CPSC 436C
Cloud Computing for Data Science

Structured Data Processing

Maryam R.Aliabadi

mraiyata@cs.ubc.ca

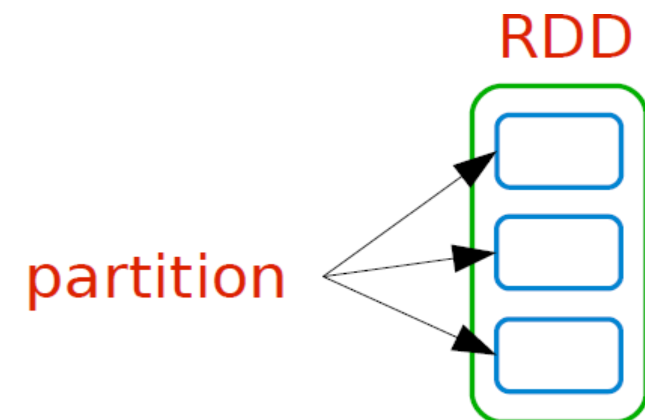
Spring 2024

Last Week's Review

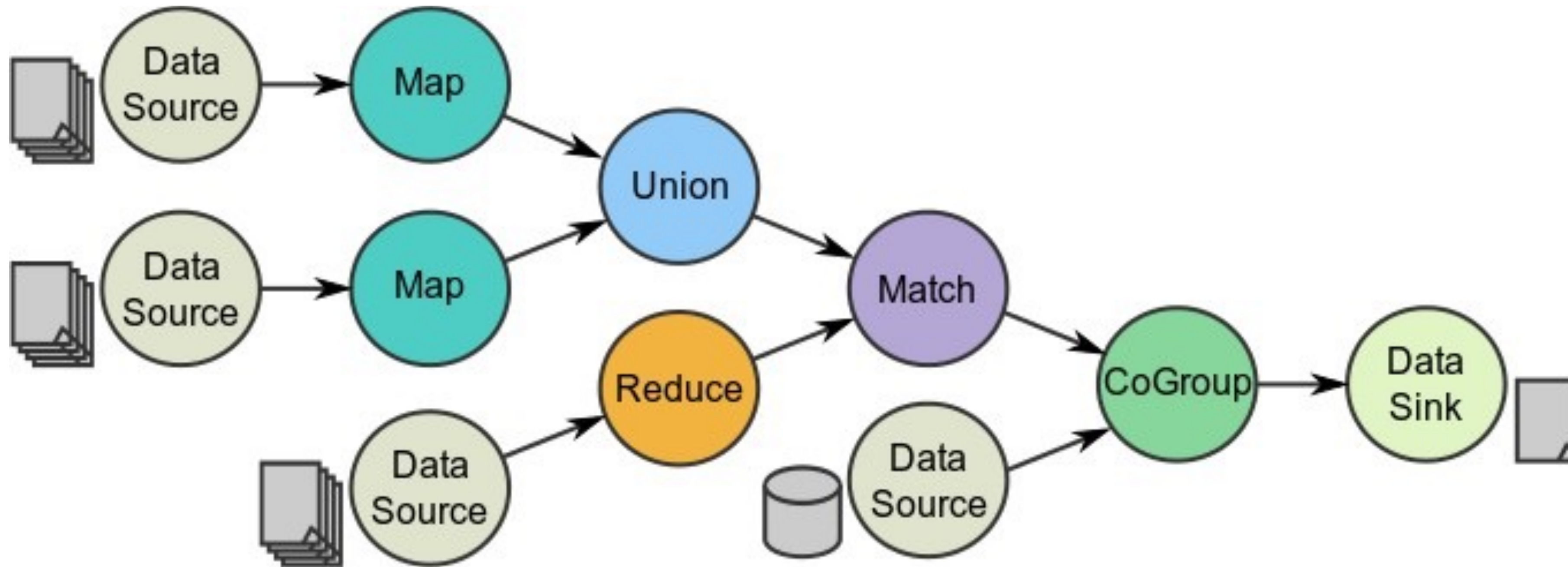
- RDD: a distributed memory abstraction that is both fault tolerant and efficient
- Spark Programming Model
  - Two types of operations: Transformations and Actions.
- Spark Execution Flow
- RDD fault tolerance: Lineage graph

Resilient Distributed Datasets (RDD)

- A distributed memory abstraction
- Immutable collections of objects spread across a cluster.
- An RDD is divided into a number of partitions, which are atomic pieces of information.
- Partitions of an RDD can be stored on different nodes of a cluster.



