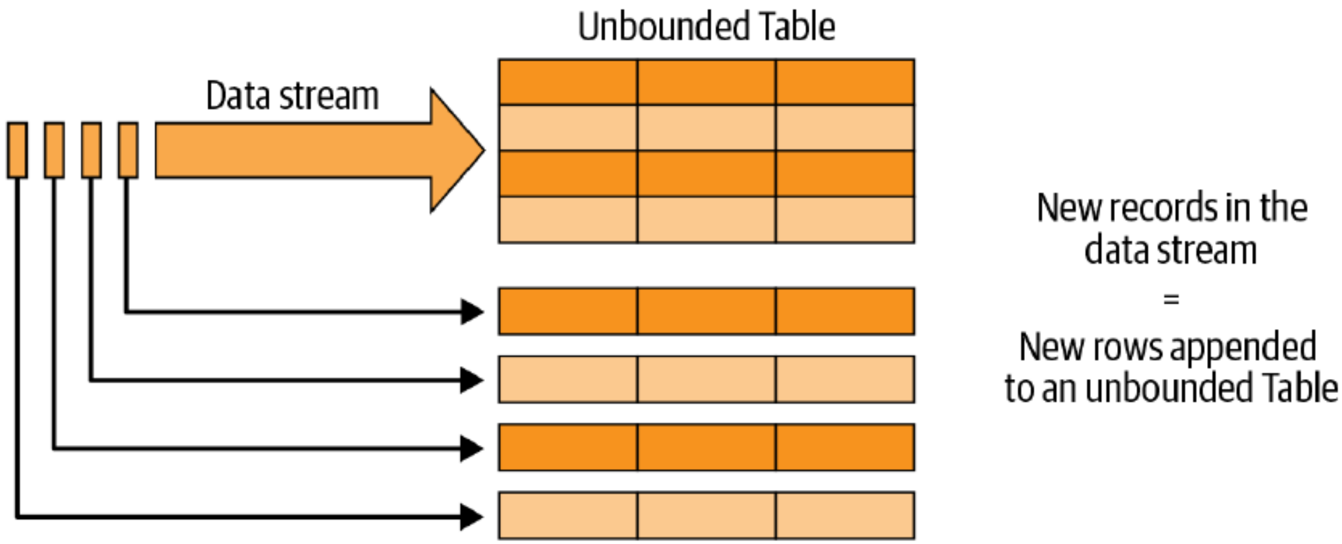
Micro-Batch Stream Processing

- Structured Streaming uses a micro-batch processing model
  - divides the data from the input stream into micro batches
  - each batch is processed in the Spark cluster in a distributed manner
  - small deterministic tasks generate the output in micro-batches
- Advantages over traditional model
  - quickly and efficiently recover from failures and straggler executors
  - deterministic nature ensures end-to-end exactly-once processing guarantees
- Disadvantages: latencies of a few seconds
  - OK for many applications
  - Application may incur more than a few seconds delay in other parts of pipeline



