Apache Spark and Scala

Reynold Xin @rxin

2017-10-22, Scala 2017

Apache Spark

Started in UC Berkeley ~ 2010

Most popular and de facto standard framework in big data

One of the largest OSS projects written in Scala (but with user-facing APIs in Scala, Java, Python, R, SQL)

Many companies introduced to Scala due to Spark

whoami

Databricks co-founder & Chief Architect

- Designed most of the major things in “modern day” Spark
- #1 contributor to Spark by commits and **net lines deleted**

UC Berkeley PhD in databases (on leave since 2013)

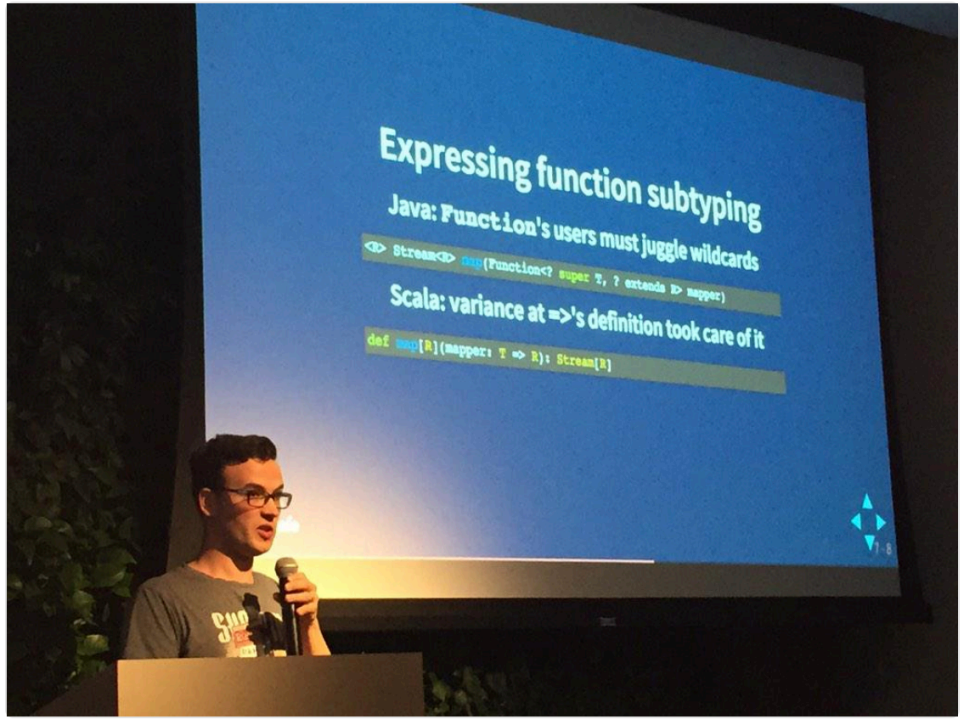
My Scala / PL background

Working with Scala day-to-day since 2010; previously mostly C, C++, Java, Python, Tcl …

Authored “Databricks Scala Style Guide”, i.e. Scala is a better Java.

No PL background, i.e. from a PL perspective, I think mostly based on experience and use cases, not first principle.

.@adriaanm at #sfscala: "I never think about variance—I just write pluses and minuses until it compiles."



8:22 PM - 17 Feb 2015 from San Francisco, CA

38 Retweets **41** Likes

How do you compare this with X?

Wasn’t this done in X in the 80s?