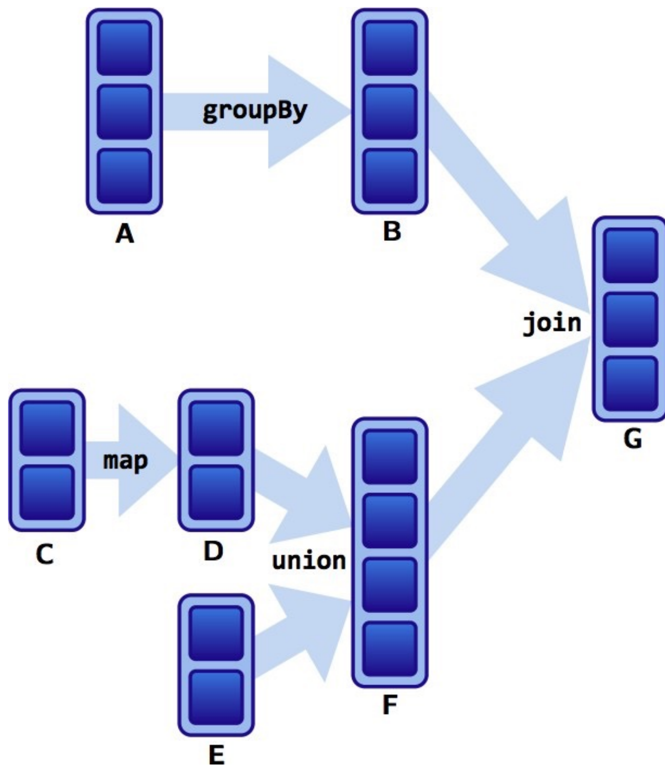
Spark Programming Model

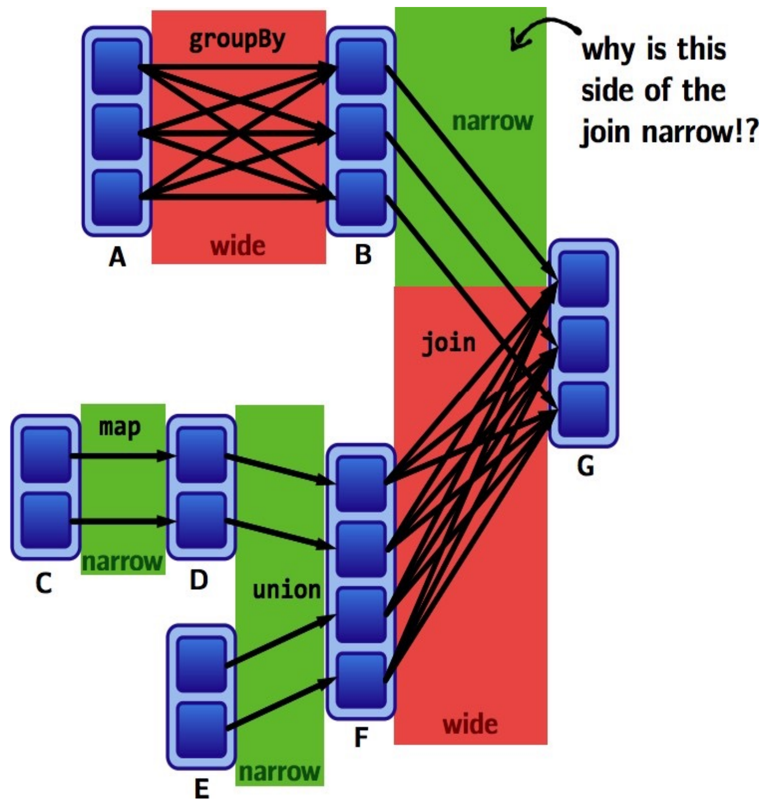


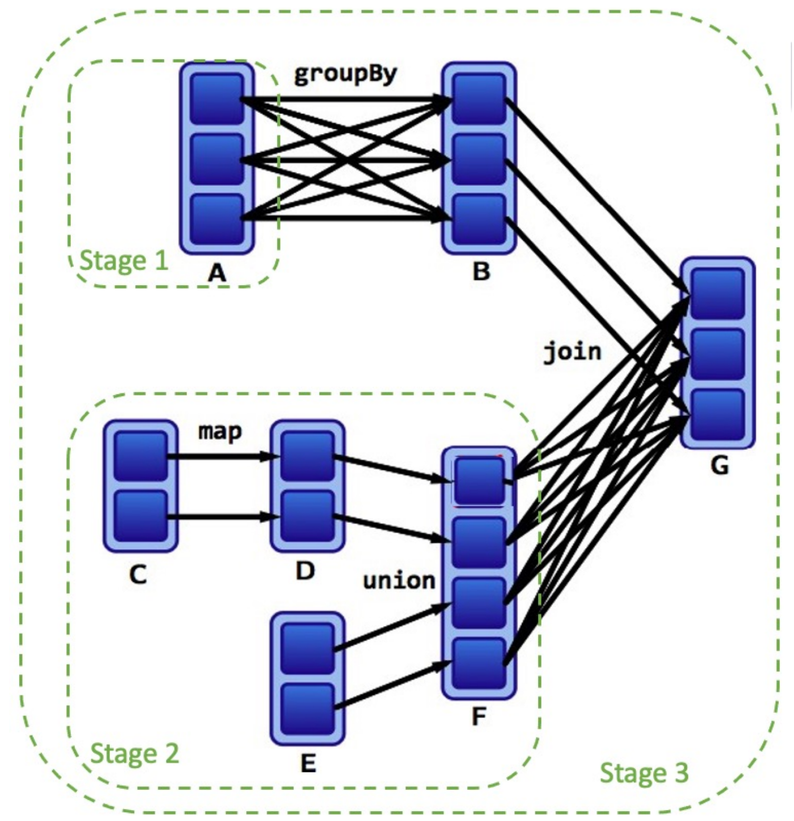
Job Execution Workflow

1. Lineage Creation
2. Dependence and Stage Identification
3. Task Scheduling
4. Shuffle
5. Execution

Job Execution Workflow

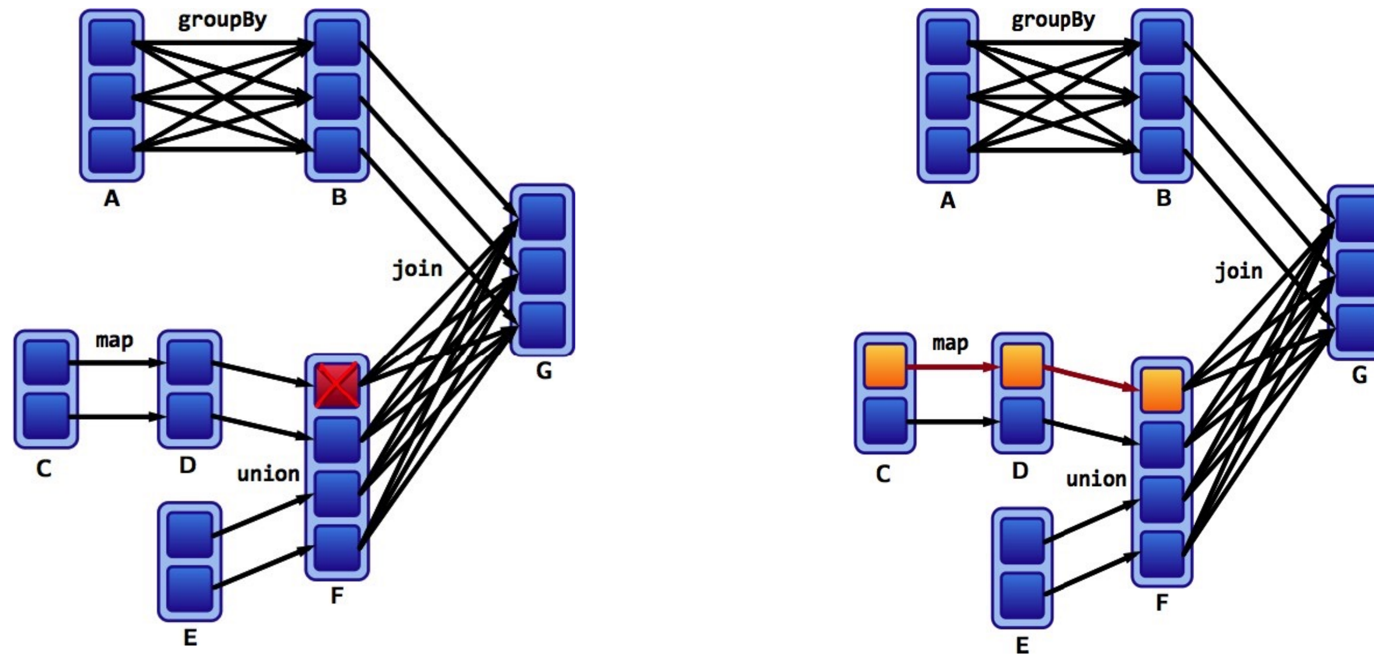






Fault Tolerance

- Assume one of the partitions fails.
- We only have to recompute the data shown below to get back on track.



Fault Tolerance

- If a task fails, it will be re-ran on another node, as long as its stages parents are available.
- If some stages become unavailable, the tasks are submitted to compute the missing partitions in parallel.
- Since RDD is in memory it will be very fast to access the RDD and repeat tranformations
- Recovery may be time-consuming for RDDs with long lineage chains and wide dependencies.
- It can be helpful to checkpoint some RDDs to stable storage.
- Decision about which data to checkpoint is left to users.

Quiz Time

- iClicker: **YFMA**

Activity: Deployment Strategy Debate

Objective

- Compare the following deployment models for a real-time image classification application in the cloud.

- Consider Deployment Strategies:
  1. *N Virtual Machines with Load Balancer*
  2. *Spark Cluster with N nodes*

Which deployment strategy would you choose ?

Option	Result
Spark cluster with N nodes **(A)**	0%
A set of N Virtual Machines (VMs) with load balancer **(B)**	0%
A Spark cluster is nothing other than a set of VMs **(C)**	0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Think Pair Share

Discuss your decision with a neighbor, focusing on the ”Management Complexity" aspect, for 30 seconds.

Which deployment strategy would you choose considering management complexity?

Option	Result
Spark cluster with N nodes	0%
A set of N Virtual Machines (VMs) with load balancer	0%
A Spark cluster is nothing other than a set of VMs	0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Think Pair Share

Discuss your decision with a neighbor, focusing on the ”Resource Utilization" aspect, for 30 seconds.

