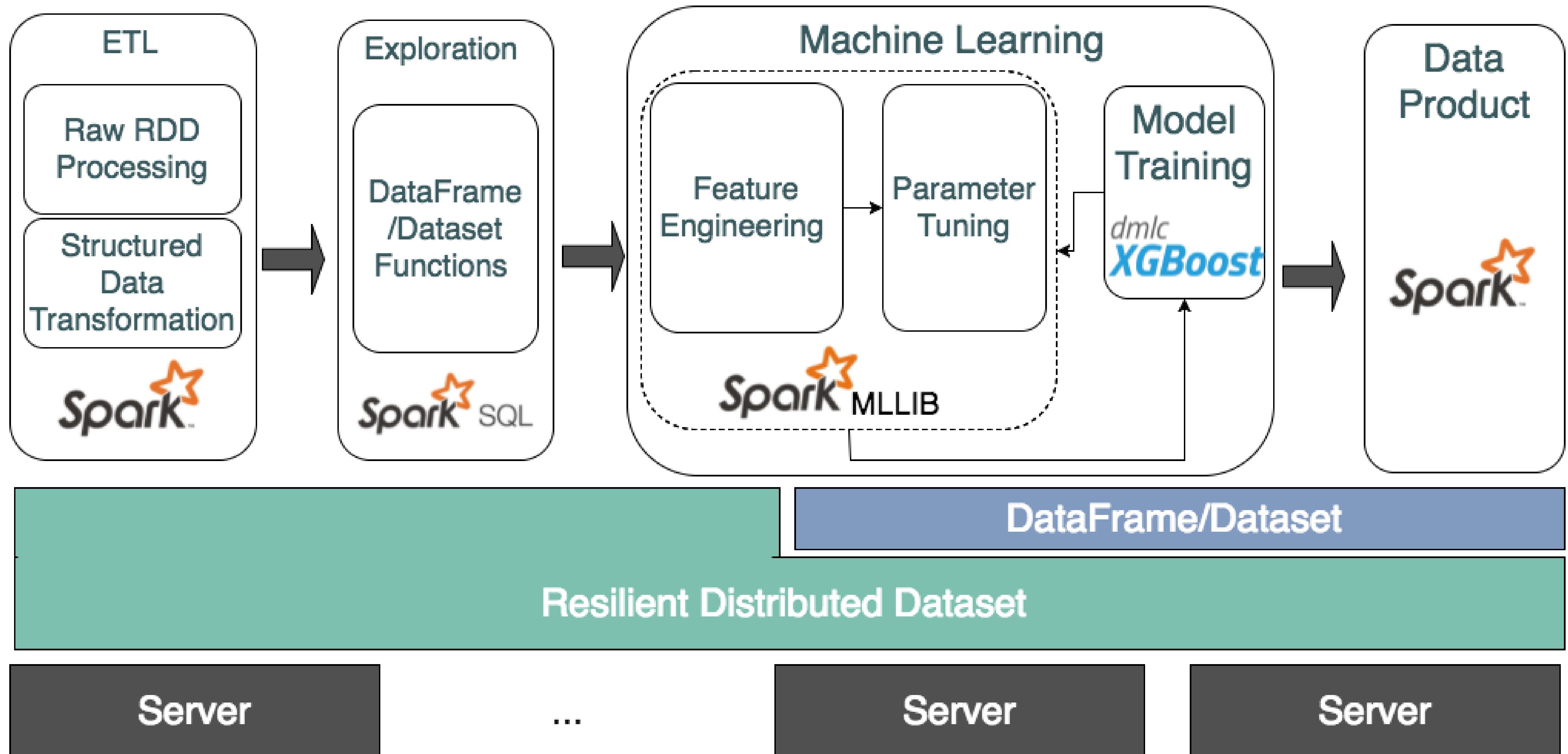


“Analytics using Apache Spark” (Lightening Fast Cluster Computing)



Spark SQL, DataFrames

Spark... Revisiting



Spark SQL... Structured Data Processing

- Spark SQL is a Spark module for structured data processing.
- Unlike the basic Spark RDD API, the interfaces provided by **Spark SQL provide Spark with more information about the structure of both the data and the computation** being performed.
- Internally, Spark SQL uses this extra information to **perform extra optimizations**.
- One use of Spark SQL is to **execute SQL queries (In parallel....faster)**
- Spark SQL can also be used **to read data from an existing Hive installation**.
- When running SQL from within another programming language **the results will be returned as a DataFrame**.
- You can also interact with the SQL interface using the command-line or over **JDBC/ODBC**.