Today's Talk

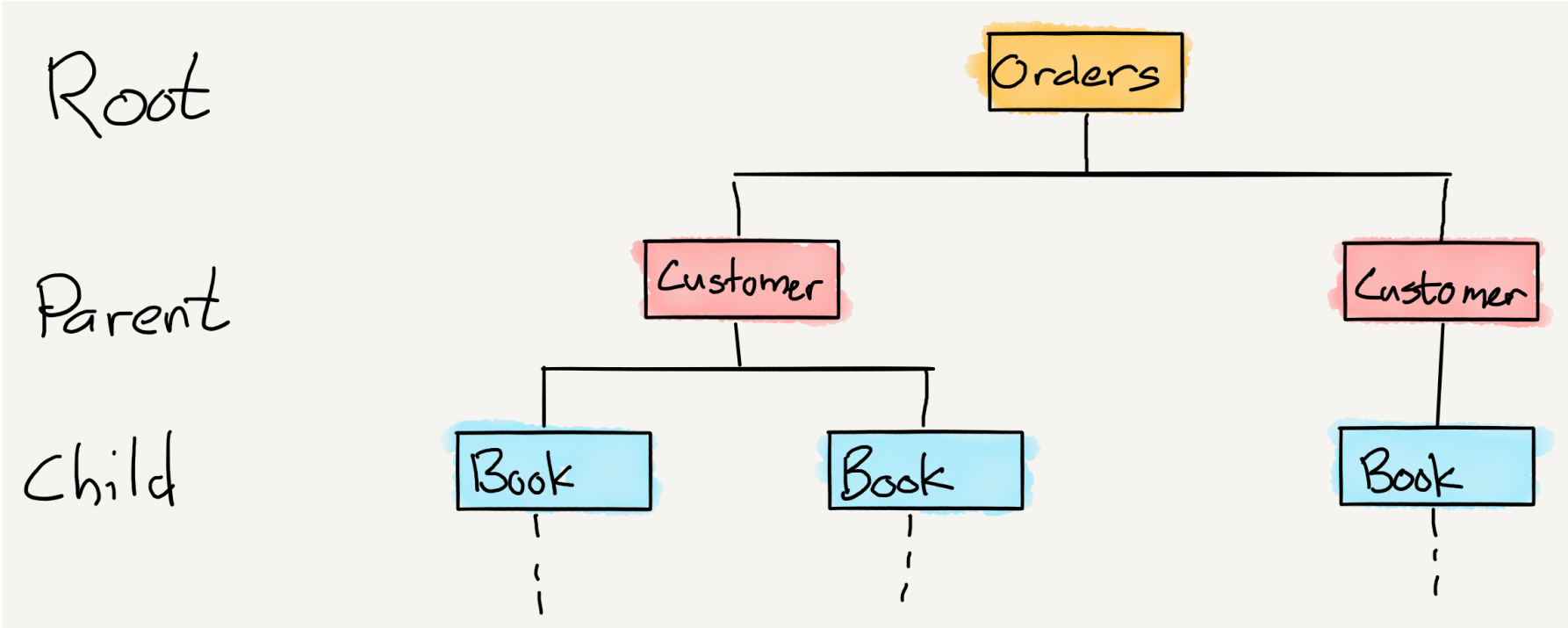
Some archaeology

- IMS, relational databases
- MapReduce
- data frames

Last 7 years of Spark evolution (along with what Scala has enabled)

Databases

IBM IMS hierarchical database (1966)



databricks

Image from https://stratechery.com/2016/oracles-cloudy-future/

A Relational Model of Data for Large Shared Data Banks

E. F. CODD
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on *n*-ary relations, a normal form for data base relations, and the concept of a universal

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for noninferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the "connection trap").

Finally, the relational view permits a clearer evaluation of the scope and logical limitations of present formatted data systems, and also the relative merits (from a logical standpoint) of competing representations of data within a single system. Examples of this clearer perspective are cited in various parts of this paper. Implementations of systems to support the relational model are not discussed.

“Future users of large data banks must be protected from having to know how the data is organized in the machine. …

most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed.”

Two important ideas in RDBMS

Physical Data Independence: The ability to change the physical data layout without having to change the logical schema.

Declarative Query Language: Programmer specifies “what” rather than “how”.

Why?

Business applications outlive the environments they were created in:

- New requirements might surface
- Underlying hardware might change
- Require physical layout changes (indexing, different storage medium, etc)

Enabled tremendous amount of innovation:

- Indexes, compression, column stores, etc

Relational Database Pros vs Cons

- Declarative and data independent
- SQL is the universal interface everybody knows

- SQL is not a “real” PL
  - Difficult to compose & build complex applications
  - Lack of testing frameworks, IDEs
- Too opinionated and inflexible
  - Require data modeling before putting any data in