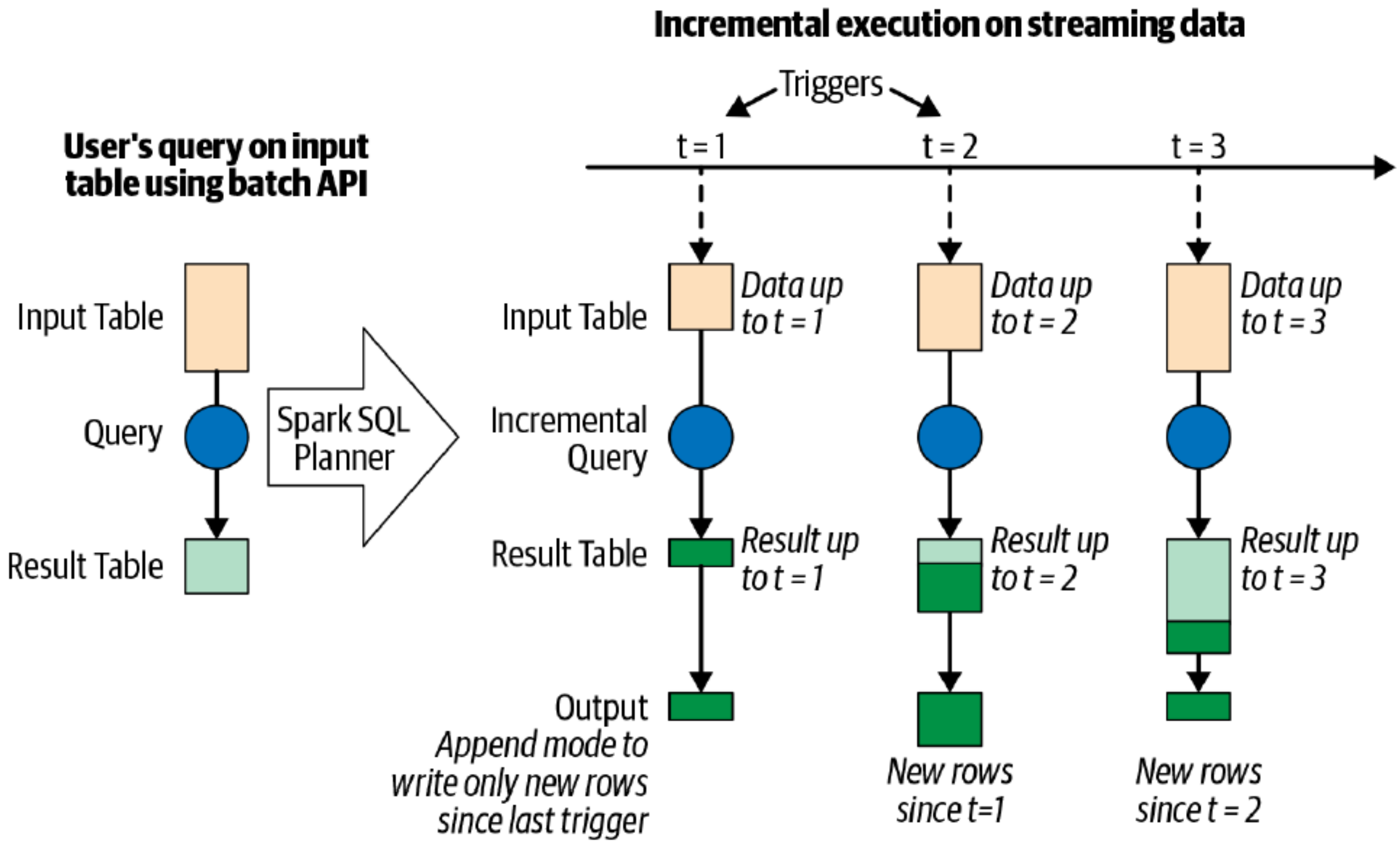
The Philosophy of Structured Streaming

- For developers, writing stream processing pipelines should be as easy as writing batch pipelines.
  - A single, unified programming model and interface for batch and stream processing
  - A broader definition of stream processing
- The Structured Streaming programming model: data stream as an unbounded table



Data stream as an unbound input table

The Structured Streaming processing model



Users express query on streaming data using a batch-like API and Structured Streaming incrementalizes them to run on streams.

Five Steps to Define a Streaming Query

- Step 1: Define input sources
- Step 2: Transform data
- Step 3: Define output sink and output mode
  - Output writing details (where and how to write the output)
  - Processing details (how to process data and how to recover from failures)
- Step 4: Specify processing details
  - Triggering details: when to trigger the discovery and processing of newly available streaming data.
  - Checkpoint Location: store the streaming query process info for failure recovery
- Step 5: Start the query

Practical_3: a simple streaming example

Practical_3 Python

File Edit View Run Help Last edit was 8 minutes ago Give feedback

Cmd 1

```python
spark.conf.set("spark.sql.shuffle.partitions", 5)
```

Command took 0.13 seconds -- by aixin@comp.nus.edu.sg at 2/13/2023, 3:35:13 PM on Test

Cmd 2

```python
static = spark.read.json("/databricks-datasets/definitive-guide/data/activity-data/")
dataSchema = static.schema

```

(3) Spark Jobs

static: pyspark.sql.dataframe.DataFrame = [Arrival_Time: long, Creation_Time: long ... 8 more fields]

Command took 38.98 seconds -- by aixin@comp.nus.edu.sg at 2/13/2023, 3:35:17 PM on Test

Cmd 3

```python
streaming = spark.readStream.schema(dataSchema).option("maxFilesPerTrigger", 1)\
  .json("/databricks-datasets/definitive-guide/data/activity-data")

```

streaming: pyspark.sql.dataframe.DataFrame = [Arrival_Time: long, Creation_Time: long ... 8 more fields]

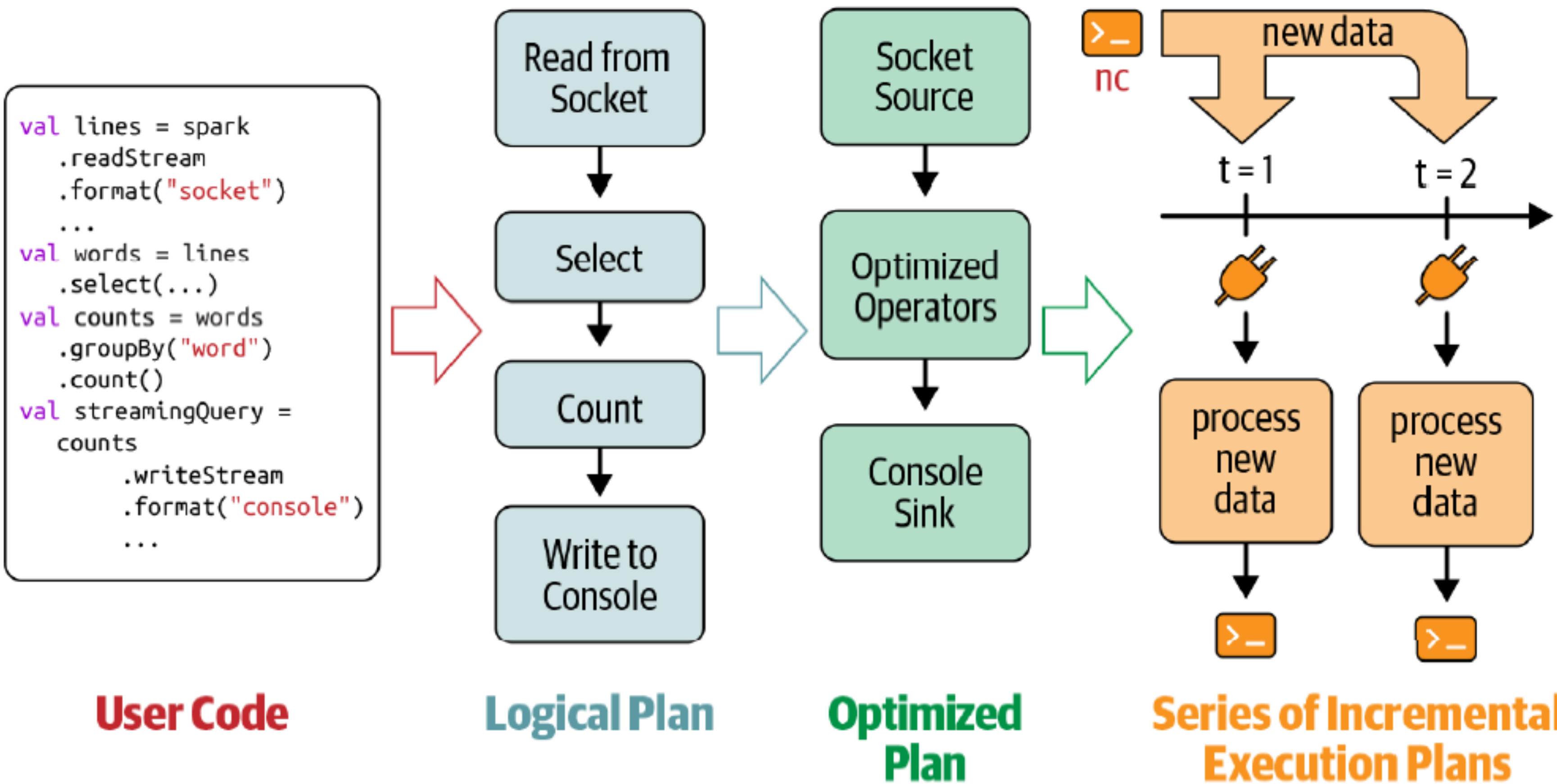
Command took 0.36 seconds -- by aixin@comp.nus.edu.sg at 2/13/2023, 3:26:19 PM on Test