Spark SQL... Data Frame

- **A DataFrame** is a dataset organized into named columns.
- It is **conceptually equivalent to a table in a relational database** or a data frame in R/Python, but with richer optimizations.
- DataFrames can be constructed from a wide array of sources such as:
 - structured data files, tables in Hive, external databases, existing RDDs.
- The DataFrame API is available in Scala, Java, **Python**, and R.
- In Scala and Java, a DataFrame is represented by a Dataset of Rows. In the Scala API, DataFrame is simply a type alias of Dataset[Row]. While, in Java API, users need to use Dataset<Row> to represent a DataFrame.
- Throughout this document, we will often refer to Scala/ Java Datasets of Rows as DataFrames.

Spark SQL Programming Interface

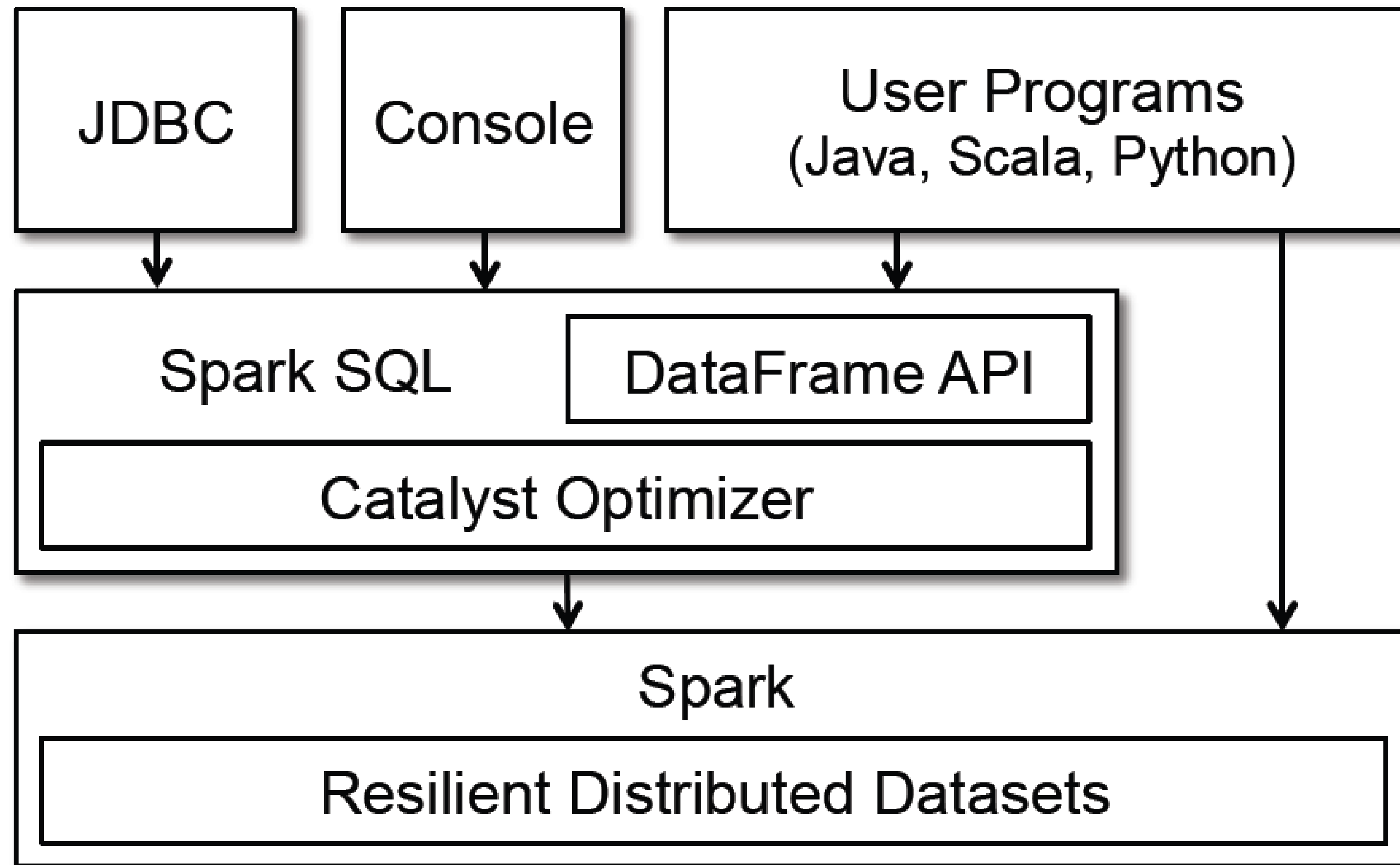


Figure 1: Interfaces to Spark SQL, and interaction with Spark.

RDDs vs DataFrames

- RDD is the building block of Spark. No matter which abstraction DataFrame or Dataset we use, internally final computation is done on RDDs.
- RDD is lazily evaluated immutable parallel collection of objects.
- RDD is simple and provides familiar OOPs style APIs with compile time safety.
- We can load any data from a source, convert them into RDD and store in memory to compute results.
- RDD can be easily cached if same set of data needs to recomputed.

Disadvantages

