W03 –
SPARK ARCHITECTURE & CORE COMPONENTS

The hoang

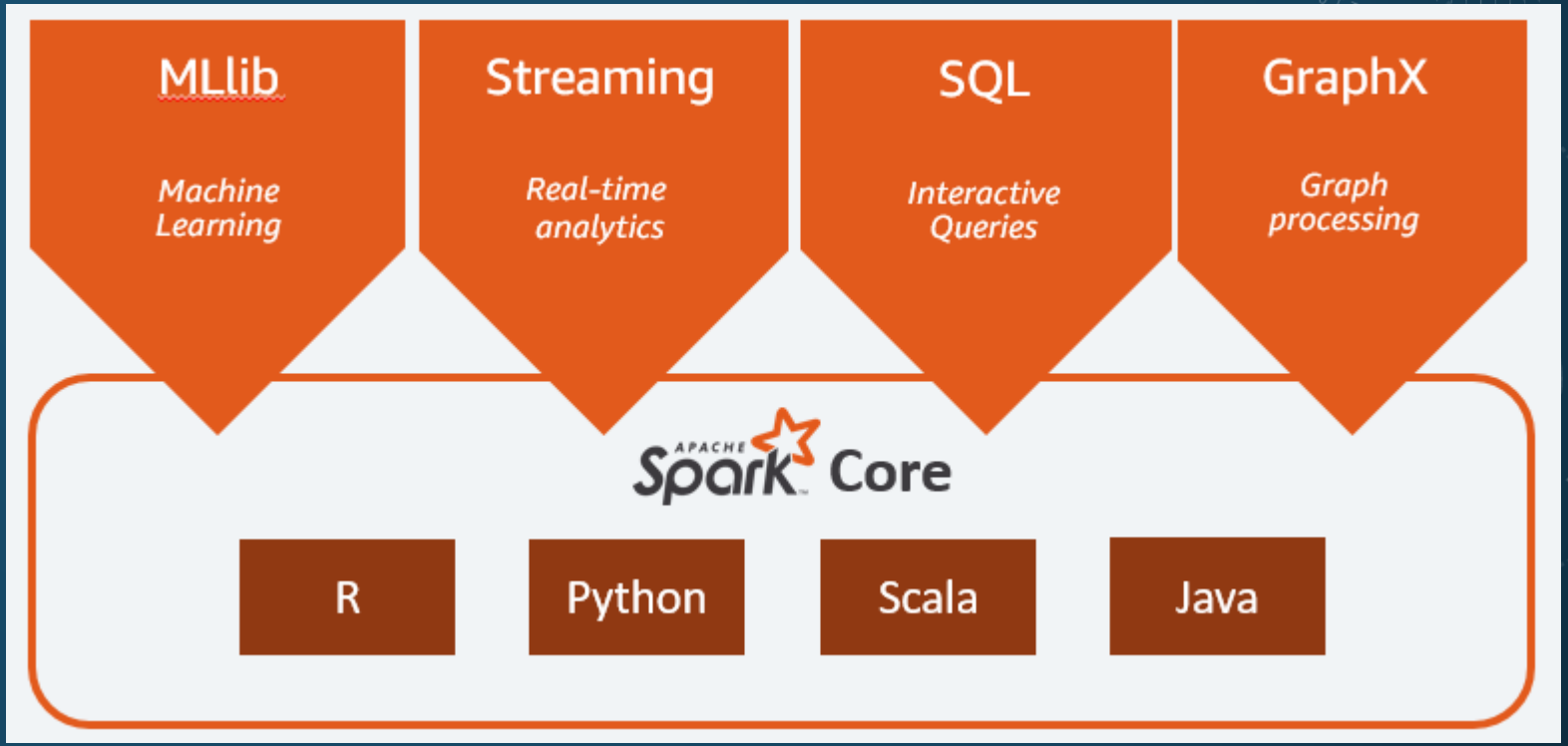
LEARNING OBJECTIVES – W03

- Understand the core architecture of Apache Spark.
- Learn about Resilient Distributed Datasets (RDDs) and their role in Spark.
- Explore DataFrames, Datasets and the benefits they offer over RDDs.
- Discover how to use Spark SQL to query structured data.
- Environment setup, hands-on activities.

Apache Spark – Introduction

Overview

Apache Spark is an *open-source* *distributed* computing system that provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.



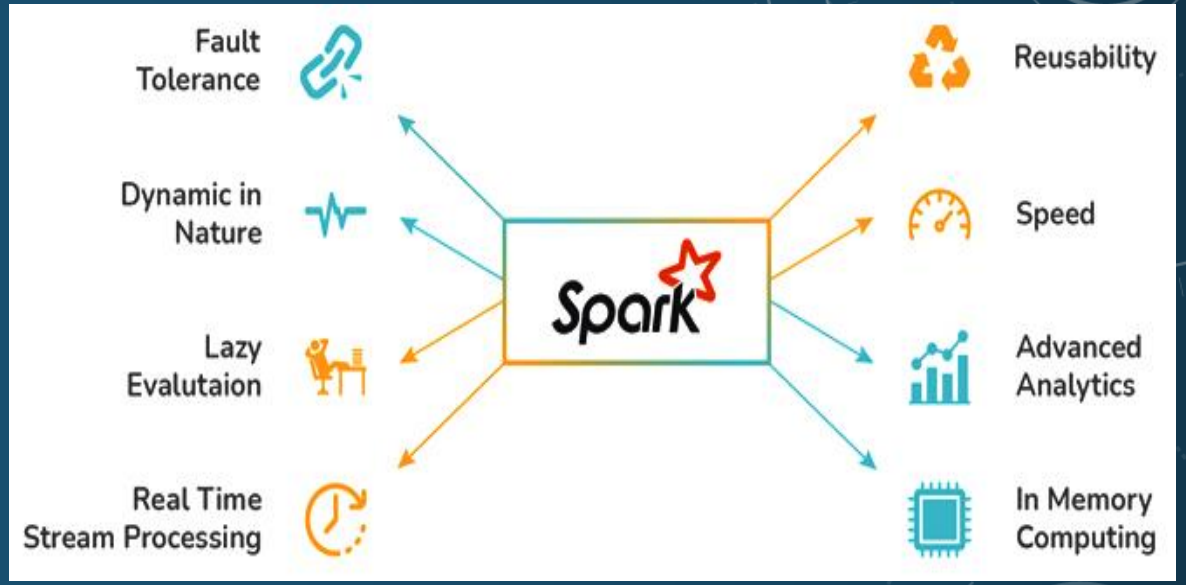
Core components:

1. *Spark Core*
2. *Spark SQL*
3. Spark Streaming
4. ML
5. Graph

Apache Spark – Introduction

Key Features

- **Fault tolerance**: automatically recover from node failures
- **Scalability**: can handle massive workloads by scaling to thousands of machines, process terabytes of data
- **Speed**: In-memory (RAM). 100x faster HADOOP
- **Muti-language support**: Java, Scala, Python, and R.
- **Lazy evaluation**
- **Unified Engine**: Single runtime & platform for different data processing tasks: batch; streaming; interactive queries; ML; Graph
- **Integration**: can integrates with other big data technologies like Hadoop, Kafka



Apache Spark – Introduction

Use Cases

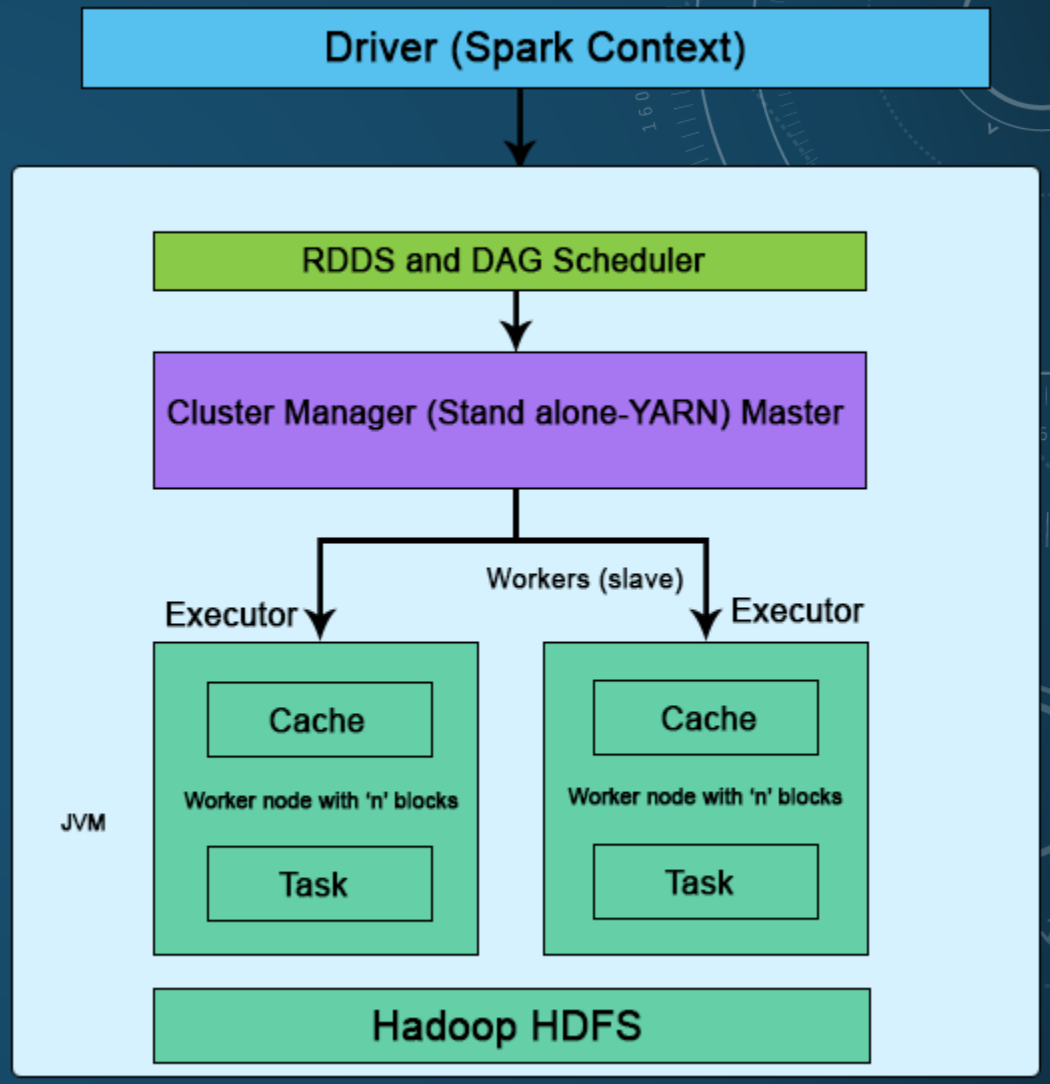
- Real-time data processing and streaming analytics
- Large-scale data transformation. ETL (Extract, Transform, Load) pipelines
- Machine learning & Advanced Analytics e.g. classification, regression, clustering, recommendation systems …
- Graph processing: Social network analysis, recommendation engine, fraud detection

Spark Architecture Overview - Core Components

Driver: The process that orchestrates the execution of the Spark application. It converts the user's code into a DAG (Directed Acyclic Graph) of tasks.

Executors: The distributed worker processes that run the tasks assigned by the driver. Each executor has its own memory and storage.

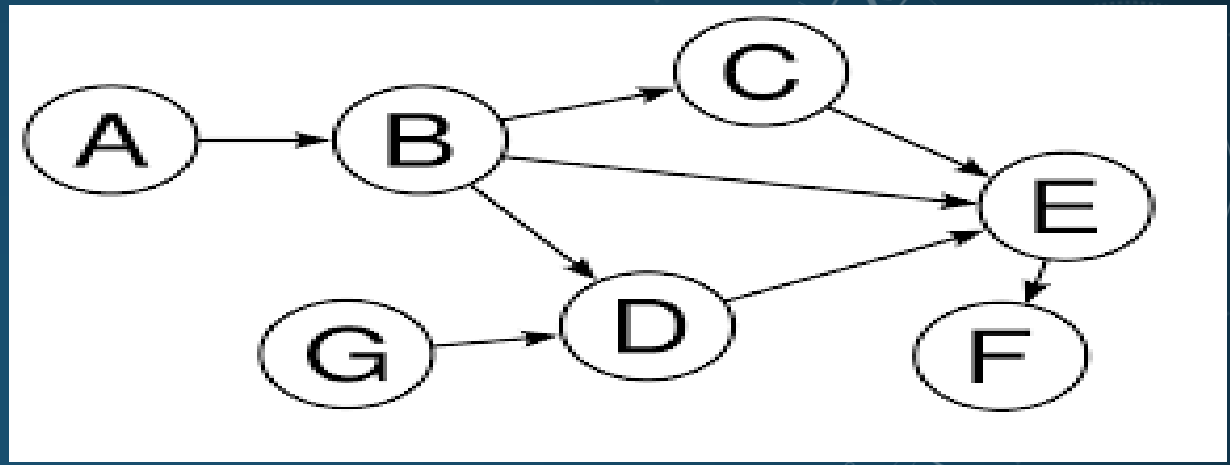
Cluster Manager: This is the component responsible for managing and allocating resources across the cluster. Examples include YARN, Mesos, and Kubernetes.

