Spark

MODIFIED BASED ON THE SLIDES FROM MATEI ZAHARIA (BERKELEY)

AND PATRICK WENDELL (FROM DATABRICKS)

Bibliography

Section “Other Considerations” of Tuning Spark. https://spark.apache.org/docs/latest/tuning.html

Understanding the Performance of Spark Applications. Patrick Mendell. https://databricks.com/session/understanding-the-performance-of-spark-applications

Other issues to take into consideration

1. Scheduling and launching tasks
2. Execution of tasks
3. Writing data between stages
4. Collecting results

Scheduling and launching tasks

Serialized task is large due to a closure

```
hash_map = some_massive_hash_map()
rdd.map(lambda x: hash_map(x)).count_by_value()
```

Using Local Variables

External variables you use in a closure will automatically be shipped to the cluster:

- query = raw_input(“Enter a query:”)
- pages.filter(x -> x.startswith(query)).count()

Some caveats:

- Each task gets a new copy (updates aren’t sent back)
- Variable must be Serializable (Java/Scala) or Pickle-able (Python)
- Don’t use fields of an outer object (ships all of it!)

Scheduling and launching tasks

Serialized task is large due to a closure

```
hash_map = some_massive_hash_map()
rdd.map(lambda x: hash_map(x)).count_by_value()
```

Detecting:
- Spark will warn you!

Fixing:
- Use broadcast variables for large object
- Make your large object into an RDD

Scheduling and launching tasks

Large number of “empty” tasks due to selective filter

```
rdd = sc.textFile(“s3n://bucket/2013-data”)
            .map(lambda x: x.split(“\t”))
            .filter(lambda parts: parts[0] == “2013-10-17”)
            .filter(lambda parts: parts[1] == “19:00”)
rdd.map(lambda parts: (parts[2], parts[3]).reduceBy…
```

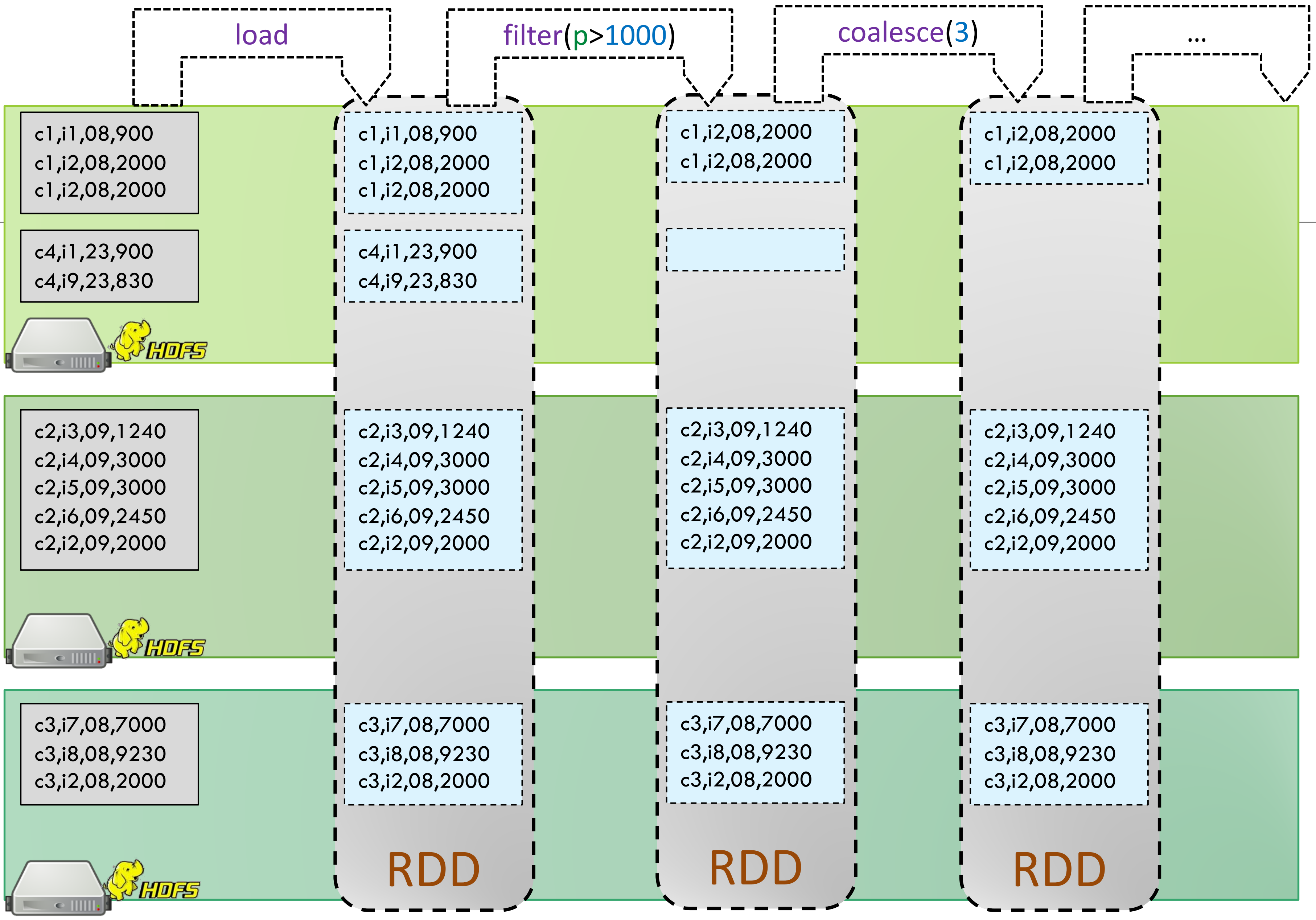
Detecting:

- Many short-lived (< 20ms) tasks

Fixing:

- Use `coalesce` or `repartition` operator to shrink RDD number of partitions after filtering:

```
rdd.coalesce(30).map(lambda parts: (parts[2]…
```


load
filter(p>1000)
coalesce(3)
...
c1,i1,08,900
c1,i2,08,2000
c1,i2,08,2000
c4,i1,23,900
c4,i9,23,830
HDFS
c2,i3,09,1240
c2,i4,09,3000
c2,i5,09,3000
c2,i6,09,2450
c2,i2,09,2000
HDFS
c3,i7,08,7000
c3,i8,08,9230
c3,i2,08,2000
HDFS
c1,i1,08,900
c1,i2,08,2000
c1,i2,08,2000
c4,i1,23,900
c4,i9,23,830
c2,i3,09,1240
c2,i4,09,3000
c2,i5,09,3000
c2,i6,09,2450
c2,i2,09,2000
c3,i7,08,7000
c3,i8,08,9230
c3,i2,08,2000
RDD
c1,i2,08,2000
c1,i2,08,2000
c2,i3,09,1240
c2,i4,09,3000
c2,i5,09,3000
c2,i6,09,2450
c2,i2,09,2000
c3,i7,08,7000
c3,i8,08,9230
c3,i2,08,2000
RDD
c1,i2,08,2000
c1,i2,08,2000
c2,i3,09,1240
c2,i4,09,3000
c2,i5,09,3000
c2,i6,09,2450
c2,i2,09,2000
c3,i7,08,7000
c3,i8,08,9230
c3,i2,08,2000
RDD

Execution of Tasks

Tasks with high per-record overhead

Detecting:

- Task run time is high

Fixing:

- Use `mapPartitions` or `mapWith` (scala)

Execution of Tasks

Skew between tasks

Detecting

- Stage response time dominated by a few slow tasks

Fixing

- Data skew: poor choice of partition key

→Consider different way of parallelizing the problem

→Can also use intermediate partial aggregations

- Worker skew: some executors slow/flakey nodes

→Set spark.speculation to true

→Remove flakey/slow nodes over time

Writing data between stages

Not setting the number of reducers

Default behavior: inherits # of reducers from parent RDD

Too many reducers:

→ Task launching overhead becomes an issue (will see many small tasks)

Too few reducers:

→ Limits parallelism in cluster

Spark SQL

https://www.edureka.co/blog/spark-sql-tutorial/

https://www.tutorialspoint.com/spark_sql/spark_sql_introduction.htm

Programming Interface

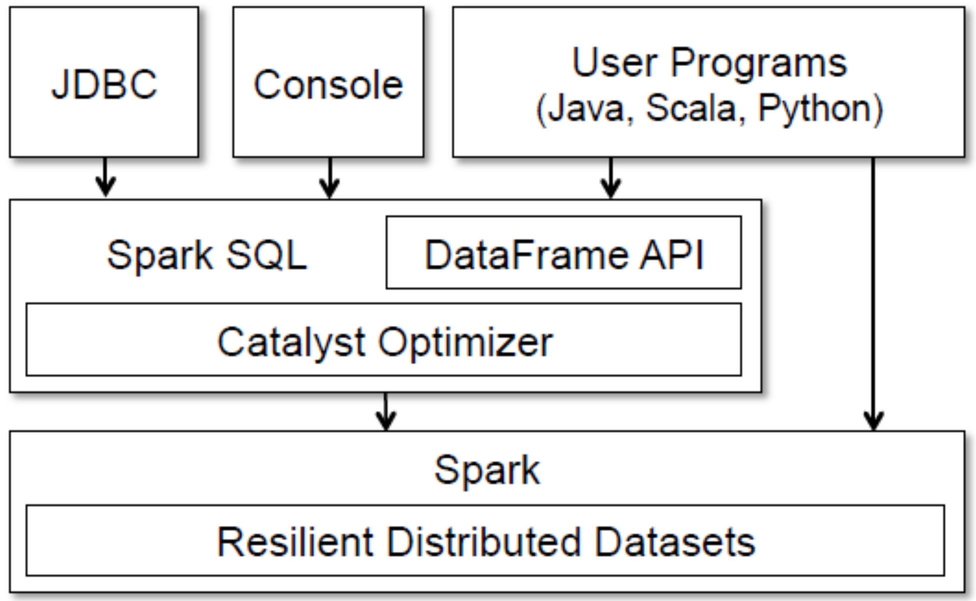

JDBC
Console
User Programs (Java, Scala, Python)
Spark SQL
DataFrame API
Catalyst Optimizer
Spark
Resilient Distributed Datasets

Figure 1: Interfaces to Spark SQL, and interaction with Spark.

DataFrame API

```
ctx = new HiveContext()
users = ctx.table("users")
young = users.where(users("age") < 21)
println(young.count())
```

Equivalent to a table in relational database

Can be manipulated in similar ways to the “native” RDD.

Data Model

Uses a nested data model based on Hive for tables and DataFrames

- Supports all major SQL data types

Supports user-defined types

Able to model data from a variety sources and formats (e.g. Hive, RDB, JSON, and native objects in Java/Scala/Python)

DataFrame Operations

Employees
- .join(dept , employees("deptId") === dept("id"))
- .where(employees("gender") === "female")
- .groupBy(dept("id"), dept("name"))
- .agg(count("name"))

All of these operators build up an abstract syntax tree (AST) of the expression, which is then passed to Catalyst for optimization.

The DataFrames registered in the catalog can still be unmaterialized views, so that optimizations can happen across SQL and the original DataFrame expressions.

Integration in a full programming language (DataFrames can be passed Inter-language but still benefit from optimization across the whole plan).

Querying Native Datasets

Allows users to construct DataFrames directly against RDDs of objects native to the programming language.

Automatically infer the schema and types of the objects.

Accesses the native objects in-place, extracting only the fields used in each query (avoid expensive conversions).

```
case class User(name: String , age: Int)

// Create an RDD of User objects

usersRDD = spark.parallelize(List(User("Alice", 22), User("Bob", 19)))

// View the RDD as a DataFrame

usersDF = usersRDD.toDF
```

In-Memory Caching

Columnar cache can reduce memory footprint by an order of magnitude

User-Defined Functions

supports inline definition of UDFs (avoid complicated packaging and registration process)

Catalyst Optimizer

Based on functional programming constructs in Scala.