- Performance limitations - Being in-memory JVM objects, RDDs involve overhead of Garbage Collection and Serialization which are expensive when data grows.
- No schema/ type

RDDs vs DataFrames

- DataFrame is an abstraction which **gives a schema view of data**, which means it gives us a view of data as columns with column name and datatypes info.
- Like RDD, execution in DataFrame is also **lazy triggered**.
- **Offers huge performance improvement** over RDDs because of 2 powerful features it has:

a) Custom Memory management (aka Project Tungsten)

- Data is stored in **off-heap memory in binary format**.
- This saves a lot of memory space.
- No Garbage Collection overhead involved.
- By knowing the schema of data in advance and storing efficiently in binary format, expensive java Serialization is also avoided.

RDDs vs DataFrames

On-Heap vs. Off-Heap

Executor acts as a JVM process, and its memory management is based on the JVM. So JVM memory management includes two methods:

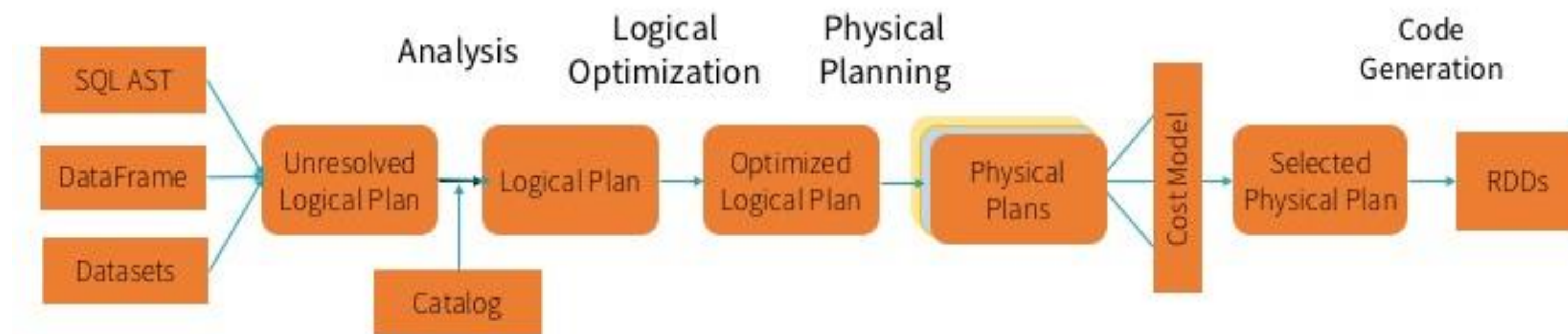
- **On-Heap memory management:** Objects are allocated on the JVM heap and bound by Garbage Collector (GC).
- **Off-Heap memory management:** Objects are allocated in memory outside the JVM by serialization, managed by the application, and are not bound by GC. This memory management method can avoid frequent GC, but the disadvantage is that you have to write the logic of memory allocation and memory release.
- In general, the objects' read and write speed is: **on-heap → off-heap → disk**