Which deployment strategy would you choose considering resource utilization?

Option	Result
Spark cluster with N nodes	0%
A set of N Virtual Machines (VMs) with load balancer	0%
A Spark cluster is nothing other than a set of VMs	0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Think Pair Share

Discuss your decision with a neighbor, focusing on the "Cost" aspect, for 30 seconds.

Which deployment strategy would you choose considering cost?

Option	Result
Spark cluster with N nodes	0%
A set of N Virtual Machines (VMs) with load balancer	0%
A Spark cluster is nothing other than a set of VMs	0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Think Pair Share

Discuss your decision with a neighbor, focusing on the ”Inference Time" aspect, for 30 seconds.

Which deployment strategy would you choose considering inference time?

Option	Result
Spark cluster with N nodes	0%
A set of N Virtual Machines (VMs) with load balancer	0%
A Spark cluster is nothing other than a set of VMs	0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Spark Cluster Or N Virtual Machines

	N Virtual Machines	Spark Cluster
Management Complexity	More complex management and scaling of virtual machines, requires handling load balancer configuration	Streamlined deployment and management process, but may require expertise in Spark setup
Resource Utilization	May experience resource wastage during periods of low traffic due to fixed number of virtual machines	Efficient utilization of resources through Spark's distributed computing
Inference Speed	Depends on the number and performance of virtual machines, potential latency due to load balancer overhead	Faster inference times possible through Spark's distributed computing
Cost	Potential higher cost due to maintaining a fixed number of virtual machines, especially during periods of low traffic	Potentially more cost-effective due to efficient resource utilization, but may require investment in Spark setup

Today’s Topics:

Stream Processing	Structured Data Processing	Machine Learning	Graph Processing
	Execution Engine		
Resource Management			
Storage (File systems and Databases)			