Big Data, MapReduce, Hadoop

The Big Data Problem

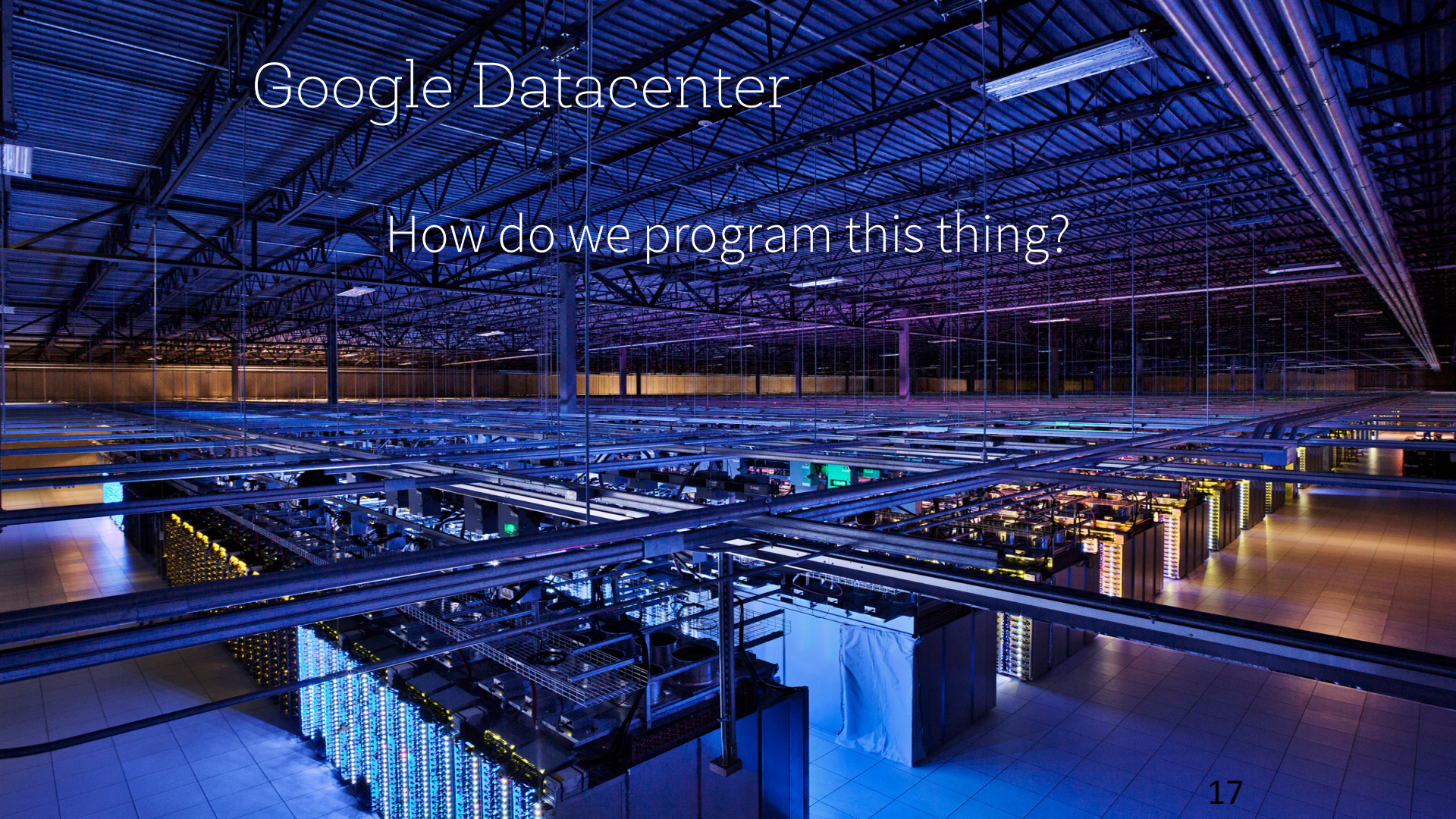
Semi-/Un-structured data doesn't fit well with databases

Single machine can no longer process or even store all the data!

Only solution is to **distribute** general storage & processing over clusters.

Google Datacenter

How do we program this thing?