RDDs vs DataFrames

b) Optimized Execution Plans (aka Catalyst Optimizer)

- **Query plans are created** for execution using Spark catalyst optimizer.
- After an optimized execution/query plan is prepared going through some steps, **the final execution happens internally on RDDs** only but that's completely hidden from the users.

Using Catalyst in Spark SQL



Analysis: analyzing a logical plan to resolve references

Logical Optimization: logical plan optimization

Physical Planning: Physical planning

Code Generation : Compile parts of the query to Java bytecode

DataFrames - Example

2 ways to define: Expression BuilderStyle and SQL Style

```
df= spark.sqlContext.read.json("file:///usr/local/spark-2.0.0-bin-hadoop2.6/examples/src/main/resources/people.json")
df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]
```

```
df.filter("age > 21"); //SQL Style
res1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [age: bigint, name: string]
```

```
df.filter("age > 21").show //SQL Style
```

```
+---+---+
|age|name|
+---+---+
| 30|Andy|
+---+---+
```

```
df.filter(df.col("age").gt(21)).show; //Expression builder style
```

```
+---+---+
|age|name|
+---+---+
| 30|Andy|
+---+---+
```


DataFrames - Disadvantages

- **Lack of Type Safety** - it doesn't seem developer friendly. Referring attribute by String names means no compile time safety. Things can fail at runtime.

Ex: If we try using some columns not present in schema, we will get problem only at runtime, i.e we try accessing salary when only name and age are present in the schema will exception like below:

```
df.filter("salary > 10000").show
org.apache.spark.sql.AnalysisException: cannot resolve '`salary`' given input columns: [age, name];
  at org.apache.spark.sql.catalyst.analysis.package$AnalysisErrorAt.failAnalysis(package.scala:42)
  at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1$$anonfun$apply$2
  at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1$$anonfun$apply$2
```

- Also, APIs doesn't look programmatic and more of SQL kind.

DataFrames – Important points to remember

- DataFrame internally does final execution on RDD objects only but the difference is users do not write code to create the RDD collections and have no control as such over RDDs.
- RDDs are created in the execution plan as last stage after deciding and going through all the optimizations.
- RDD let us decide HOW we want to do, where as DataFrame lets us decide WHAT we want to do.
- All these optimizations have been possible because data is structured and Spark knows the schema of data in advance.
- It can apply all the powerful features like Tungsten custom memory off-heap binary storage, catalyst optimizer and encoders to get the performance which was not possible if users would have been directly working on RDD.

Creating DataFrames

DataFrames

- Like Spark SQL, the **DataFrames API** assumes that the data has a table-like structure.
- Formally, a DataFrame is a size-mutable, potentially heterogeneous tabular data structure with labeled axes (i.e., rows and columns).
- Just think of it as a table in a distributed database: a distributed collection of data organized into named, typed columns.

Transformations, Actions, Laziness

DataFrames are *lazy*.

Transformations contribute to the query plan, but they don't execute anything.

Actions cause the execution of the query.

