IEMS5730/ IERG4330/ ESTR4316

Spring 2022

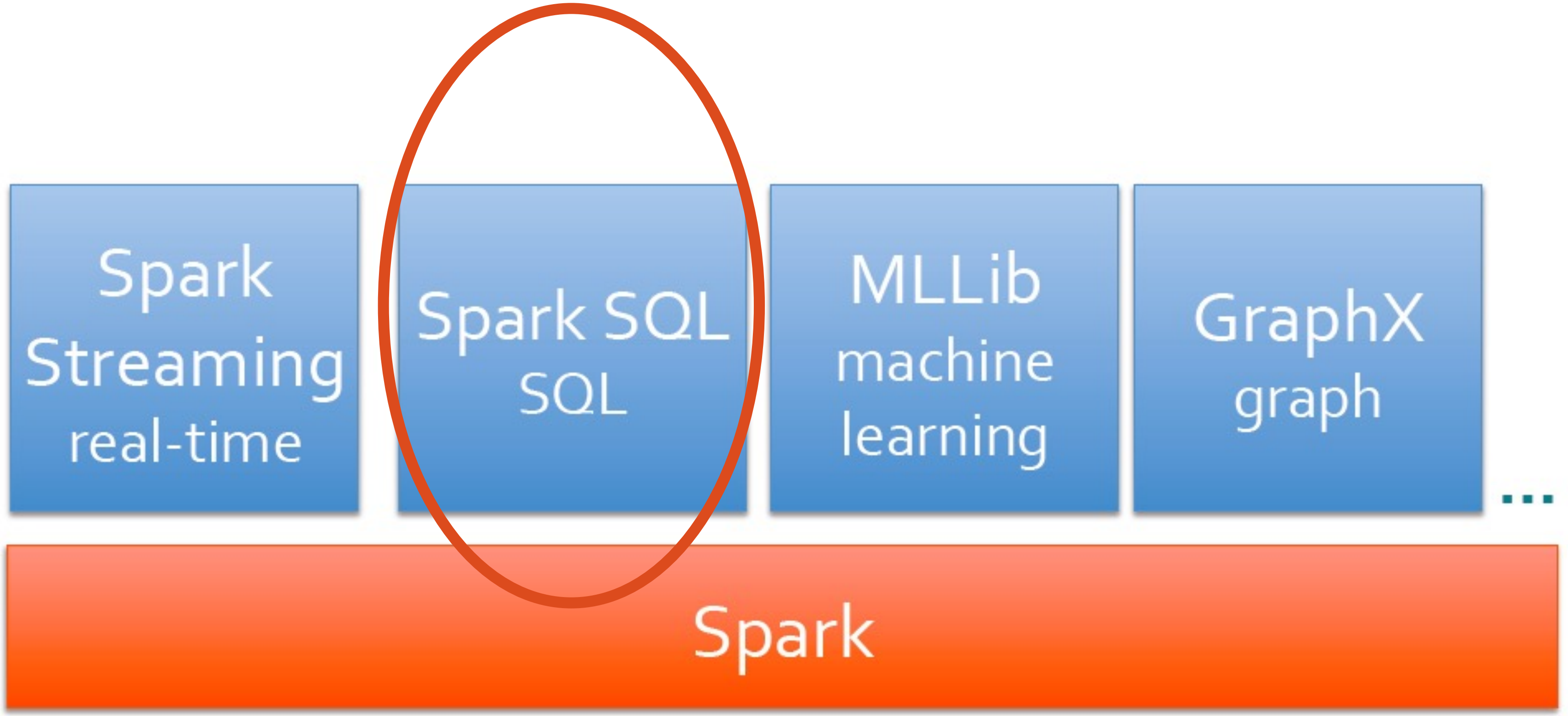
Spark SQL

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

- These slides are adapted from the following sources:
  - Matei Zaharia, “Spark 2.0,” Spark Summit East Keynote, Feb 2016.
  - Reynold Xin, “The Future of Real-Time in Spark,” Spark Summit East Keynote, Feb 2016.
  - Michael Armburst, “Structuring Spark: SQL, DataFrames, DataSets, and Streaming,” Spark Summit East Keynote, Feb 2016.
  - Ankur Dave, “GraphFrames: Graph Queries in Spark SQL,” Spark Summit East, Feb 2016.
  - Michael Armburst, “Spark DataFrames: Simple and Fast Analytics on Structured Data,” Spark Summit Amsterdam, Oct 2015.
  - Michael Armburst et al, “Spark SQL: Relational Data Processing in Spark,” SIGMOD 2015.
  - Michael Armburst, “Spark SQL Deep Dive,” Melbourne Spark Meetup, June 2015.
  - Reynold Xin, “Spark,” Stanford CS347 Guest Lecture, May 2015.
  - Joseph K. Bradley, “Apache Spark MLlib’s past trajectory and new directions,” Spark Summit Jun 2017.
  - Joseph K. Bradley, “Distributed ML in Apache Spark,” NYC Spark MeetUp, June 2016.
  - Ankur Dave, “GraphFrames: Graph Queries in Apache Spark SQL,” Spark Summit, June 2016.
  - Joseph K. Bradley, “GraphFrames: DataFrame-based graphs for Apache Spark,” NYC Spark MeetUp, April 2016.
  - Joseph K. Bradley, “Practical Machine Learning Pipelines with MLlib,” Spark Summit East, March 2015.
  - Joseph K. Bradley, “Spark DataFrames and ML Pipelines,” MLconf Seattle, May 2015.
  - Ameet Talwalkar, “MLlib: Spark’s Machine Learning Library,” AMPCamps 5, Nov. 2014.
  - Shivaram Venkataraman, Zongheng Yang, “SparkR: Enabling Interactive Data Science at Scale,” AMPCamps 5, Nov. 2014.
  - Tathagata Das, “Spark Streaming: Large-scale near-real-time stream processing,” O’Reilly Strata Conference, 2013.
  - Joseph Gonzalez et al, “GraphX: Graph Analytics on Spark,” AMPCAMP 3, 2013.
  - Jules Damji, “Jumpstart on Apache Spark 2.X with Databricks,” Spark Sat. Meetup Workshop, Jul 2017.
  - Sameer Agarwal, “What’s new in Apache Spark 2.3,” Spark+AI Summit, June 2018.
  - Reynold Xin, Spark+AI Summit Europe, 2018.
  - Hyukjin Kwon of Hortonworks, “What’s New in Spark 2.3 and Spark 2.4,” Oct 2018.
  - Matel Zaharia, “MLflow: Accelerating the End-to-End ML Lifecycle,” Nov. 2018.
  - Jules Damji, “MLflow: Platform for Complete Machine Learning Lifecycle,” PyData, Jan 2019.
-

Major Modules in Spark



Before SQL support was available from Spark

- The Spark Core Engine does not understand the structure of the data in RDDs or the semantics of user functions → limited optimization.
- However, most data is structured, e.g. JSON, CSV, Avro, Parquet, Hive, etc

=> Programming/ Operations via the RDD API inevitably ends up with a lot of tuples (_1, _2, …)

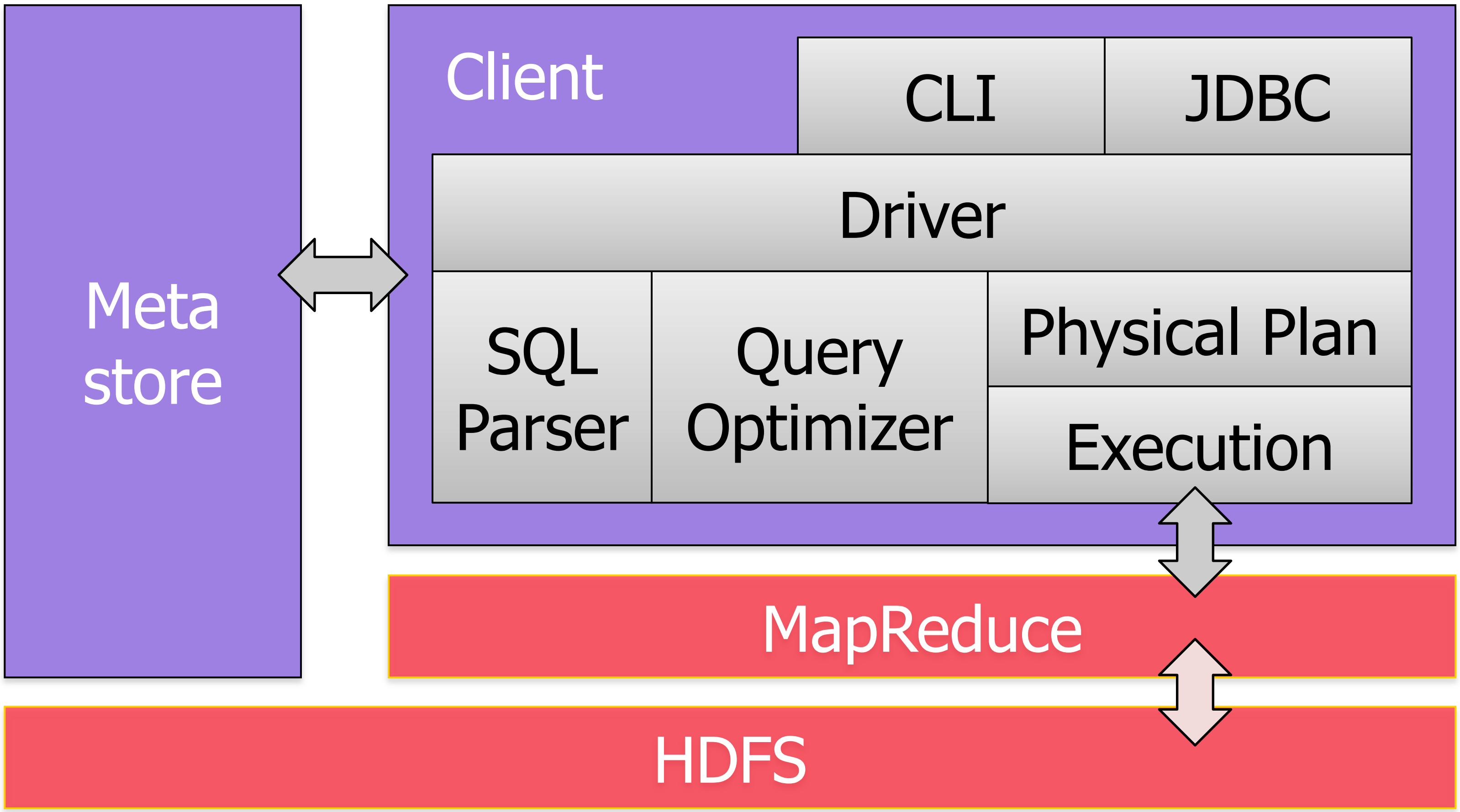
- Functional Transformations, e.g. Map/Reduce are still not as Intuitive as SQL for a lot of Experienced System/Data Analysts.