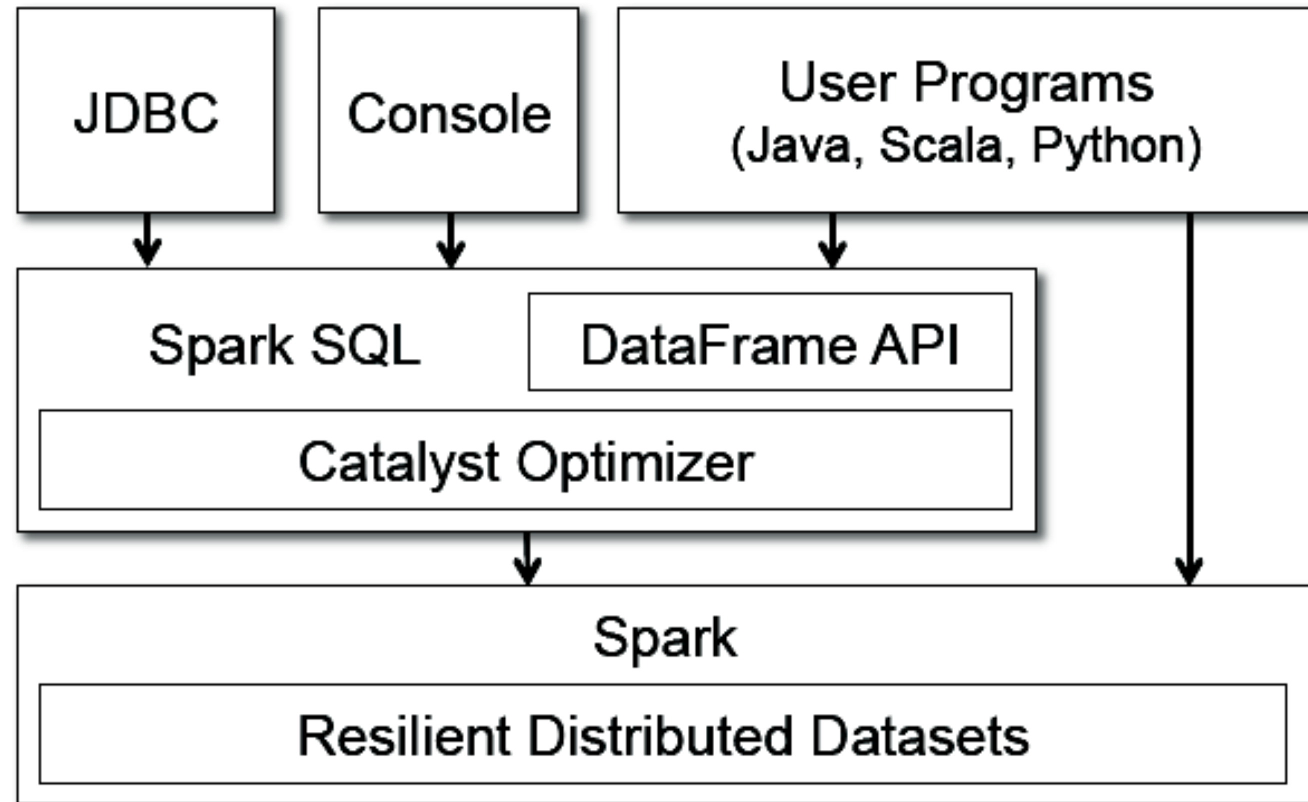
Motivation

- Users often prefer writing declarative queries.
- Lack of schema.

Spark SQL

- Spark SQL was first released in Spark 1.0 (May, 2014).
- Initial committed by Michael Armbrust & Reynold Xin from Databricks.
- Spark introduces a programming module for structured data processing called Spark SQL.
- It provides a programming abstraction called DataFrame and can act as distributed SQL query engine.

Spark and SparkSQL



Structured Data Vs. RDD

- case class Account (name: String, balance: Double, risk: Boolean)
- RDD[Account]
- RDDs don't know anything about the schema of the data it's dealing with.

Structured Data Vs. RDD

- case class Account (name: String, balance: Double, risk: Boolean)
- RDD[Account]
- A database/Hive sees it as a columns of named and typed values.

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

DataFrames and Datasets

- Spark has two notions of structured collections:
  - DataFrames
  - Datasets

- They are distributed table-like collections with well-defined rows and columns.
- They represent immutable lazily evaluated plans.
- When an action is performed on them, Spark performs the actual transformations and return the result.

DataFrames

- Consists of a series of rows and a number of columns.
- Equivalent to a table in a relational database.
- Spark + RDD: functional transformations on partitioned collections of objects.
- SQL + DataFrame: declarative transformations on partitioned collections of tuples.
- This API was designed for modern Big Data and data science applications taking inspiration from DataFrame in R Programming and Pandas in Python.

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

Schema

- Defines the column names and types of a DataFrame.
- Assume `people.json` file as an input:

```
{"name":"Michael", "age":15, "id":12}
{"name":"Andy", "age":30, "id":15}
{"name":"Justin", "age":19, "id":20}
{"name":"Andy", "age":12, "id":15}
{"name":"Jim", "age":19, "id":20}
{"name":"Andy", "age":12, "id":10}
```

```
val people = spark.read.format("json").load("people.json")
people.schema

// returns:
StructType( StructField( age, LongType, true),
StructField(id,LongType,true),
StructField(name,StringType,true))
```

Column

- They are like columns in a table.
- `col` returns a reference to a column.
- `expr` performs transformations on a column.
- `columns` returns all columns on a DataFrame

```
val people = spark.read.format("json").load("people.json")

col("age")

exp("age + 5 < 32")

people.columns
// returns: Array[String] = Array(age, id, name)
```

Column

- Different ways to refer to a column.

```
val people = spark.read.format("json").load("people.json")

people.col("name")

col("name")

column("name")

'name

$"name"

expr( "name")
```

Row

- A row is a record of data.
- They are of type Row.
- Rows do not have schemas.

```
import org.apache.spark.sql.Row

val myRow = Row("Seif", 65, 0)
```

Row

- A row is a record of data.
- They are of type Row.
- Rows do not have schemas.
  - The order of values should be the same order as the schema of the DataFrame to which they might be appended.
- To access data in rows, you need to specify the position that you would like.

```
import org.apache.spark.sql.Row

val myRow = Row("Seif", 65, 0)
```

```
myRow(0) // type Any
myRow(0).asInstanceOf[String] // String
myRow.getString(0) // String
myRow.getInt(1) // Int
```

Creating DataFrames

- Two ways to create a DataFrame:
  1. From an RDD
  2. From raw data sources

Creating a DataFrame – From an RDD

- The schema automatically inferred.
- You can use `toDF` to convert an RDD to DataFrame.

```
val tupleRDD = sc.parallelize(Array(("seif", 65, 0), ("amir", 40, 1)))
val tupleDF = tupleRDD.toDF("name", "age", "id")
```

