INTRODUCTION TO SPARK WITH SCALA

Spark SQL – DataFrame & Datasets

Spark SQL

- Overview
- Spark SQL Architecture
- Working with DataFrame
  - Data source
  - Queries & actions
- Datasets
- Summary

Overview

Spark **SQL**

is <u>more</u> than SQL

Write less code, read less data

Let the optimizer do the hard work

Overview

The easiest way to write efficient program is to not worry about it and get your programs automatically optimized

Overview

- Overview
  - Make big data processing easier for wider audience
  - Working with structured and semi-structured data
  - Leverage schema for efficient loading and querying
  - Programming abstraction is DataFrame
  - Query data through SQL – relational processing
  - Easily combine declarative queries with procedural code
  - Include a highly extensible optimizer called Catalyst
  - Support additional data sources

Integrate relational processing with Spark's functional programming API

Overview

- DataFrames
  - Inspired by data frames in R and Python
  - A distributed collection of rows organized into named columns
    - Similar to a table in RDBMS
  - Abstraction for selecting, filtering joining and aggregating
  - Very rich optimization under the hood
  - Can be constructed from many sources
    - Structured file (JSON), tables in Hive, external DB, RDD
  - More convenient and more efficient that procedural API

FKA - SchemaRDD

Overview

Spark Programming Model Shift

RDD

- Unstructured data
- Functional programming style

DataFrame

- Semi-structured data
- Structured data
- DSL or SQL

Little optimization → Lots of optimization

Overview

RDD to Spark SQL

RDD

```
pdata.map { case (dpt, age) => dpt -> (age, 1) }
     .reduceByKey { case ((a1, c1), (a2, c2)) => (a1 + a2, c1 + c2)}
     .map { case (dpt, (age, c)) => dpt -> age/ c }
```

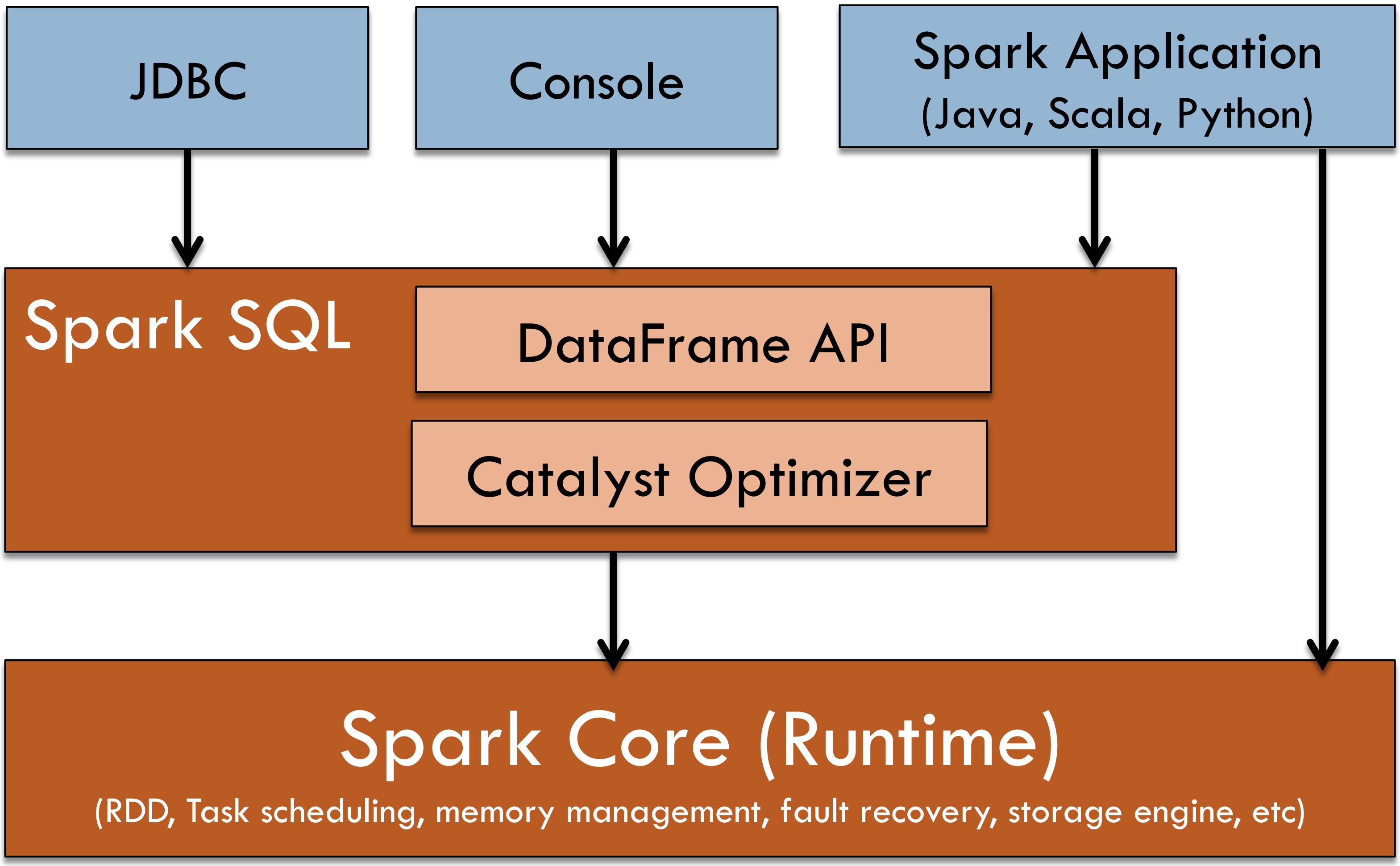
Dataframe

```
data.groupBy("dept").avg("age")
```