SQL support in Spark - Take 1: The Shark Story

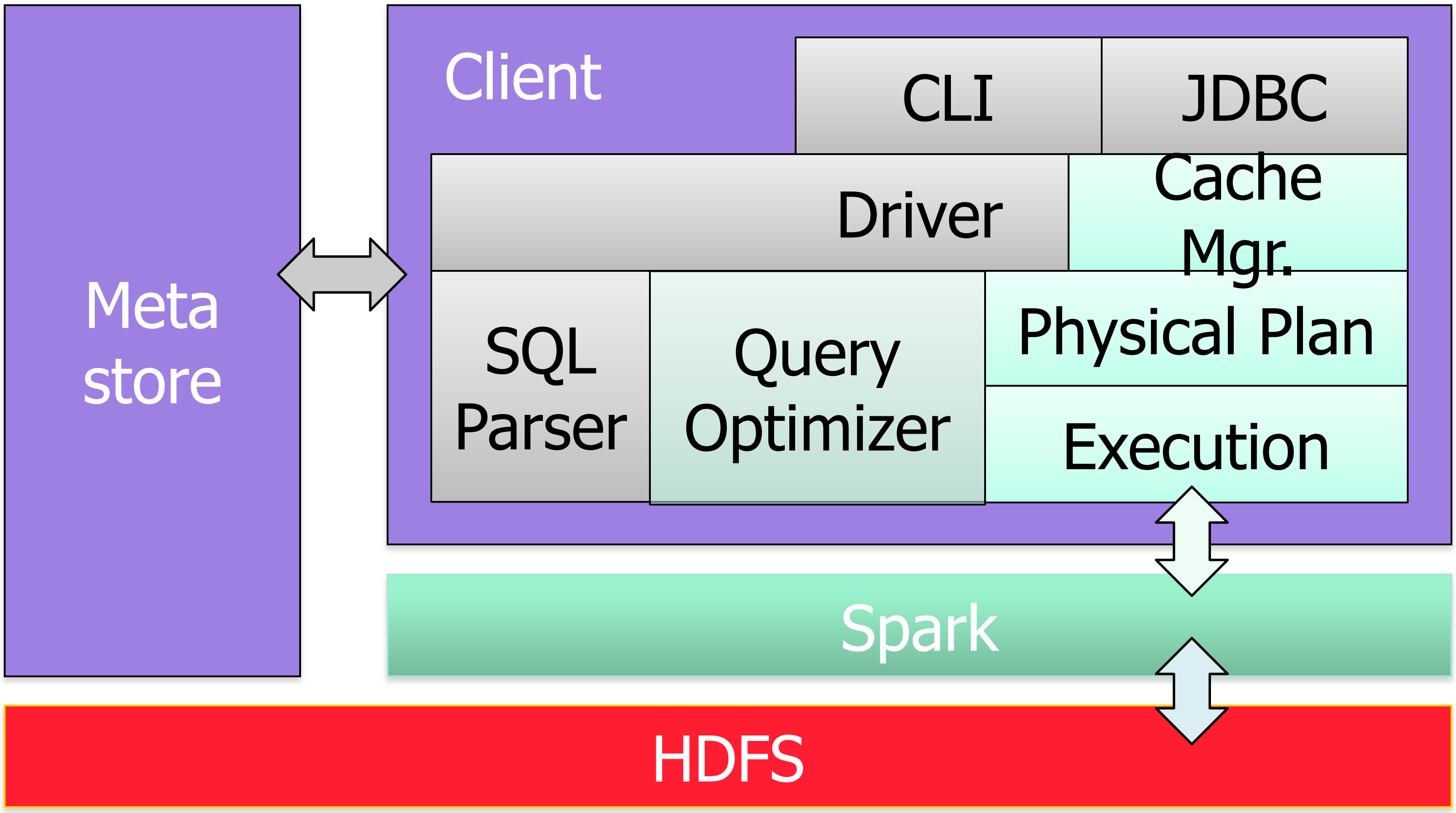
- Hive is great, but Hadoop’s execution engine makes even the smallest queries take minutes
- Scala is good for programmers, but many data users only know SQL
- **Initial Approach: Make Hive to run on Spark**

SHARK = Hive on Spark

Original Hive Architecture



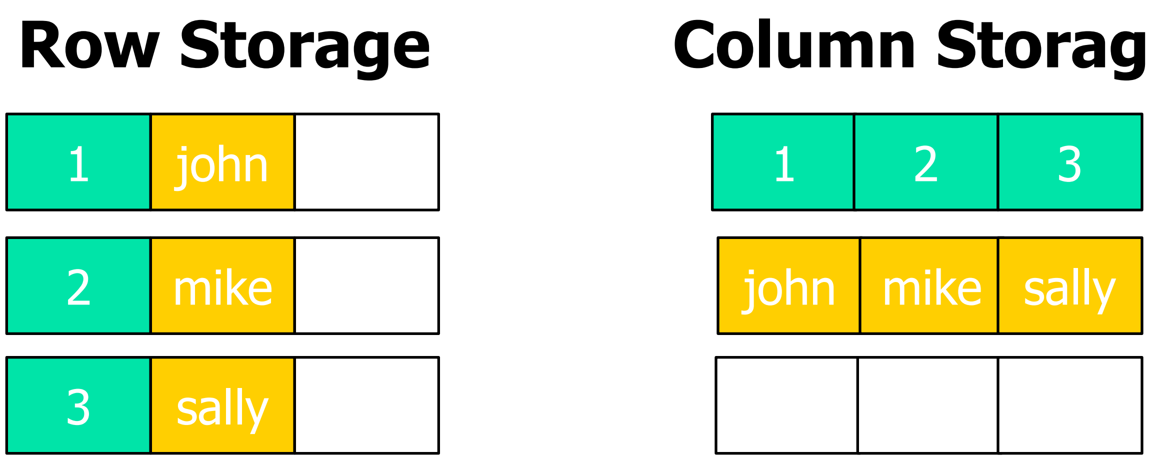
Shark Architecture



[Engle et al, SIGMOD 2012]

Efficient In-Memory Storage

- Simply caching Hive records as Java objects is inefficient due to high per-object overhead
- Instead, Shark employs column-oriented storage using **arrays of primitive types**



Efficient In-Memory Storage

- Simply caching Hive records as Java objects is inefficient due to high per-object overhead
- Instead, Shark employs column-oriented storage using **arrays of primitive types**

Row Storage **Column Storage**

Benefit: similarly compact size to serialized data,
but >5x faster to access

But Shark was short-lived (2011-2014)

Limitations of SHARK

- Can only be used to query external data in Hive catalog → limited data sources
- Can only be invoked via SQL string from Spark
  → error prone
- Hive optimizer tailored for MapReduce
  → difficult to extend

- As a result, BDAS Project decided to switch to Spark SQL and stopped development of Shark in 2014
  - The Apache Hive community still runs a Hive-over-Spark effort, as well as the Stinger/ Stinger.Next efforts to make Hive/HiveQL to be SQL compatible and low-latency

Take 2: Spark SQL Overview

- Part of the core distribution since Spark 1.0 (April 2014)
  - Optionally alongside or replacing existing Hive deployments
- Run SQL/ HiveQL queries including UDFs, UDAFs and SerDes, e.g.

```
SELECT COUNT(*)
FROM hiveTable
WHERE hive_udf(data)
```

- Connect existing Business Intelligence (BI) tools to Spark through JDBC
- Bindings in Python, Scala and Java

The Approach of Spark SQL

- Introduce a Tightly Integrated way to work with a new abstraction of Structured Data called SchemaRDD, which is a Distributed Collection of Rows (i.e. a Table) with Named Columns
  - SchemaRDD was renamed to DataFrame in Spark 1.3

- Support the Transformation of RDDs using SQL: In particular, DataFrames (aka SchemaRDDs) is an abstraction which supports:
  - Selecting, Filtering, Aggregating and Plotting Structured data (cf. R or Python-based Pandas)
- Evaluated lazily → unmaterialized *logical* plan

- Data source integration Support for: Hive, Parquet, JSON and …

Relationship between Spark SQL and Shark

- Shark modified the Hive backend to run over Spark but had two challenges:
  - Limited integration with Spark programs
  - Hive Optimizer not designed for Spark

- Spark SQL reuses some parts of Shark by

Borrowing:
  - Hive Data Loading
  - In-memory Column-store

while Adding:
  - RDD-aware Optimizer
  - Richer Language Interfaces

What is an RDD ?