Cmd 4

```python
activityCounts = streaming.groupBy("gt").count()

```

Source: https://github.com/databricks/Spark-The-Definitive-Guide

Incremental execution of streaming queries

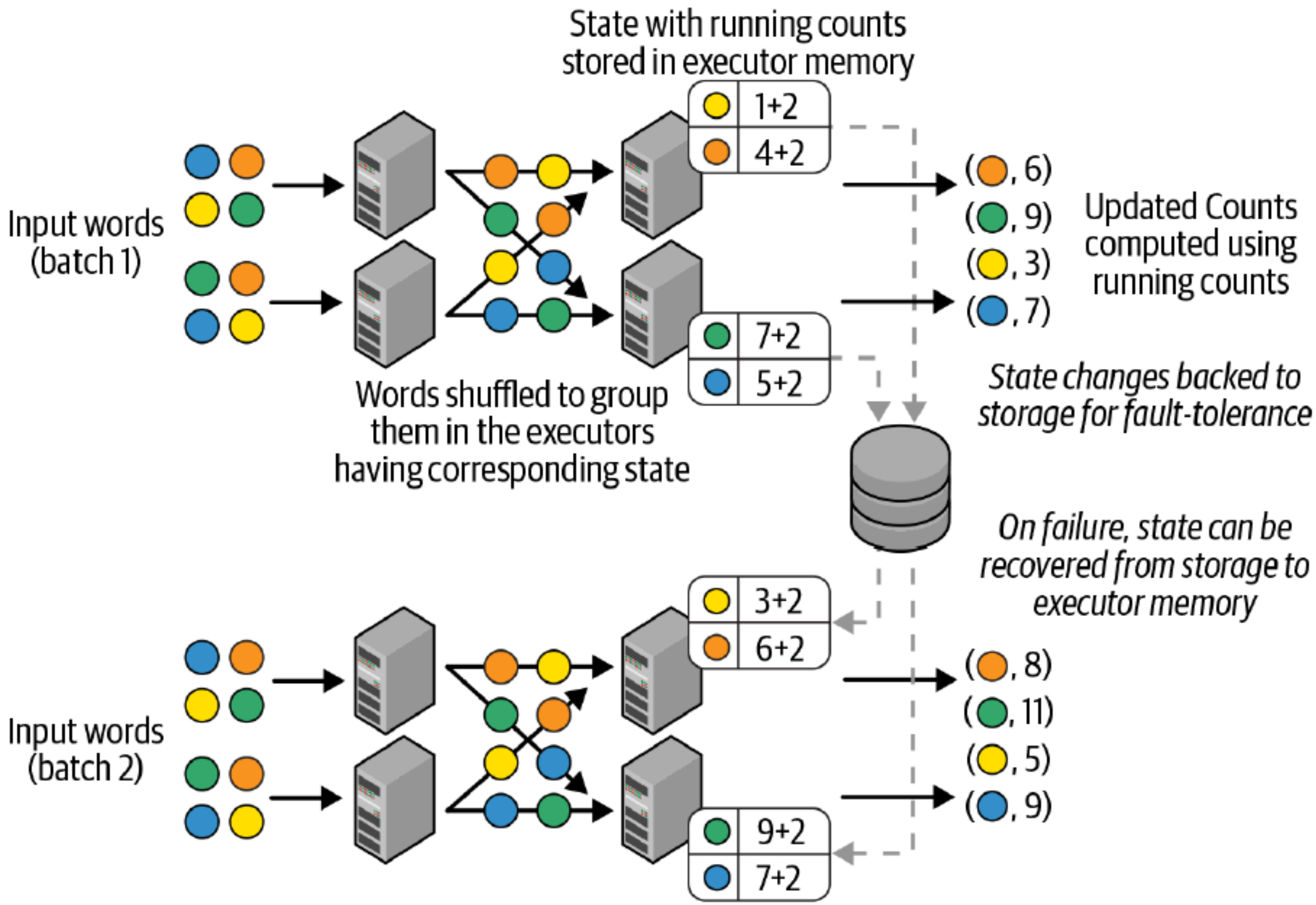


Data Transformation

- Stateless Transformation
  - Process each row individually without needing any information from previous rows
  - Projection operations: select(), explode(), map(), flatMap()
  - Selection operations: filter(), where()

- Stateful Transformation
  - A simple example: DataFrame.groupBy().count()
  - In every micro-batch, the incremental plan adds the count of new records to the previous count generated by the previous micro-batch
  - The partial count communicated between plans is the state
  - The state is maintained in the memory of the Spark executors and is checkpointed to the configured location to tolerate failures.

Distributed state management in Structured Streaming



Stateful Streaming Aggregations

- Aggregations Not Based on Time
  - Global aggregations

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

  - Grouped aggregations

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

  - All built-in aggregation functions in DataFrames are supported
    - sum(), mean(), stddev(), countDistinct(), collect_set(), approx_count_distinct(), and etc.

