CS 555: Distributed Systems [Spark]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Why use Hadoop if Spark is so much faster?

Topics covered in this lecture

- Orchestration Plans
- Transformations and Dependencies
- Spark Resilient Distributed Datasets

A simple Scala word count example

```
def simpleWordCount( rdd: RDD[ String]): RDD[( String, Int)] = {
        val words = rdd.flatMap(_. split(" "))
        val wordPairs = words.map((_, 1))
        val wordCounts = wordPairs.reduceByKey(_ + _)
        wordCounts
}
```


Orchestration Plans

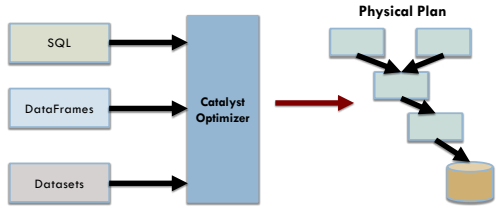
Executing Spark code in clusters: Overview

- Write DataFrame/Dataset/SQL Code.
- If valid code, 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 (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



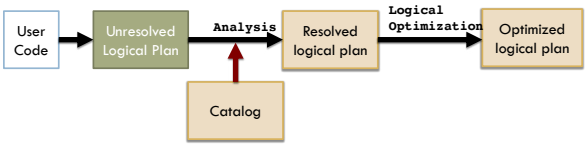
Logical Planning

- 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

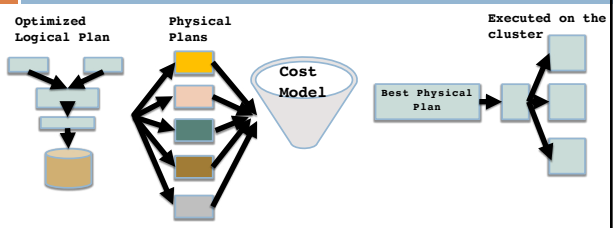
Physical Planning [1/2]

- 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

The Physical Planning Process



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

WIDE AND NARROW TRANSFORMATIONS

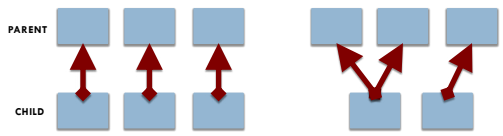
Transformations and Dependencies

- Two categories of dependencies
  - Narrow
    - Each parent partition is used by **at most one child partition**
  - Wide
    - **Multiple** child partitions may depend on a single parent 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 input partition contributes to only one output partition
- 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



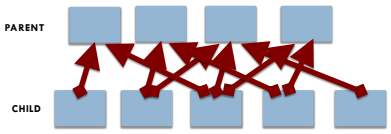
Wide Transformations

- A wide dependency (or wide transformation) style transformation will have input partitions contributing to many output partitions
- Also referred to as a **shuffle** whereby Spark will **exchange partitions** across the cluster

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

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

Other implications of narrow and wide transformations

- Narrow transformations
  - Spark automatically performs **pipelining**
    - If we specify multiple filters on DataFrames, they'll all be performed in-memory
- Wide transformations
  - When we perform a shuffle Spark writes the results to disk

One of the key optimizations that Spark performs is **pipelining**

- Occurs at and below the RDD level
- Any sequence of operations that feed data directly into each other, without needing to move it across nodes?
  - This sequence is collapsed into a single stage of tasks that do all the operations together

An example of pipelining

- If you write an RDD-based program that does a map , then a filter, then another map?
  - These will result in a single stage of tasks that will immediately read each input record, pass it through the first map , pass it through the filter, and pass it through the last map function if needed
  - This pipelined version of the computation is **much faster** than writing the intermediate results to memory or disk after each step
- The same kind of pipelining happens for a DataFrame or SQL computation that does a select , filter, and select

Let's look at a case when pipelining is infeasible

- When Spark needs to run an operation that has to **move data across nodes**, such as a reduce-by-key operation
  - Where input data for each key needs to first be brought together from many nodes
  - The engine can't perform pipelining anymore
  - Instead it performs a **cross-network shuffle**

