RDDs in Parallel Programming and Spark

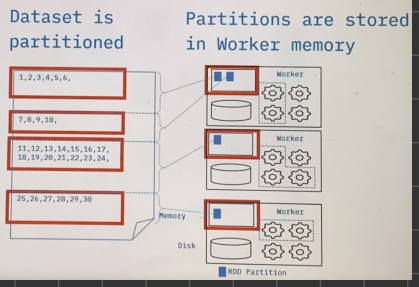
Objectives

After watching this video, you will be able to:

- Describe Resilient Distributed Datasets (RDDs)
- Explain how to use Resilient Distributed Datasets in Spark
- Explain RDD transformation and actions and list basic transformations and actions

Resilient Distributed Datasets

- Are Spark's primary data abstraction
- Are partitioned across the nodes of the cluster



RDD Transformations

Transformations:

- Create a new RDD from existing one
- Are "lazy" because the results are only computed when evaluated by actions

RDD Transformations

Ex

The map transformation passes each element of a dataset through a function and returns a new RDD

Example

map()
transformation

RDD Actions

to evaluate a transformation in Spark

Actions return a value to driver program after running a computation

Example

reduce() ← reduce action which

An action that aggregates all RDD elements

and return the result to the driver program

The Directed Acyclic Graph (DAG)

to do transformations and actions

- A graphical data structure with edges and vertices
- Every new edge is obtained from an older vertex

In general, the vertices and edges are sequential

- In Apache Spark DAG, the vertices represents RDDs and the edges represent operations such as transformations or actions
- If a node goes down, Spark replicates the DAG and restores the node.

Transformations & Actions

1. Spark creates the DAG when creating an RDD
2. Spark enables the DAG Schedular to perform a transformation and updates the DAG
3. The DAG now points to the new RDD
4. The pointer that transforms RDD is returned to the Spark driver program
5. If there is an action, the driver program that calls the action evaluates the DAG only after Spark completes the action

Transformation examples

Transformation	Description
map (*func*)	Returns a new distributed dataset formed by passing each element of the source through a function *func*
filter (*func*)	Returns a new dataset formed by selecting those
distinct ([numTasks]))	Returns a new dataset that contains the distinct elements of the source dataset
flatmap (*func*)	Similar to map (*func*) Can map each input item to zero or more output items *Func* should return a Seq rather than a single item

essential operation and capable of expressing all transformations needed

Action examples

Action	Description
reduce(*func*) aggregates dataset elements using the function *func*	*func* takes two arguments and returns one • Is commutative • Is associative • Can be computed correctly in parallel
take(*n*)	Returns an array with the first *n* element
collect()	Returns all the elements as an array WARNING: Make sure that ? will fit in driver program
takeOrdered (*n*,key=*func*)	Returns *n* elements ordered in ascending order or as specified by the optional key function

A Transformation with an Action

Transformations map operations

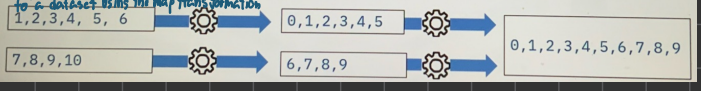
Actions return computed values to the driver program

Map(f): Each task makes a new partition by calling f(e) on each entry e in the original partition

Collect(): Gathers the entries from all the partitions into the driver.

The driver program receives the results

apply this function as a transformation to a dataset using the map transformation



Data Frames and Datasets

Objectives

After watching this video, you will be able to:

- Describe dataset features and benefits within Apache Spark
- Explain three ways you can create datasets for use in Spark
- Summarize the differences between datasets and DataFrames

Datasets are the newest Spark data abstraction

A dataset is a distributed collection of data that:

- Consists of a collection of strongly typed JVM objects (meaning type safe, and the datasets of datatype is made explicit during it's creation)
- Provides the combined benefits of both RDDs and Spark SQL

Datasets Features benefits

- Are immutable, meaning that data cannot be deleted or lost
- Feature an encoder that converts JVM objects to a tabular representation
- Extend DataFrame type-safe and object-oriented API capabilities
- Work with both Scala and Java APIs ← only works with statically typed (R and Python are not supported ↓ Dynamically typed)

Datasets in Spark - Benefits

- Provide compile-time type safety (because datasets are statically typed)
- Compute faster than RDDs
- Offer the benefits of Spark SQL and DataFrames
- Optimize queries using Catalyst and Tungsten
- Enable improved memory usage and caching
- Use dataset API functions for aggregate operations including sum, average, join and group by

Creating a dataset

Apply the toDS()function to create a dataset from a sequence

```
// Create a Dataset from a sequence of primitive datatype
val ds = Seq("Alpha", "Beta", "Gamma").toDS()
```