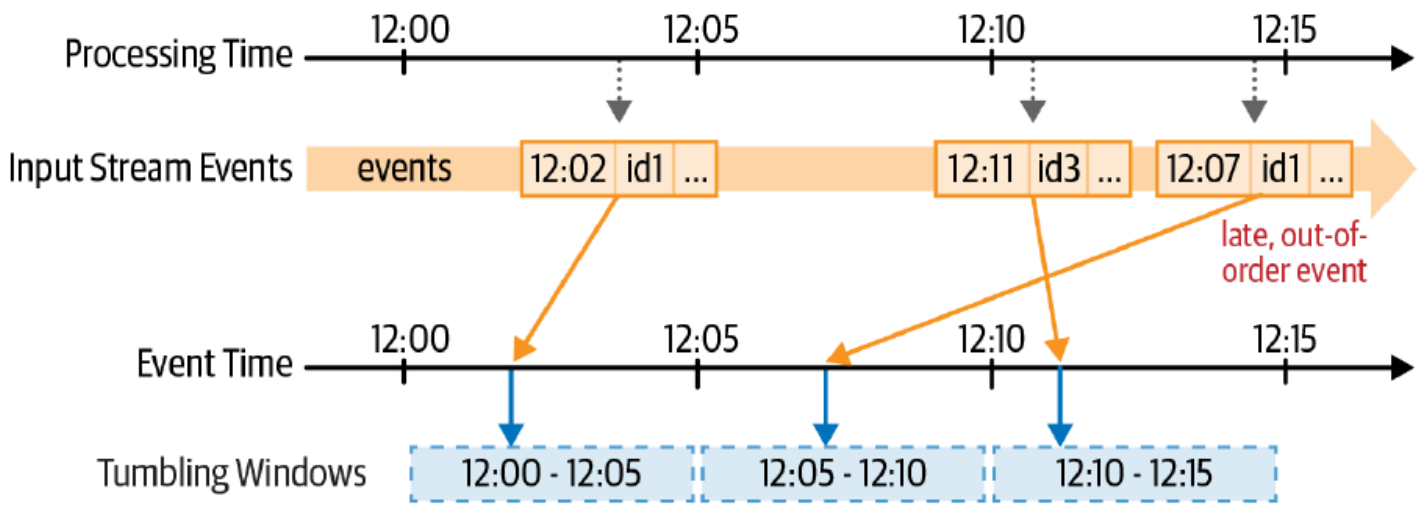
  - You can apply multiple aggregation functions to be computed together

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

Stateful Streaming Aggregations

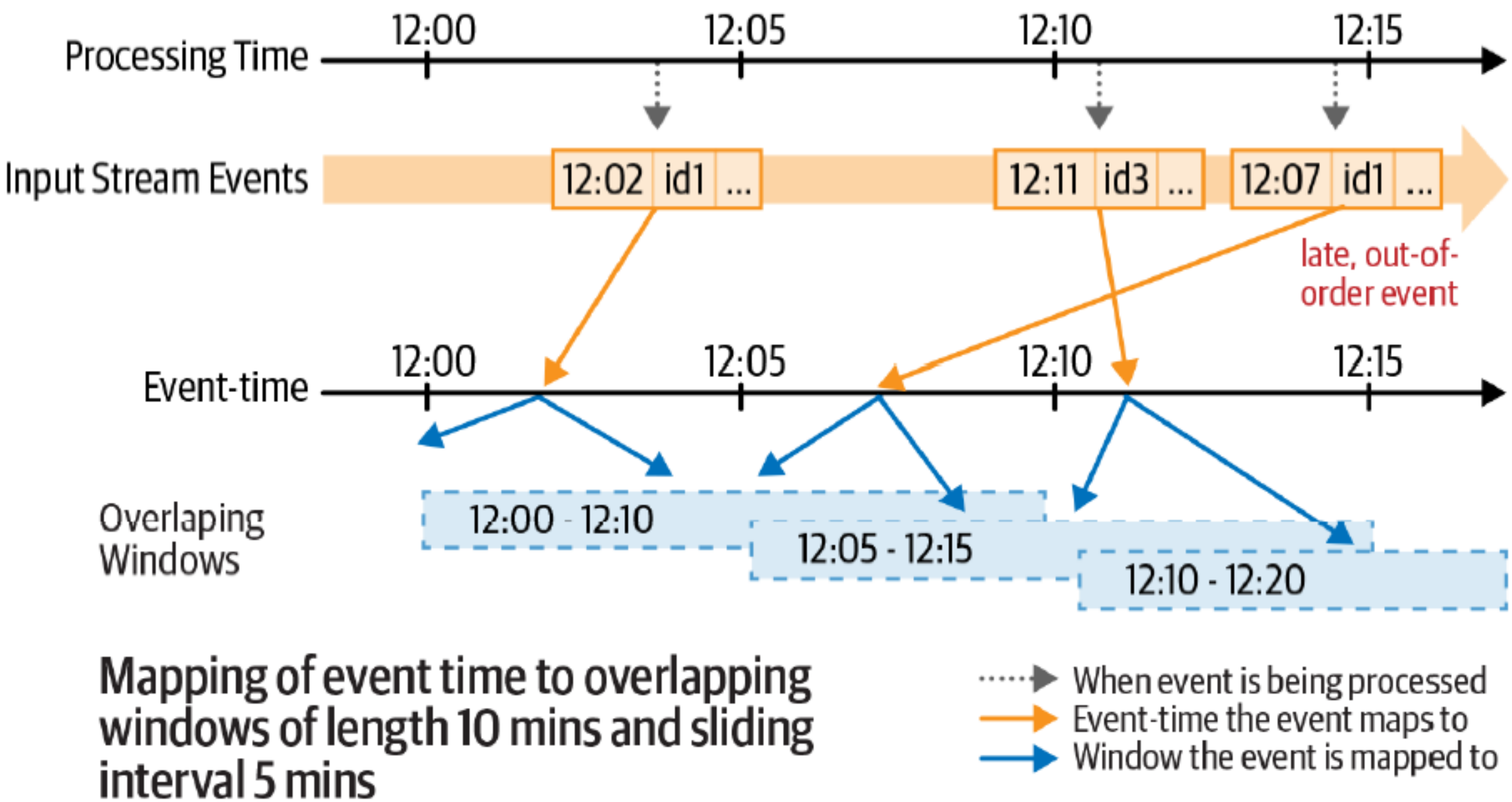
- Aggregations with Event-Time Windows

```
(sensorReadings
  .groupBy("sensorId", window("eventTime", "5 minute"))
  .count())
```

