Lecture 9

Apache Spark

Iztok Savnik, FAMNIT

December, 2025.

Sources

Zaharia, et.al., Spark: cluster computing with working sets, HotCloud, 2010.

Zaharia, et al., Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, NSDI, 2012.

Armbrust, et.al, Spark SQL: Relational Data Processing in Spark, SIGMOD, 2015.

Zaharia, et.al., Apache Spark: A Unified Engine for Big Data Processing, CACM, 2016.

Early Spark

Big data applications require a mix of processing techniques, data sources and storage formats.

- Earliest systems: **MapReduce** is powerful, low-level, procedural programming interface.
  - Offers low level, manual optimization.
- Multiple new systems sought to provide a more productive **relational interfaces** to big data.
  - Pig, Hive, Dremel and Shark: take advantage of declarative queries to provide richer automatic optimizations.

Users often prefer writing declarative queries.

On the other hand, the relational approach is insufficient for many big data applications.

Spark SQL

Relational approach is **insufficient** for many big data applications.

1) Users want to perform ETL to and from various data sources that might be semi- or un-structured, requiring custom code.
2) Users want to perform advanced analytics, such as machine learning and graph processing.

Most data pipelines would ideally be expressed with a combination of both **relational queries** and complex **procedural algorithms**.

Spark SQL lets users seamlessly intermix the two.

Resilent Distributed Dataset (RDD)

- **Immutable**, **distributed** collection of objects that Spark can process in parallel across a cluster.
- **Resilent** -- Spark can recompute lost partitions using the lineage (the history of operations applied to the data)
- Created from HDFS files (text, JSON, CSV, etc.)
  - One HDFS segment (128MB) is one partition

File.txt → Partition 0 → Executor 1 → map → filter → output
Partition 1 → Executor 2 → map → filter → output
Partition 2 → Executor 3 → map → filter → output

Resilient Distributed Dataset (RDD)

An RDD is a read-only, partitioned collection of records.

- Created through deterministic operations on either (1) data in stable storage or (2) other RDDs.
- We call these operations **transformations** (map,filter,join).

RDDs do not need to be materialized at all times.

- RDD has information about how it was derived from other datasets (its **lineage**).
- It can be **reconstructed** from data on disk.

Users can control two other aspects of RDDs:

- **persistance**, and **partitioning**.
- Determine which RDDs, and how they are going to be stored.

Spark Programming Interface

Start by defining RDDs through **transformations** on data from stable storage (e.g., map, filter, join).

Use **operations** that return a value to the application or export data to a storage system (e.g., count, collect, save).

Programmers can call a method **persist** to indicate which RDDs they want to reuse in future operations.

RDDs: Transformations and actions

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] ⇒ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] ⇒ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] ⇒ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	:	RDD[T] ⇒ Long
	$collect()$	:	RDD[T] ⇒ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] ⇒ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Example: Console Log Mining

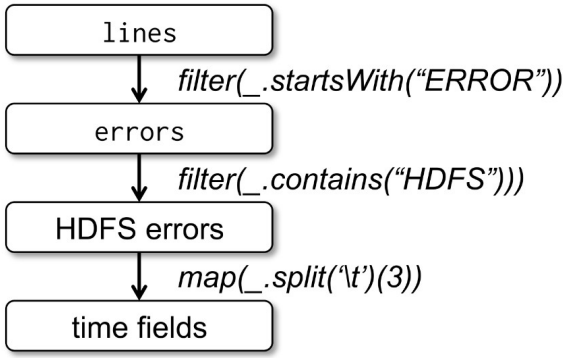
Web service is experiencing errors and an operator wants to search TB of logs in the HDFS to find the cause.
Scala code:

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

- RDD 'lines' backed by an HDFS file.
- RDD with lines is filtered into RDD 'errors'.
- RDD 'errors' (partitions) are instructed to stay in memory (to be further used in queries).

Example: Console Log Mining

User can also perform further transformations on RDD:




```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
        .map(_.split('\t')(3))
        .collect()
```

– Note that collect() returns the elements themselves (to an app).