- Dependencies
- Partitions (with optional locality information)
- Compute Function: `Partition=>Iterator[T]`

Opaque Computation

What is an RDD ?

- Dependencies
- Partitions (with optional locality information)
- Compute Function: `Partition=>Iterator[T]`

Opaque Data

Why Structure ?

- What do we mean by “Structure” [verb] ?:
  - Construct or Arrange according to a plan ; Give a pattern or organization to.
- By definition, structure will LIMIT what can be expressed.
- In practice, it is still possible to accommodate a vast majority of computations

BUT

- By Limiting the space of what can be expressed ENABLES Optimization

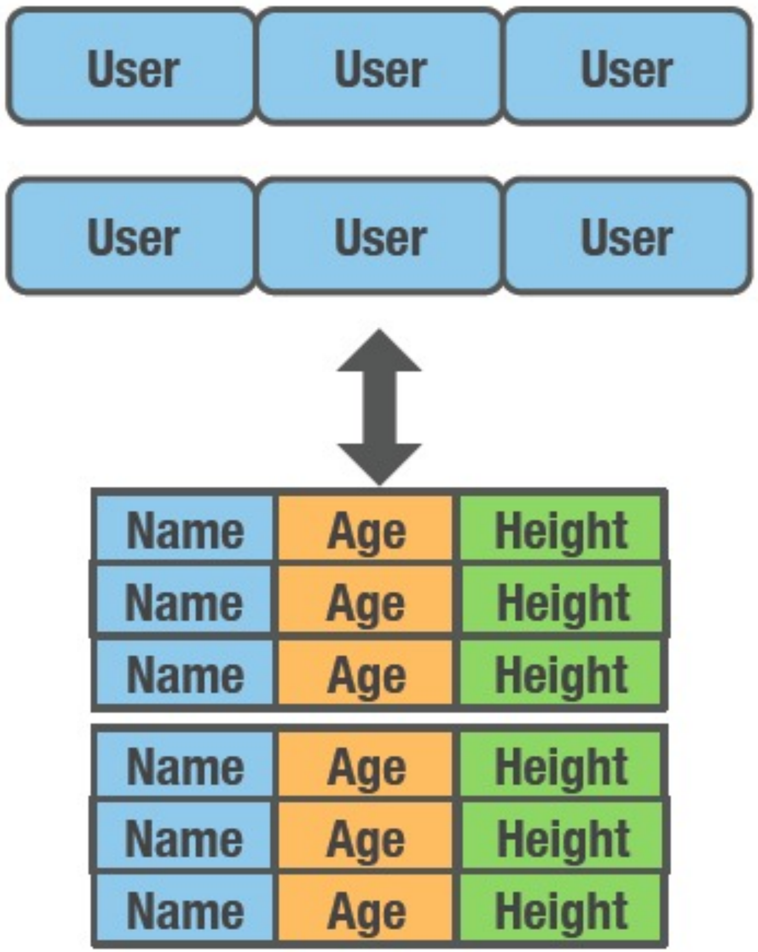
Adding Schema to RDDs

Spark + RDDs

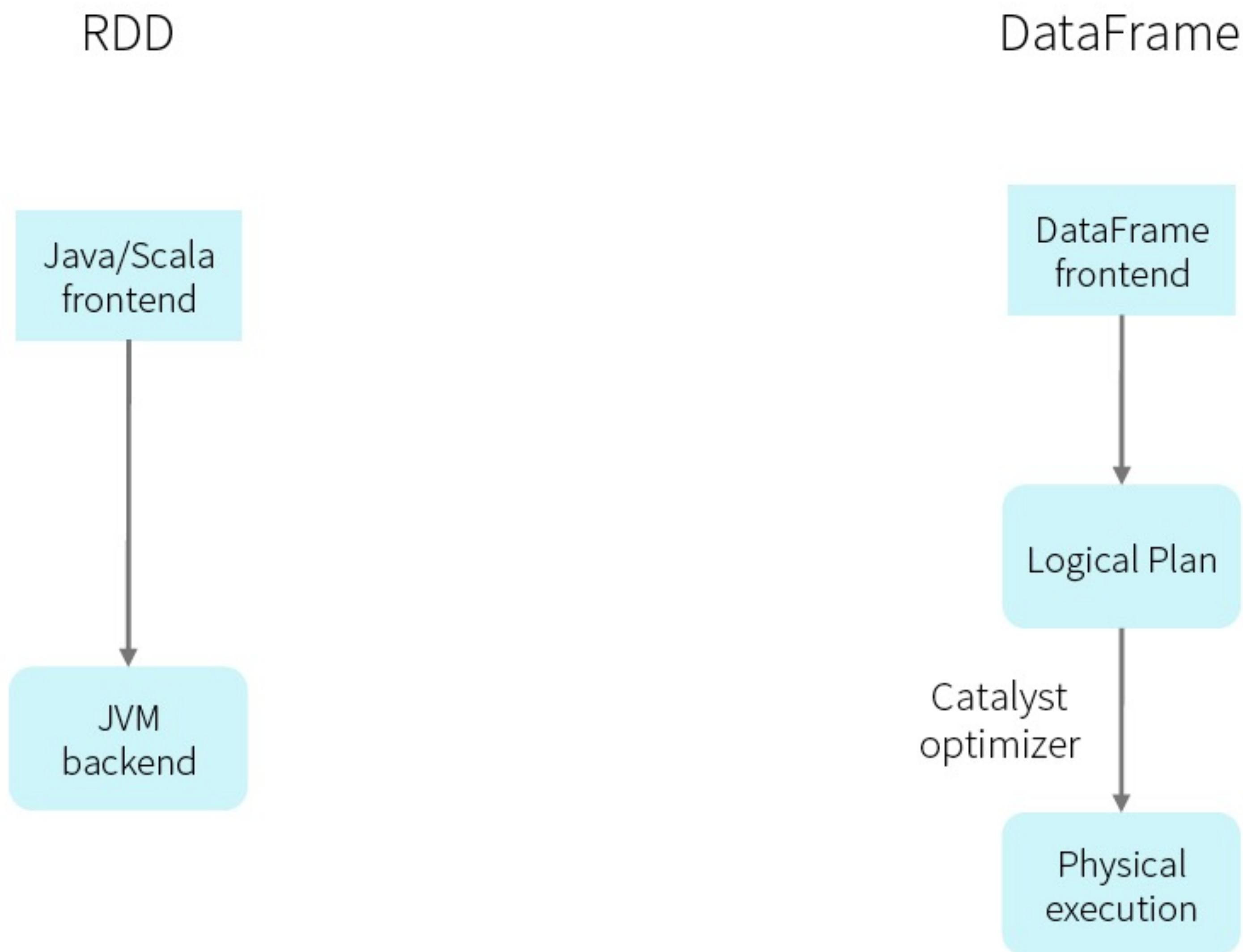
- Functional transformations on Partitioned Collections of *Opaque Objects*

SQL + DataFrames (aka SchemaRDDs)

- Declarative transformations on Partitioned Collections of *Tuples*



Comparing the Approaches of RDD vs. DataFrame



Data Model for DataFrame

- Nested data model
- Supports both primitive SQL types (boolean, integer, double, decimal, string, data, timestamp) and complex types (structs, arrays, maps, and unions); also user defined types.
- First class support for complex data types

DataFrame Operations

- Relational operations (select, where, join, groupBy) via a DSL
- Operators take *expression* objects
- Operators build up an abstract syntax tree (AST), which is then optimized by *Catalyst*.

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

- Alternatively, register as temp SQL table and perform traditional SQL query strings

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

Programming Interface for Spark SQL

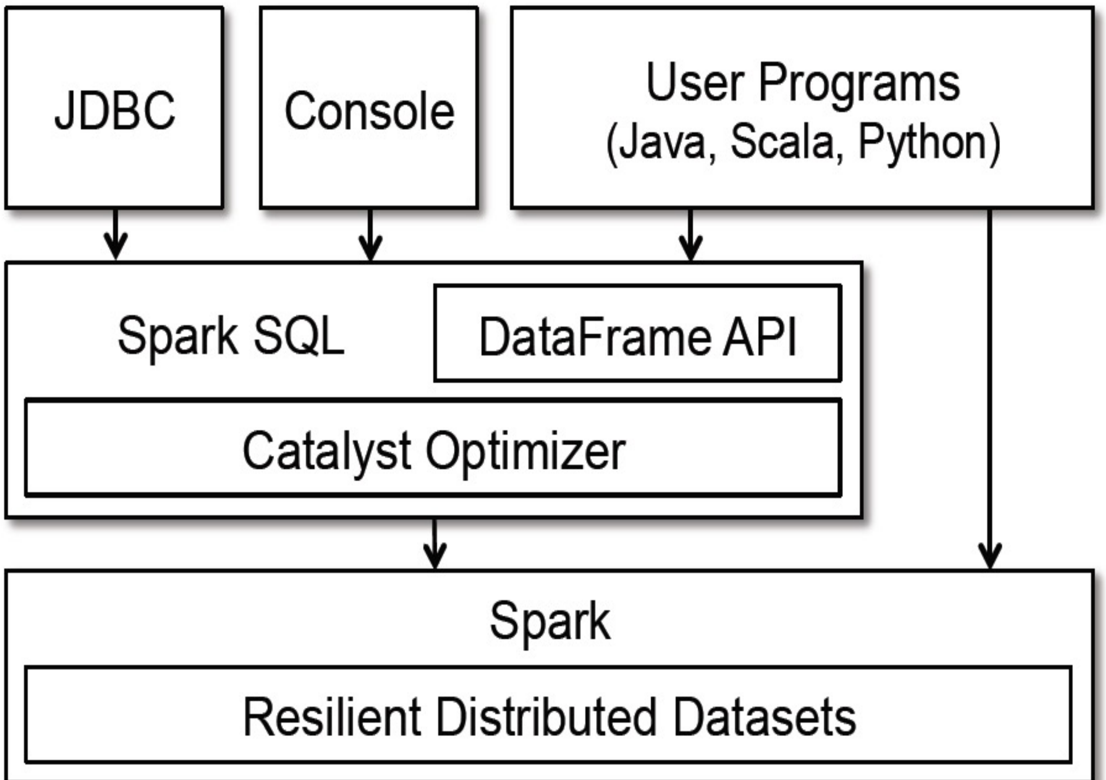
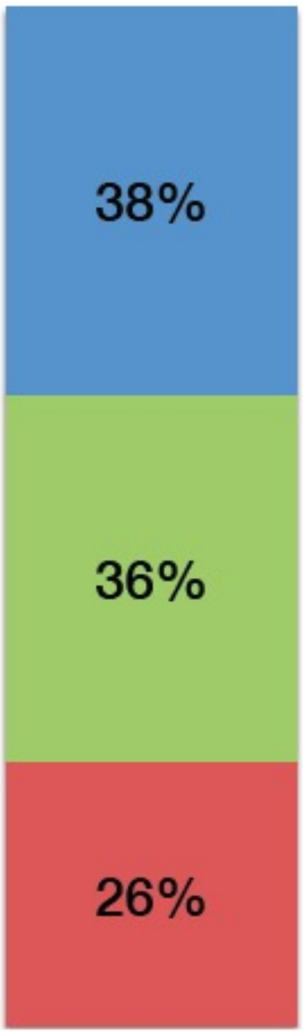


