CS X55: DISTRIBUTED SYSTEMS [SPARK]

Transformations: Narrow and Wide

Though their numbers are few
Don't let them beguile you
Innocuous though
they may seem
The wrong invocation
Is all it takes
To amplify inefficiencies
And protract computations

Shrideep Pallickara
Computer Science
Colorado State University

COMPUTER SCIENCE DEPARTMENT
COLORADO STATE UNIVERSITY

1

Frequently asked questions from the previous class survey

COLORADO STATE UNIVERSITY
Professor: SHRIDEEP PALLICKARA
COMPUTER SCIENCE DEPARTMENT
SPARK
L32.2

2

Topics covered in this lecture

- Data Frames
  - Column manipulations
- Orchestration Plans

Column Manipulations [1/4]

- **withColumn(columnName, func)**
  - Return a Dataframe with the additional column
  - Invocation: `df.withColumn("dogYears", df.age / 7)`
- **dropColumn(columnName)**
  - Return a Dataframe without the column
  - Invocation: `df.dropColumn("age")`

Column Manipulations [2/4]

- **select(columnNames)**
  - Return a DataFrame with the specified columns
  - Invocation: `df.select("firstName", "age")`
- **describe(columnName)**
  - Compute summary statistics over DataFrame columns
  - Invocation: `df.describe("age"), df.describe()`

Column Manipulations [3/4]

```
val df = Seq(
    ("Peterson", "Marcus", 54),
    ("Batey", "Edward", 36),
    ("Bruce", "Karen", 35)
  ).toDF("lastName", "firstName", "age")

  df.withColumn("dogYears", df.age / 7.0)
  df.describe("age", "dogYears")
```

Column Manipulations [4/4]

```
+-------+---------+---------+
|summary|      age| dogYears|
+-------+---------+---------+
|  count|        3|        3|
|   mean|  41.6667|  5.95238|
| stddev| 10.69268|  1.52753|
|    min|       35|        5|
|    max|       54| 7.714286|
+-------+---------+---------+
```


Dataframe joins

- `join(other, <columnComparison>, <joinType>)`
  - Performs a join between 2 Dataframes
  - Invocation: `df1.join(df2, Seq("id"))`

Join column comparison

- Supports a variety of criteria
  - Sequence of column names (e.g., `Seq(“id”, “age”)`)
  - Elaborate comparison definitions (e.g., `df1(“age”) >= df2(“age”)`)

Join Type

- `DataFrames` may perform multiple styles of join
  - Inner: typical dataset join with key-to-key match
  - Outer, left-outer, right-outer: result contains all rows, filling in columns with ‘null’ values where data doesn’t exist
  - Left-semi, right-semi: similar to outer join, but result only contains rows in specified source dataset

Example: Spark SQL

```
val df = Seq(
    ("Peterson", "Marcus", 54),
    ("Batey", "Edward", 36),
    ("Bruce", "Karen", 35)
  ).toDF("lastName", "firstName", "age")

  df.createOrReplaceTempView("people")
  spark.sql("SELECT firstName, age, age / 7.0 as dogYears
    FROM people where age < 50")
```

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT SPARK L32.11

11

TUNING THE LEVEL OF PARALLELISM

12

Tuning the level of parallelism

- Every RDD has a **fixed number of partitions**
  - Determine the degree of parallelism when executing operations
- During aggregations or grouping operations, you can ask Spark to use a specific number of partitions
  - This will override defaults that Spark uses

13

Example: Tuning the level of parallelism

```
data = [("a", 3), ("b", 4), ("a", 1)]

sc.parallelize(data).
       reduceByKey(lambda x, y: x + y) #default

sc.parallelize(data).
       reduceByKey(lambda x, y: x + y, 10) #Custom
```


14

What if you want to tune parallelism outside of grouping and aggregation operations?

- There is `repartition()`
  - **Shuffles data across the network** to create a new set of partitions
  - Very **expensive operation!**
- There is the `coalesce()` operation
  - Allows avoiding data movement
    - But only if you are **decreasing** the number of partitions
  - Check `rdd.getNumPartitions()` and make sure you are coalescing to fewer partitions than current