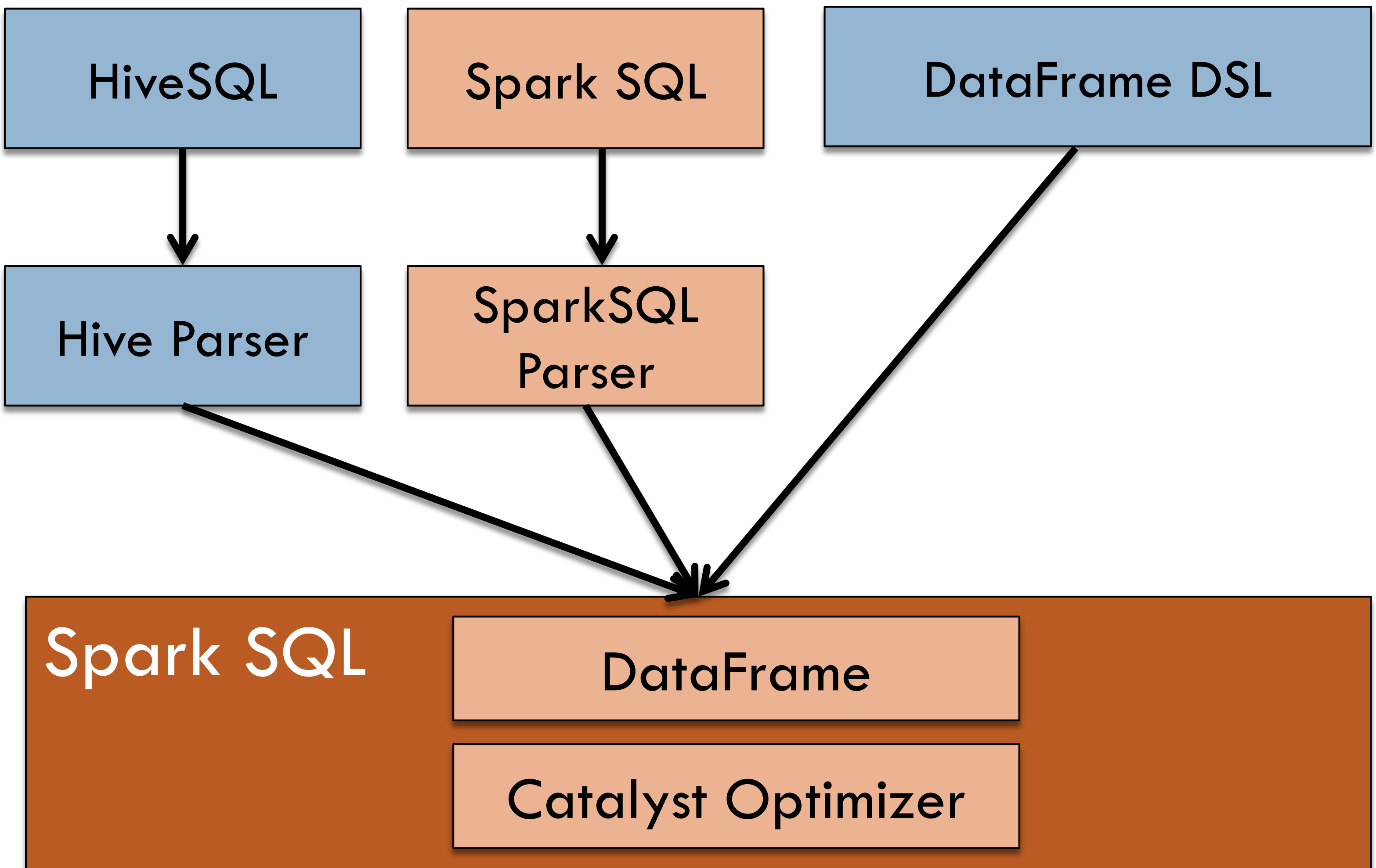
SQL

```
select dept, avg(age) from data group by 1
```

Spark SQL Architecture



Spark SQL Architecture



Spark SQL Architecture

Supported Data Formats and Sources

Parquet

HIVE

MySQL

PostgreSQL

and more ...

{ JSON }

amazon web services™ | S3

openstack

HDFS

External

APACHE HBASE

cassandra

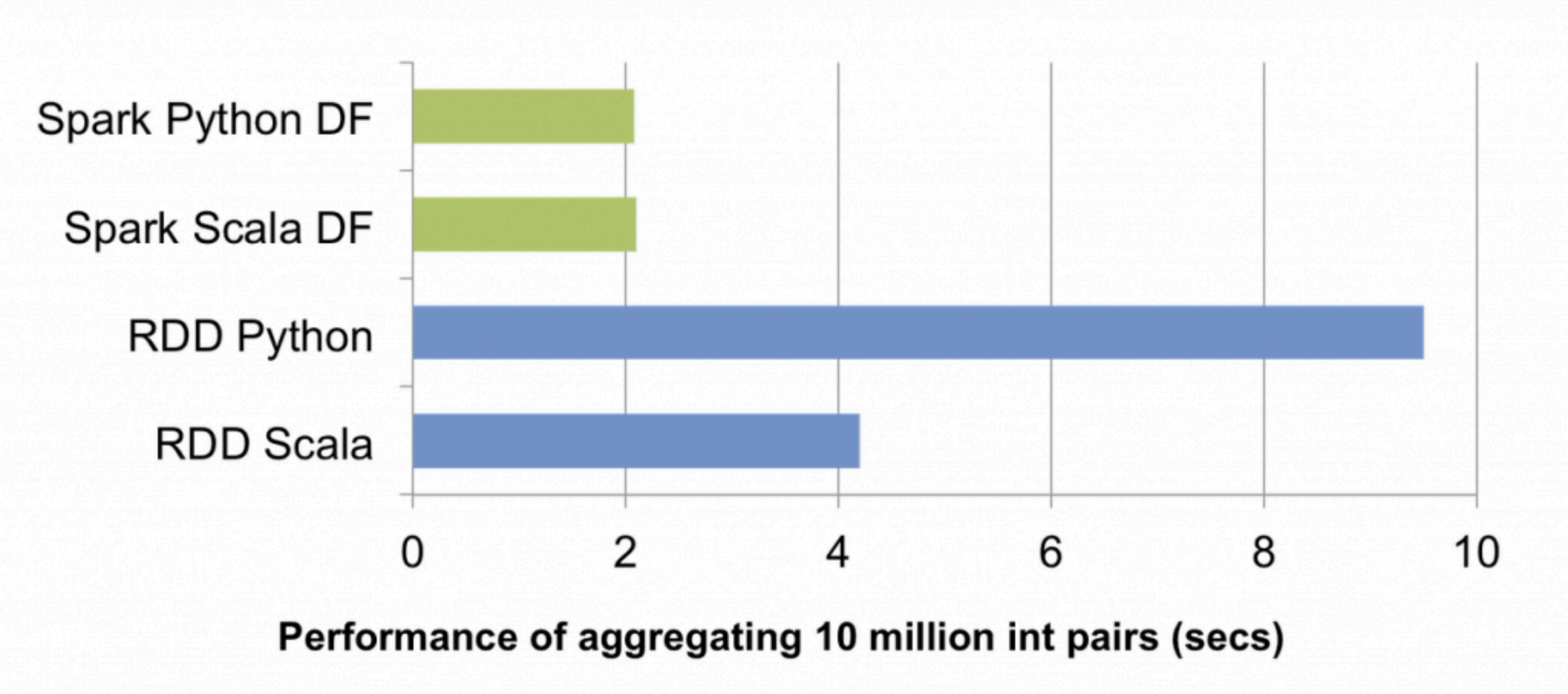
amazon web services

Amazon Redshift

elasticsearch.

Spark SQL Architecture

Democratizing Speed



DataFrame APIs

- Data Sources
  - Ability to combine data from multiple sources
  - Loading and saving data
  - DataFrameReader
    - jdbc(…..), json(…..)
    - parquet(…), orc(….)
  - DataFrameWriter
    - jdbc(…..), json(…..)
    - parquet(…), orc(….)