Create a dataset from a text file

Create a dataset from a text file

```
// Create a Dataset from a file for a primitive datatype
val ds =
spark.read.text("/text_folder/file.txt").as[String]
```

Create a Dataset from a JSON file

Create a dataset using a JSON file

```
// Create a Dataset from a file for a custom datatype
case class Customer(name: String, id: Int, phone: Double)
val ds_cust =
spark.read.json("/customer.json").as[Customer]
```

Datasets & DataFrames Compared

Datasets are:

- Strongly-typed
- Use unified Java and Scala APIs
- Built on top of DataFrames and the latest data abstraction added to Spark

DataFrames are:

- Not typesafe
- Use APIs in Java, Scala, Python and R
- Built on top of RDDs and added in earlier Spark versions

Catalyst and Tungsten

Objectives

After watching this video, you will be able to:

- Describe the goals of Apache Spark SQL optimization
- Describe how Catalyst and Tungsten benefit Spark SQL
- Explain how Spark performs SQL and memory optimization using Catalyst and Tungsten

Spark SQL Optimization goals

Reduce

- Query time
- Memory consumption

Catalyst defined

- Is the Spark SQL built-in *rule-based query* optimizer
- Based on functional programming constructs in Scala
- Supports the addition of new optimization techniques and features
- Enables developers to add data source-specific rules and support new data types

SQL Optimization explained

Rule-based optimization defines how to run the query

Examples

- Is the table is indexed?
- Does the query contain only the required columns?

SQL Optimization explained

Cost-based optimization equals time + memory a query consumes

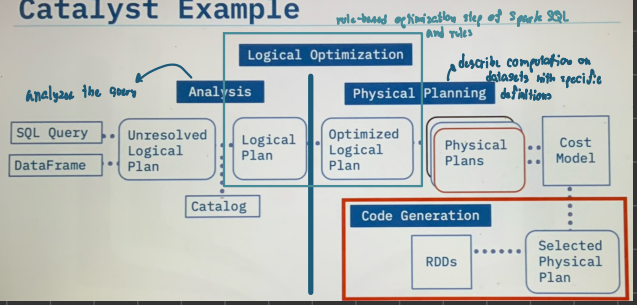
Example

What are the best paths for multiple datasets to use when querying data?

Catalyst query optimization

- Uses a tree data structure and a set of rules
  - Four major phases of query execution

Catalyst Example



Tungsten defined

Spark's *cost-based* optimizer that maximizes *CPU and memory* performance

Tungsten Features

- Manages memory explicitly and does not rely on the JVM object model or garbage collection
- Enables cache-friendly computation of algorithms and data structures using both STRIDE-based memory access
- Supports on-demand JVM byte code generation
- Does not generate virtual function dispatches
- Places intermediate data in CPU registers
- Enables Loop unrolling

ETL with DataFrames

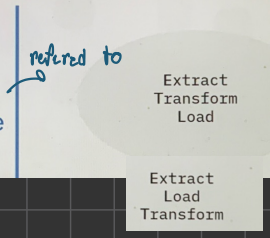
Objectives

After watching this video, you will be able to:

- Identify the basic DataFrame operations
- Apply basic DataFrame operations on real world data
- Apply Spark DataFrames to real world data

Basic DataFrame operations

- Read the data
- Analyze the data
- Transform the data
- Load data into a database
- Write data back to disk



Read the data

- Create a DataFrame
- Create a DataFrame from an existing DataFrame

load dataset into a Pandas DataFrame

```
import pandas as pd
mtcars = pd.read_csv('mtcars.csv')
sdf = spark.createDataFrame(mtcars)
```

load to spark DataFrame

Read the data

View the dataset parameters

```
[, 1] mpg Miles/(US) --> gallon
[, 2] cyl -> Number of cylinders
[, 3] disp --> Displacement (cu.in.)
[, 4] hp -> Gross horsepower
[, 5] drat -> Rear axle ratio
[, 6] wt -> Weight (lb/1000)
[, 7] qsec -> ¼ mile time
[, 8] vs --> V/S
[, 9] am --> Transmission (0 = automatic, 1 = manual)
[,10] gear -> Number of forward gears
[,11] carb -> Number of carburetors
```

Read the data

View the first few dataset rows

	Unnamed: 0	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.9	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.6	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet SportAbout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

Analyze the data using printschema

View the schema

```
sdf.printSchema()
```

```
root
|-- Unnamed: 0: string (nullable = true)
|-- mpg: double (nullable = true)
|-- cyl: long (nullable = true)
|-- disp: double (nullable = true)
|-- hp: long (nullable = true)
|-- drat: double (nullable = true)
|-- wt: double (nullable = true)
|-- qsec: double (nullable = true)
|-- vs: long (nullable = true)
|-- am: long (nullable = true)
|-- gear: long (nullable = true)
|-- carb: long (nullable = true)
```