How shuffles work

- Spark executes shuffles by first having the "source" tasks (those sending data) **write shuffle files to their local disks** during their execution stage
- Then, the stage that does the grouping and reduction launches and runs tasks
  - E.g., fetches and processes the data for a specific range of keys)

How writing the shuffle to disk helps

- Saving the shuffle files to disk lets Spark run this stage later in time than the source stage
  - E.g., if there are not enough executors to run both at the same time
  - Also lets the engine **re-launch reduce tasks on failure** without rerunning all the input tasks

Other benefits of shuffle persistence [1/2]

- Running a new job over data that's already been shuffled does not rerun the "source" side of the shuffle
  - Because the shuffle files were already written to disk earlier
  - Spark knows that it can use them to run the later stages of the job
    - It need not redo the earlier ones
  - In the Spark UI and logs, you will see the pre-shuffle stages marked as "skipped"
- This automatic optimization can save time in a workload that runs multiple jobs over the same data
- For even better performance?
  - You can perform your own caching with the DataFrame or RDD cache method

Other benefits of shuffle persistence [2/2]

- Running a new job over data that's already been shuffled does not rerun the "source" side of the shuffle
  - Because the shuffle files were already written to disk earlier
  - Spark knows that it can use them to run the later stages of the job
    - It need not redo the earlier ones
  - In the Spark UI and logs, you will see the pre-shuffle stages marked as "skipped"
- This automatic optimization can save time in a workload that runs multiple jobs over the same data
- For even better performance?
  - You can perform your own caching with the DataFrame or RDD cache method

RESILIENT DISTRIBUTED DATASET [RDD]

Resilient Distributed Dataset (RDD)

- RDD is an **immutable, distributed collection** of objects
  - Lazily evaluated, statically typed
- Each RDD is split into *multiple partitions*
  - Maybe computed on different nodes in the cluster
- Can contain any type of Java, Scala, or Python objects
  - Including user-defined classes

RDDs support …

- A number of predefined "coarse-grained" **transformations** (functions that are applied to the entire dataset),
  - Such as map, join, and reduce to manipulate the distributed datasets
- I/ O functionality to read and write data between the distributed storage system and the Spark JVMs

RDDs: memory residency and immutability implications

- Spark can keep an RDD loaded in-memory on the executor nodes throughout the life of a Spark application for faster access in **repeated computations**
- RDDs are immutable, so **transforming an RDD returns a new RDD** rather than the existing one
- Cross-cutting implications?
  - Lazy evaluation, in-memory storage, and immutability allows Spark to be easy-to-use, fault-tolerant, scalable, and efficient

Creation of RDDs

1. Loading an external dataset
2. Distributing a collection of objects via the driver program

```
>>> lines = sc.textFile("README.md")
```

Once created, RDDs offer two types of operations

- **Transformations**
  - Construct a new RDD from a previous one
  - E.g.: Filtering data that matches a predicate
- **Actions**
  - Compute a result based on an RDD
  - Return result to the driver program or save it in external storage system (HDFS)

Some more about RDDs

- Although you can define new RDDs anytime
  - Spark computes them in a **lazy fashion**
  - When?
    - The first time they are used in an *action*
- Loading lazily allows transformations to be performed *before* the action

Lazy loading allows Spark to see the whole chain of transformations

- Allows it to **compute just the data needed** for the result
- Example:

```
lines = sc.textFile("README.md")
pythonLines= lines.filter(lambda line: "Python" in line)
```

- If Spark were to load and store all lines in the file, as soon as we wrote `lines=sc.textFile()`?
  - Would waste a lot of storage space, since we immediately filter out a lot of lines

RDD and actions

- RDDs are **recomputed** (by default) every time you run an action on them
- If you wanted to reuse an RDD?
  - Ask Spark to **persist** it using `RDD.persist()`
  - After computing it the first time, Spark will store RDD contents in memory (*partitioned* across cluster machines)
  - Persisted RDD is used in future actions