Working with DataFrame

Data Sources

Name	Description
spark.read.json(path) spark.read.csv(path)	Read JSON Read CSV
spark.read.parquet(path) spark.read.orc(path)	Read Parquet Read ORC
spark.read.jdbc(driverClass, table, properties)	Read data through JDBC
df.write.json(path), df.write.csv(path)	Write JSON, CSV
df.write.parquet(path), df.write.orc(path)	Write Parquet, ORC

Built-in supported formats: csv, json, parquet, orc, jdbc

DataFrame APIs

- Hive integration
  - Use HiveSQL
  - Access to Hive UDFs
  - Read from Hive tables

```
import sqlContext.implicits._

val sqlContext = new org.apache.spark.sql.HiveContext(sc)

sqlContext.sql("SELECT name, email from users")
          .collect().foreach(println)
```

Working with DataFrame

- Create DataFrame through SparkSession
  - Existing RDD
  - From data sources

```
# from JSON data source
val df = spark.read.json("/<path>/movies-json")

# display top 20 rows in tabular form
df.show()
```

Working with DataFrame

Create DataFrame from existing RDD using Case class

```
case class Movie(actor: String, title:String, year: Int)

val movies = spark.read.textFile("movies")
                  .map(_.split("\t"))

val validMovies = movies.filter(m => m.length == 3)
      .map(p => Movie(p(0), p(1), p(2).trim.toInt))

val movieDF = validMovies.toDF()

movieDF.printSchema

movieDF.show()
```

Working with DataFrame

Create DataFrame from existing RDD using inline schema

```
import scala.util.Random

val rdd = sc.parallelize(1 to 100)
              .map(x => (x, Random.nextInt(100)* x))

val kvdf = rdd.toDF("key","value")

kvdf.printSchema

kvdf.show()
```

Working with DataFrame

Create DataFrame – create schema programmatically

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._
val rdd = sc.parallelize(Array(
  Row(1, "John Doe",  30),
  Row(2, "Mary Jane", 25)
))

val schema = StructType(Array(
   StructField("id", LongType, true),
   StructField("name", StringType, true),
   StructField("age", BigIntType, true)
))

val df = spark.createDataFrame(rdd, schema)
df.printSchema
df.show
```

Working with DataFrame

Create DataFrame – create schema programmatically

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._

val data = Seq((1, "John Doe",  30),
               (2, "Mary Jane", 25))

val df = data.toDF("id", "name", "age")

df.printSchema
df.show
```

Working with DataFrame

DataFrame APIs

Name	Description
agg(expr, exprs)	Aggregate on the entire DF
col(name)	Return an instance of Column
cube(col1, cols)	Create a multi-dimensional cube using specified columns
distinct	Return DF that contains only unique rows
filter(conditionExpr) where(condition)	Filter rows based on given expression Filter rows with given condition
groupBy(cols)	Group by one or more columns to perform aggregation
limit(n)	Taking first N rows
select(col1, col2)	Select a set of column
sort(col1, col2)	Sort based one or more columns

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrame

Working with DataFrame

DataFrame APIs

Name	Description
registerTempTable(tableName)	Register this DF as a temporary table
join(df2, joinExpr, joinType)	Join with another DF. joinType - outer, left_outer, right_outer, leftsemi

```
val left = sc.parallelize(Seq((1,2),(2,3))).toDF("key","value")
val right = sc.parallelize(Seq((1,10),(2,15)
                           (2,20))).toDF("key","value")

left.registerTempTable("kv")
spark.sql("select * from kv")

left.join(right, left("key") === right("key"), "inner").show
left.join(right, left("key") === right("key"), "leftsemi").show
```

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrame

Working with DataFrame

DataFrame Actions

Name	Description
collect()	Return an array of Row objects
count()	Return number of rows
describe(cols)	Compute statistics for numeric rows – count, mean, stddev, min, max
first()	Return first row
head(n)	Return the first n rows
show()	Display first 20 rows in tabular format
show(n)	Display first n rows in tabular format
take(n)	Return first N rows

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrame

Working with DataFrame

DataFrame Aggregation APIs

```
movies.groupBy("year").count()

// useful for after grouping
//(count(),avg(columName))
```

Name	Description
avg(columnNames)	Compute average value for each numeric column
count()	Count # of rows per group
max(columnNames)	Compute max value for each numeric column
mean(columnNames)	Compute mean for each numeric column
min(columnNames)	Compute min value for each numeric column
sum(columnNames)	Compute sum value for each numeric column

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.GroupedData

Working with DataFrame

DataFrame Aggregation APIs

```
import org.apache.spark.sql.functions._

movies.groupBy("year").agg(
   max("rating").as("max_rating"),
   min("rating").as("min_rating")
)
```

Working with DataFrame

Useful Column APIs

Name	Description
contains(other)	Contains other element
desc	Return an ordering used in sorting
endsWith(str)	String ends with another string literal
startsWith(Str)	String starts with
equal(to)	Equality test
isNotNull, isNull	True if not null, true is null
like(literal)	SQL like expression

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column

Working with DataFrame

Working with column

```
val left = Seq((1,2),(2,3)).toDF("key","value")

left.select("key").show
left.select(col("key")).show
left.select(left("key")).show
left.select($"key").show
left.select('key).show


left.select($"key", $"key" > 1).show
+---+---------+
|key|(key > 1)|
+---+---------+
|  1|    false|
|  2|     true|
+---+---------+
```

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column

Working with DataFrame

Working with column

```
val mixedData = List(("1", "2.5",
                      "2017-07-31", "2017-07-31 15:04:58.865"))

val mixedDataDF = spark.sparkContext.parallelize(mixedData)
                   .toDF("intCol", "floatCol", "dateCol", "tsCol")

val typeMixedDataDF = mixedDataDF.select(
          $"intCol".cast("int"), $"floatCol".cast("float"),
          $"dateCol".cast("date"), $"tsCol".cast("timestamp"))

typeMixedDataDF.printSchema
root
 |-- intCol: integer (nullable = true)
 |-- floatCol: float (nullable = true)
 |-- dateCol: date (nullable = true)
 |-- tsCol: timestamp (nullable = true)
```

Working with DataFrame


```
val movies = spark.read.json("/movies-json")

movies.count()
movies.show()
movies.printSchema

movies.select("title", "year").show

movies.filter($"year" === 2010).show
movies.filter($"year" =!= 2001).show

movies.filter($"actor".contains("aron") && $"year" >
2000).show

movies.filter($"actor" === "Jolie, Angelina").show()

movies.groupBy("year").count().show()
```

Working with DataFrame


```
// distinct
movies.select("actor").count                  // 31393
movies.select("actor").distinct.count         // 6527

// limit
movies.limit(20).show

// sort
movies.sort("actor").show
movies.sort($"actor".desc).show

movies.sort(desc("actor")).show
movies.orderBy("actor").orderBy("title")).show

// isNotNull
movies.filter(col("title").isNotNull).show
```

Working with DataFrame


```
// join
val movies = spark.read.json("/movies-json")
val movieRatings = spark.read.json("/movieRatings-json")

// logically doesn't make sense
val joinedMovies = movies.join(movieRatings, movies("title")
=== movieRatings("title"))

joinedMovies.printSchema
joinedMovies.count
joinedMovies.show



val bestMoviesPerYear =
joinedMovies.groupBy(movies.col("year")).agg(
   min(movieRatings("rating")).alias("minRating"),
   max(movieRatings("rating")).alias("maxRating")
)
```

Working with DataFrame

- Caching data in memory
  - Spark SQL uses in-memory columnar format
  - Scan only needed columns
  - Compression to minimize memory usage

```
val userDF = spark.read.json("/data/people.json")
// where is identical to filter
val youngDF = userDF.where($"age" < 21)

// persist  dataframe
youngDF.persist()

// un-persist dataframe
youngDF.unpersist()
```