Figure 1: Interfaces to Spark SQL, and interaction with Spark.

Spark SQL Components



- Catalyst Optimizer
  - Relational algebra + expressions
  - Query optimization
- Spark SQL Core
  - Execution of queries as RDDs
  - Reading in Parquet, JSON …
- Hive Support
  - HQL, MetaStore, SerDes, UDFs

Getting Started: Spark SQL

- `SQLContext/ HiveContext`
  - Entry point for all SQL functionality
  - Wraps/Extend existing Spark Context

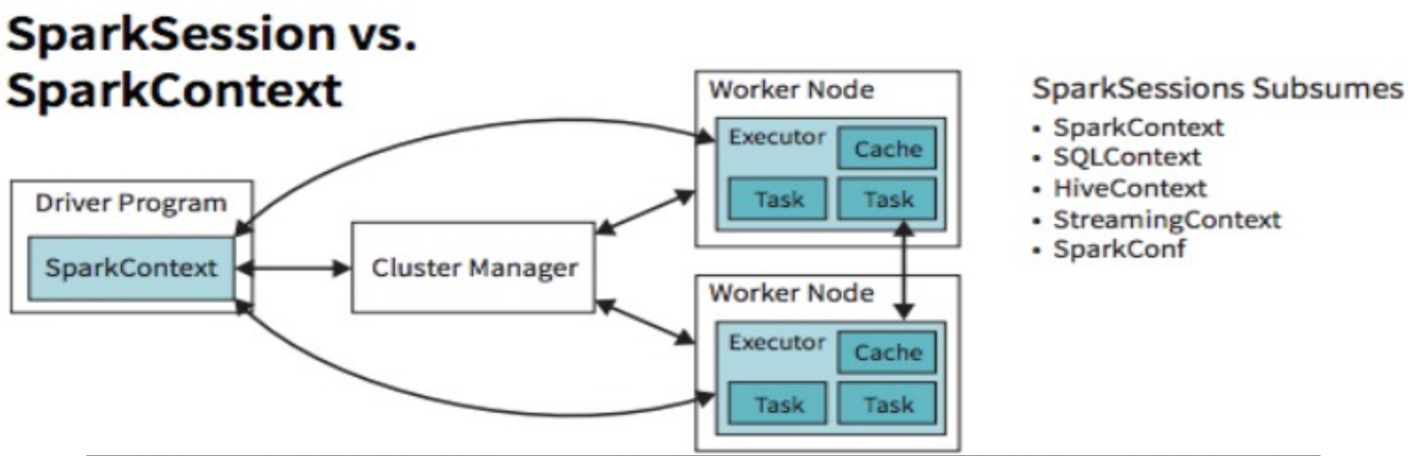
```
from pyspark.sql import SQLContext
sqlCtx = SQLContext(sc)
```

OR

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

SparkContext subsumed by SparkSession since Spark v2.0 !

- Starting v2.0, SparkSession becomes the unified entry point, i.e. a Conduit, to Spark
  - Create Datasets/ DataFrames
  - Read/Write Data,
  - Access services of all Spark modules like SparkSQL, Streaming, …
  - Work with metadata
  - Set/Get Spark Configuration ; Driver uses for Cluster Resource Management




```scala
val warehouseLocation = "file:${system:user.dir}/spark-warehouse"

val spark = SparkSession
  .builder()
  .appName("SparkSessionZipsExample")
  .config("spark.sql.warehouse.dir", warehouseLocation)
  .enableHiveSupport()
  .getOrCreate()
```

Sample Input Data

- A text file filled with people's names and ages:

```
Michael, 30
Andy, 31
…
```

RDDs as Relations (Scala)

```scala
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._

// Define the schema using a case class.
case class Person(name: String, age: Int)
// Create an RDD of Person objects and register it as a table.
val people =
  sc.textFile("examples/src/main/resources/people.txt")
    .map(_.split(","))
    .map(p => Person(p(0), p(1).trim.toInt))

people.registerAsTable("people")
```

RDDs as Relations (Python)

```
# Load a text file and convert each line to a dictionary.
lines = sc.textFile("examples/…/people.txt")

parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0],age=int(p[1])))

# Infer the schema, and register the SchemaRDD as a table
peopleTable = sqlCtx.inferSchema(people)
peopleTable.registerAsTable("people")
```

RDDs as Relations (Java)

```
public class Person implements Serializable {
  private String _name;
  private int _age;
  public String getName() { return _name;  }
  public void setName(String name) { _name = name; }
  public int getAge() { return _age; }
  public void setAge(int age) { _age = age; }
}

JavaSQLContext ctx = new org.apache.spark.sql.api.java.JavaSQLContext(sc)
JavaRDD<Person> people = ctx.textFile("examples/src/main/resources/
people.txt").map(
  new Function<String, Person>() {
    public Person call(String line) throws Exception {
      String[] parts = line.split(",");
      Person person = new Person();
      person.setName(parts[0]);
      person.setAge(Integer.parseInt(parts[1].trim()));
      return person;
    }
  });
JavaSchemaRDD schemaPeople = sqlCtx.applySchema(people, Person.class);
```

Querying using Spark SQL (Python)

```
# SQL can be run over SchemaRDDs that have been registered
# as a table.
teenagers = sqlCtx.sql("""
  SELECT name FROM people WHERE age >= 13 AND age <= 19""")

# The results of SQL queries are RDDs and support all the normal
# RDD operations.
teenNames = teenagers.map(lambda p: "Name: " + p.name)
```

Support of Existing Tools, and New Data Sources

- SparkSQL includes a server that exposes its data using JDBC/ODBC
  - Query data from HDFS/S3
  - Including formats like Hive/Parquet/JSON
  - Support for caching data IN-MEMORY