Advantages of RDD Model

Compare RDDs against distributed shared memory (DSM).

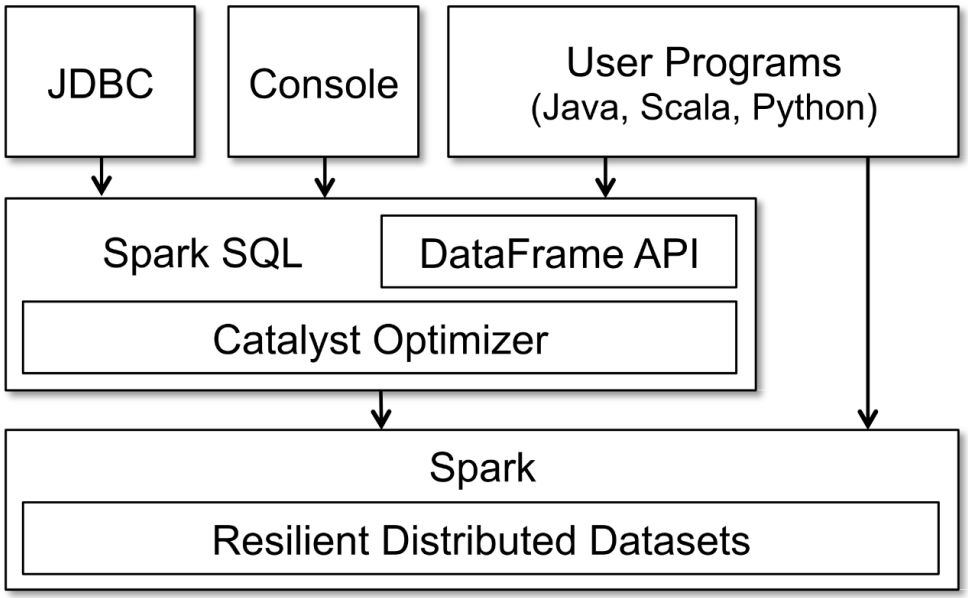
Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Spark SQL

Spark SQL runs as a library on top of Spark.

Interfaces of Spark SQL:

- JDBC/ODBC
- Command-line console
- DataFrame API integrated into Spark's supported programming languages.



DataFrame

A distributed collection of data organized into named columns, similar to a table in a relational database.

Some properties:

- Can be created from **various data sources**, such as CSV, JSON, Parquet, JDBC databases, or existing RDDs.
- DFs provide a **high-level API** for querying, filtering, aggregating, and transforming data using DSL expressions or SQL.
- They benefit from **Catalyst Optimizer**, which automatically optimizes query execution for better performance.
- DF includes a **schema**: each column has a name and a data type.
- DFs can be **cached** or **stored** as Parquet, JSON, CSV, Hive table.

DataFrame API

Main abstraction in Spark SQL's API is a DataFrame (DF).

- DF is equivalent to a table in a relational database.
- Manipulated in similar ways to the "native" Spark RDDs.
- DFs keep track of their **schema**.
- Support various **relational operations** that lead to more optimized execution.

DFs can be constructed from

- Tables in a system catalog (based on external data sources) or
- RDDs of native Java/Python objects.

DFs can be manipulated with

- various relational operators, such as where and groupBy, which take expressions in a domain-specific language (DSL), and
- All operations on RDDs (map,reduce,etc.)

DataFrame API

Spark DFs are **lazy**.

- DF object represents a logical plan to compute a dataset.
- **No execution** occurs until the user calls a special “output operation” (action) such as save.
- Enables rich **optimization** across operations used to build DF.
- DF object represents a **logical plan** to compute a dataset.
- Example:

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

Data model

Spark SQL uses a **nested data model** based on Hive for tables and DataFrames.

Supports all major SQL data types, complex types and UDT

- Basic: boolean, integer, double, decimal, string, date, and timestamp.
- Complex data types: structs, arrays, maps and unions.
  - Provides first-class support for complex data types in the query language and the API.
- In addition, Spark SQL also supports user-defined types (UDT).

DataFrame Operations

Relational operations used on DataFrames.