Working with DataFrame

- Cache tables using in-memory columnar format
  - Instead of JVM objects
- Will require less memory footprint
- Automatically tune compression
- Scan only required columns
- Applicable for interactive and iterative workload

```
// caching
val movies = spark.read.json("/movies-json")
movies.persist()

movies.unpersist()
```

Working with DataFrame

- Register as temporary table to use SQL

```
val movies = spark.read.json("/movies-json")

movies.createOrReplaceTempView("movies")

spark.sql("select * from movies where year > 2010").show

val newMovies = spark.sql("select * from movies where year >
2010")

newMovies.map(t => "Title: " +
t.getAs[String]("title")).collect().foreach(println)
```

Datasets


```
DataFrame = DataSet[Row]

DataSet[Row] = DataFrame
```

Spark 2.0 Unified APIs

Datasets

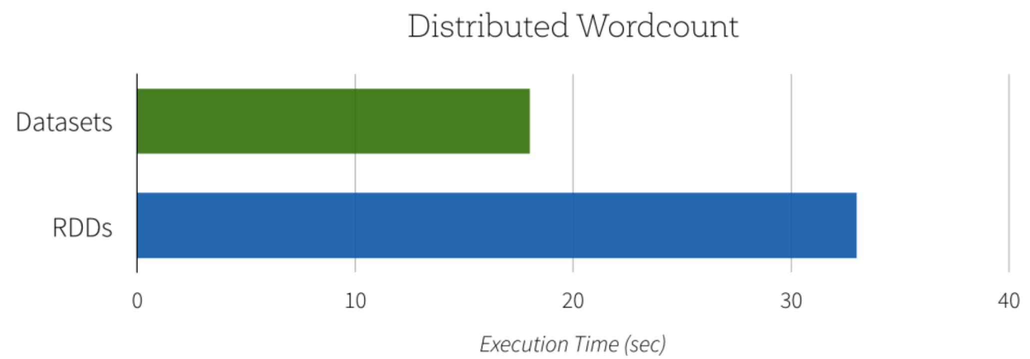
- Pushing Spark's usability & performance
- Support type-safe, object-oriented programming
- Same underlying components
  - Catalyst optimizer & Tungsten's fast in memory encoding
- Work alongside with RDD API
- Benefits
  - Compile-time type safety
  - Direct operations over user-defined classes

Datasets

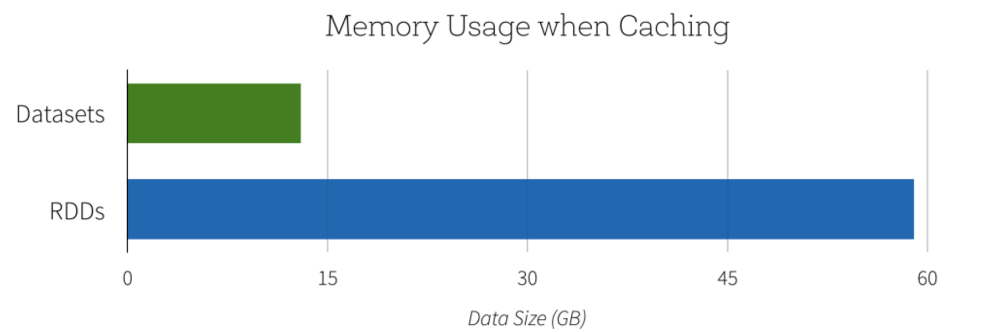
- A strongly-typed, immutable collection of objects
- Smart Encoder
  - Converting between JVM objects and tabular format
  - Auto-generated for widely used types
    - Scala case classes and Java Beans
  - Skip de-serializing when performing filtering, sorting, and hashing operation
- A specialized DataFrame – elements map to specific JVM object type

Datasets

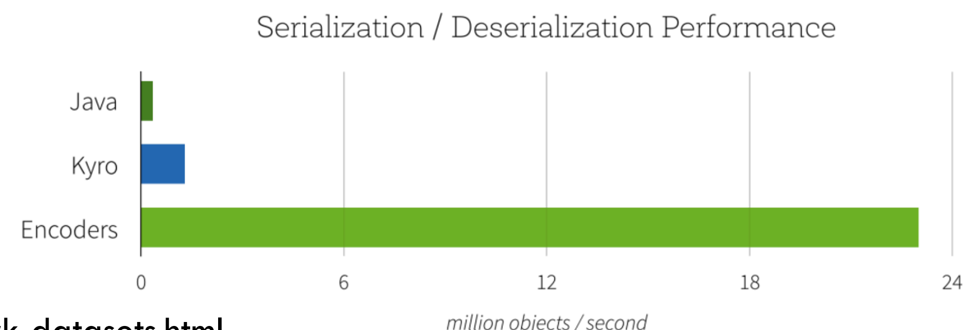
- Execution Speed
  - Built-in aggregation



- Space Usage
  - Optimal memory layout



- Encoder Speed
  - Custom bytecode



https://databricks.com/blog/2016/01/04/introducing-spark-datasets.html

Datasets

- Encoders
  - Translating between domain objects and Spark's internal format

JVM Object

```
Movie("Jessica Tuck", "Super 8", 2011)
```

Tunsteng Internal Representation

0x0	12	"Jessica Tuck"	7	"Super 8"	680	2011

Datasets

Working with DataSet

```
case class Movie(actor: String, title:String, year: Long)

val movieDF = sqlContext.read.json(”<path>/movies-json")
// based on column name
val movieDS = movieDF.as[Movie]

movieDS.printSchema

# movies > 2010
movieDS.filter(m => m.year > 2010).show

# group by actor
movieDS.groupBy(m => m.actor).count().show

# convert back to DF
val actorDF = movieDS.groupBy(“actor”).count().toDF()
```