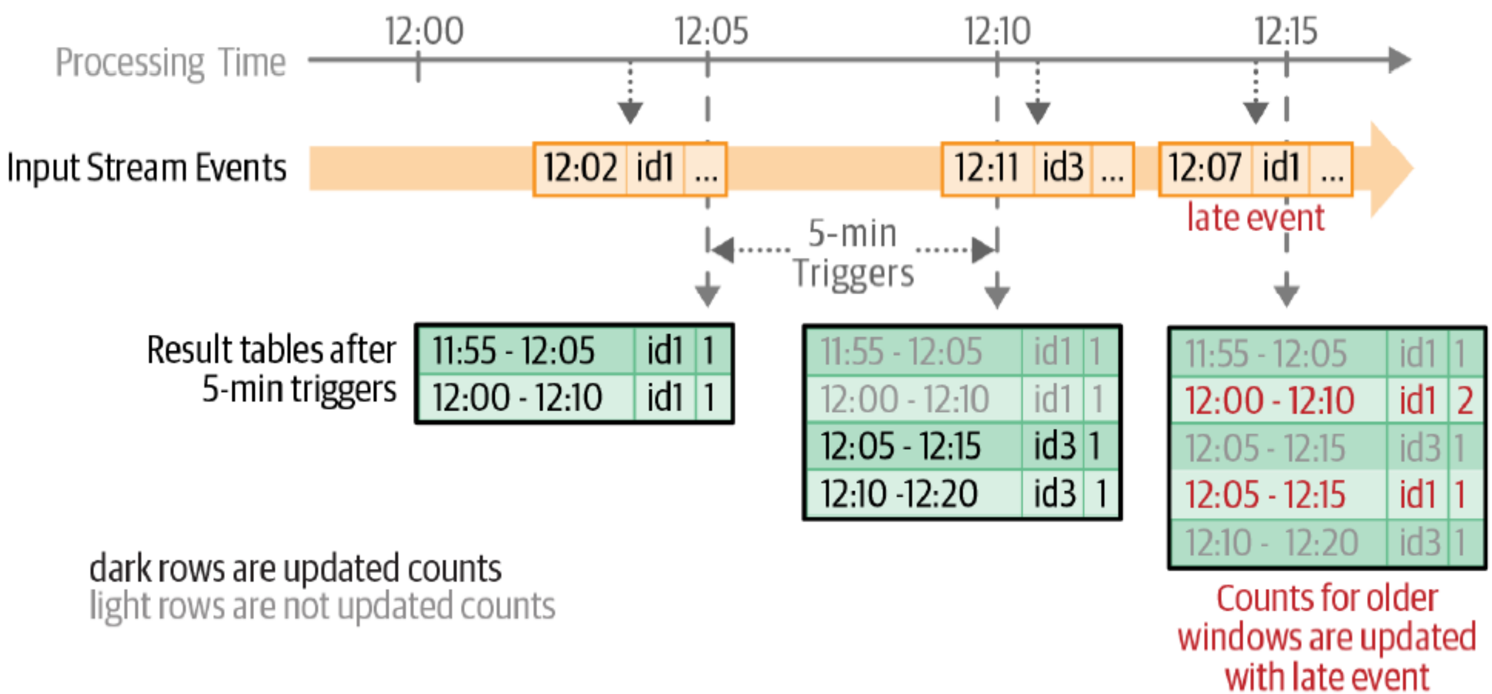


Mapping of event time to 5-min tumbling windows

```
(sensorReadings
  .groupBy("sensorId", window("eventTime", "10 minute", "5 minute"))
  .count())
```