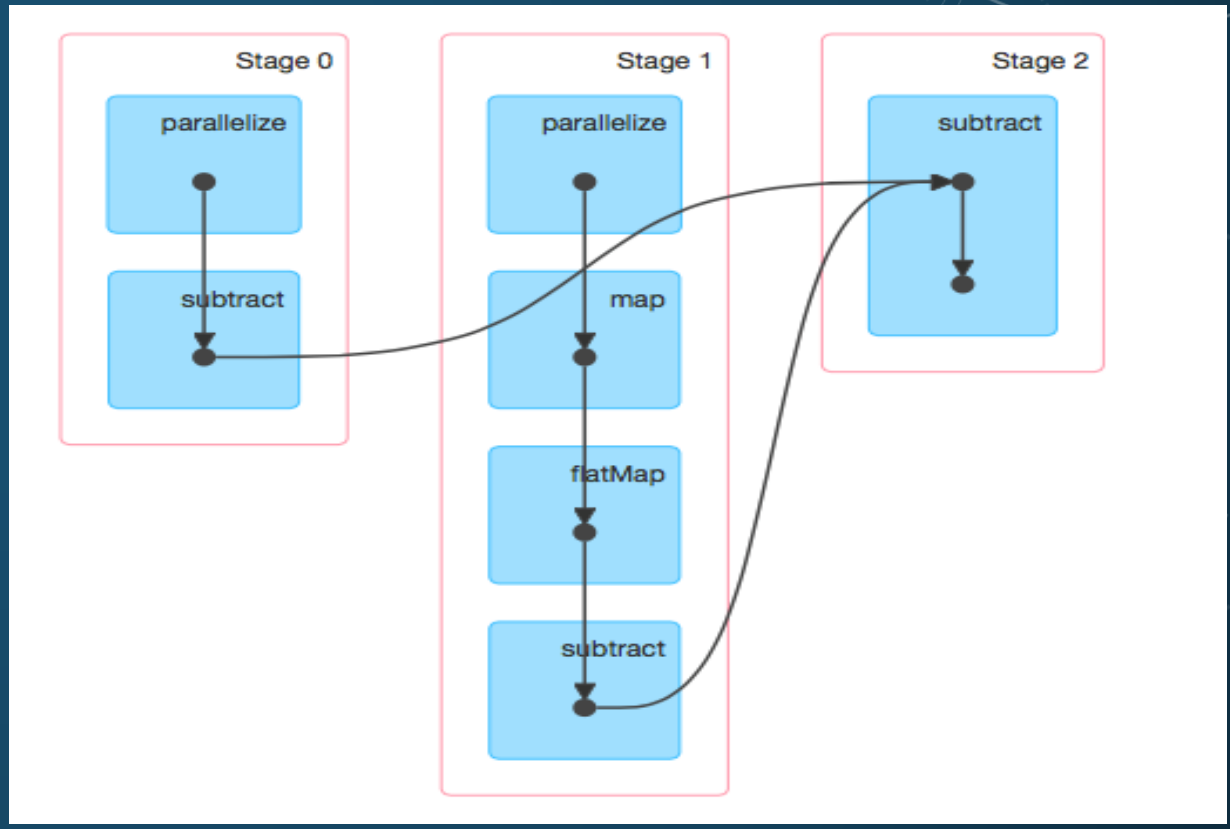


Spark Architecture Overview - DAG

A *directed* *acyclic* graph (DAG) is a conceptual representation of a series of activities. The order of the activities is depicted by a graph, which is visually presented as a set of circles, each representing an activity, some of which are connected by lines, representing the flow from one activity to another.

In Apache Spark, DAG is a fundamental concept used by Spark's execution engine to represent and optimize the flow of operations in a data processing job.





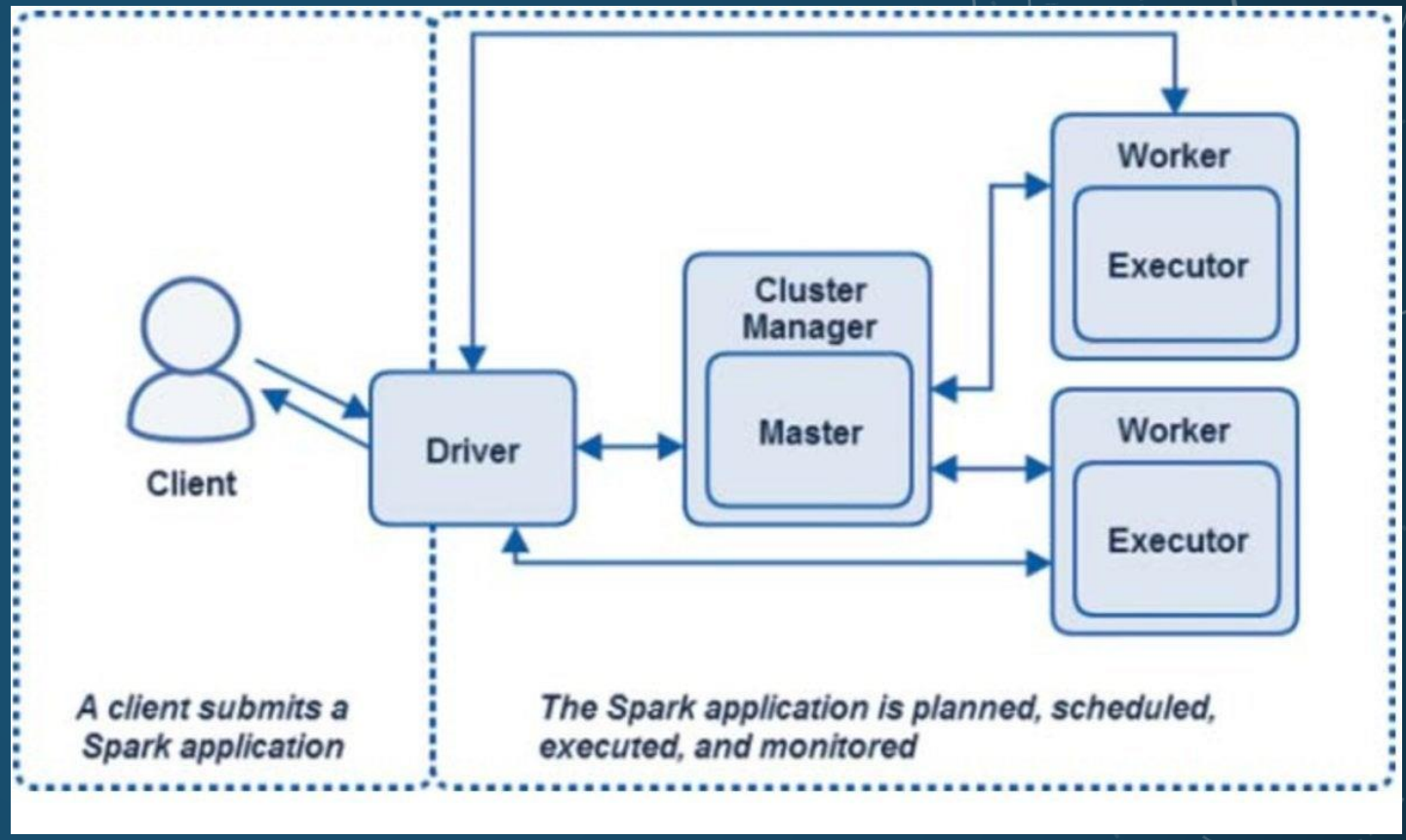
Spark Architecture Overview - Execution Flow