Data-Parallel Models

Restrict the programming interface so that the system can do more automatically

"Here's an operation, run it on all of the data"

- I don't care *where* it runs (you schedule that)
- In fact, feel free to run it *twice* on different nodes
- Leverage key concepts in functional programming
- Similar to "declarative programming" in databases

MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM | Permalink | Comments (44) | TrackBacks (1)
[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we'll begin here v
to discuss it, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradig
processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a larg
much smaller number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to te
software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the Ma

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represen
data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. A sub-optimal implementation, in that it uses brute force instead of indexing
3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
4. Missing most of the features that are routinely included in current DBMS

MapReduce Pros vs Cons

+ Massively parallel
+ Flexible programming model & schema-on-read
+ Type-safe programming language (great for large eng projects)
- Bad performance
- Extremely verbose
- Hard to compose, while most real apps require multiple MR steps
  - 21 MR steps -> 21 mapper and reducer classes

R, Python, data frame

Data frames in R / Python

Developed by stats community & concise syntax for ad-hoc analysis

Procedural (not declarative)

```
> head(filter(df, df$waiting < 50))  # an example in R
##  eruptions waiting
##1     1.750      47
##2     1.750      47
##3     1.867      48
```

Traditional data frames

+ Built-on “real” programming languages
+ Easier to learn

- No parallelism & doesn’t work well on med/big data
- Lack sophisticated query optimization

• No compile-time type safety (great for data science, not so great for data eng)

“Are you going to talk about Spark at all!?”

Which one is better?

Databases, R, MapReduce?
Declarative, functional, procedural?

A slide from 2013 …

Spark

Fast and expressive cluster computing system interoperable with Apache Hadoop

Improves efficiency through:
- » In-memory computing primitives
- » General computation graphs

→ Up to 100× faster (2-10× on disk)

Improves usability through:
- » Rich APIs in Scala, Java, Python
- » Interactive shell

→ Often 5× less code

Spark's initial focus: a better MapReduce

Language-integrated API (RDD): similar to Scala's collection library using functional programming; incredibly powerful and composable

```
lines = spark.textFile("hdfs://...")          // RDD[String]
points = lines.map(line => parsePoint(line))  // RDD[Point]
points.filter(p => p.x > 100).count()
```

Better performance: through a more general DAG abstraction, faster scheduling, and in-memory caching (i.e. "100X faster than Hadoop")

Programmability

```
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

WordCount in 50+ lines of Java MR

```
1  val f = sc.textFile(inputPath)
2  val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
3  w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

databricks

Why Scala (circa 2010)?

JVM-based, integrates well with existing Hadoop stack

Concise syntax

Interactive REPL

Challenge 1. Lack of Structure

Most data is structured (JSON, CSV, Parquet, Avro, …)

- Defining case classes for every step is too verbose
- Programming RDDs inevitably ends up with a lot of tuples (_1, _2, …)

Functional transformations not as intuitive to data scientists

- E.g. map, reduce

```
data
  .map(x => (x.dept, (x.age, 1)))
  .reduceByKey((v1, v2) => ((v1._1 + v2._1), (v1._2 + v2._2)))
  .map { case(k, v) => (k, v._1.toDouble / v._2) }
  .collect()
```

```
data.groupby("dept").avg()
```

Challenge 2. Performance

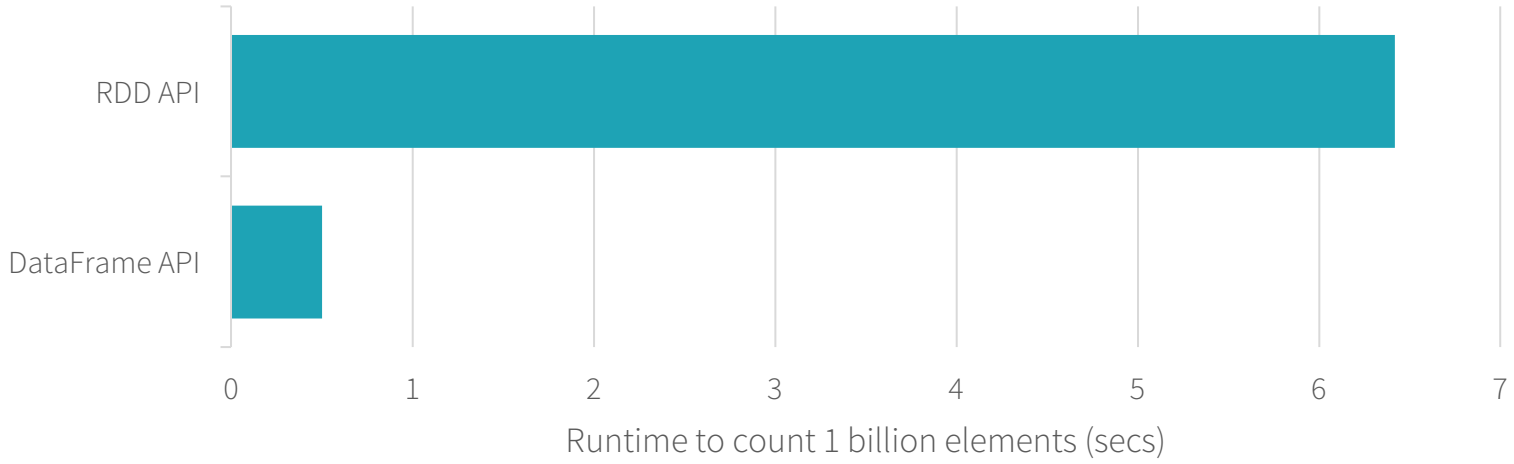
Closures are black boxes to Spark, and can't be optimized