Transformation examples

- filter
- select
- drop
- intersect
- join

Action examples

- count
- collect
- show
- head
- take

Write Less Code: Input & Output

Unified interface to reading/writing data in a variety of formats.

```
df = sqlContext.read\  
    .format("json") \  
    .option("samplingRatio", "0.1") \  
    .load("/Users/spark/data/stuff.json")  
  
df.write \  
    .format("parquet") \  
    .mode("append") \  
    .partitionBy("year") \  
    .saveAsTable("faster--stuff")
```



Write Less Code: High-level Operations

Solve common problems concisely using DataFrame functions:

- Selecting columns and filtering
- Joining different data sources
- Aggregation (count, sum, average, etc.)
- Plotting results (e.g., with Pandas)

Creating a DataFrame in Python



```
from pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)

sqlContext = SQLContext(sc)

df = sqlContext.read.parquet("/path/to/data.parquet")
df2 = sqlContext.read.json("/path/to/data.json")
```

Use DataFrames



Create a new DataFrame that contains only "young" users

```
young = users.filter(users["age"] < 21)
```

Alternatively, using a Pandas-like syntax

```
Young = users[users.age < 21]
```

Increment everybody's age by 1

```
young.select(young["name"], young["age"] + 1)
```

Count the number of young users by gender

```
young.groupBy("gender").count()
```

Join young users with another DataFrame, logs

```
young.join(log, logs["userId"] == users["userId"], "left_outer")
```

DataFrames and Spark SQL

```
young.registerTempTable("young")  
sqlContext.sql("SELECT count(*) FROM young")
```

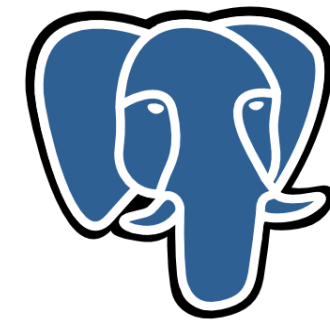
Data Sources supported by DataFrames

Built-in



JDBC

{ JSON }



PostgreSQL



External



and more ...

Schema Inference

What if your data file doesn't have a schema? (e.g., You're reading a CSV file or a plain text file.)

Two ways in which Scheme can be created:

1. You can create an RDD of a particular type and let Spark infer the schema from that type.
We'll see how to do that in a moment.
2. You can use the API to specify the schema programmatically.
(It's better to use a schema-oriented input source if you can, though.)

Schema Inference Example

Suppose you have a (text) file that looks like this:

```
Erin,Shannon,F,42
Norman,Lockwood,M,81
Miguel,Ruiz,M,64
Rosalita,Ramirez,F,14
Ally,Garcia,F,39
Claire,McBride,F,23
Abigail,Cottrell,F,75
José,Rivera,M,59
Ravi,Dasgupta,M,25
...
```

The file has no schema, but it's obvious there is one:

First name: String

Last name: String

Gender: String

Age: Integer

How Spark infers schema from the type of fields?

We can also force schema inference without creating our own People type, by using a fixed-length data structure (such as a tuple) and supplying the column names to the `toDF()` method.

Schema Inference – Python version

```
rdd = sc.textFile("people.csv")

def line_to_person(line):
    cells = line.split(",")
    return tuple(cells[0:3] + [int(cells[3])])

peopleRDD = rdd.map(line_to_person)
df = peopleRDD.toDF(("first_name", "last_name", "gender", "age"))
```

It takes the type of each data element and assigns the same datatype in DataFrame.

What can we do with DataFrames?

Columns

- Once you have a DataFrame, there are a number of operations you can perform.
- A DataFrame *column* is an abstraction. It **provides a common column-oriented view of the underlying data**, *regardless* of how the data is really organized.
- A place (a cell) for a data value to reside, within a row of data. This cell can have several states:
 - empty (null)
 - missing (not there at all)
 - contains a (typed) value (non-null)
 - A collection of those cells, from multiple rows
 - A syntactic construct we can use to specify or target a cell (or collections of cells) in a DataFrame query

Columns

Input Source Format	Data Frame Variable Name	Data									
JSON	dataFrame1	[{"first": "Amy", "last": "Bello", "age": 29 }, {"first": "Ravi", "last": "Agarwal", "age": 33 }, ...]									
CSV	dataFrame2	first, last, age Fred, Hoover, 91 Joaquin, Hernandez, 24 ...									
SQL Table	dataFrame3	<table><tr><th>first</th><th>last</th><th>age</th></tr><tr><td>Joe</td><td>Smith</td><td>42</td></tr><tr><td>Jill</td><td>Jones</td><td>33</td></tr></table>	first	last	age	Joe	Smith	42	Jill	Jones	33
first	last	age									
Joe	Smith	42									
Jill	Jones	33									

dataFrameX
Column: "first"

Columns

Assume we have a DataFrame, **df**, that reads a data source that has "first", "last", and "age" columns.