Job Submission: The user submits a Spark application.

DAG Creation: The driver converts the operations into a DAG.

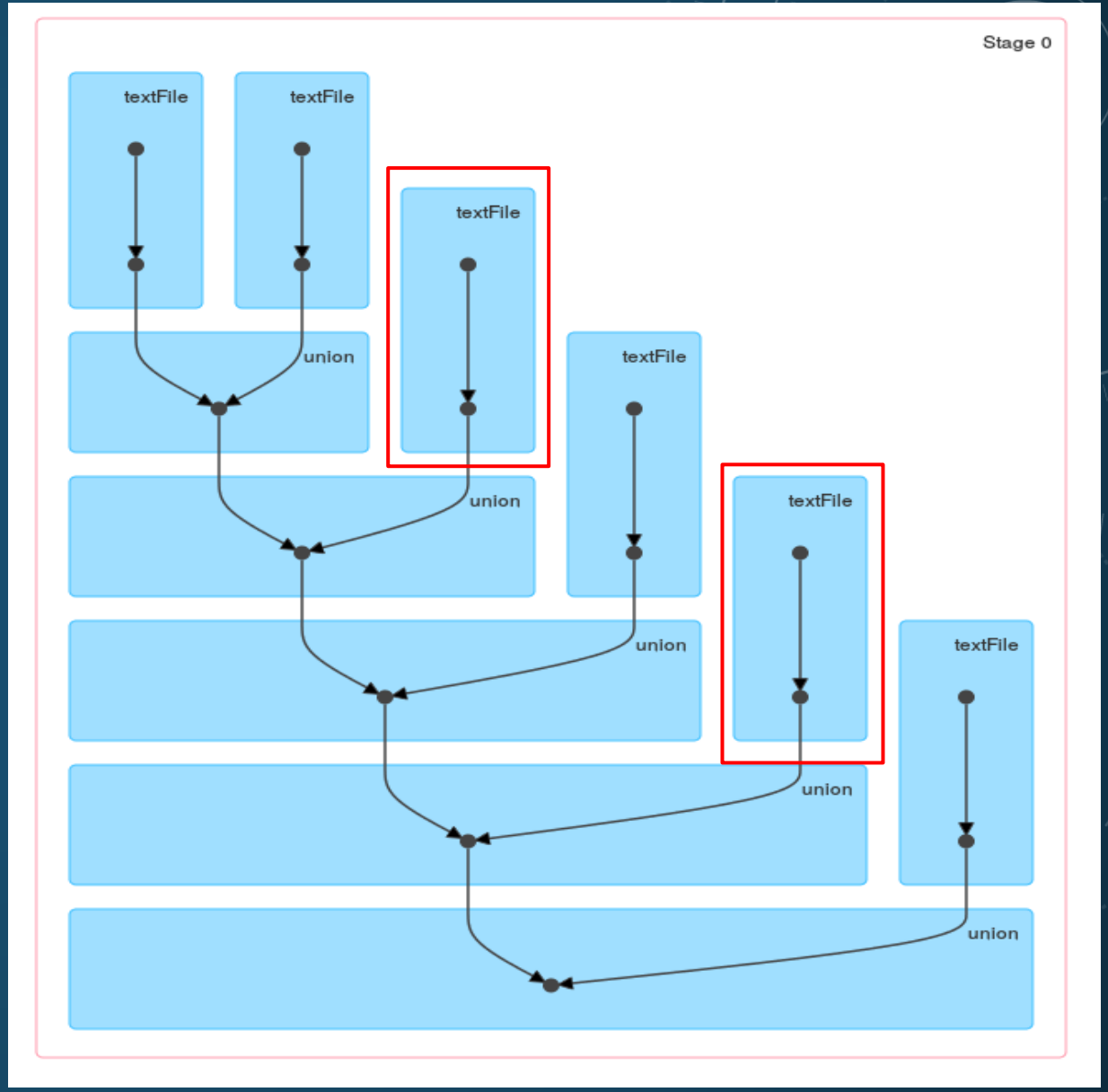
Task Scheduling: The DAG is broken down into tasks, which are scheduled on executors.

Task Execution: Executors execute the tasks and return the results to the driver.