On data-heavy computation, small overheads add up

- Iterators
- Null checks
- Physical immutability, object allocations

Python/R (the data science languages) 10X slower than Scala

Demo


RDD API
DataFrame API
0
1
2
3
4
5
6
7
Runtime to count 1 billion elements (secs)

Solution:

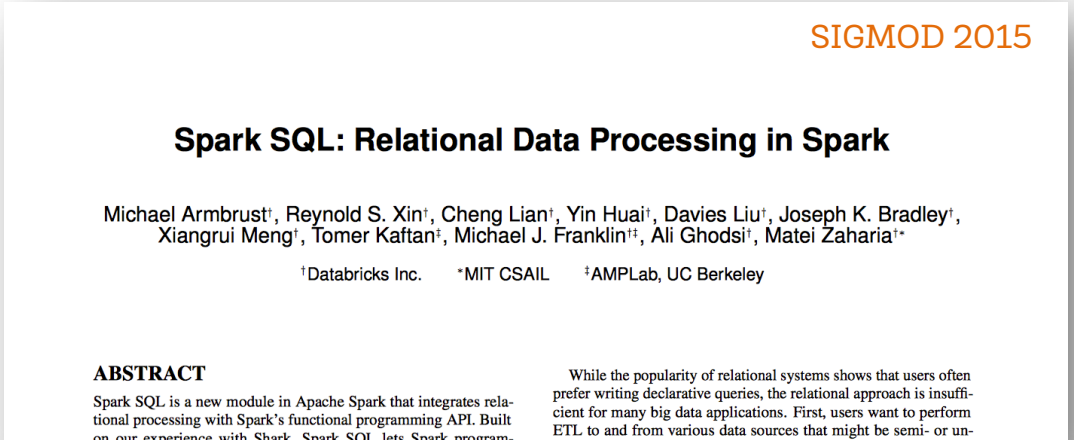
Structured APIs
DataFrames + Spark SQL

DataFrames and Spark SQL

Efficient library for structured data (data with a known schema)

- Two interfaces: SQL for analysts + apps, DataFrames for programmers

Optimized computation and storage, similar to RDBMS



SIGMOD 2015

Spark SQL: Relational Data Processing in Spark

Michael Armbrust†, Reynold S. Xin†, Cheng Lian†, Yin Huai†, Davies Liu†, Joseph K. Bradley†, Xiangrui Meng†, Tomer Kaftan‡, Michael J. Franklin†‡, Ali Ghodsi†, Matei Zaharia†*