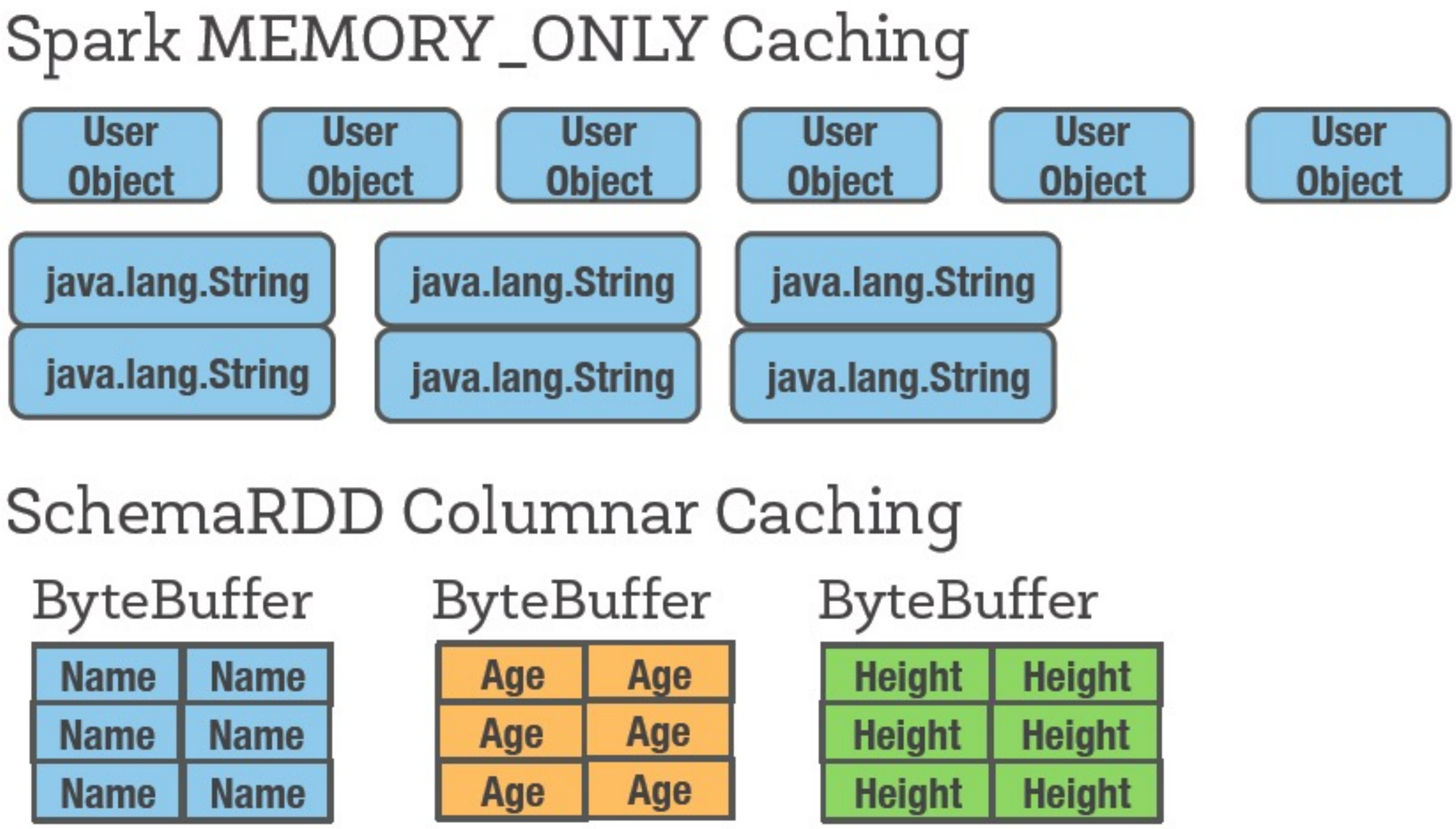
Caching Tables In-Memory

- SparkSQL can cache tables using an in-memory columnar format:
  - Scan only required columns
  - Fewer allocated objects (less Garbage Collection)
  - Automatically selects best compression

  e.g.

  cacheTable("people") or dataframe.cache( )

Caching Comparison



Language Integrated UDFs

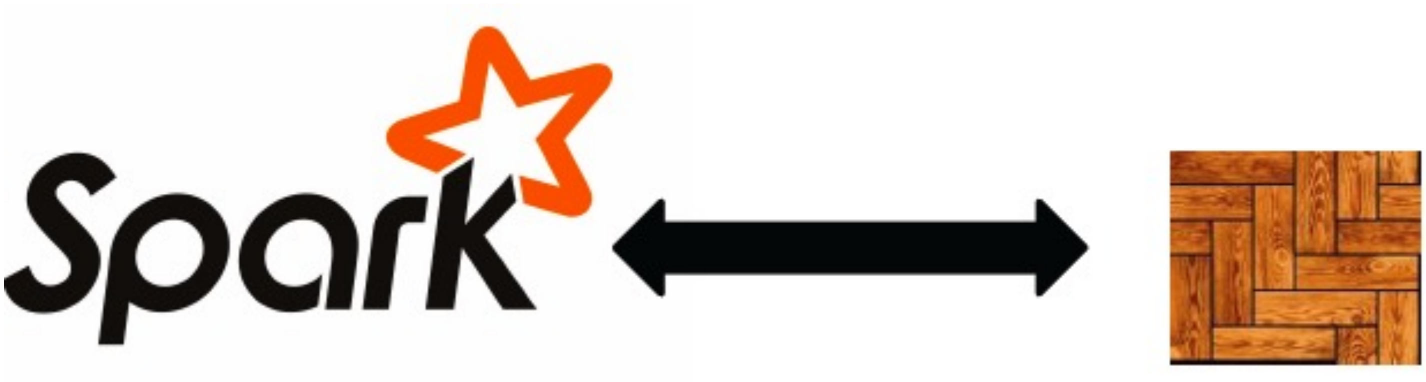
```
registerFunction("countMatches",
  lambda (pattern, text):
    re.subn(pattern, '', text)[1])

sql("SELECT countMatches('a', text)…")
```

Reading Data stored in Hive

```
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)

hiveCtx.hql("""
  CREATE TABLE IF NOT EXISTS src (key INT, value STRING)""")

hiveCtx.hql("""
  LOAD DATA LOCAL INPATH 'examples/…/kv1.txt' INTO TABLE src""")

# Queries can be expressed in HiveQL.
results = hiveCtx.hql("FROM src SELECT key, value").collect()
```

Parquet Compatibility

- Native support for reading data in Parquet
  - Columnar storage avoids reading unneeded data
  - RDDs can be written to Parquet files, preserving the schema
  - Convert other slower formats into Parquet for repeated querying



Using Parquet

```
# SchemaRDDs can be saved as Parquet files, maintaining the
# schema information.
peopleTable.saveAsParquetFile("people.parquet")

# Read in the Parquet file created above.  Parquet files are
# self-describing so the schema is preserved. The result of
# loading a parquet file is also a SchemaRDD.
parquetFile = sqlCtx.parquetFile("people.parquet")

# Parquet files can be registered as tables used in SQL.
parquetFile.registerAsTable("parquetFile")
teenagers = sqlCtx.sql("""
  SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19""")
```

JSON Support

- Use jsonFile or jsonRDD to convert a collection of JSON objects into a DataFrame
- Infer and Union the schema of each record
- Maintain nested structures and arrays

JSON Example

```
# Create a SchemaRDD from the file(s) pointed to by path
people = sqlContext.jsonFile(path)

# Visualized inferred schema with printSchema().
people.printSchema()
# root
#  |-- age: integer
#  |-- name: string

# Register this SchemaRDD as a table.
people.registerTempTable("people")
```

Data Sources API

- Allow easy integration with new sources of structured data:

```
CREATE TEMPORARY TABLE episodes
USING com.databricks.spark.avro
OPTIONS (
  path ”./episodes.avro”
)
```

https://github.com/databricks/spark-avro

Much More than SQL: DataFrames as A Unified Interface for the Processing of Structured Data

Image credit: http://barrymieny.deviantart.com/

Much More than SQL: Simplifying Inputs and Outptuts

Spark SQL's Data Source API can read and write DataFrames using a variety of formats.

Unified and Simplified Interface to Read/ Write Data in Many different Formats

```
df = sqlContext.read \
  .format("json") \
  .option("samplingRatio", "0.1") \
  .load("/home/michael/data.json")

df.write \
  .format("parquet") \
  .mode("append") \
  .partitionBy("year") \
  .saveAsTable("fasterData")
```

`read` and `write` functions create new builders for doing I/O

Unified and Simplified Interface to Read/ Write Data in Many different Formats

```
df = sqlContext.read \
  .format("json") \
  .option("samplingRatio", "0.1") \
  .load("/home/michael/data.json")

df.write \
  .format("parquet") \
  .mode("append") \
  .partitionBy("year") \
  .saveAsTable("fasterData")
```

Builder methods are used to specify:

- Format
- Partitioning
- Handling of existing data
- and more

Unified and Simplified Interface to Read/ Write Data in Many different Formats

```
df = sqlContext.read \
  .format("json") \
  .option("samplingRatio", "0.1") \
  .load("/home/michael/data.json")

df.write \
  .format("parquet") \
  .mode("append") \
  .partitionBy("year") \
  .saveAsTable("fasterData")
```

load(…), save(…) or saveAsTable(…) functions create new builders for doing I/O

ETL using Custom Data Sources

```
sqlContext.read
  .format("com.databricks.spark.jira")
  .option("url", "https://issues.apache.org/jira/rest/api/latest/search")
  .option("user", "marmbrus")
  .option("password", "********")
  .option("query", """
    |project = SPARK AND
    |component = SQL AND
    |(status = Open OR status = "In Progress" OR status = Reopened)""".stripMargin)
  .load()
  .repartition(1)
  .write
  .format("parquet")
  .saveAsTable("sparkSqlJira")
```

Write Less Codes with DataFrames

- Common operations can be expressed concisely as higher level operation calls to the DataFrame API:
  - Selecting required Columns
  - Joining Different Data Sources
  - Aggregation (Count, Sum, Average, etc)
  - Filtering

Write Less Codes:
An Example of Computing Average

```java
private IntWritable one =
  new IntWritable(1)
private IntWritable output =
  new IntWritable()
proctected void map(
    LongWritable key,
    Text value,
    Context context) {
  String[] fields = value.split("\t")
  output.set(Integer.parseInt(fields[1]))
  context.write(one, output)
}

IntWritable one = new IntWritable(1)
DoubleWritable average = new DoubleWritable()

protected void reduce(
    IntWritable key,
    Iterable<IntWritable> values,
    Context context) {
  int sum = 0
  int count = 0
  for(IntWritable value : values) {
     sum += value.get()
     count++
    }
  average.set(sum / (double) count)
  context.Write(key, average)
}
```

```python
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [x.[1], 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] +
y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

Write Less Code:
Example of Computing Average

Using RDDs

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

Using SQL

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

Using Pig

```
P = load '/people' as (name, name);
G = group P by name;
R = foreach G generate … AVG(G.age);
```

Using DataFrames

```
sqlCtx.table("people") \
   .groupBy("name") \
   .agg("name", avg("age")) \
   .collect()
```

Read Less Data with DataFrames & SparkSQL

"The fastest way to process big data is to never read it."

- SparkSQL can help the program to read less data automatically by performing BEYOND naïve scanning:
  - Using Columnar formats (e.g. Parquet) and prune irrelevant Columns and Blocks of data
  - Push filters to the source
  - Converting to more efficient formats, e.g. turning string comparisons into integer comparisons for dictionary encoded data
  - Using Partitioning (i.e., /year=2-14/month=02/..)
  - Skipping data using statistics (i.e. min, max)
  - Pushing predicates into storage systems (i.e. JDBC)

Intermix DataFrame Operations with Custom Codes (Python, Java, R, Scala)