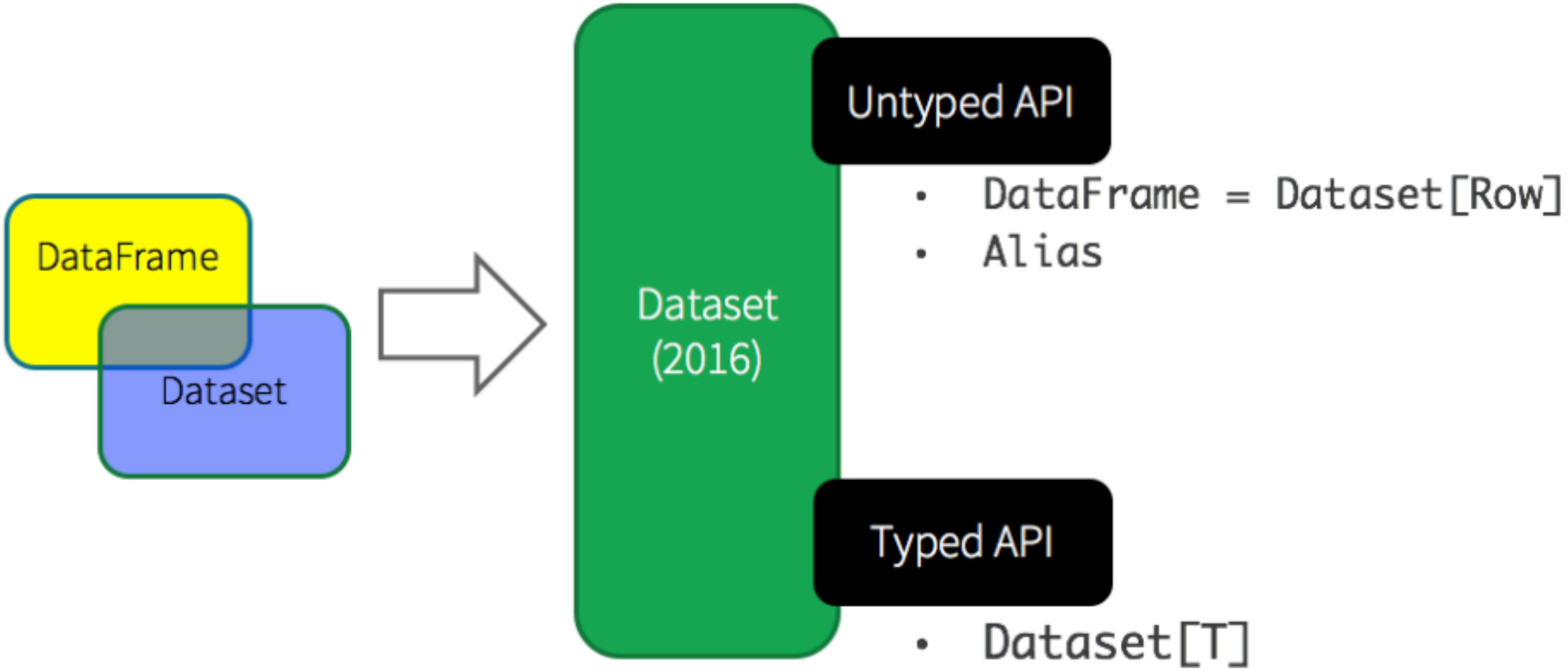
Datasets

Unified APIs

	SQL	DataFrame	Dataset
Syntax Error	Runtime	Compile time	Compile time
Analysis Error	Runtime	Runtime	Compile time

Datasets

Unified Apache Spark 2.0 API

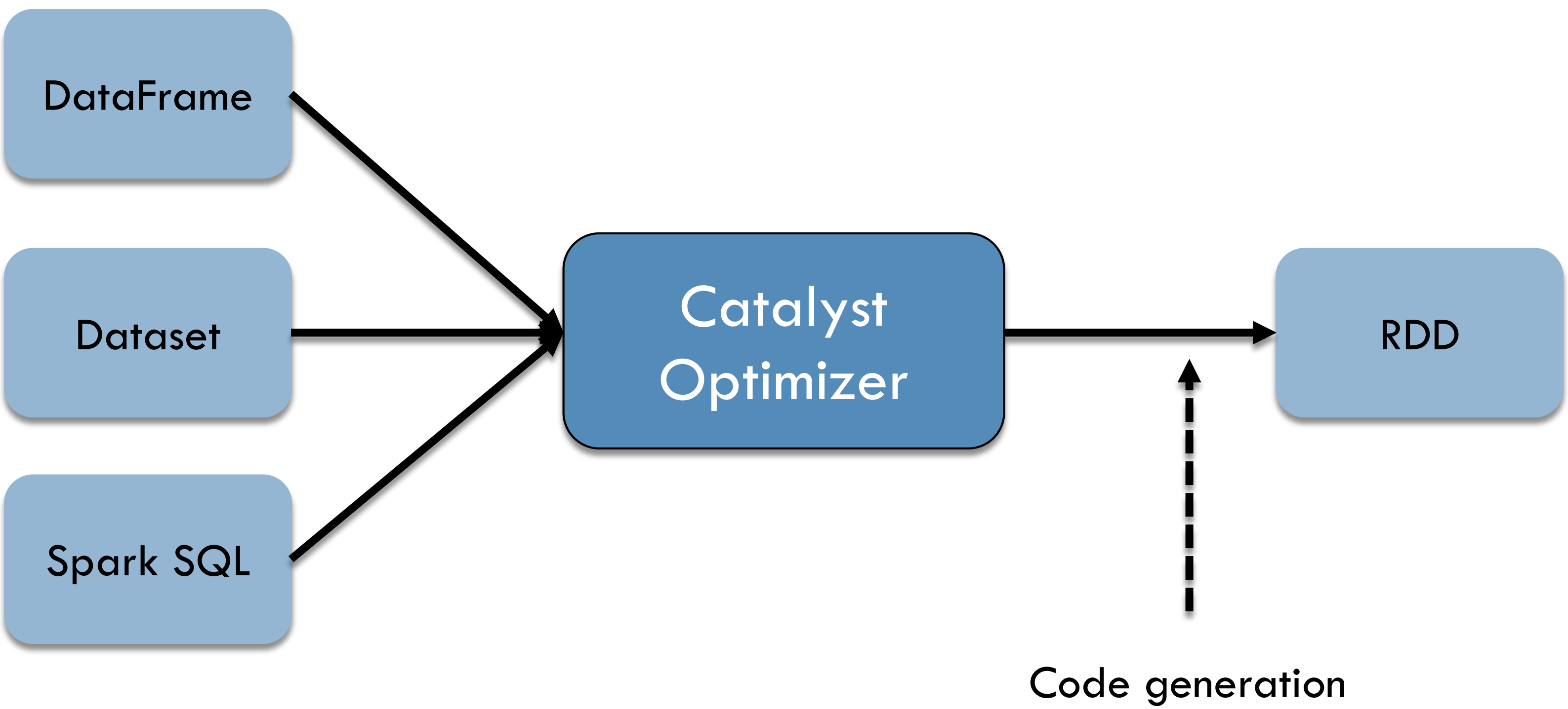


Spark SQL Catalyst

- Intelligent Optimization
  - Understand the semantics of operations and knowledge of data structure
  - Optimization types
    - Predicate pushdown
    - Column pruning
    - Generate JVM bytecode during physical plan step
    - Choosing the right kind of join
      - Broadcast join vs shuffle join
    - Reducing virtual function calls and object allocations