Spark Architecture Overview - Fault Tolerance

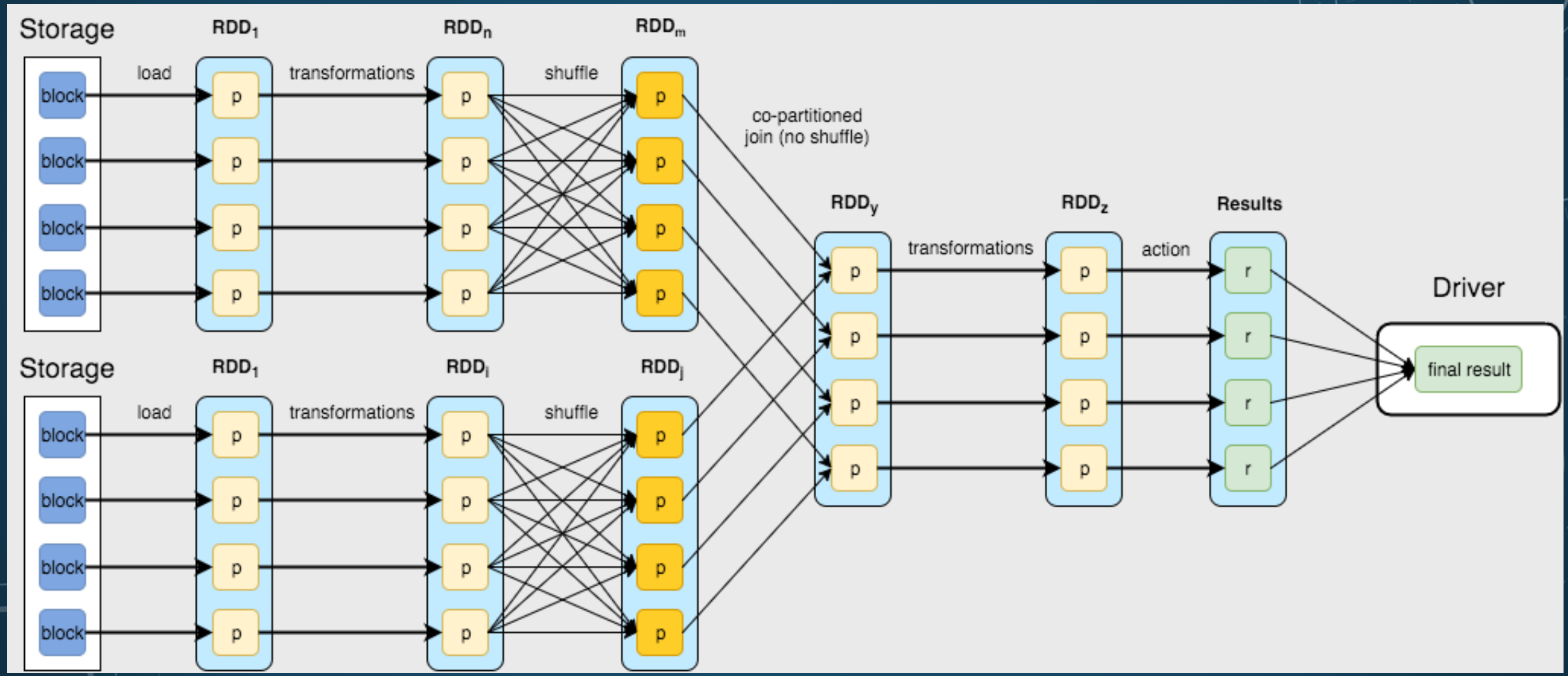
Spark automatically retries failed tasks and uses *lineage* information to recompute lost data.



Resilient Distributed Datasets (RDDs)

- The fundamental data structure in Spark.
- An *immutable* collection of elements *distributed* across multiple nodes.
- Supports *transformations* (operations that create new RDDs) and *actions* (operations that compute and return results).
- There are 2 types of transformation: *wide transformation* and *narrow transformation*
  - Narrow: each input partition contributes to only one output partition; do not shuffle; executed in single stage.
  - Wide: one input partition can contribute to multiple output partitions; data is shuffled, resulting in a stage boundary.
- RDDs maintain a *lineage* graph, which records the sequence of transformations that created the RDD.
- Lazy Evaluation: Transformations on RDDs are *not executed immediately*. They are only executed *when an action is called*.
- Examples of transformations
  - Narrow: map, filter, flatMap, union, coalesce, …
  - Wide: groupByKey, reduceByKey, join, cogroup, distinct, …
- Examples of actions: *collect*, count, first(n), take(n), reduce.

Resilient Distributed Datasets (RDDs)



Resilient Distributed Datasets (RDDs) – Examples

RDD Creation: An RDD can be created from a list of elements, a file, or another RDD.

Python

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("RDDEExample").getOrCreate()

# Create an RDD from a list
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])

# Create an RDD from a file
textRDD = spark.sparkContext.textFile("path/to/your/file.txt")
```

Results:

- rdd = [1, 2 , 3, 4, 5]
- The textRDD variable will contain an RDD where each element is a line from the specified file.

Resilient Distributed Datasets (RDDs) - Examples

RDD Transformations: Create new RDDs based on existing RDDs.

```python
doubledRDD = rdd.map(lambda x: x * 2)
filteredRDD = rdd.filter(lambda x: x > 3)
```

Results:

- doubledRDD: [2, 4, 6, 8, 10]
- filteredRDD: [4, 5]

Resilient Distributed Datasets (RDDs) - Examples

RDD Actions: Trigger the execution of transformations and return a result.