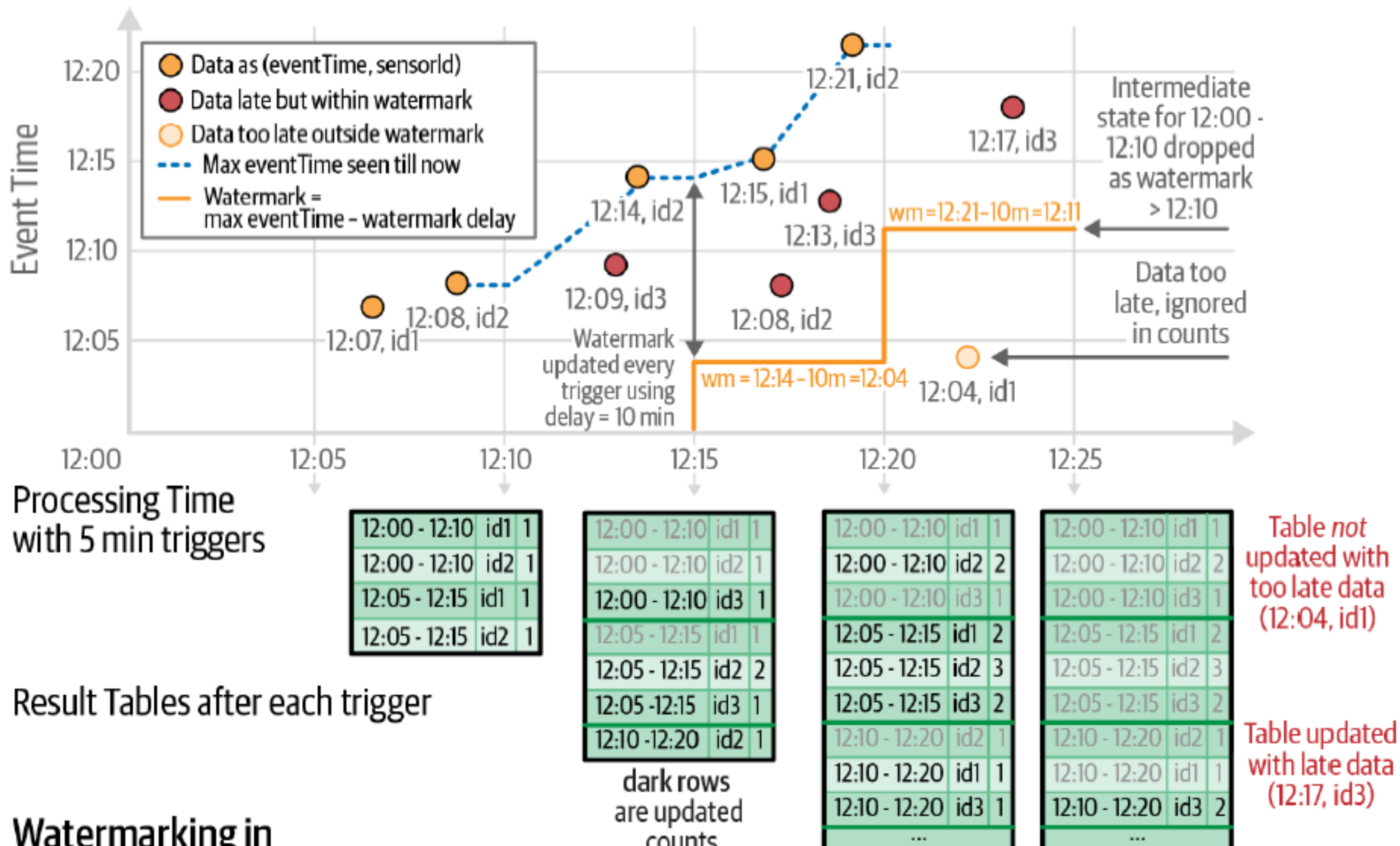


- Updated counts in the result table after each five-minute trigger



- ## Handling Late Data with Watermarks

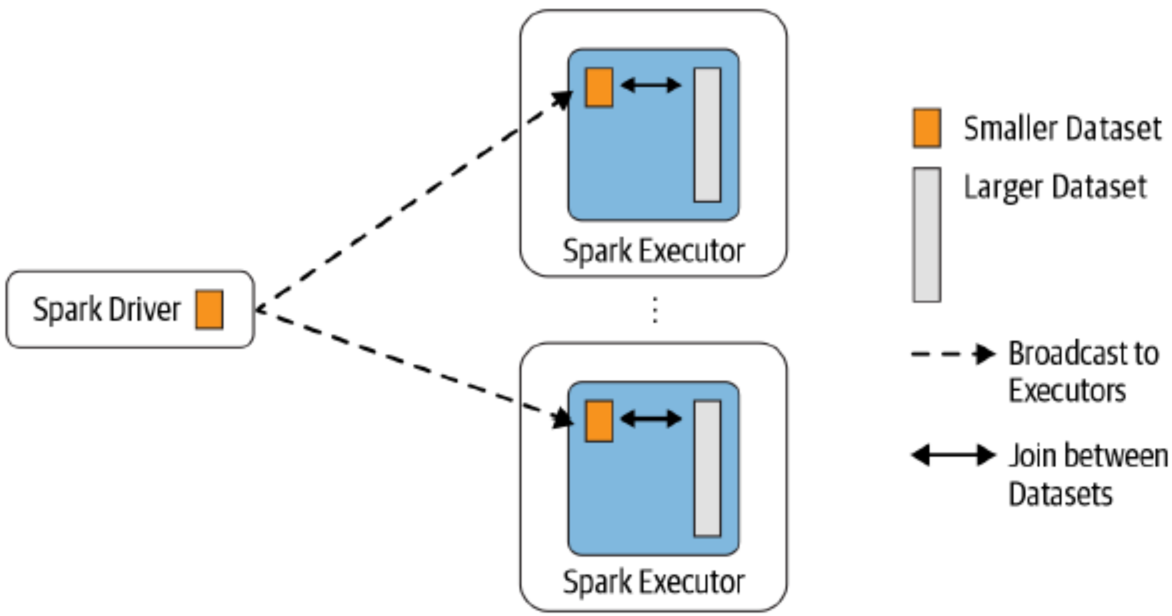
```
(sensorReadings
  .withWatermark("eventTime", "10 minutes")
  .groupBy("sensorId", window("eventTime", "10 minutes", "5 minutes"))
  .count())
```



Watermarking in Windowed Grouped Counts

Spark Join

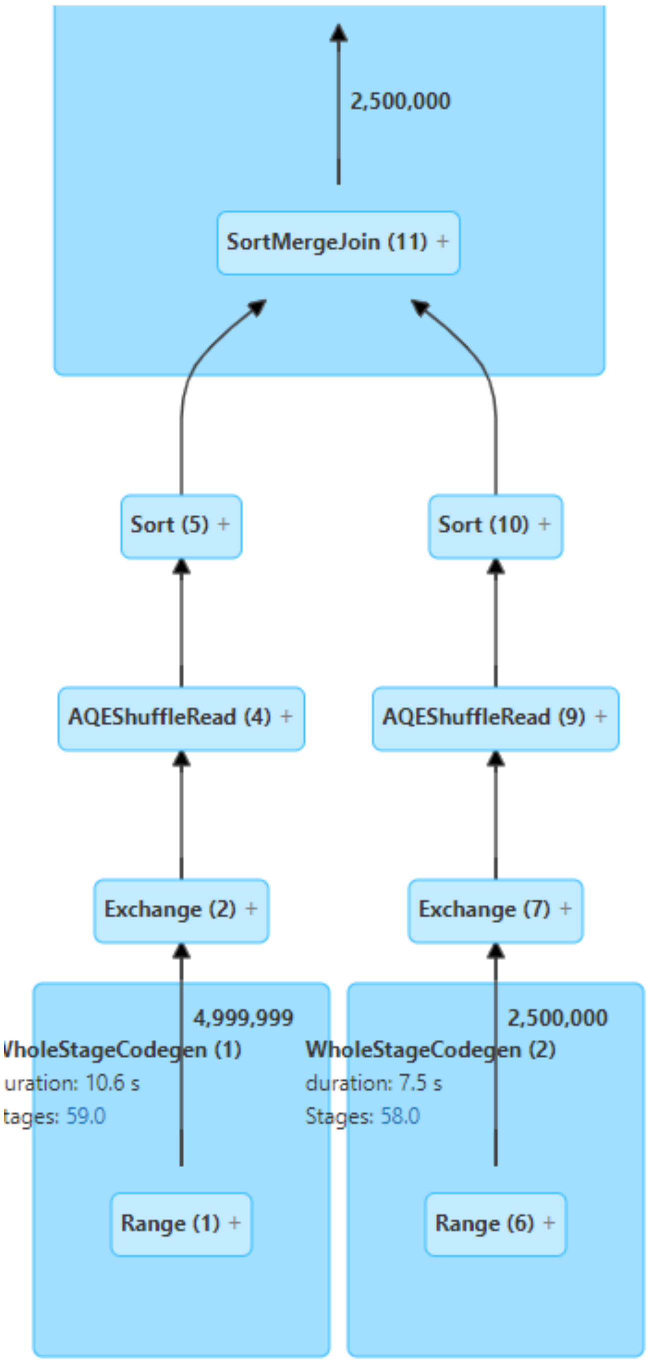
- Broadcast Hash Join (a.k.a. map-side-only join)
  - the smaller data set is broadcast to all executors