15

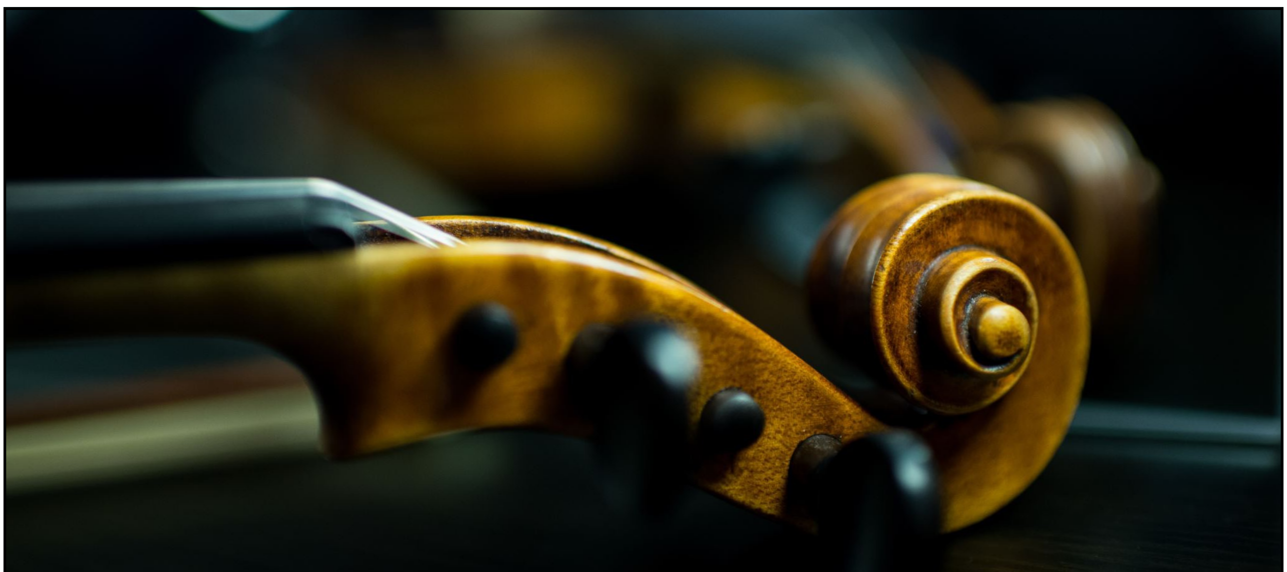
DATASETS VS DATAFRAMES

16

Datasets vs DataFrames

- In Spark's supported languages, Datasets make sense only in Java and Scala, whereas in Python and R only DataFrames make sense
- This is because Python and R are **not compile-time type-safe**
  - types are dynamically *inferred* or assigned during execution, not during compile time
- The reverse is true in Scala and Java: types are bound to variables and objects at compile time

17



ORCHESTRATION PLANS

18

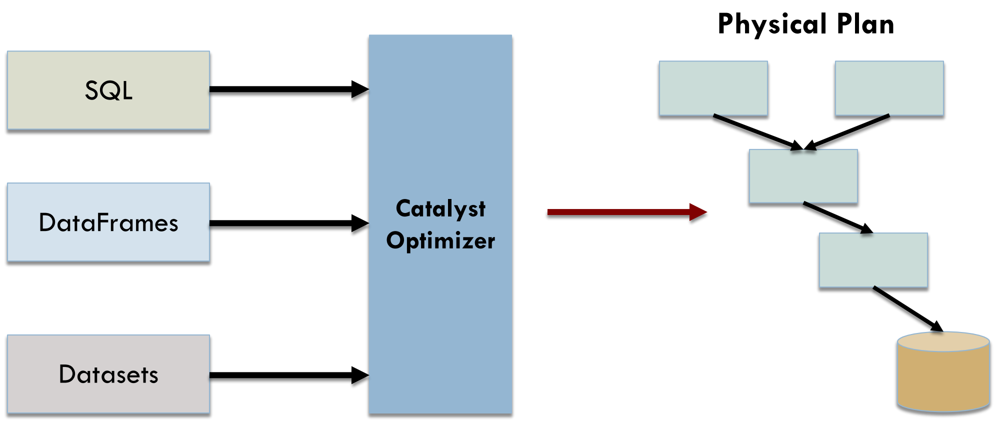
Executing Spark code in clusters: Overview

- Write DataFrame/Dataset/SQL Code
- If the code is valid, Spark converts this to a **Logical Plan**
- Spark transforms this Logical Plan to a **Physical Plan**, checking for optimizations along the way
- Spark then executes this Physical Plan (which involves RDD manipulations) on the cluster

Once you have the code ready

- Code is submitted either through the console or via a submitted job
- This code passes through the **Catalyst Optimizer**
  - Decides *how* the code should be executed
  - Lays out a plan for doing so before, finally, the code is run
    - And the result returned to the user