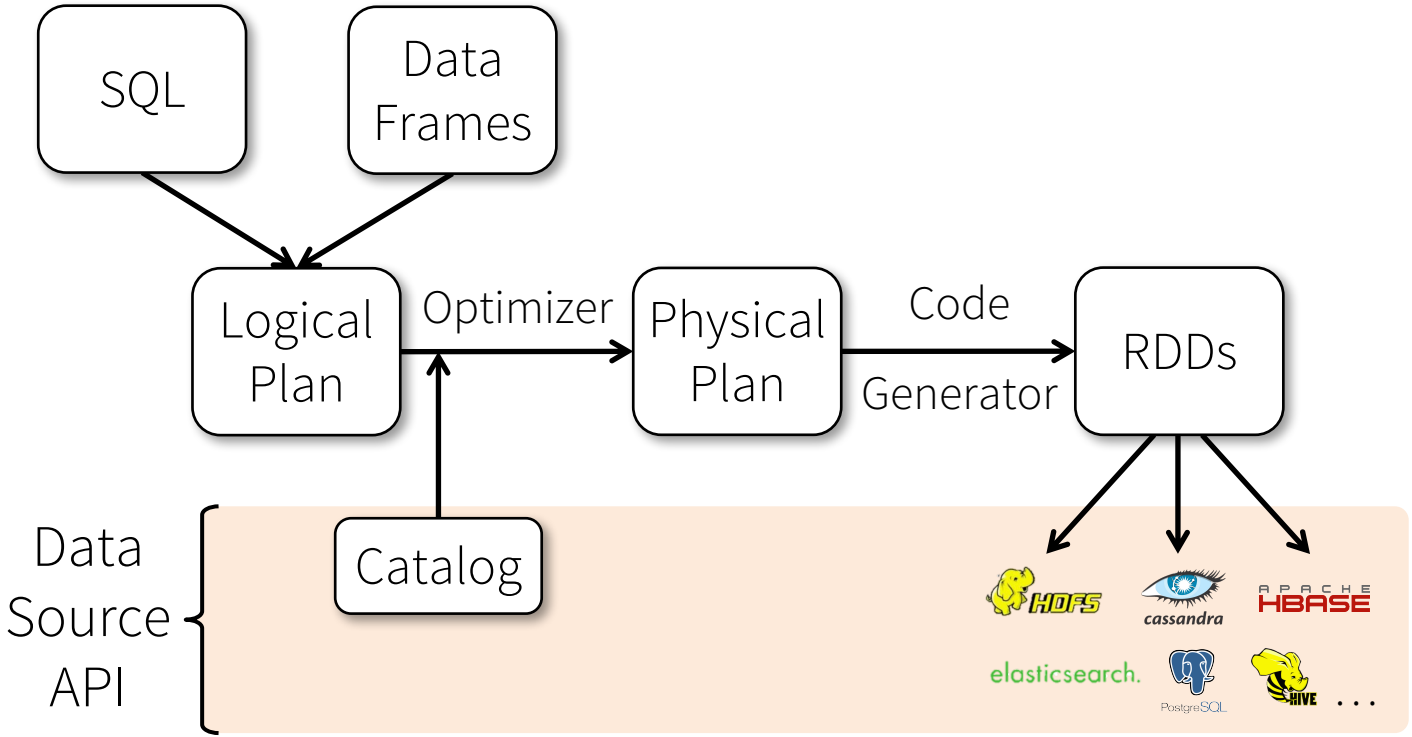
†Databricks Inc. *MIT CSAIL ‡AMPLab, UC Berkeley

ABSTRACT

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built

While the popularity of relational systems shows that users often prefer writing declarative queries, the relational approach is insufficient for many big data applications. First, users want to perform ETL to and from various data sources that might be semi- or un-