```
zipToCity = udf(lambda zipCode: <custom logic here>)

def add_demographics(events):
   u = sqlCtx.table("users")
   events \
     .join(u, events.user_id == u.user_id) \
     .withColumn("city", zipToCity(df.zip))
```

Augments any DataFrame that contains `user_id`

Takes and returns a DataFrame

Integration with RDDs

- Internally, DataFrame execution is done with Spark RDDs

=> Easy Interoperation with outside sources and custom algorithms

External Input

```
def buildScan(
    requiredColumns: Array[String],
    filters: Array[Filter]): RDD[Row]
```

Custom Processing

```
queryResult.rdd.mapPartitions { iter =>

  … Your code here …

}
```

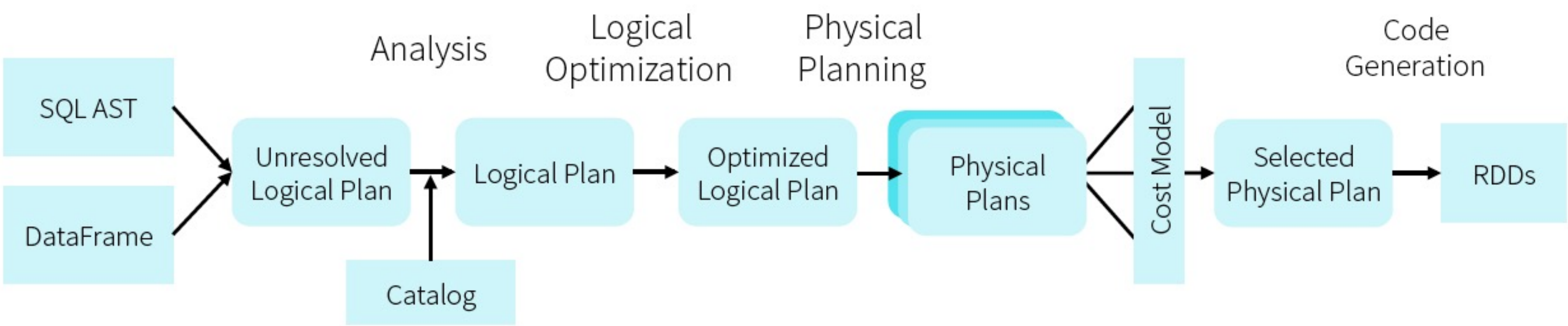
DataFrame & SparkSQL Demo

Demo:

- *Using Spark SQL to read, write, and transform data in a variety of formats:*

http://people.apache.org/~marmbrus/talks/dataframe.demo.pdf

Plan Optimization and Execution for the entire Pipelines



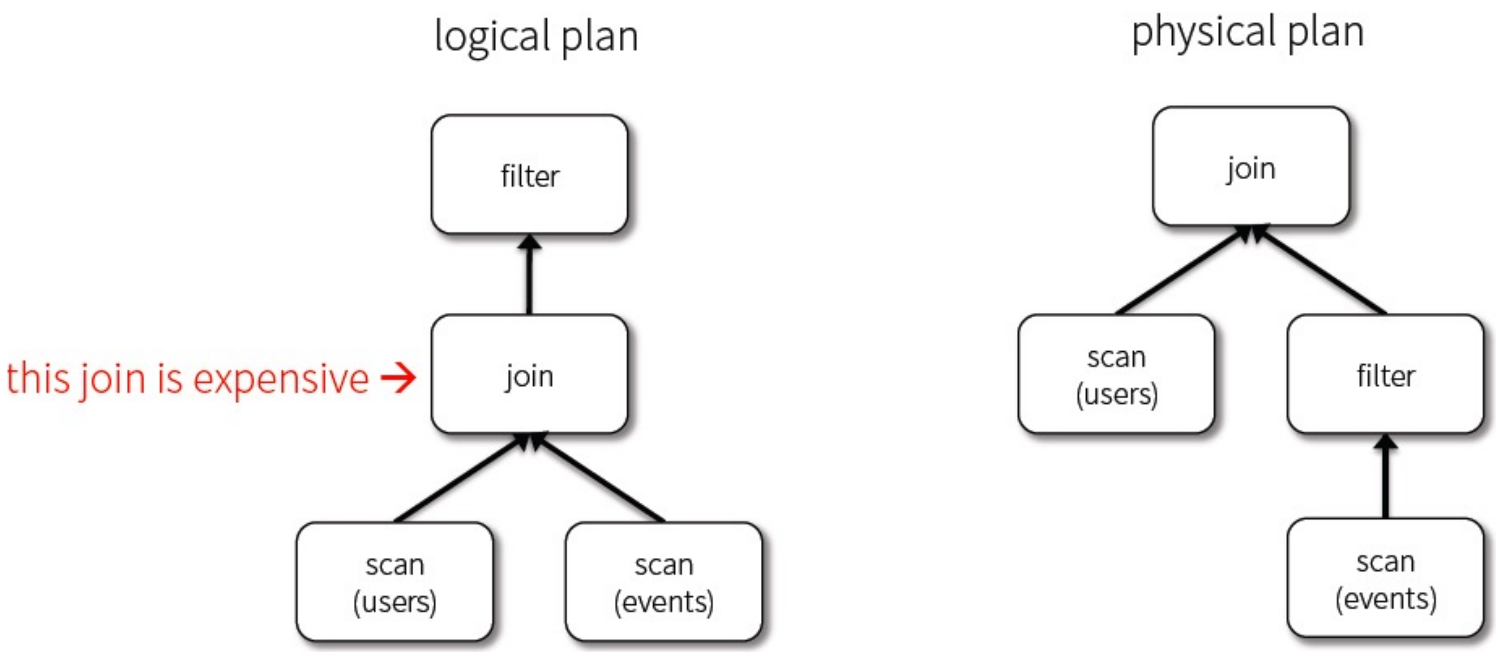
DataFrames and SQL share the same optimization/execution pipeline

- Optimization happens as late as possible

=> Spark SQL can optimize *even across different* functions !

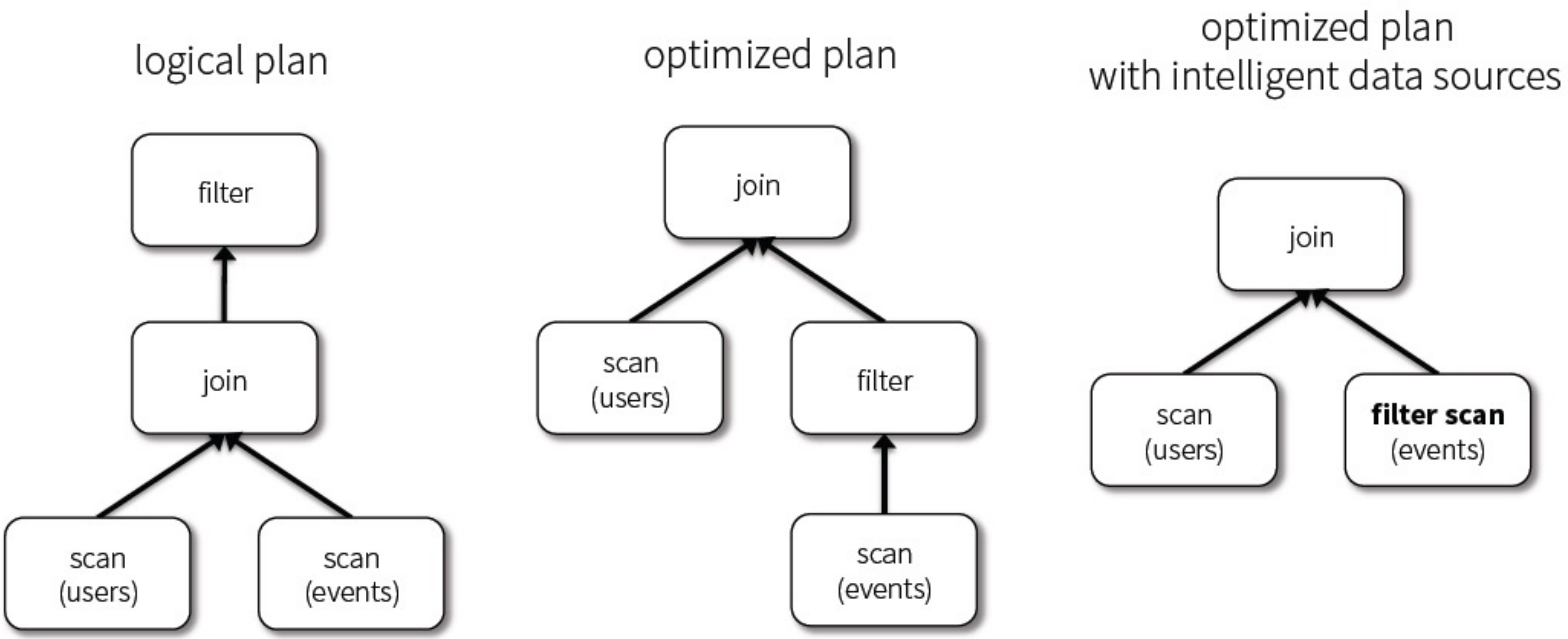
Optimization Example

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



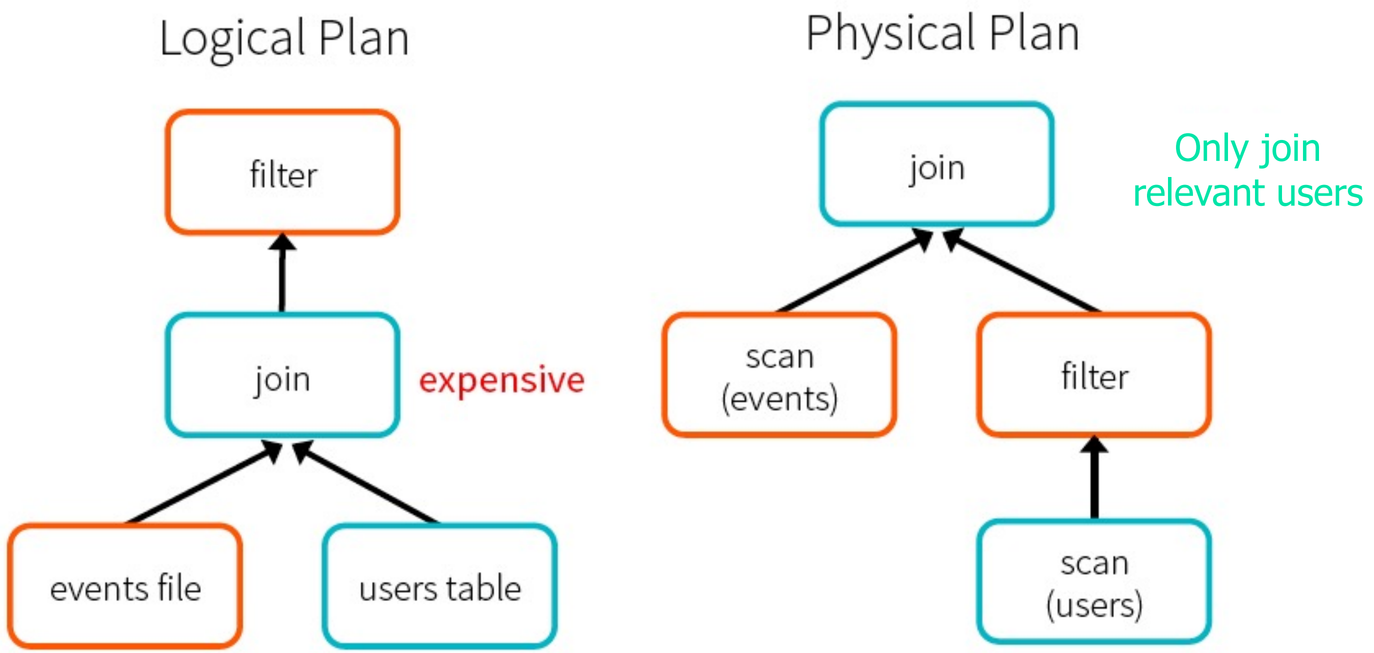
Optimization Example

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date > "2015-01-01")
```