The Catalyst Optimizer



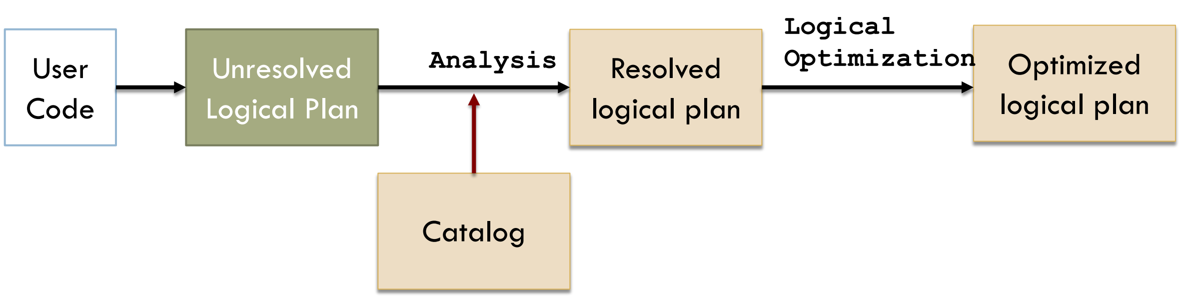
Logical Planning

- The **logical plan** only represents a set of abstract transformations
  - Does not refer to executors or drivers
  - Simply converts the user's set of expressions into the most optimized version
- Converting user's code into an *unresolved* logical plan
  - This plan is unresolved because although your code might be valid, the tables or columns that it refers to might or might not exist

How are columns and tables resolved?

- Spark uses the **catalog**, a repository of all table and DataFrame information, to resolve columns and tables in the analyzer optimizations
- The analyzer might reject the unresolved logical plan if the required table or column name does not exist in the catalog
- If the analyzer can resolve it, the result is passed through the Catalyst Optimizer

The Structured API Logical Planning Process



Catalyst Optimizer

- A **collection of rules** that attempt to optimize the logical plan by pushing down predicates or selections
- Catalyst is **extensible**
  - Users can include their own rules for domain-specific optimizations

Physical Planning [1/2]

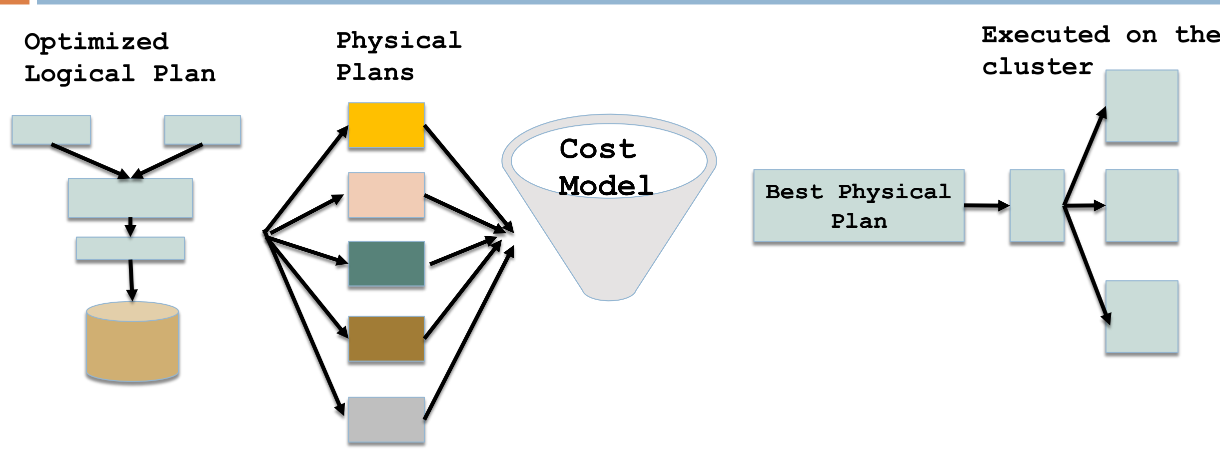
- The physical plan specifies how the logical plan will execute on the cluster
- Involves generating *different* physical execution strategies and comparing them through a **cost model**
- An example of the cost comparison might be choosing how to perform a given join by looking at the physical attributes of a given table
  - How big the table is or
  - How big its partitions are

Physical Planning [2/2]

- Physical planning results in a series of RDDs and transformations
- This is why Spark is also referred to as a compiler
  - Takes queries in DataFrames, Datasets, and SQL and compiles them into RDD transformations

27

The Physical Planning Process



28

Execution

- Spark performs further optimizations at runtime
- Generating native Java bytecode that can remove entire tasks or stages during execution
- Finally, the result is returned to the user

29

30

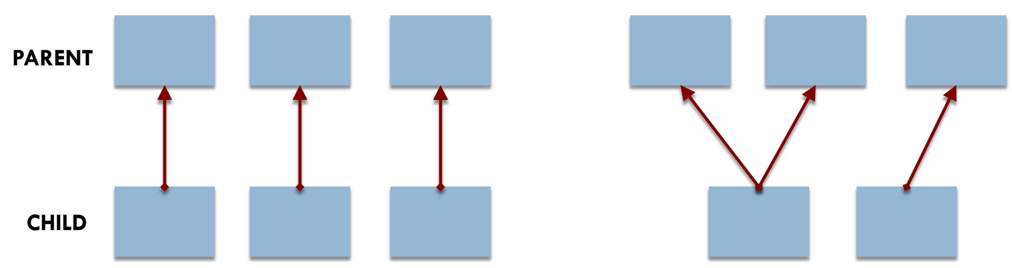
Transformations and Dependencies

- Two categories of **dependencies**
  - Narrow
    - Each partition of the parent RDD is used by **at most one partition of the child** RDD
  - Wide
    - Multiple child RDD partitions may depend on a single parent RDD partition
- The narrow versus wide distinction has significant implications for the way Spark evaluates a transformation and …
  - consequently, for its performance