```python
sum = rdd.reduce(lambda x, y: x + y)
first = rdd.first()
```

Results:

- sum: 15
- first: 1

Resilient Distributed Datasets (RDDs) – Use cases

Low-level operations: RDDs provide more granular control over data processing and can be useful for complex transformations.

Custom data structures: If you need to work with custom data structures that don't fit well into DataFrames or Datasets, RDDs can be a good option.

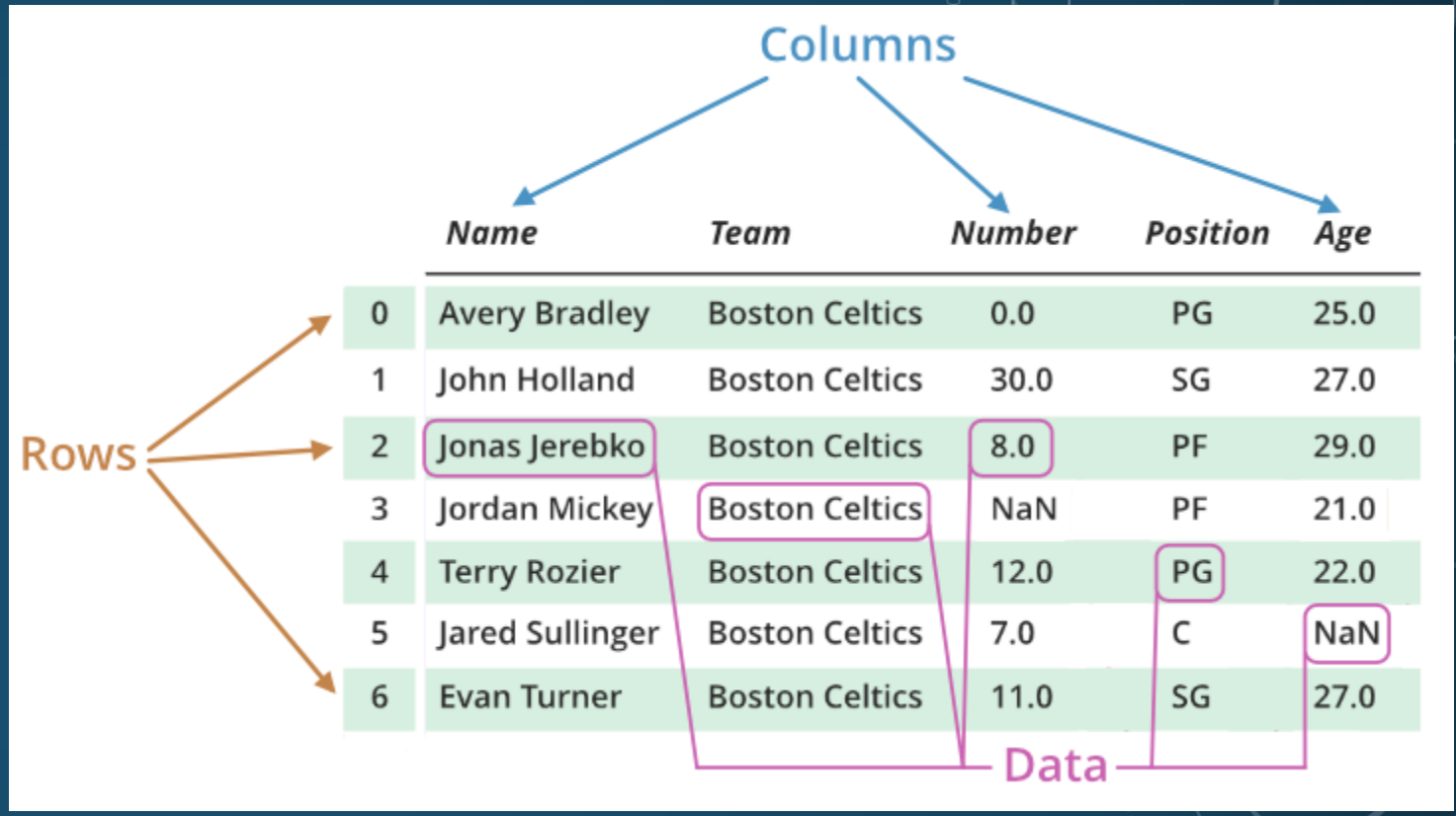
Legacy code: If you have existing Spark code that uses RDDs, you may need to continue using them for compatibility reasons.

However, in most cases, DataFrames and Datasets are preferred due to their higher-level API, performance optimizations, and integration with Spark SQL.

DataFrames – definition

DataFrames are distributed collections of data organized into named columns, similar to a table in a relational database or a DataFrame in Python's Pandas library.

- A higher-level abstraction built on top of RDDs.
- Represent structured data with named columns and data types.



	Name	Team	Number	Position	Age
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

DataFrames – key features

Schema: defined schema, i.e. column names and data types

=> easier to work with structured data

Python

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("DataFrameExample").getOrCreate()

# Create a DataFrame from a list of dictionaries
data = [{"name": "Alice", "age": 30, "city": "New York"},
        {"name": "Bob", "age": 25, "city": "Los Angeles"}]
df = spark.createDataFrame(data)

# Print the DataFrame's schema
df.printSchema()
```

Output:

```
root
 |-- name: string (nullable=true)
 |-- age: long (nullable=true)
 |-- city: string (nullable=true)
```

DataFrames – key features

- **SQL-like interface**: can be manipulated using SQL-like syntax. (DML)
- **Integration with Spark SQL**: DataFrames are tightly integrated with Spark SQL, allowing for efficient query processing and optimization.

Creating a DataFrame from a list of dictionaries:

Python

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("DataFrameExample").getOrCreate()

data = [{"name": "Alice", "age": 30}, {"name": "Bob", "age": 25}]
df = spark.createDataFrame(data)
```

Python

```python
df.createOrReplaceTempView("people")
result_df = spark.sql("SELECT * FROM people WHERE age > 30")
```

DataFrames – key features

Domain-specific language (DSL): provides functions that are specifically designed for data analysis tasks, such as filtering, grouping, joining, and aggregating data.

Filtering data:

Python

```python
filtered_df = df.filter(df.age > 30)
```

Grouping and aggregating data:

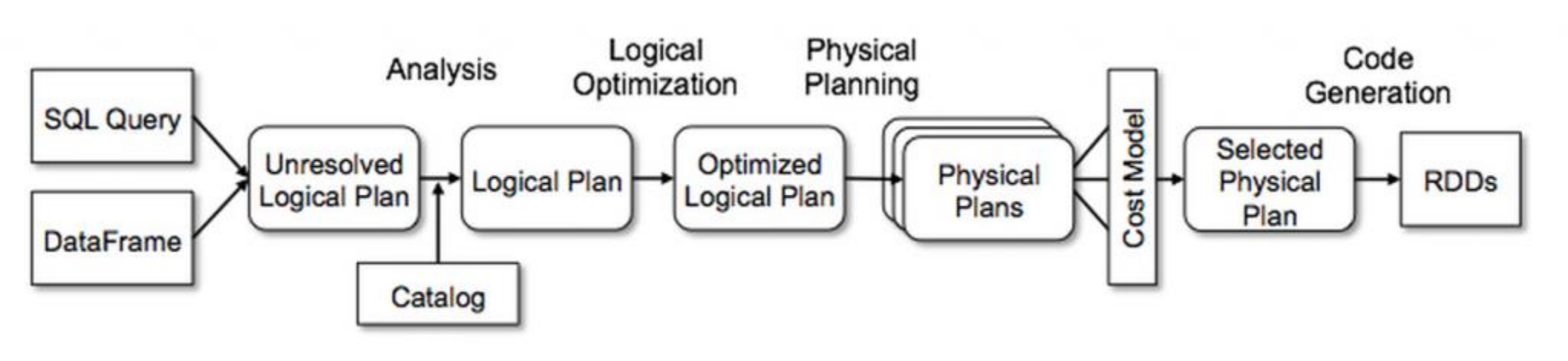
Python