A Closer look at RDD Operations

RDDs support two types of operations

- Transformations
  - Operations that **return a new RDD**. E.g.: `filter()`
- Actions
  - Operations that **return a result** to the driver program or write to storage
  - Kicks of a computation. E.g.: `count()`
- Distinguishing aspect?
  - Transformations return RDDs
  - Actions return *some other* data type

Transformations

- Many transformations are **element-wise**
  - Work on only one element at a time
- Some transformations are not element-wise
  - E.g.: We have a logfile, *log.text*, with several messages, but we only want to select error messages

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```


In our previous example …

- `filter` **does not mutate** `inputRDD`
  - Returns a pointer to an entirely new RDD
  - `inputRDD` can still be reused later in the program
- We could use `inputRDD` to search for lines with the word "warning"
  - While we are at it, we will use another transformation, `union()`, to print number of lines that contained either

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badlinesRDD = errorsRDD.union(warningsRDD)
```

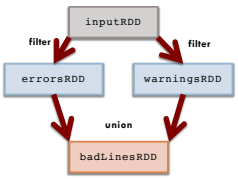

In our previous example

- Note how `union()` is different from `filter()`
  - Operates on 2 RDDs instead of one
- Transformations can actually operate on **any number** of RDDs

RDD Lineage graphs

- As new RDDs are derived from each other using transformations, Spark *tracks dependencies*
  - **Lineage graph**
- Uses lineage graph to
  - Compute each RDD on demand
  - Recover lost data if part of persistent RDD is lost

RDD lineage graph for our example



Actions

- We can create RDDs from each other using transformations
- At some point, we need to actually **do something** with the dataset
  - Actions
- Forces *evaluations of the transformations* required for the RDD they were called on

Each Spark program must contain an **action**

- Actions either:
  - Bring information back to the driver or
  - Write the data to stable storage
- Actions are what force evaluation of a Spark program
- Persist calls also force evaluation, but usually do not mark the end of Spark job
- Actions that bring data back to the driver include `collect`, `count`, `collectAsMap`, `sample`, `reduce`, and `take`.

Let's try to print information about badlinesRDD

```
print "Input had " + badLinesRDD.count() + "concerning lines"
print "here are 10 examples:"
for line in badLinesRDD.take(10)
    print  line
```


RDDs also have a `collect` to retrieve the entire RDD

- Useful if program filters RDD to a very small size and you want to deal locally
  - Your entire dataset must fit in memory on a single machine to use `collect()` on it
    - Should NOT be used on large datasets
- In most cases, RDDs **cannot be** `collect()`ed to the driver
  - Common to write data out to a distributed storage system … HDFS or S3

A caveat about actions and scaling

- Some of these actions do not scale well, since they can cause memory errors in the driver
- In general, it is best to use actions like `take`, `count`, and `reduce`, which bring back a fixed amount of data to the driver, rather than `collect` or `sample`.

Lazy Evaluation

- Transformations on RDDS are **lazily evaluated**
  - Spark will not begin to execute until it sees an action
- Uses this to **reduce the number of passes** it has to take over data by grouping operations together
- What does this mean?
  - When you call a transformation on an RDD (for e.g. `map`) the operation is not immediately performed
  - Spark internally records metadata that operation is requested

How you should think of RDDs

- Rather than thinking of it as containing specific data
  - Best to think of it as **containing instructions on how to compute the data** that we build through transformations
- Loading data into a RDD is lazily evaluated just as transformations are

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

- *Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. Learning Spark: Lightning-Fast Big Data Analysis. 1st Edition. O'Reilly. 2015. ISBN-13: 978-1449358624.* [Chapters 1-4]
- 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]
- Chambers, Bill, Zaharia, Matei. Spark: The Definitive Guide: Big Data Processing Made Simple. O'Reilly Media. ISBN-13: 978-1491912218. 2018. [Chapters 2, 3, 4, 15, and 16].