Spark SQL Catalyst

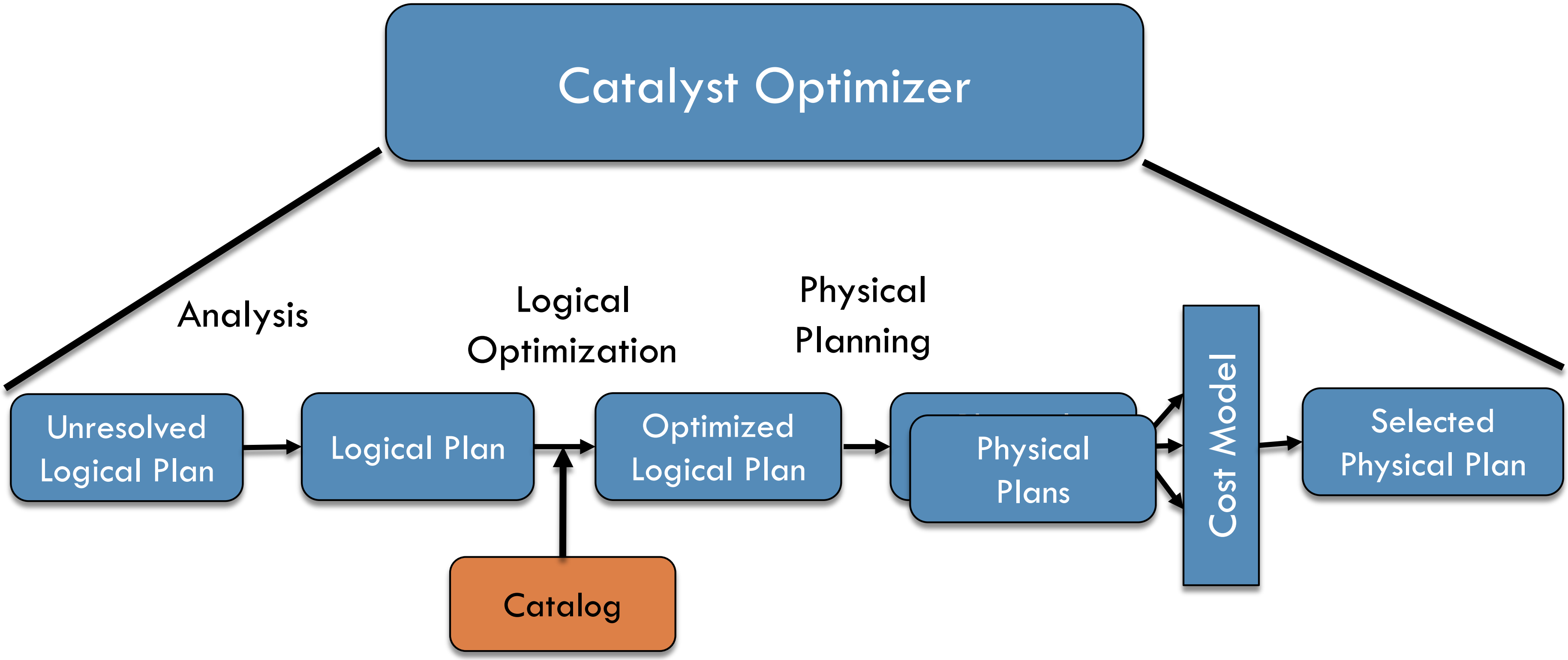
Catalyst Optimizer



Representing query plans as trees and applying optimization rules

Spark SQL Catalyst

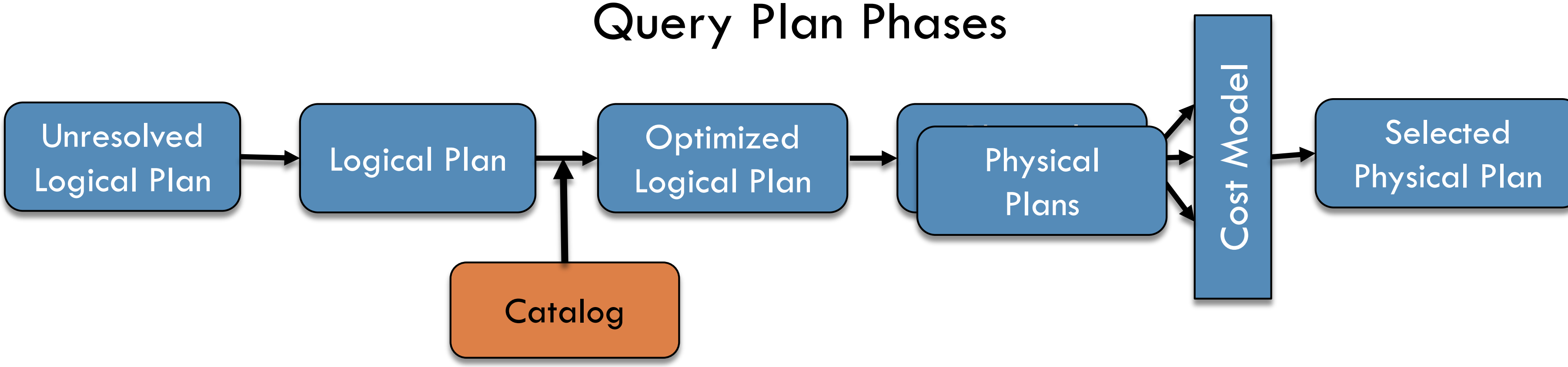
Catalyst Optimizer Components



Intelligent optimization by understanding operation semantics & data structure

Spark SQL Catalyst

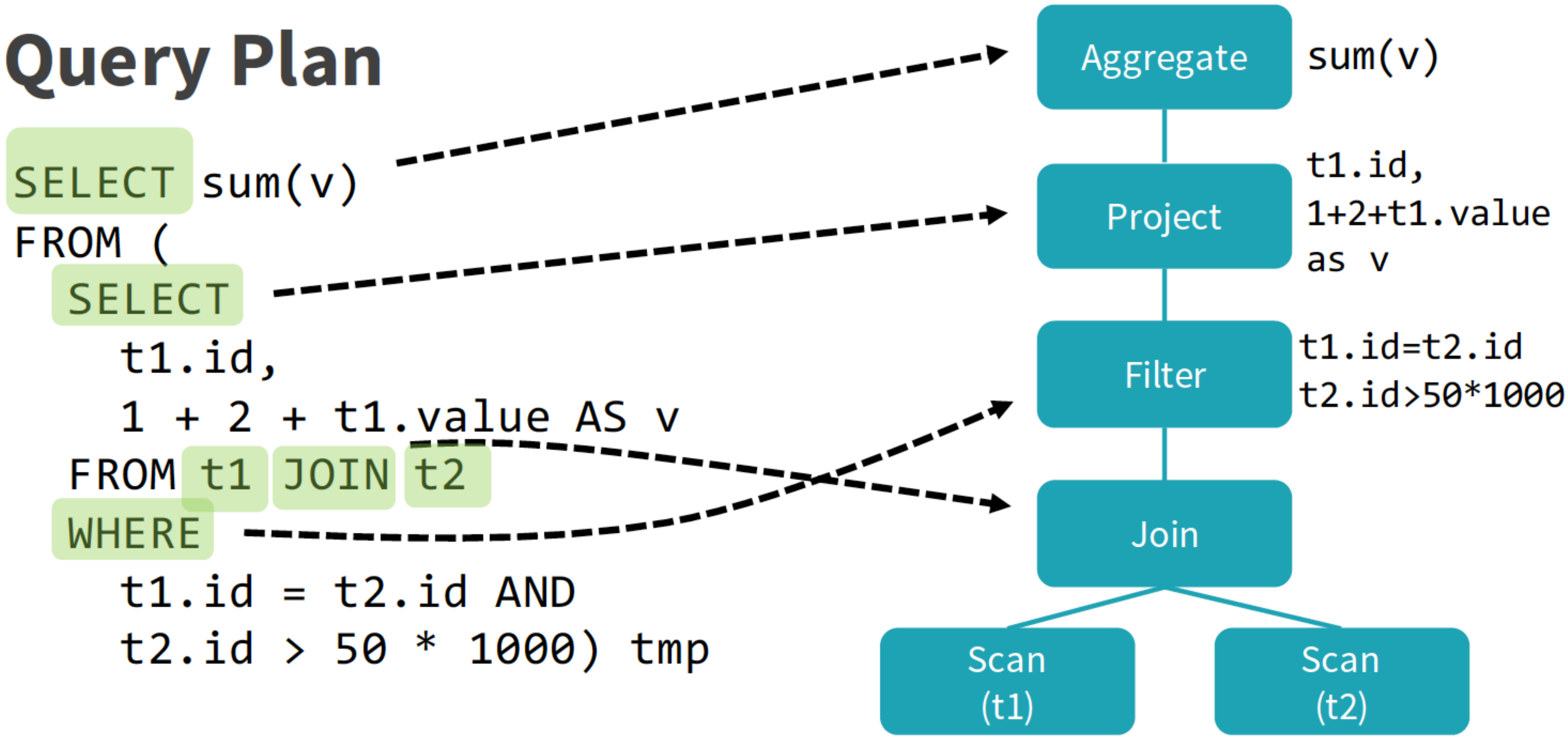
Query Plan Phases



- Analysis
  - Transform unresolved logical plan to resolved logical plan
  - Verify table, column and qualified names
- Logical optimization
  - Transform resolved logical plan to optimized logical plan
  - Re-arrange of steps i.e move filter operation before a join
- Physical plan
  - Transform a optimized logical plan to physical plan
  - Select optimal kind of join – broadcast join instead of shuffle join

Spark SQL Catalyst

Abstraction of User Query



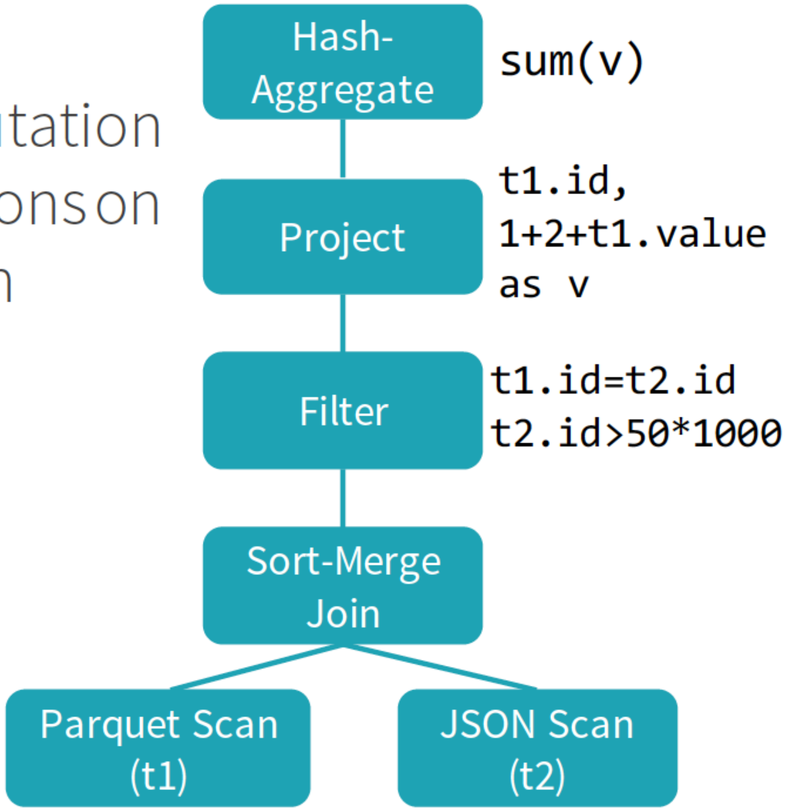
http://www.slideshare.net/databricks/deep-dive-into-catalyst-apache-spark-20s-optimizer

Spark SQL Catalyst

Physical Plan