```python
grouped_df = df.groupBy("department").agg({"salary": "avg"})
```

DataFrames - key features

FYI - Versioning in OSS: **Major**.Minor.Bugfixes

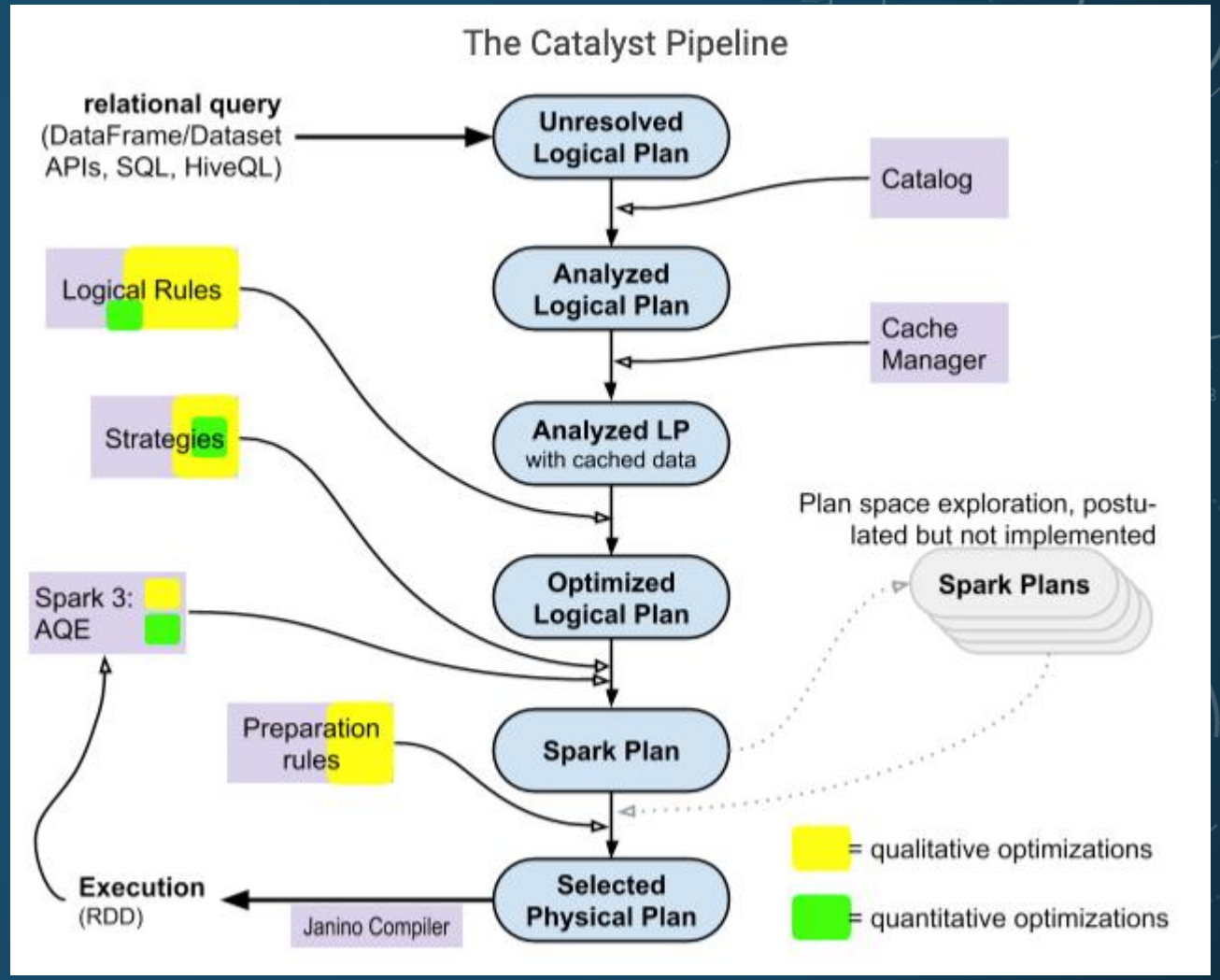
Catalyst optimizer: automatically optimize queries and improve performance.



- **Unresolved Logical Plan** - Contains unresolved references.
- **Analyzed Logical Plan** - All references are resolved (usually by looking into the data catalog)
- **Optimized Logical Plan** - Optimized plan by the Catalyst optimizer for better performance, such as pushdown filters, combining operations, selecting efficient execution plans
- **Physical plans** - execution plans for the query, including the sequence of operations and the specific algorithms to be used.
- **Cost model** - A model used to estimate the cost of different physical plans based on number of rows, data distribution, and available resources.
- **AQE** (* from Spark version 3): A feature that allows Spark to dynamically adjust the execution plan based on runtime conditions e.g. data distribution, resource availability ..
- **Selected physical plan** - The physical plan chosen by the Catalyst optimizer to execute the query based on outputs from the Cost model and AQE.
- **Code generation** - The process of generating optimized machine code from the selected physical plan.

DataFrames - key features

- **Performance**: more performant than RDDs for certain operations, especially when working with structured data thanks to the Catalyst optimization.



DataFrames – Use cases

DataFrames in Apache Spark are well-suited for a wide range of use cases involving structured and semi-structured data

- Data Analysis and Exploration
- Machine Learning
- Data Warehousing and Business Intelligence
- Real-time Data Processing
- Graph processing
- Natural language processing (NLP)
- Recommendation systems
- …

DataFrames – example / classroom practice

Loading a CSV file into a DataFrame and performing operations, such as filtering and aggregating data.

Dataset – definition

Datasets in Spark are a *typed version* of DataFrames. They provide a more concise and expressive API for working with structured data, and they enforce type safety to prevent errors caused by incorrect data types.

Note: Since Dataset is strong typed, it's not available in untyped languages such as Python (Pyspark). In Python you can you typespark instead.

Scala

```scala
val df = spark.read.json("people.json")
val ds = df.as[Person] // Assuming Person is a case class
```

```scala
case class Person(name: String, age: Int)