Execution Steps



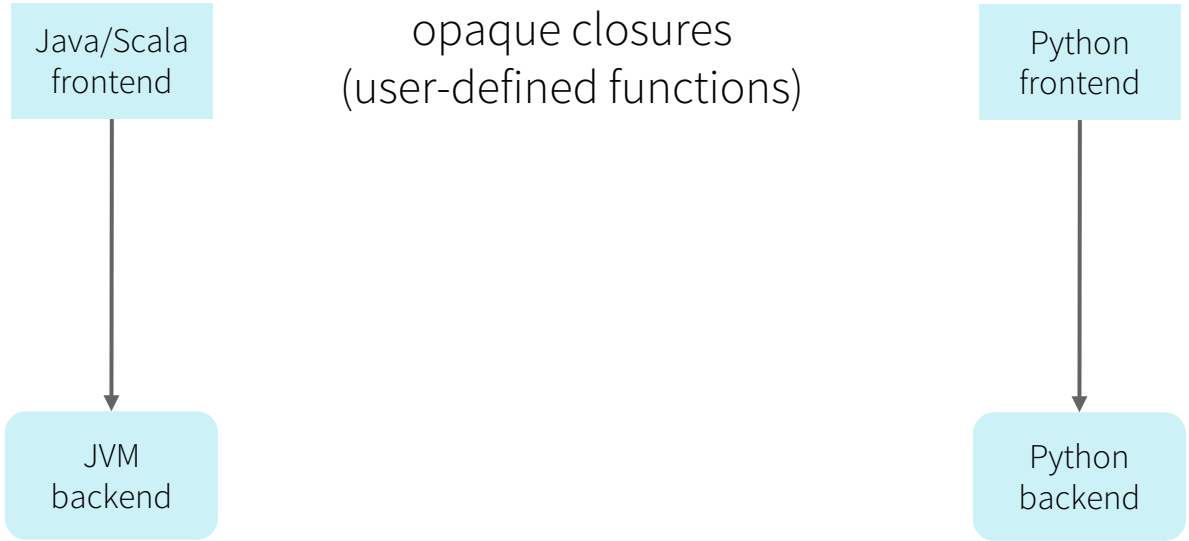
DataFrame API

DataFrames hold rows with a known schema and offer relational operations on them through a DSL

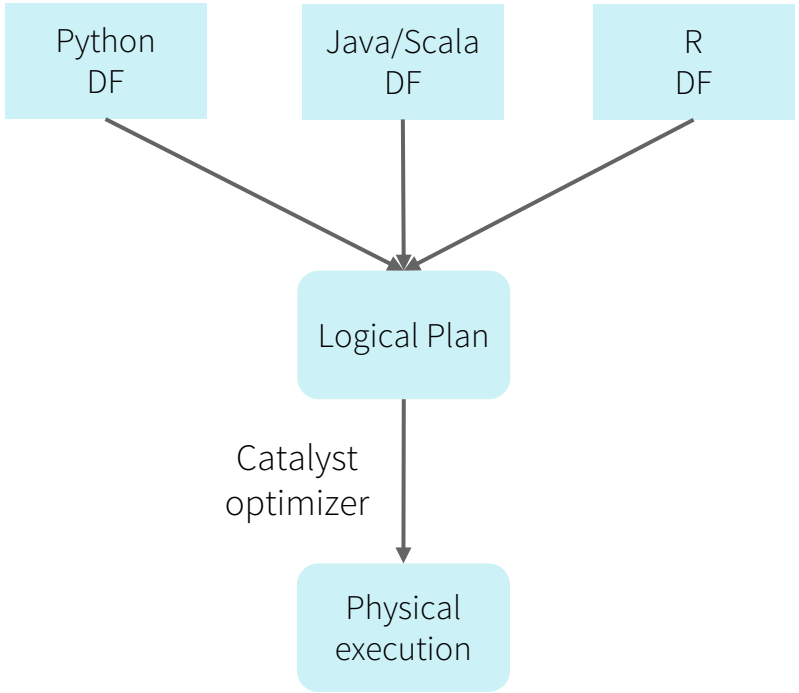
```
val users = spark.sql(“select * from users”)

val massUsers = users('country === “Canada”)

massUsers.count()

massUsers.groupBy(“name”).avg(“age”)
```

Expression AST

Spark RDD Execution



Spark DataFrame Execution

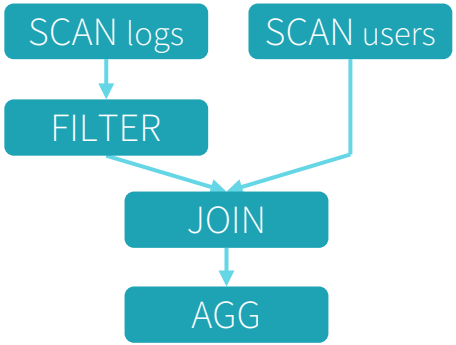


Simple wrappers to create logical plan

Intermediate representation for computation

Structured API Example

```
events =
 sc.read.json(“/logs”)

stats =
 events.join(users)
  .groupBy(“loc”,“status”)
  .avg(“duration”)

errors = stats.where(
 stats.status == “ERR”)
```




```
while(logs.hasNext) {
  e = logs.next
  if(e.status == “ERR”) {
    u = users.get(e.uid)
    key = (u.loc, e.status)
    sum(key) += e.duration
    count(key) += 1
  }
}
...
```

DataFrame API

Optimized Plan

Generated Code*

databricks

* Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. VLDB 2011.

What has Scala enabled?

Spark becomes effectively a compiler.

Pattern matching, case classes, tree manipulation invaluable.

Much more difficult to express the compiler part in Java.

Type-safety strikes back

DataFrames are runtime type checked; harder to ensure correctness for large data engineering pipelines.

Lack the ability to reuse existing classes and functions.