Python	Java	Scala	R
<code>df["first"]</code> <code>df.first[†]</code>	<code>df.col("first")</code>	<code>df("first")</code> <code>\$"first"[‡]</code>	<code>df\$first</code>

- [†]In Python, it's possible to access a DataFrame's columns either by attribute (`df.age`) or by indexing (`df['age']`). While the former is convenient for interactive data exploration, you should use the index form. It's future proof and won't break with column names that are also attributes on the DataFrame class.
- [‡]The \$ syntax can be ambiguous, if there are multiple DataFrames in the lineage.

show()

You can look at the first n elements in a DataFrame with the `show()` method. If not specified, n defaults to 20.

This method is an *action*: It:

- reads (or re-reads) the input source
- executes the RDD DAG across the cluster
- pulls the n elements back to the driver JVM
- displays those elements in a tabular form

cache()

- Spark can cache a DataFrame, using an in-memory columnar format, by calling `df.cache()` (which just calls `df.persist(MEMORY_ONLY)`).
- Spark will scan only those columns used by the DataFrame and will automatically tune compression to minimize memory usage and GC pressure.
- You can call the `unpersist()` method to remove the cached data from memory.

select()

select() is like a SQL SELECT, allowing you to limit the results to specific columns.

```
In[1]: df.select(df['first_name'], df['age'] > 49).show(5)
```



first_name	age	(age > 49)
Erin	42	false
Claire	23	false
Norman	81	true
Miguel	64	true
Rosalita	14	false

select() – SQL way

And, of course, you can also use SQL.

(This is the Python API, but you issue SQL the same way in Scala and Java.)

```
In[1]: df.registerTempTable("names")
```

```
In[2]: sqlContext.sql("SELECT first_name, age, age > 49 FROM names").show(5)
```

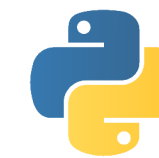


first_name	age	_c2
Erin	42	false
Claire	23	false
Norman	81	true
Miguel	64	true
Rosalita	14	false

filter()

The **filter()** method allows you to filter rows out of your results.

```
In[1]: df.filter(df['age'] > 49).select(df['first_name'],  
df['age']).show()
```



```
+-----+-----+  
|firstName|age|  
+-----+-----+  
|   Norman|  81|  
|   Miguel |  64|  
|   Abigail |  75|  
+-----+-----+
```

filter() – SQL version

```
In[1]: SQLContext.sql("SELECT first_name, age FROM names " + \
Age > 49"). Show()
```



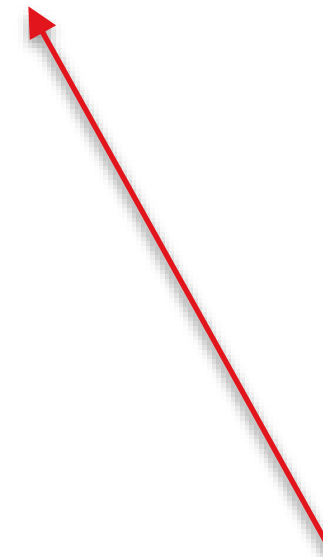
```
+---+---+
|firstName|age|
+---+---+
|   Norman|  81|
|   Miguel |  64|
|  Abigail |  75|
+---+---+
```

orderBy()

The **orderBy()** method allows you to sort the results.


```
In [1]: df.filter(df['age'] > 49).\n        select(df['first_name'], df['age']).\n        orderBy(df['age'].desc(), df['first_name']).show()
```

```
++  
|first_name|age|  
++  
|    Norman| 81|  
|   Abigail| 75|  
|    Miguel| 64|  
++
```



orderBy()