- A Physical Plan describes computation on datasets with specific definitions on how to conduct the computation

- A Physical Plan is executable



Spark SQL Catalyst

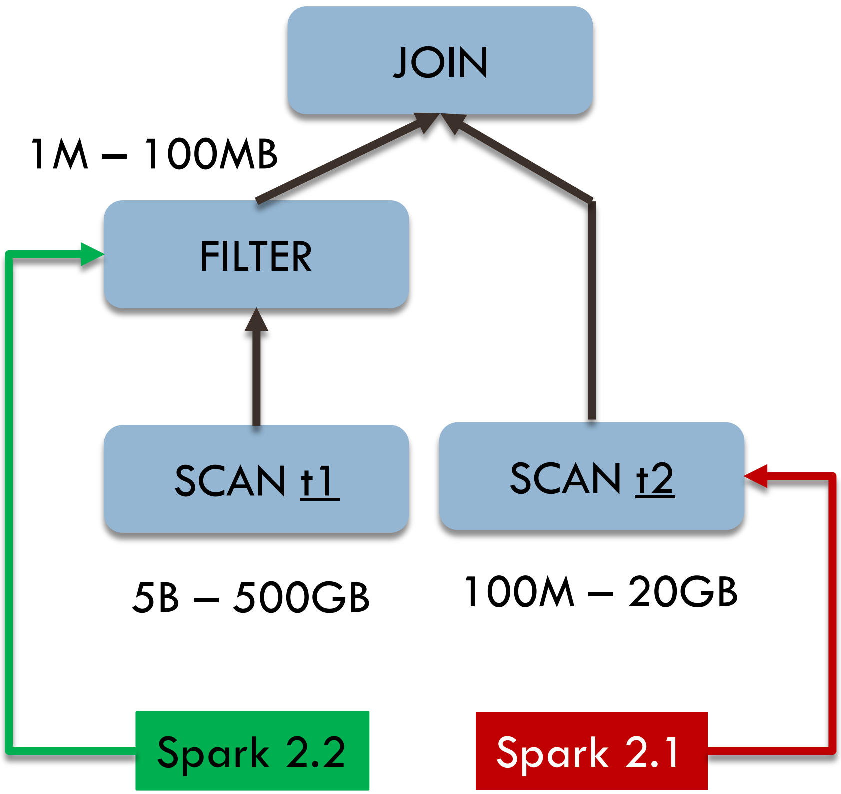
Catalyst Cost Based Optimizer – Spark 2.2

- Detailed column statistics
  - Cardinality
  - Number of distinct values
  - Max/min, average/man length
- Smart about choosing the right join type
  - Broadcast hash-join vs shuffled hash-join
  - Adjusting multi-way join order

Spark SQL Catalyst

Catalyst Cost Based Optimizer

Spark uses hash join by choosing the smaller table as the build side




```
select count(*)
from t1 join t2
on t1.id = t2.id
where t1.age > 30
```

Spark SQL Catalyst

Examine Execution Plan

Name	Description
explain	Print physical plan to console
explain(true)	Print logical and physical plan to console

```
val movies = sqlContext.read.json("/movies-json")

moviesDF.createOrReplaceTempView("movies")

val groupByYear = spark.sqlContext.sql("select year, count(*) from
movies group by year")

groupByYear.explain(true)
```