- Operations are expressed in a DSL.
  - PySpark and Scala
  - Similar to R data frames and Python Pandas
- DFs support all common relational operators.
  - Projection (select), filter (where), join, and aggregations (groupBy).
  - All these operators take *expression* objects in a limited DSL

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

DataFrame Operations

DFs can be registered as **temporary tables** in the system catalog and queried using SQL.

```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("SELECT count (*), avg(age) FROM young")
```

- SQL is convenient for computing multiple aggregates concisely.
- SQL allows programs to expose datasets through JDBC/ODBC.

DataFrame Execution

Distributed, immutable, lazy evaluation, optimized execution.

DataFrame execution

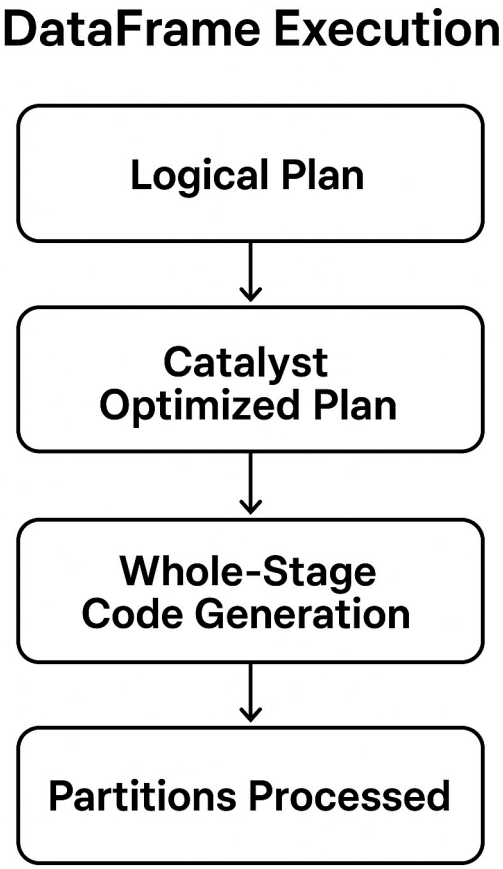
- Query, Logical Plan, Catalyst Optimized Plan, Physical Plan, Whole-Stage Code Generation, Execution on Partitions

Stages are compiled into JVM code by using Tungsten.

Include **stage boundaries** for shuffles.

Operations on DataFrames are lazy

- Executed when an action like show() or collect() is called.



Catalyst Optimizer

A new extensible optimizer, Catalyst, based on functional programming constructs in Scala.

- Easy to add new optimization techniq. and features to Spark SQL.
  - Handle problems with "big data" (e.g., semi-structured data).
- Enable external developers to extend the optimizer.
  - Adding data source specific rules (e.g., low-level filtering).

Catalyst uses standard features of Scala, such as pattern-matching.

- Rules are easy to specify.

Catalyst Optimizer

A general library for representing trees and applying rules to manipulate them.

- On top of this framework, we have built libraries specific to relational query processing.

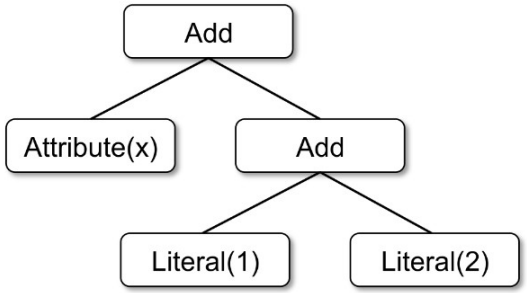
Several sets of rules that handle different phases of query execution:

- Analysis,
- Logical optimization,
- Physical planning, and
- Code generation to compile parts of queries to Java bytecode.

Catalyst offers several public extension points.

- External data sources and user-defined types.

Trees and rules



The main data type in Catalyst is a **tree** composed of node objects.

- **Nodes**: Literal(value: Int), Attribute(name: String),
  Add(left: TreeNode, right: TreeNode)
- These classes can be used to build up trees, e.g., for expr. X+(1+2).
- *Example*: Add(Attribute(x), Add(Literal(1) , Literal(2)))

Rules are functions from a tree to another tree.