Analyze the data – the show function

Apply the show() function

```
sdf.show(5)
```

Unnamed: 0	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.9	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.6	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet SportAbout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

Analyze the data - the select function

Apply the select() function to view a specific column

```
sdf.select('mpg').show(5)
```

```
+----+
| mpg|
+----+
|21.0|
|21.0|
|22.8|
|21.4|
|18.7|
+----+
only showing top 5 rows
```

Transform the data - guidelines

- Keep only the relevant data
- Apply filters, joins, sources and tables, column operations, grouping and aggregations and other functions
- Apply domain-specific data augmentations processes

Transform the data using a filter

```
sdf.filter(sdf['mpg'] < 18).show(5)
```

Unnamed: 0	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Duster	14.3	8	160	110	3.90	2.620	16.46	0	1	4	4
Merc 280C	17.8	6	160	110	3.90	2.875	17.02	0	1	4	4
Merc 450SE	16.4	8	108.9	93	3.85	2.320	18.61	1	1	4	1
Merc 450SL	17.3	8	258.0	110	3.08	3.215	19.44	1	0	3	1
Merc 450SLC	15.2	8	360.0	175	3.15	3.440	17.02	0	0	3	2

Transform data using aggregation

```
car_counts = sdf.groupby(['cyl']).agg({"wt": "count"})\ .sort("count(wt)", ascending=False).show(5)
```

```
+---+---------+
|cyl|count(wt)|
+---+---------+
|  8|       14|
|  4|       11|
|  6|        7|
+---+---------+
```

Loading or Exporting the Data

Is the final step of ETL pipeline

- Export to another database
- Export to disk as JSON files
- Save the data to a Postgres database
- Use an API to export data

Real-World Usage SparkSQL

Objectives

After watching this video, you will be able to:

- Define Spark SQL
- Create a table view in Spark SQL
- Explain how to aggregate data using Spark SQL
- Explain the various data sources Spark SQL supports

Spark SQL - revisited

- Is a Spark module for structured data processing
- Runs SQL queries on Spark DataFrames
- Usable in Java, Scala, Python and R

Creating a View in Spark SQL

- Creating a table view in Spark SQL is required to run SQL queries programmatically on a DataFrame
- A view is a temporary table to run SQL queries
  - A Temporary view provides local scope within the current Spark session
  - A Global Temporary view provides global scope within the Spark application

Creating a View in Spark SQL

temporary

```
# Create DataFrame from file
df =
spark.read.json("people.json")

# Create a temp view
df.createTempView("people")
# Run SQL Query
spark.sql("SELECT * FROM
people").show()
```

```
+----+-------+
| age| name  |
+----+-------+
|null|Michael|
| 30 | Andy  |
| 19 | Justin|
+----+-------+
```

Creating a View in Spark SQL

```
# Create a global view
df.createGlobalTempView("people")
# Run SQL Query
spark.sql("SELECT * FROM
global_temp.people").show()
```

```
+----+-------+
| age| name  |
+----+-------+
|null|Michael|
| 30 | Andy  |
| 19 | Justin|
+----+-------+
```

Aggregating Data

Used to aggregate data over columns

- DataFrames contain inbuilt common aggregation functions - count(), countDistinct(), avg(), max(), min(), and others
- Alternatively, aggregate using SQL queries and tableviews

Aggregating Data - An Example

```
import pandas as pd
mtcars = pd.read_csv('mtcars.csv')
sdf = spark.createDataFrame(mtcars)
```

Aggregating data – An Example

```
sdf.select('mpg').show(5)
```

```
[, 1] mpg Miles/(US) --> gallon
[, 2] cyl -> Number of cylinders
[, 3] disp -> Displacement (cu.in.)
[, 4] hp -> Gross horsepower
[, 5] drat --> Rear axle ratio
```

Aggregating data – An Example

```
sdf.select('mpg').show(5)
```

```
+----+
| mpg|
+----+
|21.0|
|21.0|
|22.8|
|21.4|
|18.7|
+----+
only showing top 5 rows
```

Aggregating data - An Example

```
# Using a DataFrame function
car_counts =
sdf.groupby(['cyl']).agg({"wt":"count"}).sort("count(wt)",
ascending=False).show(5)


# Using an SQL Query + Table View
sdf.createTempView("cars")
sql("SELECT cyl, COUNT(*) FROM cars GROUPBY cyl ORDER by 2
DESC")
```

Spark SQL Data Sources

- Parquet Files
  - Supports reading/ writing and preserving data schema.
  - Spark SQL can also run queries without loading the file
- JSON Datasets: Spark infers the schema and loads the dataset as a DataFrame
- Hive Tables: Spark supports reading and writing data stored in Apache Hive