val people = Seq(Person("Alice", 30), Person("Bob", 25))
val ds = spark.createDataset(people)
```

Dataset – Key features

Type safety: Datasets enforce type safety, ensuring that data is of the correct type.

Concise API: Datasets provide a more concise API compared to RDDs, making it easier to work with structured data.

Integration with Spark SQL: Datasets are tightly integrated with Spark SQL, allowing for efficient query processing and optimization.

Dataset – Use cases

Working with complex data structures: When dealing with complex data structures that require strong type safety, Datasets can help prevent errors and improve code readability.

Enforcing data quality: Datasets can be used to enforce data quality rules and prevent invalid data from entering your system.

Improving performance: Datasets can sometimes offer better performance than DataFrames, especially when working with complex data structures.

RDD vs. DataFrame vs. Dataset – In summary

- **RDDs** are the fundamental building blocks of Spark, providing a low-level API for working with distributed data.
- **DataFrames** are a higher-level abstraction that provide a SQL-like interface for working with structured data.
- **Datasets** are a typed version of DataFrames, offering additional type safety and expressiveness.

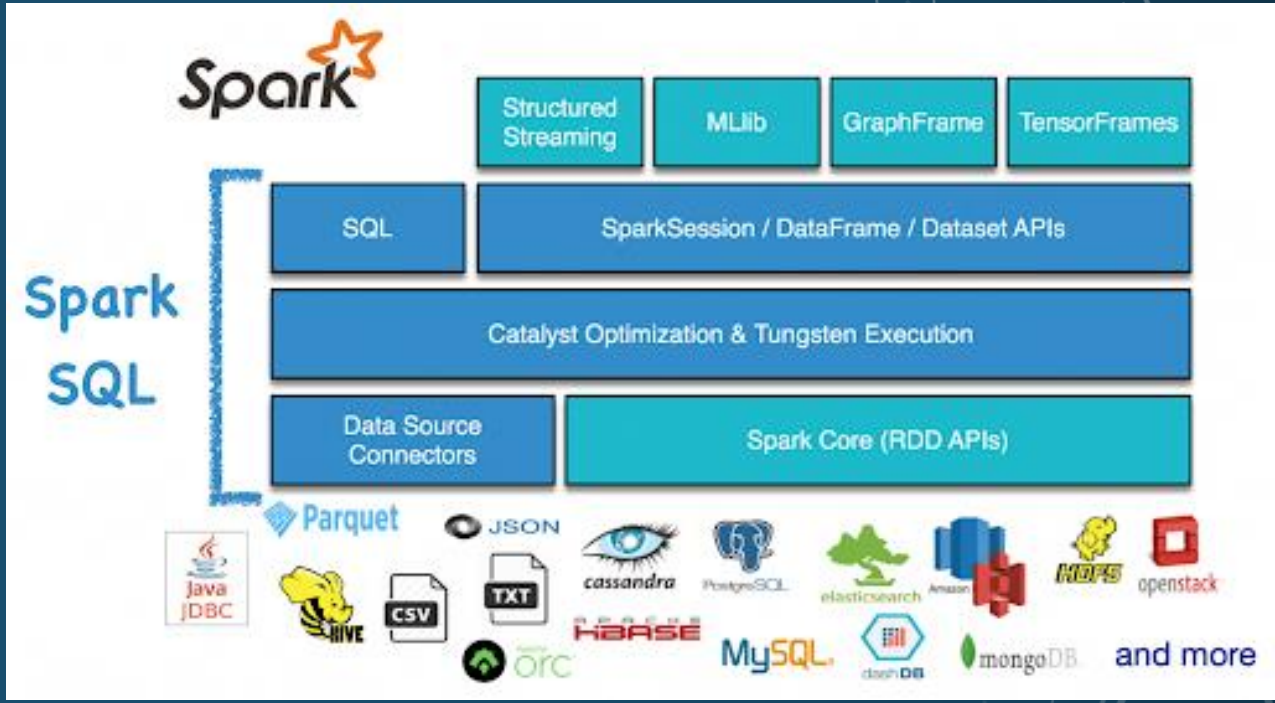
RDD	Dataframe	Dataset
Program how to do	Program what to do	Program what to do
OOPs Style API	SQL Style API	OOPs Style API
On-heap JVM objects	Off-heap also used	Off heap also used
Serialization unavoidable	Serialization can be avoided(off heap)	Serialization can be avoided(encoder)
GC impacts performance	GC impact mitigated	GC impact mitigated
Strong Type Safety	Less Type Safety	Strong Type Safety
No optimation	Catalyst Optimizer	Optimization
Compile time error	Run time error	Compile time error
Java, Scala, Python, and R	Java, Scala, Python, and R	Scala and Java
No Schema	Schema Structured	Schema Structured

RDD vs. DataFrame vs. Dataset – Key take aways

- DataFrames and Datasets are built on top of RDDs.
- DataFrames and Datasets provide a more convenient and efficient way to work with structured data compared to RDDs.
- Datasets offer additional type safety compared to DataFrames.

Spark SQL – Introduction

- Spark SQL is a module for structured data processing. It provides an API for querying data using SQL, along with the ability to mix SQL queries with Spark's RDD and DataFrame APIs.



Spark SQL – Key features

- **SQL Interface**: Allows you to query and manipulate data using familiar SQL syntax.
- **DataFrames and Datasets**: Provides DataFrame and Dataset abstractions for structured data, offering columnar storage and rich APIs.
- **Integration with Spark Core**: Leverages the same distributed processing engine, ensuring scalability and performance.
- **Optimized Query Execution**: Employs advanced query optimization techniques for efficient data processing.
- **Built-in functions**: Math, statistics, …
- **UDFs and UDAFs**: Supports user-defined functions (UDFs) and user-defined aggregate functions (UDAFs) to extend functionality. -> use with cares

Spark SQL – Example

Scenario

Analyzing a DataFrame of sales data using SQL queries. (data can be downloaded from Kaggle, …)

Steps

- Register DataFrame as Table: df.createOrReplaceTempView("sales")
- Run SQL Query: result = spark.sql("SELECT region, SUM(sales) AS total_sales FROM sales GROUP BY region HAVING total_sales > 10000")
- Show Results: result.show()

Spark SQL – Example

Explanation

The DataFrame is registered as a temporary SQL table.

A SQL query is run to group sales by region and filter regions with total sales greater than 10,000.

The results are displayed using the show() action.

Benefits

Familiar SQL syntax for querying large datasets.

Seamless integration with other Spark components.

Spark SQL – Benefits

- **Simplified Data Processing**: SQL syntax is more intuitive for many users, making data processing tasks easier.
- **Scalability**: Handles large datasets efficiently due to its distributed processing capabilities.
- **Performance**: Optimized query execution ensures fast data processing.
- **Integration with Other Spark Components**: Works seamlessly with other Spark modules like Spark Streaming and MLlib.
- **Rich Functionality**: Provides a wide range of SQL operations and APIs for data manipulation – built-in functions

Spark SQL – Online materials

- *Official document*
- *Spark SQL Built-in Functions*

Practices & prepare for the next sessions

1) Setup Pyspark environment – must be ready before tutorial sessions
    1) *Setup for Windows*
    2) *Setup for MacOS (Linux)*
    3) *Advanced: Setup Pyspark Notebook using Docker*
    4) *Advanced: Setup a Spark cluster using Docker*
2) Launch simple interactive Pyspark-shell
3) Starting point: Get SparkSession
4) Creating DataFrames
5) DataFrame Operations
6) Running SQL Queries
7) Global Temporary View
8) Interoperating with RDDs
9) Scalar Functions
10) Aggregate Functions
11) UDFs – User Defined Functions
12) UDAFs - User Defined Aggregation Functions
13) View & Understand SparkUI