Spark Join

- Shuffle Sort Merge Join
  - an efficient way to merge two large data sets over a common key that is sortable, unique, and can be assigned to or stored in the same partition

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
df3 = df1.join(df2, ["id"])
df3.count()
```



Streaming Join

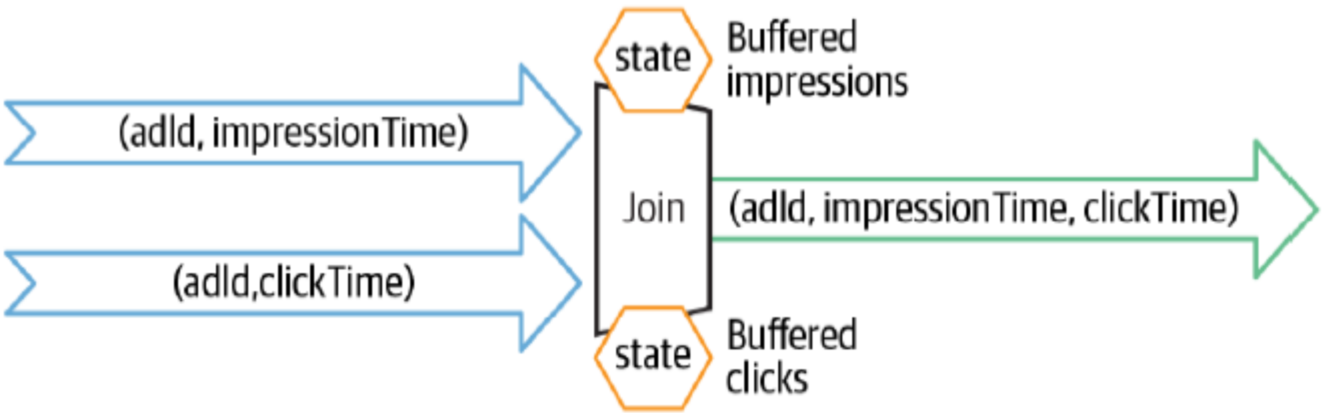
- Stream-Static Join

```
# Static DataFrame [adId: String, impressionTime: Timestamp, ...]
# reading from your static data source
impressionsStatic = spark.read. ...

# Streaming DataFrame [adId: String, clickTime: Timestamp, ...]
# reading from your streaming source
clicksStream = spark.readStream. ...