Creating a DataFrame – From Data Source

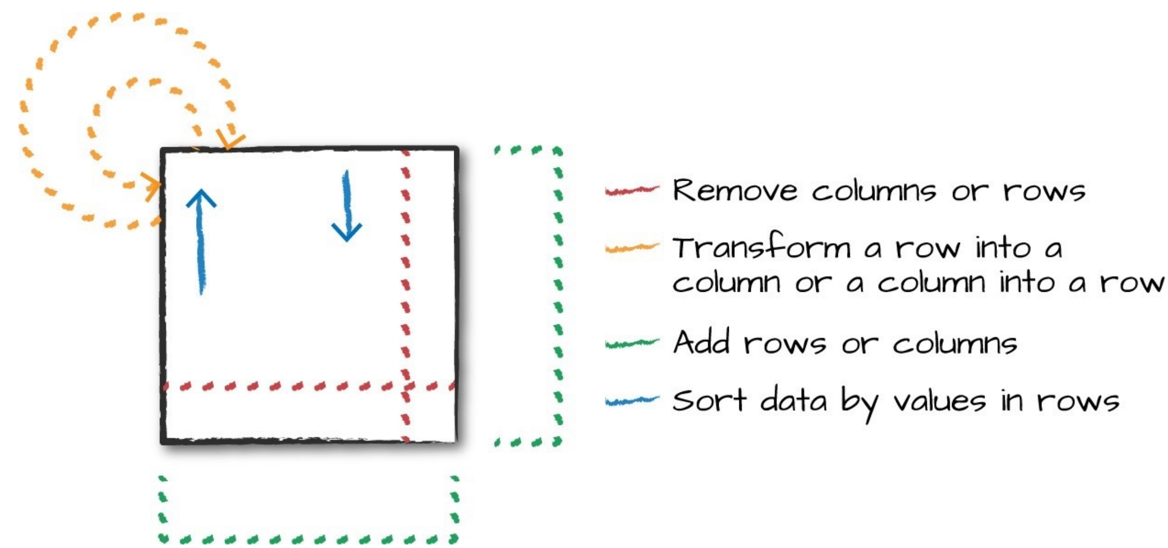
- Data sources supported by Spark.
  - CSV, JSON, Parquet, ORC, JDBC/ODBC connections, Plain-text files
  - Cassandra, HBase, MongoDB, AWS Redshift, XML, etc.

```
val peopleJson = spark.read.format("json").load("people.json")

val peopleCsv = spark.read.format("csv")
  .option("sep", ";")
  .option("inferSchema", "true")
  .option("header", "true")
  .load("people.csv")
```

```
val peopledelta = spark.read.format("delta").load("/path/to/delta/table")
```

DataFrame Transformation

- Add and remove rows or columns
- Transform a row into a column (or vice versa)
- Change the order of rows based on the values in columns



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

DataFrame Transformation

- select and selectExpr allow to do the DataFrame equivalent of SQL queries on a table of data.

```
// select
people.select("name", "age", "id").show(2)
```

```
// selectExpr
people.selectExpr("*", "(age < 20) as teenager").show()
people.selectExpr("avg(age)", "count(distinct(name))", "sum(id)").show()
```

DataFrame Transformation

- `filter` and where both filter rows.

- `distinct` can be used to extract unique rows.

```
people.filter("age < 20").show()

people.where("age < 20").show()

people.select("name").distinct().show()
```

DataFrame Transformation

- withColumn adds a new column to a DataFrame.
- withColumnRenamed renames a column.
- drop removes a column.

```
// withColumn
people.withColumn("teenager", expr("age < 20")).show()

// withColumnRenamed
people.withColumnRenamed("name", "username").columns

// drop
people.drop("name").columns
```

What is the output?

```
people.withColumn("teenager", expr("age < 20")).show()

+---+---+-------+
|age| id|   name|
+---+---+-------+
| 15| 12|Michael|
| 30| 15|   Andy|
| 19| 20| Justin|
+---+---+-------+
```

```
Option 1
+---+---+-------+--------+
|age| id|   name|teenager|
+---+---+-------+--------+
| 15| 12|Michael|    true|
| 30| 15|   Andy|   false|
| 19| 20| Justin|    true|
+---+---+-------+--------+
```

```
Option 2
+---+---+-------+--------+
|age| id|   name|teenager|
+---+---+-------+--------+
| 15| 12|Michael|    true|
| 19| 20| Justin|    true|
+---+---+-------+--------+
```

DataFrame Actions

- Like RDDs, DataFrames also have their own set of actions.
- `collect`: returns an array that contains all of rows in this DataFrame.
- `count`: returns the number of rows in this DataFrame.
- `first` and head: returns the first row of the DataFrame.
- show: displays the top 20 rows of the DataFrame in a tabular form.
- `take`: returns the first n rows of the DataFrame.

Aggregation

Aggregation

- In an aggregation you specify
  - A key for grouping
  - An aggregation function
- The given function must produce one result for each group.

Summarizing a complete DataFrame

- count returns the total number of values.
- countDistinct returns the number of unique groups.
- first and last return the first and last value of a DataFrame.

```
val people = spark.read.format("json").load("people.json")

people.select(count("age")).show()

people.select(countDistinct("name")).show()

people.select(first("name"), last("age")).show()
```

Summarizing a complete DataFrame

- min and max extract the minimum and maximum values from a DataFrame.
- sum adds all the values in a column.
- avg calculates the average.

```
val people = spark.read.format("json").load("people.json")

people.select(min("name"), max("age"), max("id")).show()

people.select(sum("age")).show()

people.select(avg("age")).show()
```

Group By

- Perform aggregations on groups in the data.
- Typically on categorical data.
- We do this grouping in two phases:
  1. Specify the column(s) on which we would like to group.
  2. Specify the aggregation(s).