In SQL, it's pretty normal looking:



```
scala> sqlContext.SQL("SELECT first_name, age FROM names " +  
    |   "WHERE age > 49 ORDER BY age DESC, first_name").show()  
++  
|first_name|age|  
++  
|   Norman| 81|  
|  Abigail| 75|  
|   Miguel| 64|  
++
```

groupBy()

Often used with `count()`, `groupBy()` groups data items by a specific column value.

```
In [5]: df.groupBy("age").count().show()
```

```
+++
```

```
|age|count|
```

```
+++
```

```
| 39|      1|
```

```
| 42|      2|
```

```
| 64|      1|
```

```
| 75|      1|
```

```
| 81|      1|
```

```
| 14|      1|
```

```
| 23|      2|
```

```
+++
```



This is Python. Scala and Java are similar.

as() or alias()

`as()` or `alias()` allows you to rename a column. It's especially useful with generated columns.


```
In [7]: df.select(df['first_name'],\
                  df['age'],\
                  (df['age'] < 30).alias('young')).show(5)
```



```
+++
|first_name|age|young|
+++
|      Erin|  42|false|
|    Claire|  23|  true|
|   Norman|  81|false|
|   Miguel|  64|false|
|  Rosalita|  14|  true|
+++
```

Note: In Python, you *must* use `alias`, because `as` is a keyword.

And, of course, SQL:



```
scala> sqlContext.sql("SELECT firstName, age, age < 30 AS young " +  
  |                      "FROM names")  
+++++  
| first_name|age| young|  
+++++  
|          |    |      |  
|      Erin| 42|false|  
|    Claire| 23| true|  
|   Norman| 81|false|  
|   Miguel| 64|false|  
|   Rosalita|14| true|  
+++++
```

Other Useful Transformations

Method	Description
<code>limit(n)</code>	Limit the results to <i>n</i> rows. <code>limit()</code> is not an action, like <code>show()</code> or the RDD <code>take()</code> method. It returns another <code>DataFrame</code> .
<code>distinct()</code>	Returns a new <code>DataFrame</code> containing only the unique rows from the current <code>DataFrame</code>
<code>drop(column)</code>	Returns a new <code>DataFrame</code> with a column dropped. <i>column</i> is a name or a <code>Column</code> object.
<code>intersect(dataframe)</code>	Intersect one <code>DataFrame</code> with another.
<code>join(dataframe)</code>	Join one <code>DataFrame</code> with another, like a SQL join. We'll discuss this one more in a minute.

Writing DataFrames

- You can write DataFrames out, as well. When doing ETL, this is a very common requirement.
- In most cases, if you can read a data format, you can write that data format, as well.
- If you're writing to a text file format (e.g., JSON), you'll typically get multiple output files.

```
In [20]: df.write.format("json").save("/path/to/directory")  
In [21]: df.write.format("parquet").save("/path/to/directory")
```



Writing DataFrames: Save modes

Save operations can optionally take a **SaveMode** that specifies how to handle existing data if present.

Scala/ Java	Python	Meaning
SaveMode.ErrorIfExists (default)	"error"	If output data or table already exists, an exception is expected to be thrown.
SaveMode.Append	"append"	If output data or table already exists, append contents of the DataFrame to existing data.
SaveMode.Overwrite	"overwrite"	If output data or table already exists, replace existing data with contents of DataFrame.
SaveMode.Ignore	"ignore"	If output data or table already exists, do not write DataFrame at all.



Warning: These save modes do not utilize any locking and are not atomic.

- Thus, it is not safe to have multiple writers attempting to write to the same location.
- When performing a overwrite, the data will be deleted before writing out the new data.

Concluding: SQL and RDDs

Recall: **Spark SQL operations generally return DataFrames**, i.e, you can freely mix DataFrames and SQL.

Because SQL queries return DataFrames, and DataFrames are built on RDDs, you can use normal RDD operations on the results of a SQL query.

However, as with any DataFrame, it's best to stick with DataFrame operations.

Thank You