

# Spark SQL

## Architecture and Data Structures

Pietro Michiardi

Eurecom

- M. Aembrust, et. al., “Spark SQL: Relational Data Processing in Spark”, in Proc. of ACM Sigmod 2015
- Lookup for slides on SlideShare, and videos around the web!

# Introduction

# Introduction

- **SQL as an high-level programming language**

- ▶ Expressiveness, succinctness
- ▶ Enables compatibility with existing tools, e.g. BI using JDBC
- ▶ Large pool of engineers proficient in SQL

- **Project goals**

- ▶ Write less code
- ▶ Read less data
- ▶ Let the optimizer do the hard work

- **Design philosophy**

- ▶ SparkSQL is a Library
- ▶ Uses the SparkContext to interact with Spark

# Challenges

- **Variety of data source formats**

- ▶ ETL workloads often involve working with various kinds of data
- DataSource API

- **SQL implementation**

- ▶ Extensibility, e.g. to cover SQL standard
- DataFrame API
- ▶ Efficiency
- Catalyst optimizer

# Outline

- **Spark and SparkSQL data structures**
- **Functional architecture, with a RDBMS flavor**
- **Performance**

# Data Representation

## DataSource API

- Read and write with a variety of formats

Built-In

{ JSON }



JDBC

Parquet



MySQL



PostgreSQL



amazon  
web services | S3



External



dBase

APACHE  
HBASE

elasticsearch.



amazon  
web services  
Amazon Redshift

and more...



## DataSource API

- Unified interface to reading data
- **read** function creates new I/O builders
- **load** function creates new I/O builders

```
df = sqlContext.read \  
  .format('json') \  
  .option('samplingRatio', '0.1') \  
  .load('data.json')
```

## DataSource API

- Unified interface to writing data
- **Write** function creates new I/O builders
- **save** function creates new I/O builders

```
df.write \  
  .format('parquet') \  
  .mode('append') \  
  .partitionBy('year') \  
  .saveAsTable('myData')
```

# DataSource API

## ● **Builder methods**

- ▶ Specify data format
- ▶ Define data partitioning
- ▶ Handle existing data
- ▶ ... and much more

# DataFrame

- **Schema to the rescue**

- ▶ A distributed collection of rows organized into named columns
- ▶ Schema inference can be automatic

- **Structured data**

- ▶ An abstraction for selecting, filtering, aggregating and plotting structured data

# DataFrame

- **General idea borrowed from Python Pandas**

- ▶ Tabular data with an API
- ▶ Math, stats, algebra, ...

- **Relation to a low-level RDD**

- ▶ Introduces structure to the data
- ▶ Specific relational operators
  - ★ Select required columns
  - ★ Join different data sources
  - ★ Aggregation operations
  - ★ Filtering

## DataFrame API

- Example using RDDs

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

## DataFrame API

- Example using SQL

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

## DataFrame API

- Example using DataFrames

```
sqlContext.table('people') \  
  .groupBy('name') \  
  .agg('name', avg('age')) \  
  .collect()
```



# Architecture

## Background and roadmap

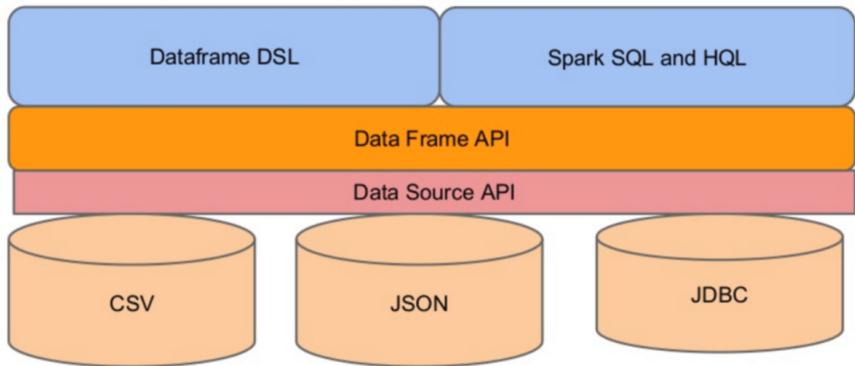
- **Reminiscent of traditional database systems**

- ▶ Abstract representation of SQL expressions
- ▶ Optimizations for efficiency and performance
- ▶ Sophisticated cost model

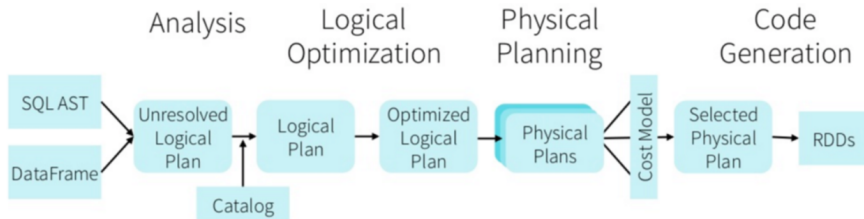
- **Focus on optimizations**

- ▶ Logical plan
- ▶ Physical plan
- ▶ Cost-based vs. Rule-based

## Global view



# SparkSQLContext



# Catalyst optimizer

- **Overall goals**

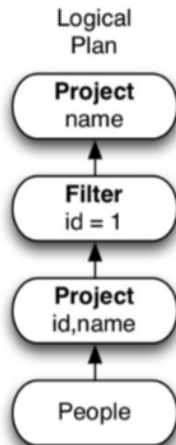
- ▶ Optimize logical plan
- ▶ Convert logical to physical plan
- ▶ Optimize physical plan
- ▶ Code generation