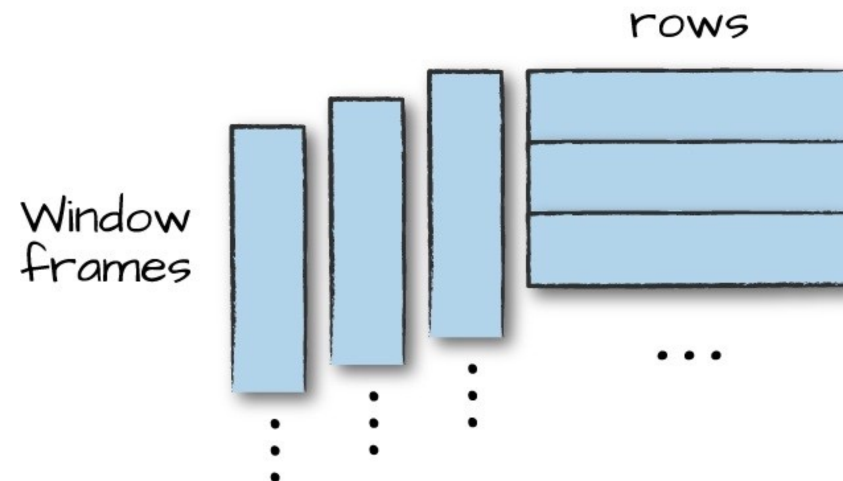
Group By

- Grouping with expressions
  - Rather than passing that function as an expression into a `select` statement, we specify it as within `agg`.

```
val people = spark.read.format("json").load("people.json")

people.groupBy("name").agg(count("age").alias("ageagg")).show()
```

Windowing

- Computing some aggregation on a specific window of data.
- The window determines which rows will be passed in to this function.
- You define them by using a reference to the current data.
- A group of rows is called a frame.



Windowing

- Unlike grouping, here each row can fall into one or more frames.

```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions.col

val people = spark.read.format("json").load("people.json")

val windowSpec = Window.rowsBetween(-1, 1)
val avgAge = avg(col("age")).over(windowSpec)
people.select(col("name"), col("age"), avgAge.alias("avg_age")).show
```

SQL

- You can run SQL queries on views/tables via the method `sql` on the `SparkSession` object.sofar we have used specific api

```
spark.sql("SELECT * from people_view").show()
```

```
+---+---+-------+
|age| id|   name|
+---+---+-------+
| 15| 12|Michael|
| 30| 15|   Andy|
| 19| 20| Justin|
| 12| 15|   Andy|
| 19| 20|    Jim|
| 12| 10|   Andy|
+---+---+-------+
```

Temporary View

- createOrReplaceTempView creates (or replaces) a lazily evaluated view.
- You can use it like a table in Spark SQL.

```
people.createOrReplaceTempView("people_view")

val teenagersDF = spark.sql("SELECT name, age FROM people_view WHERE age BETWEEN 13 AND 19")
```

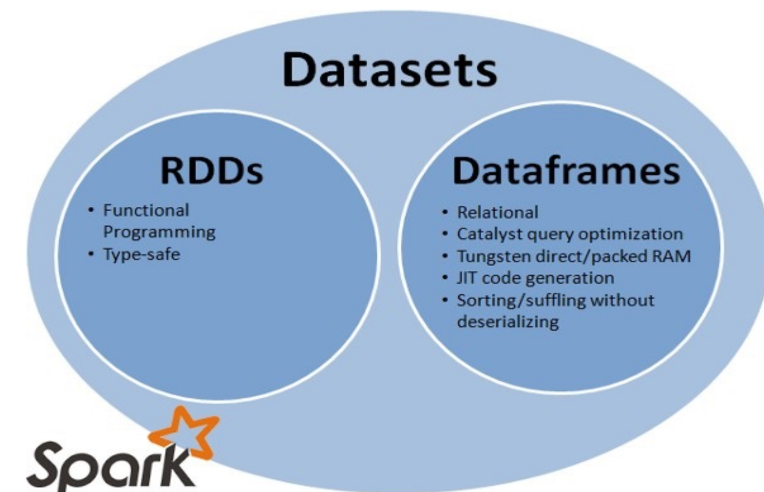
DataSets

Dataset

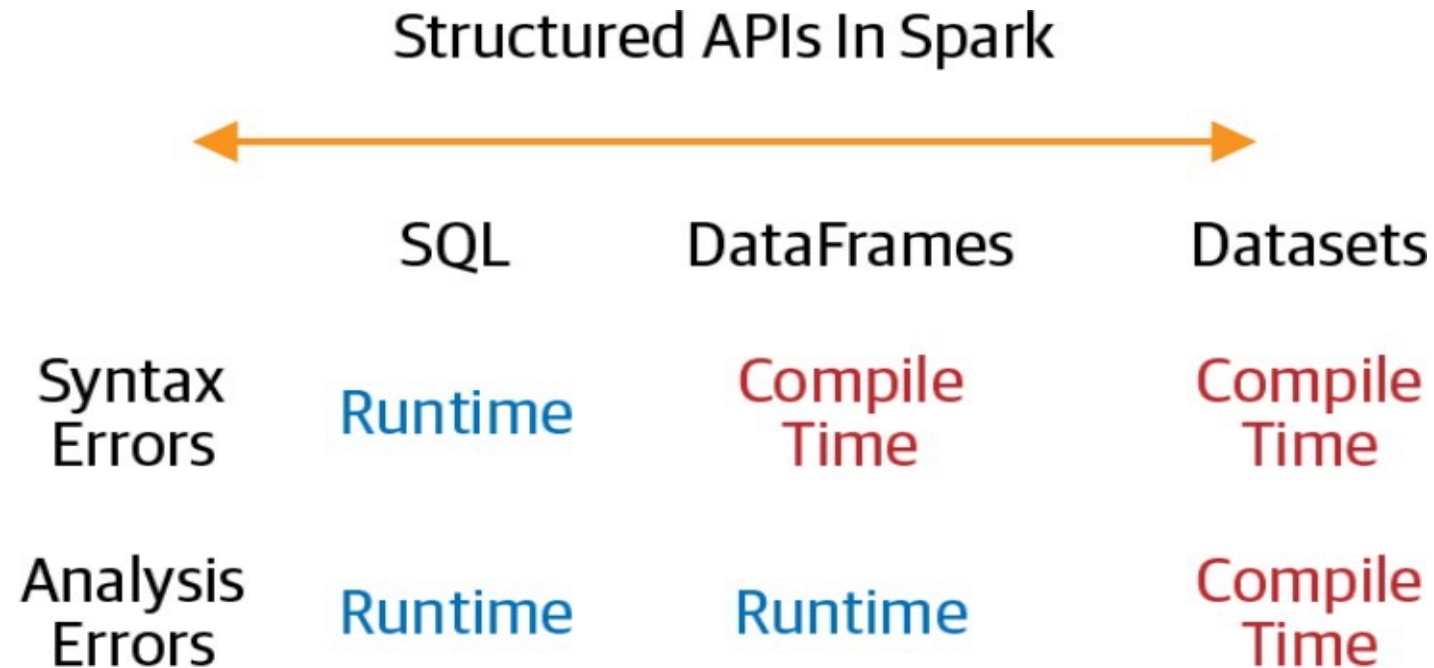
- Datasets can be thought of as typed distributed collections of data.
- Dataset API unifies the DataFrame and RDD APIs.
- You can consider a `DataFrame` as an alias for `Dataset[Row]`

```
type DataFrame = Dataset[Row]
```