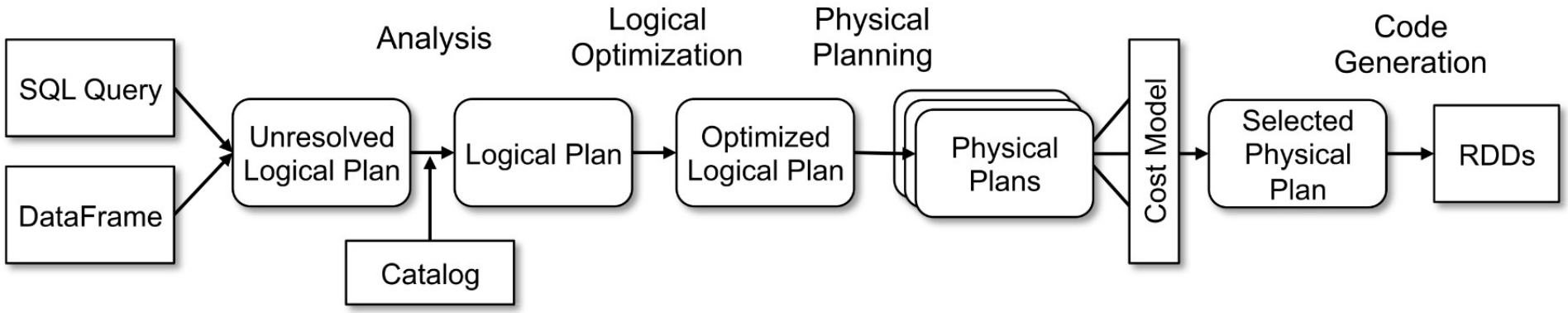
- Pattern matching allows extracting values from potentially nested structures of algebraic data types.
- *Example*: x+(1+2) => x+3

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

Using Catalyst in Spark SQL

Catalyst's general tree transformation framework is used in four phases.

- (1) analyzing a logical plan to resolve references, (2) logical plan optimization, (3) physical planning, and (4) code generation



Catalyst - Analysis

Spark SQL begins with a relation to be computed:

- From an abstract syntax tree (AST) returned by a SQL parser, or from a DataFrame object constructed by API.

Relation may contain unresolved attribute references or relations.

Input: Unresolved Logical Plan

Tasks: Resolve table names, column names, and functions, validate schemas and data types

Output: Analyzed Logical Plan

Catalyst - Logical Optimization

Applies standard rule-based optimizations to the logical plan.

- Constant folding, predicate pushdown, projection pruning, null propagation, Boolean expression simplification, and other rules.
- It is extremely simple to add rules for a wide variety of situations.

Input: Analyzed Logical Plan

Rules: Constant folding, Predicate pushdown, Projection pruning, Boolean simplification, Join reordering

Output: Optimized Logical Plan

Catalyst - Physical Planning

Takes a logical plan and generates one or more physical plans, using physical operators.

Input: Optimized Logical Plan

Tasks: Generate multiple physical plans, Apply cost-based optimization, Select best physical plan

Output: Selected Physical Plan

- Cost-based optimization is only used to select join algorithms.
- Rule-based physical optimizations.
  - Pipelining filters and projections into one map() operation.
  - Push operations from the logical plan into data sources.

Catalyst - Code Generation

Generating Java bytecode to run on each machine.

Spark SQL often operates on in-memory datasets.

- Rule-based physical optimizations.
- Processing is CPU-bound.
- Wanted to support code generation to speed up execution.

Input: Physical Plan

Task: Generate optimized bytecode for execution (via Tungsten)

Output: Spark RDD/DAG ready for execution

Conclusions

Spark SQL extends Spark with a declarative DataFrame API to allow relational processing, offering benefits such as automatic optimization, and letting users write complex pipelines that mix relational and complex analytics.

Spark SQL is based on an extensible optimizer called Catalyst that makes it easy to add optimization rules, data sources and data types by embedding into the Scala programming language.