Datasets

RDDs

```
val lines = sc.textFile("/wikipedia")

val words = lines

  .flatMap(_.split(" "))

  .filter(_ != "")
```

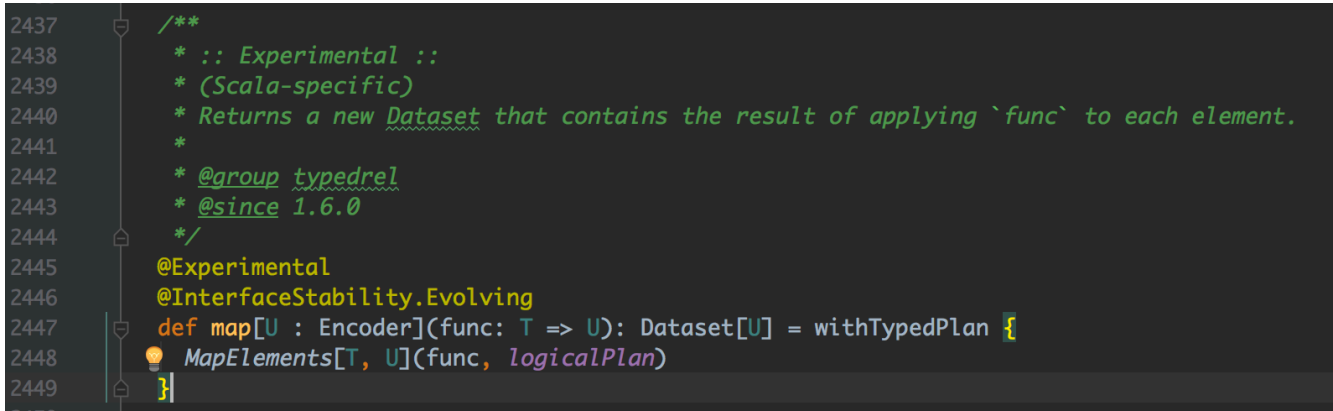
Datasets

```
val lines = sqlContext.read.text("/wikipedia").as[String]

val words = lines

  .flatMap(_.split(" "))

  .filter(_ != "")
```

Dataset API

Runs on the same optimizer and execution engine as DataFrames

“Encoder” (context bounds) describes the structure of user-defined classes to Spark, and code-gens serializer.



What are Spark's structured APIs?

Multi-faceted APIs for different big data use cases:

- SQL: "lingua franca" of data analysis
- R / Python: data science
- Scala Dataset API: type safety for data engineering

Internals that achieve this:

- declarativity & data independence from databases – easy to optimize
- flexibility & parallelism from MapReduce – massively scalable & flexible

Future possibilities from decoupled frontend/backend

Spark as a fast, multi-core data collection library

- Spark running on my laptop is already much faster than Pandas

Spark as a performant streaming engine

Spark as a GPU/vectorized engine

All using the same API

No language is perfect, but things I wished were designed differently in Scala

(I realize most of them have trade-offs that are difficult to make)

Binary Compatibility

Scala's own binary compatibility (2.9 -> 2.10 -> 2.11 -> 2.12 . . .)

- Huge maintenance cost for PaaS provider (Databricks)

Case classes

- Incredibly powerful for internal use, but virtually impossible to guarantee forward compatibility (i.e. add a field)

Traits with default implementations

Java APIs

Spark defines one API usable for both Scala and Java

- Everything needs to be defined twice (APIs, tests)
- Have to use weird return types, e.g. array
- Docs don’t work for Java
- Kotlin’s idea to reuse Java collection library can simplify this (although it might come with other hassles)

Exception Handling

Often use lots of Java libraries, especially for disk I/O, network

No good way to ensure exceptions are handled correctly:

- Create Scala shims for all libraries to turn return types into Try's
- Write low level I/O code in Java and rely on checked exceptions

Tooling so project can be more opinionated

Need to restrict and enforce consistency

- Otherwise impossible to train 1000+ OSS contributors (or even 100+ employees) on all language features properly

Lack of **great** tooling to enforce standards or disable features

Recap

Latest Spark take the best ideas out of earlier systems

- data frame from R as the “interface” – easy to learn
- declarativity & data independence from databases -- easy to optimize & future-proof
- parallelism from functional programming -- massively scalable & flexible

Scala’s a critical part of all of these!

Thank you & we are hiring!

@rxin