matched = clicksStream.join(impressionsStatic, "adId")
```

  - Stateless Operations, no need watermarking
  - Cache the static DataFrame to speed up the repeatedly reads

Streaming Join

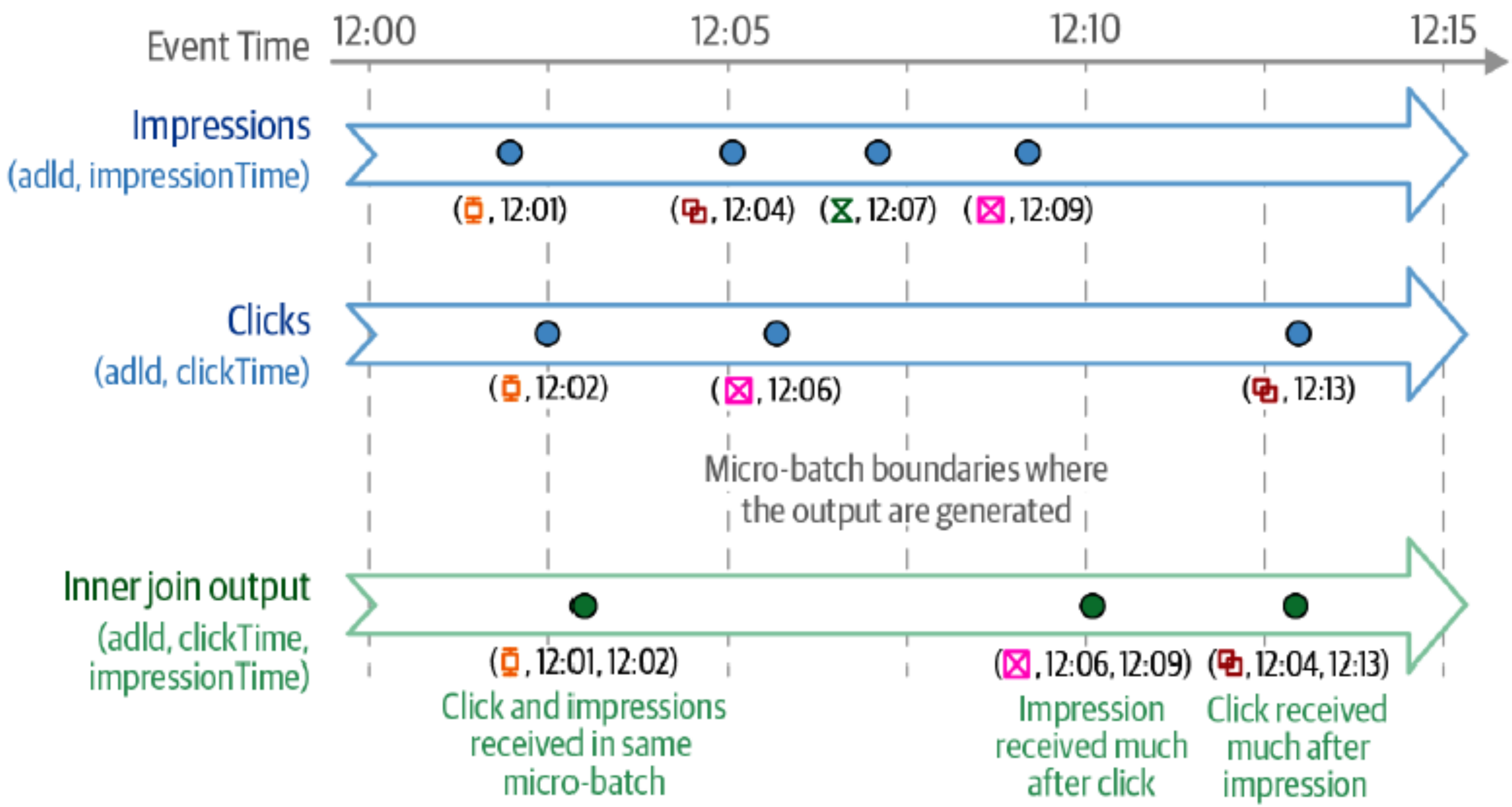
- Stream-Stream join
  - Finding the matching events from the two buffered streams



Stream−stream join use case: Ad monetization (joining ad clicks to impressions)

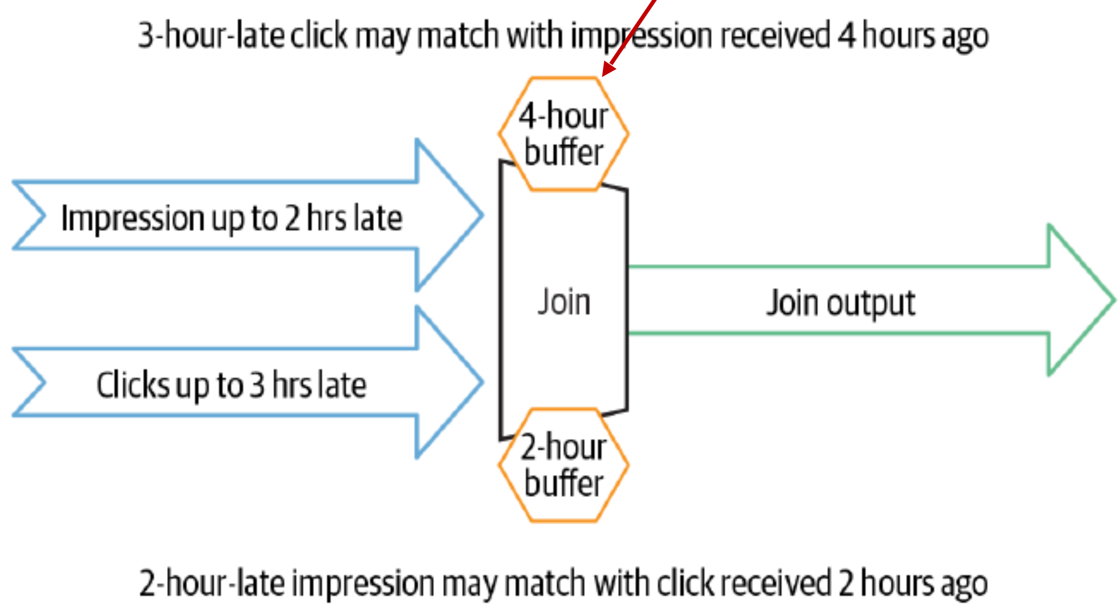
```
# Streaming DataFrame [adId: String, impressionTime: Timestamp, ...]
impressions = spark.readStream. ...

# Streaming DataFrame[adId: String, clickTime: Timestamp, ...]
clicks = spark.readStream. ...
matched = impressions.join(clicks, "adId")
```

- Inner joins with optional watermarking



```python
# Define watermarks
impressionsWithWatermark = (impressions
  .selectExpr("adId AS impressionAdId", "impressionTime")
  .withWatermark("impressionTime", "2 hours"))

clicksWithWatermark = (clicks
  .selectExpr("adId AS clickAdId", "clickTime")
  .withWatermark("clickTime", "3 hours"))

# Inner join with time range conditions
(impressionsWithWatermark.join(clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime BETWEEN impressionTime AND impressionTime + interval 1 hour""")))
```

3 hours late click + up to 1 hour delay between the impression and click



Outer joins with watermarking

- For correct outer join results and state cleanup, the watermarking and event-time constraints must be specified.
  - for generating the NULL results, the engine must know when an event is not going to match with anything else in the future

```
# Define watermarks
impressionsWithWatermark = (impressions
  .selectExpr("adId AS impressionAdId", "impressionTime")
  .withWatermark("impressionTime", "2 hours"))

clicksWithWatermark = (clicks
  .selectExpr("adId AS clickAdId", "clickTime")
  .withWatermark("clickTime", "3 hours"))


# Left outer join with time range conditions
(impressionsWithWatermark.join(clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime BETWEEN impressionTime AND impressionTime + interval 1 hour"""),
  "leftOuter"))  # only change: set the outer join type
```

Performance Tuning

- Besides tuning Spark SQL engine, a few other considerations
  - Cluster resource provisioning appropriately to run 24/7
  - Number of partitions for shuffles to be set much lower than batch queries
  - Setting source rate limits for stability
  - Multiple streaming queries in the same Spark application

Acknowledgements

- CS4225 slides by He Bingsheng and Bryan Hooi
- Jules S. Damji, Brooke Wenig, Tathagata Das & Denny Lee, "Learning Spark: Lightning-Fast Data Analytics"
- Bill Chambers, Matei Zaharia, "Spark: The Definitive Guide"
- Spark SQL: Relational Data Processing in Spark, SIGMOD'15
- https://spark.apache.org/docs/latest/ml-pipeline.html