Easy to add new optimization techniques and features,

- Especially to tackle various problems when dealing with “big data”(e.g. semi-structured data and advanced analytics)

Enable external developers to extend the optimizer.

- Data source specific rules that can push filtering or aggregation into external storage systems
- Support for new data type

Supports rule-based and cost-based optimization

Tree

Scala Code:

Add(Attribute(x), Add(Literal(1), Literal(2)))

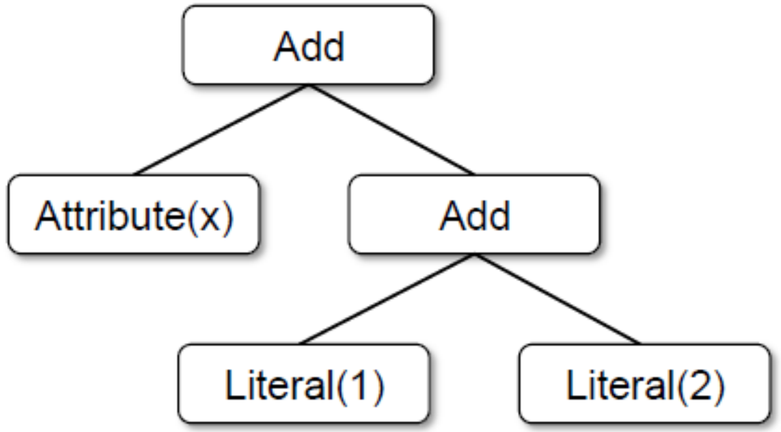


Figure 2: Catalyst tree for the expression x+(1+2)**.**

Rules

Trees can be manipulated using rules, which are functions from a tree to another tree.

- Use a set of pattern matching functions that find and replace subtrees with a specific structure.

```
tree.transform{
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
  case Add(left , Literal(0)) => left
         case Add(Literal(0), right) => right
}
```

Catalyst groups rules into batches, and executes each batch until it reaches a fixed point.

Using Catalyst

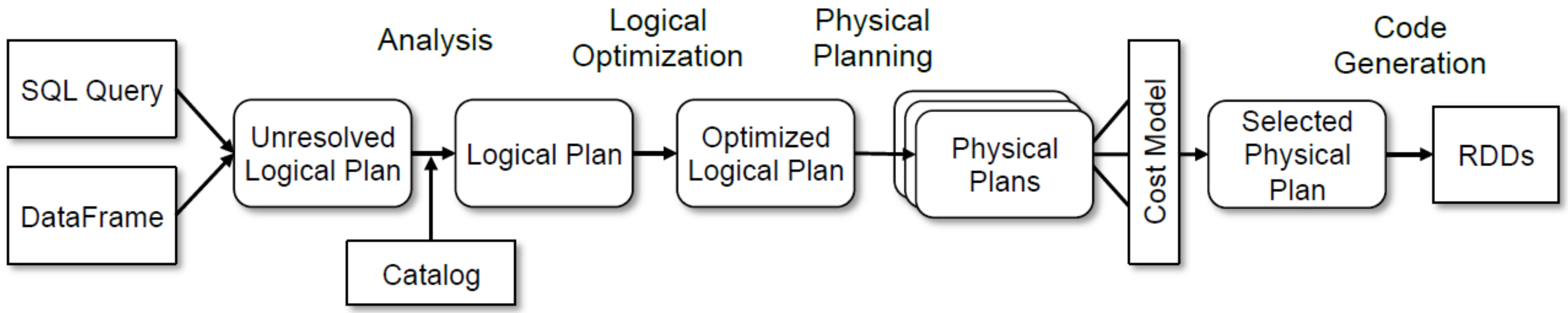


Figure 3: Phases of query planning in Spark SQL. Rounded rectangles represent Catalyst trees.

Advanced Analytics Features
Specifically designed to handle “big data”

A schema inference algorithm for JSON and other semi-structured data.

A new high-level API for Spark’s machine learning library.

Supports query federation, allowing a single program to efficiently query disparate sources.