Spark SQL Catalyst

Examine Execution Plan

```
== Parsed Logical Plan ==
'Aggregate ['year], ['year, unresolvedalias('count(1), None)]
+- 'UnresolvedRelation `movies`

== Analyzed Logical Plan ==
year: bigint, count(1): bigint
Aggregate [year#2323L], [year#2323L, count(1) AS count(1)#2483L]
+- SubqueryAlias movies
   +- Relation[actor#2321,title#2322,year#2323L] json

== Optimized Logical Plan ==
Aggregate [year#2323L], [year#2323L, count(1) AS count(1)#2483L]
+- Project [year#2323L]
   +- Relation[actor#2321,title#2322,year#2323L] json

== Physical Plan ==
*HashAggregate(keys=[year#2323L], functions=[count(1)], output=[year#2323L,
count(1)#2483L])
+- Exchange hashpartitioning(year#2323L, 200)
   +- *HashAggregate(keys=[year#2323L], functions=[partial_count(1)],
output=[year#2323L, count#2485L])
   +- *Scan json [year#2323L] Format: JSON, InputPaths: dbfs/movies.json,
PartitionFilters: [], PushedFilters: [], ReadSchema: struct<year:bigint>
```

Project Tungsten

Project Tungsten

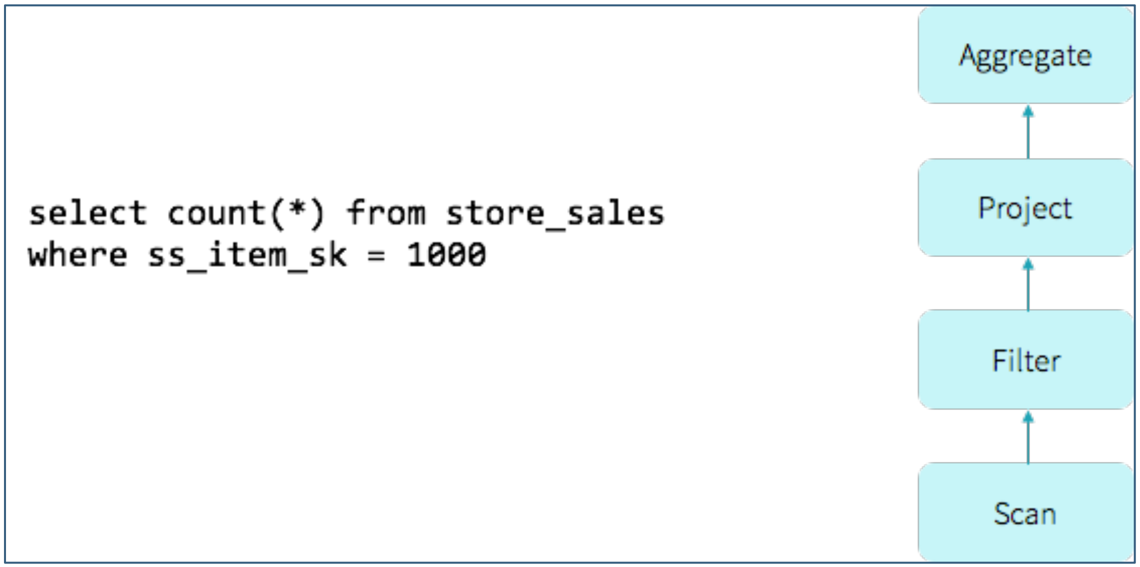
Pushing Spark performance closer to hardware limits

Phase I	Phase II
• Memory management • Code generation • Cache-aware algorithms	• Whole stage codegen • Vectorized processing

Improving the efficiency of memory & CPU

Project Tungsten

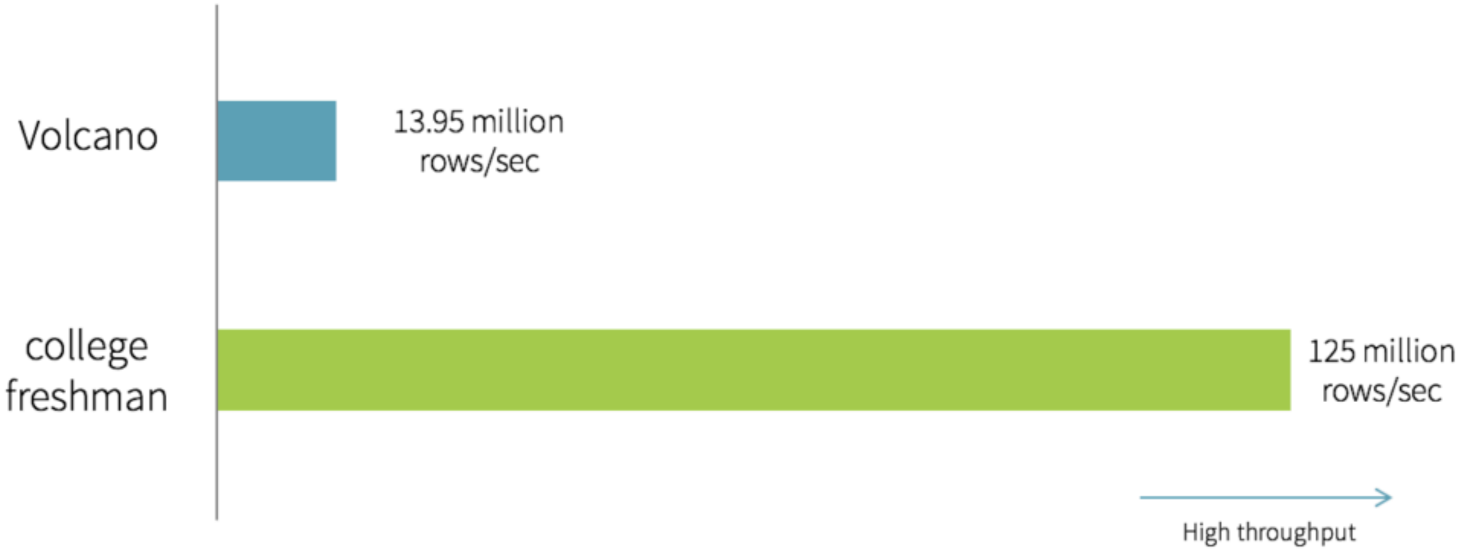
Volcano Iterator Model vs Hand-written Model




```
var count = 0
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

Project Tungsten

Volcano Iterator Model vs Hand-written Model



Volcano Iterator Model	Hand-written Model
• Too many virtual function call • Intermediate data in memory	• No virtual function call • Intermediate data in CPU registers • CPU – SIMD, pipelining, prefetching for loops

Project Tungsten

Vectorization

- Instead of processing one row at a time
- Batch multiple rows together in columnar format
- Each operator loops over data in a batch

Summary

RDD vs DataFrame & Datasets

RDD	DataFrame & Datasets
Low-level transformation & action	Rich semantic, high-level abstractions, DSL
Data is unstructured, i.e media	Higher degree of type-safety at compile time
Use functional programming constructs	Use high-level expressions – averages, sum, SQL queries
Don't care about imposing a schema	Simplification of APIs across Spark libraries
Forgo some optimization & performance benefits	Take advantage of Catalyst optimization & Tungsten's efficient code generation

When in doubt, use DataFrame or Datasets