- A new interface added in Spark 1.6
- Static-typing and runtime type-safety



Structured API in Spark



Creating Datasets

- To convert a sequence or an RDD to a Dataset, we can use `toDS()`.
- You can call `as[SomeCaseClass]` to convert the DataFrame to a Dataset.

```
case class Person(name: String, age: BigInt, id: BigInt)
val personSeq = Seq(Person("Max", 33, 0), Person("Adam", 32, 1))
```

```
val ds1 = sc.parallelize(personSeq).toDS
```

```
val ds2 = spark.read.format("json").load("people.json").as[Person]
```

Dataset Transformation

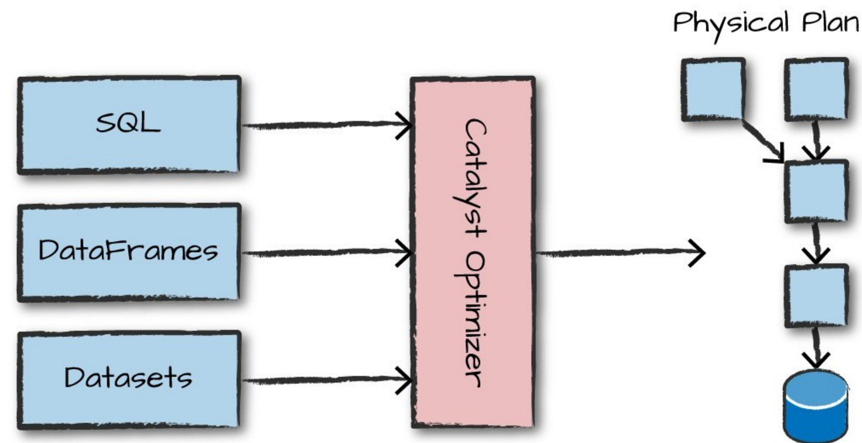
- Transformations on Datasets are the same as those that we had on DataFrames.

```
case class Person(name: String, age: BigInt, id: BigInt)

val people = spark.read.format("json").load("people.json").as[Person]

people.filter(x => x.age < 40).show()

people.map(x => (x.name, x.age + 5, x.id)).show()
```

Structured Data Execution

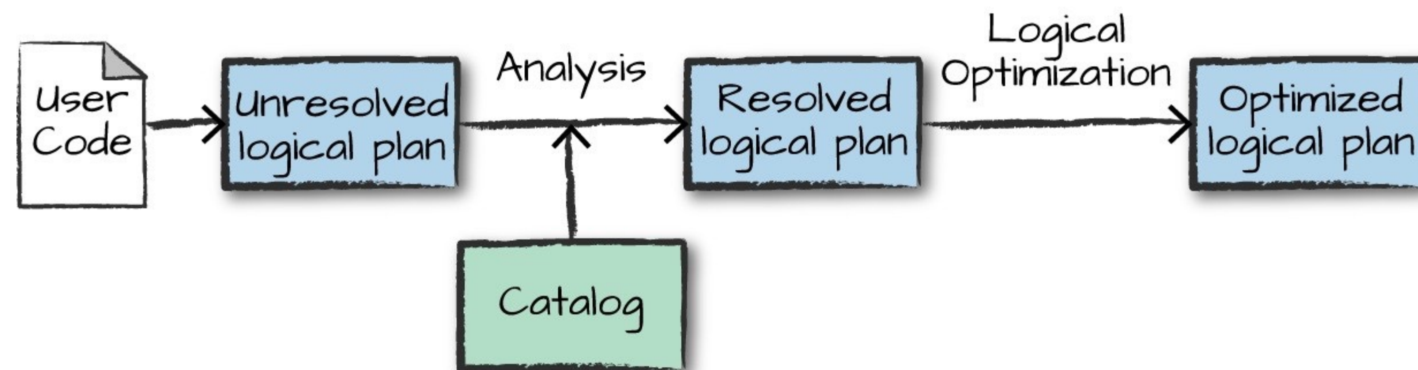
Structured Data Execution - Steps

- 1. Write DataFrame/Dataset/SQL Code.
- 2. If valid code, Spark converts this to a logical plan.
- 3. Spark transforms this logical plan to a Physical Plan
  - Checking for optimizations along the way.
- 4. Spark then executes this physical plan (RDD manipulations) on the cluster.