- **Exploit `scala` language features**

- ▶ `Quasiquotes`
- ▶ Abstract syntax tree
- ▶ Tree manipulation library
- ▶ Optimizations rules implemented as tree transformations

## Example query

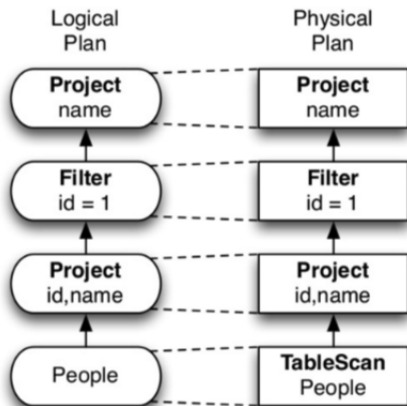
```
SELECT name
FROM (
    SELECT id, name
    FROM People) p
WHERE p.id = 1
```



## Example query

### Native query planning

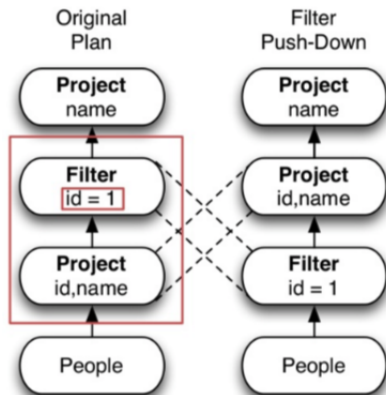
```
SELECT name
FROM (
    SELECT id, name
    FROM People) p
WHERE p.id = 1
```



## Example query

### Optimization rules example

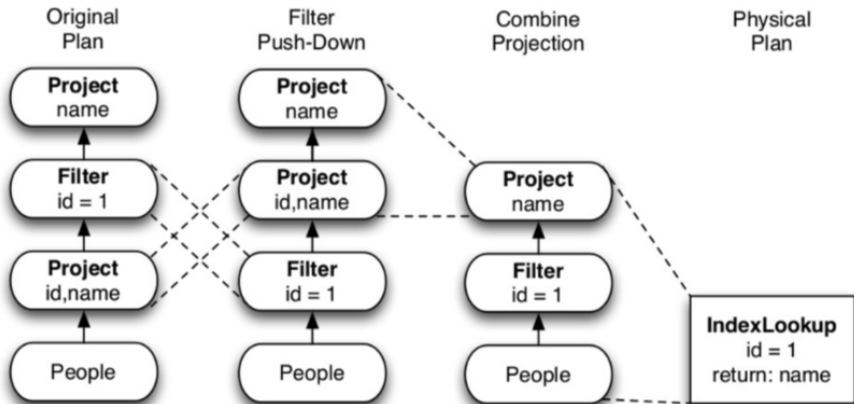
- Find filters on top of projections
- Check that filters can be evaluated without the result of the projection
- If yes, switch operators





## Example query

- Definition of custom rules



## Example Optimization Rules

- Eliminate subqueries
- Constant folding
- Simplify filters
- PushPredicate through filter
- Project collapsing

# Project Tungsten

- **Runtime code generation**

- ▶ Using code generation to exploit modern compilers and CPUs

- **Cache locality**

- ▶ Algorithms and data structures to exploit memory hierarchy

- **Off-heap memory management**

- ▶ Leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection

## Advanced features

- Consider string "abcd": this is 4 bytes in UTF-8

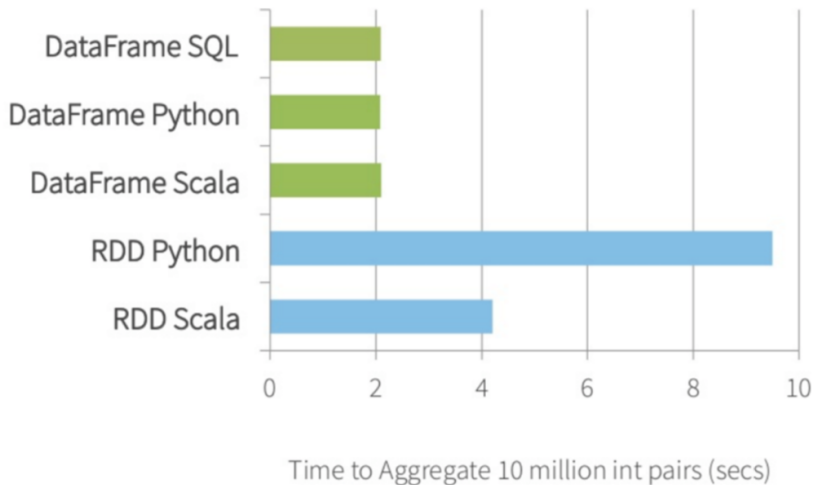
java.lang.String object internals:

OFFSET	SIZE	TYPE	DESCRIPTION	VALUE
0	4		(object header)	...
4	4		(object header)	...
8	4		(object header)	...
12	4	char[]	String.value	[]
16	4	int	String.hash	0
20	4	int	String.hash32	0

Instance size: 24 bytes (reported by Instrumentation API)

# Performance

## Performance comparisons



# Conclusion

## Conclusion

- **Short overview of SparkSQL**

- ▶ Borrows many ideas from traditional RDBMs
- ▶ Standard SQL
- ▶ Cost-based optimization
- ▶ Project Tungsten

- **Very useful tool for**

- ▶ Extract, transform workloads
- ▶ Simple descriptive statistics
- ▶ Data exploration, cleaning

- **Laboratory on SparkSQL**

- ▶ Work on a highly-dimensional dataset
- ▶ Typical BI queries