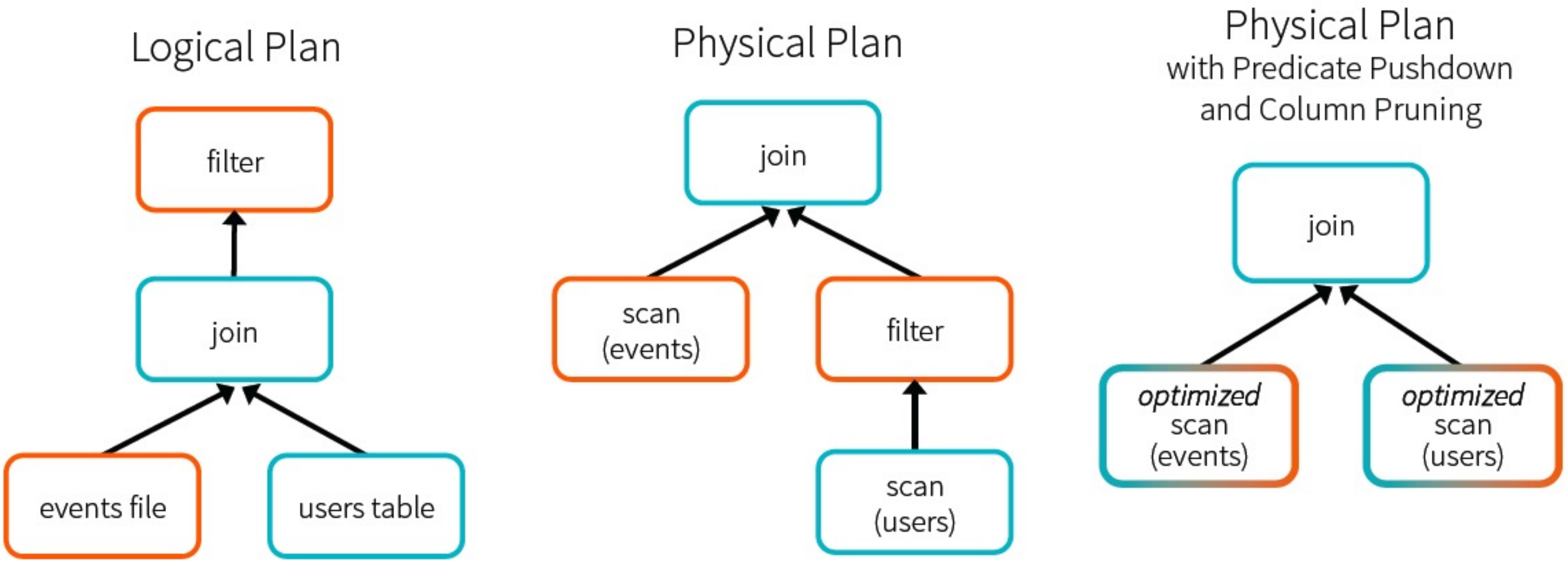
Optimization Example

```
def add_demographics(events):
    u = sqlCtx.table("users")                        # Load Hive table
    events \
      .join(u, events.user_id == u.user_id) \       # Join on user_id
      .withColumn("city", zipToCity(df.zip))        # Run udf to add city column
events = add_demographics(sqlCtx.load("/data/events", "json"))
training_data = events.where(events.city == "New York").select(events.timestamp).collect()
```

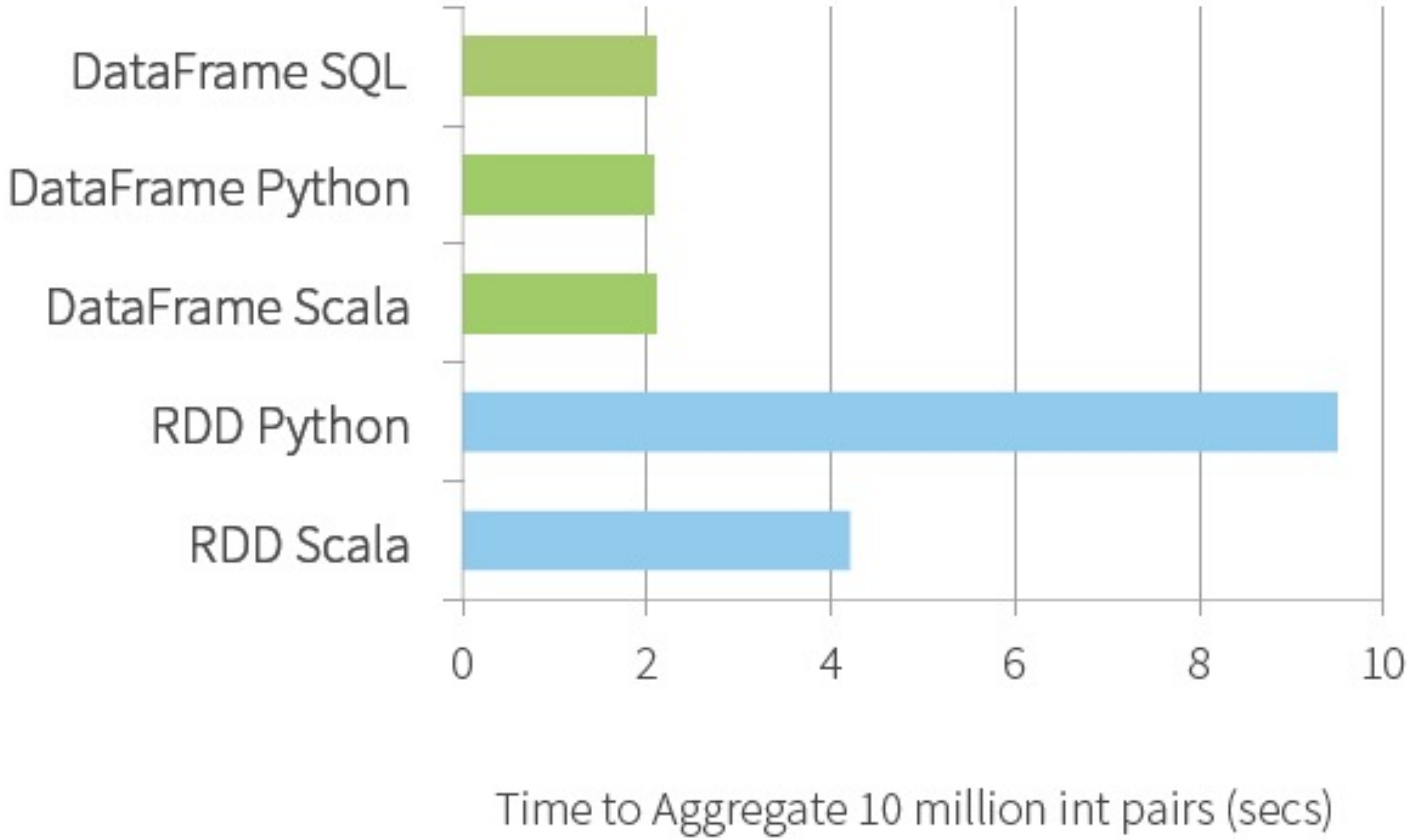


Optimization Example

```
def add_demographics(events):
    u = sqlCtx.table("users")                    # Load partitioned Hive table
    events \
      .join(u, events.user_id == u.user_id) \    # Join on user_id
      .withColumn("city", zipToCity(df.zip))     # Run udf to add city column
events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "New York").select(events.timestamp).collect()
```