Logical Planning

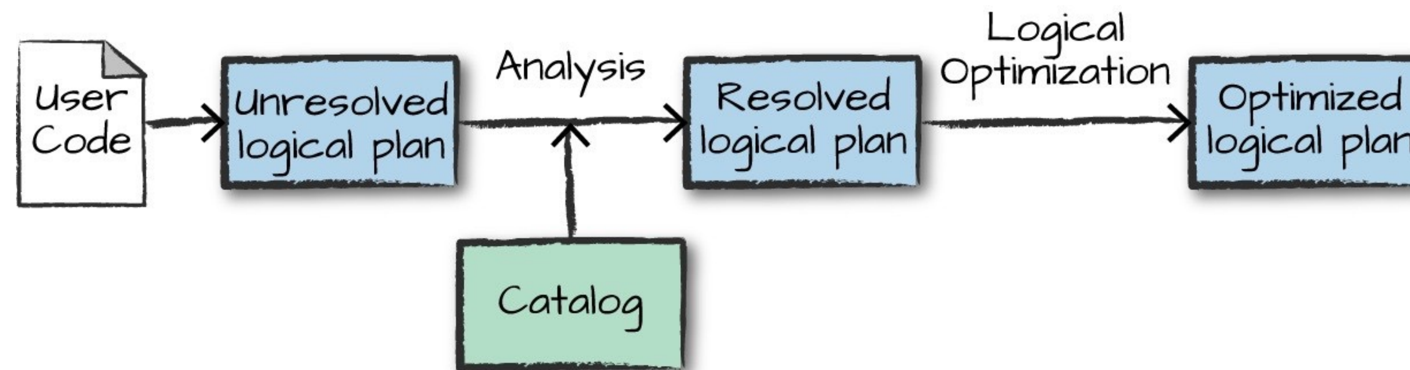
- The logical plan represents a set of abstract transformations.
- This plan is unresolved.
  - The code might be valid, the tables/columns that it refers to might not exist.
- Spark uses the catalog, a repository of all table and DataFrame information, to resolve columns and tables in the analyzer.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

Logical Planning

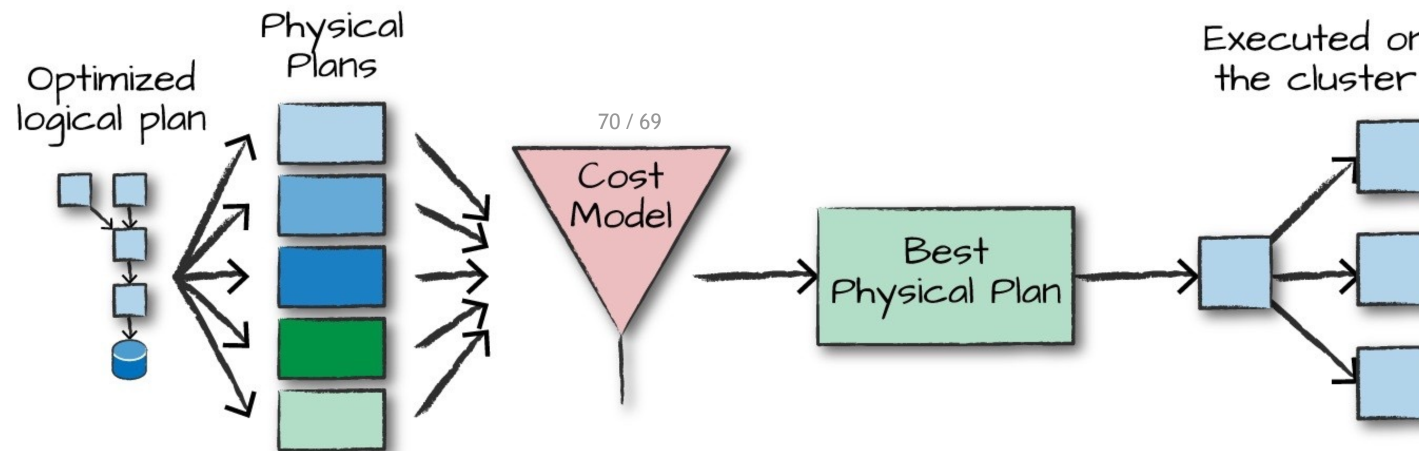
- The analyzer might reject the unresolved logical plan.
- If the analyzer can resolve it, the result is passed through the Catalyst optimizer.
- It converts the user's set of expressions into the most optimized version.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

Physical Planning

- The physical plan specifies how the logical plan will execute on the cluster.
- Physical planning results in a series of RDDs and transformations.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

Execution

- Upon selecting a physical plan, Spark runs all of this code over RDDs.
- Spark performs further optimizations at runtime.
- Finally, the result is returned to the user.

Spark SQL; The Whole Story

- Create and Run Spark Programs Faster:

  1. Write less code.
  2. Read less data.
  3. Let the optimizer do the hard work.

Example: Compute an average

```java
private IntWritable one =  new IntWritable(1)
private IntWritable output =  new IntWritable()
proctected void map(  LongWritable key,
    Text value,  Context context) {
  String[] fields = value.split("\t")
  output.set(Integer.parseInt(fields[1]))
  context.write(one, output)
}
IntWritable one = new IntWritable(1) DoubleWritable average
= new DoubleWritable()

protected void reduce(  IntWritable key,
    Iterable<IntWritable> values,  Context
    context) {
  int sum = 0  int count = 0
  for(IntWritable value : values) {  sum +=
     value.get()
     count++
    }
  average.set(sum / (double) count)
  context.Write(key, average)
```

```python
data = sc.textFile(...).split("\t")
data.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]]) \
   .collect()
```

Example: Compute an average

Using RDDs

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(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]]) \
   .collect()
```

Using SQL

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

Using DataFrames

```
sqlCtx.table("people") \
   .groupBy("name") \
   .agg("name", avg("age")) \
   .collect()
```

Recap

- Structured Data processing using SparkSQL
- RDD vs. DataFrame vs. DataSet
- Logical and physical plans

Next class:

Distributed Machine Learning