Narrow Transformations

- Narrow transformations are those in which each partition in the child RDD has simple, finite dependencies on partitions in the parent RDD
- Dependencies can be **determined at design time**, irrespective of the values of the records in the parent partitions
- Partitions in narrow transformations can either depend on:
  - One parent (such as in the `map` operator), or
  - A unique subset of the parent partitions that is known at design time (`coalesce`)
- Narrow transformations can be executed on an arbitrary subset of the data without any information about the other partitions.

Dependencies between partitions for **narrow** transformations



COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT SPARK L32.33

33

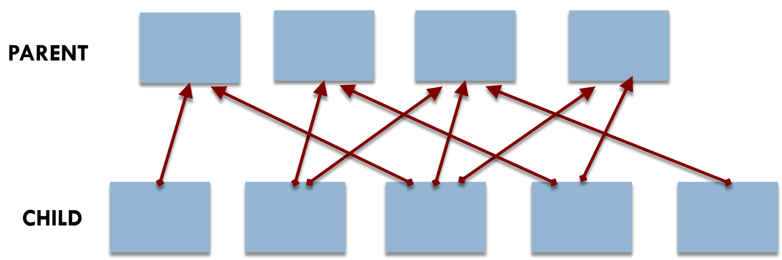
Wide Transformations

- Transformations with **wide dependencies** cannot be executed on arbitrary rows
- Require the data to be partitioned in a particular way, e.g., according the **value** of their key
  - In sort, for example, records have to be partitioned so that keys in the same range are on the same partition
- Transformations with wide dependencies include `sort`, `reduceByKey`, `groupByKey`, `join`, and anything that calls the `rePartition` function

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT SPARK L32.34

34

Dependencies between partitions for wide transformations



Wide dependencies **cannot be known fully before the data is evaluated**

The dependency graph for any operations that cause a **shuffle** (such as `groupByKey`, `reduceByKey`, `sort`, and `sortByKey`) follows this pattern

35

PAIR RDDS: WHAT TO WATCH FOR

36

Despite their utility, key/value operations can lead to a number of performance issues

- Most expensive operations in Spark fit into the key/value pair paradigm
  - Because **most wide transformations** are key/ value transformations,
    - And most require some fine tuning and care to be performant

In particular, operations on key/value pairs can cause …

1. Out-of-memory errors in the driver
2. Out-of-memory errors on the executor nodes
3. Shuffle failures
4. "Straggler tasks" or partitions, which are especially slow to compute

- The last three performance issues are all most often caused by **shuffles associated with the wide transformations**

Memory errors in the driver, are usually caused by actions

- Several key/value actions (including `countByKey`, `countByValue`, `lookUp`, and `collectAsMap`) return data to the driver
- In most instances they return *unbounded* data since the number of keys and the number of values are unknown
- In addition to number of records, the size of each record is an important factor in causing memory errors

Preventing out-of-memory errors with aggregation operations [1/2]

- `combineByKey` and all of the aggregation operators built on top of it (`reduceByKey`, `foldLeft`, `foldRight`, `aggregateByKey`) may lead to memory errors if they cause the accumulator to become too large for one key
- What about `groupByKey`?
  - It is actually implemented using `combineByKey` where the accumulator is an iterator with all the data.

Preventing out-of-memory errors with aggregation operations [2/2]

- Use functions that implement **map-side combinations**
  - Meaning that records with the same key are combined before they are shuffled
  - This can greatly reduce the shuffled read
- The following four functions are implemented to use map-side combinations
  - `reduceByKey`
  - `treeAggregate`
  - `aggregateByKey`
  - `foldByKey`

The contents of this slide-set are based on the following references

- *Learning Spark: Lightning-Fast Big Data Analysis. 1st Edition. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. O'Reilly. 2015. ISBN-13: 978-1449358624.* [Chapters 1-4, 10]
- Chambers, Bill, and Zaharia, Matei. Spark: The Definitive Guide: Big Data Processing Made Simple. O'Reilly Media. ISBN-13: 978-1491912218. 2018. [Chapters 5 and 9].
- Karau, Holden; Warren, Rachel. High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark. O'Reilly Media. 2017. ISBN-13: 978-1491943205. [Chapter 2]