Named Columns (vs. Opaque Objects in RDDs) Enable Performance Optimization



More Performance Comparison

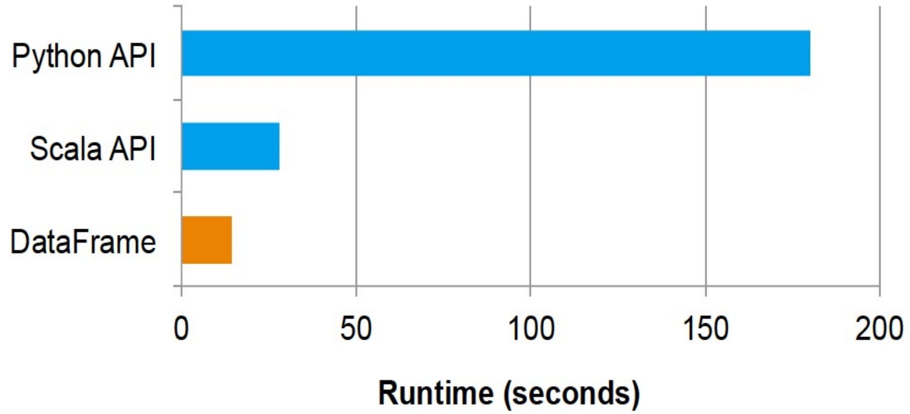


Figure 9: Performance of an aggregation written using the native Spark Python and Scala APIs versus the DataFrame API.

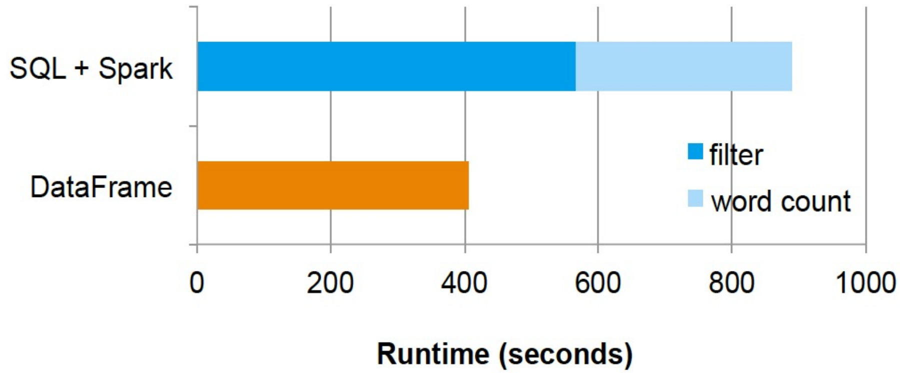


Figure 10: Performance of a two-stage pipeline written as a separate Spark SQL query and Spark job (above) and an integrated DataFrame job (below).

Datasets: Another Structured Data Abstraction and its API in Spark

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

Analysis errors reported before a distributed job starts

More info at:
https://techvidvan.com/tutorials/apache-spark-dataframe-vs-datasets/

Datasets vs. DataFrames

- DataFrames are collections of rows with a schema
- Datasets add static types, e.g. Dataset[Person]
- Spark 2.0 has merged these APIs:

```
Dataframe = Dataset[Row]
```

Benefits of Merging

- Simpler to understand
  - Only kept Dataset separate to keep binary compatibility in Spark 1.x
- Libraries can take data of both forms
- With Streaming, same API will also work on streams

Datasets vs. DataFrames

Source: Chapter 4, p.g. 50 of "Spark - The Definitie Guide" by Bill Chambers & Matei Zaharia

"In essence, within the Structured APIs, there are two more APIs, the "untyped" DataFrames and the "typed" Datasets. To say that DataFrames are untyped is aslightly inaccurate; they have types, but Spark maintains them completely and only checks whether those types line up to those specified in the schema at runtime. Datasets, on the other hand, check whether types conform to the specification at compile time.

Datasets are only available to Java Virtual Machine (JVM)–based languages (Scala and Java) and we specify types with case classes or Java beans. For the most part, you're likely to work with DataFrames. To Spark (in Scala), Data-Frames are simply Datasets of Type Row. The "Row" type is Spark's internal representation of its optimized in-memory format for computation. This format makes for highly specialized and efficient computation because rather than using JVM types, which can cause high garbage-collection and object instantiation costs, Spark can operate on its own internal format without incurring any of those costs. To Spark (in Python or R), there is no such thing as a Dataset: everything is a DataFrame and therefore we always operate on that optimized format."

Example for Datasets and DataFrames

```
case class User(name: String, id: Int)
case class Message(user: User, text: String)

dataframe = sqlContext.read.json("log.json")      // DataFrame, i.e. Dataset[Row]
messages = dataframe.as[Message]                  // Dataset[Message]

users = messages.filter(m => m.text.contains("Spark"))
                .map(m => m.user)                 // Dataset[User]

pipeline.train(users)           // MLlib takes either DataFrames or Datasets
```

Datasets API

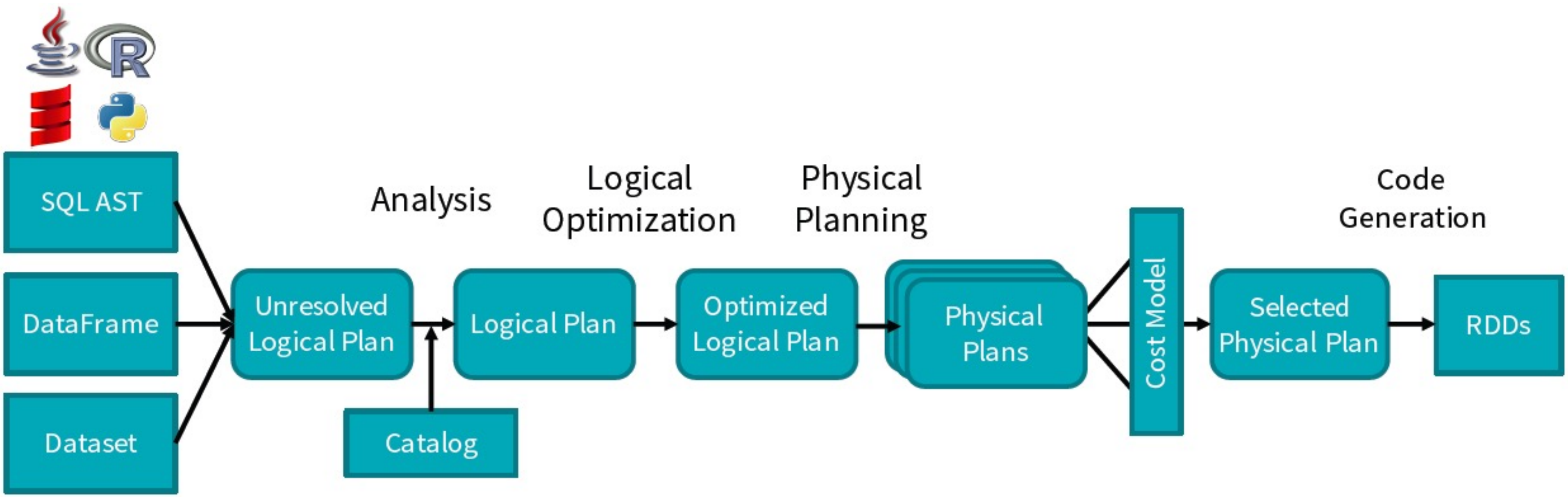
- Type-safe: Operate on domain objects with compiled lambda functions

```
val df = ctx.read.json("people.json")

// Convert data to domain objects.
case class Person(name: String, age: Int)
val ds: Dataset[Person] = df.as[Person]
ds.filter(_.age > 30)

// Compute histogram of age by name.
val hist = ds.groupBy(_.name).mapGroups {
  case (name, people: Iter[Person]) =>
    val buckets = new Array[Int](10)
    people.map(_.age).foreach { a =>
      buckets(a / 10) += 1
    }
    (name, buckets)
}
```

Long-Term Direction

- RDD will remain the low-level API in Spark
- Datasets and DataFrames give richer semantics and optimizations
  - New libraries will increasingly use these as interchange format, e.g. Structured Streaming, MLlib and GraphFrames

Shared Optimization and Execution



DataFrames, Datasets and SQL
share the same optimization/execution pipeline

Towards to the Support of SQL 2003

- Since 2017, Spark can run all 99 TPC-DS queries
- Have a standard compliant parser
- Subqueries (correlated & uncorrelated)
- Approximate Aggregate Stats
  - https://databricks.com/blog/2016/06/17/sql-subqueries-in-apache-spark-2-0.html

Lessons Learnt from Spark SQL

- SQL is wildly popular and important for real-world customers
- Schema is very useful
  - In most data pipelines, even the ones that start with unstructured data end up having some implicit structure
  - Key-value abstraction (under RDD) is too limited
  - Nevertheless, Support for Semi/Un-structured data is critical !
- Separation of Logical vs. Physical Plan is important for Performance Optimizations, e.g. join selection.