Functional Query Optimization with

SQL

Michael Armbrust
@michaelarmbrust

DATABRICKS

spark.apache.org

What is Apache Spark?

Fast and general cluster computing system interoperable with Hadoop

Improves efficiency through:

» In-memory computing primitives
» General computation graphs

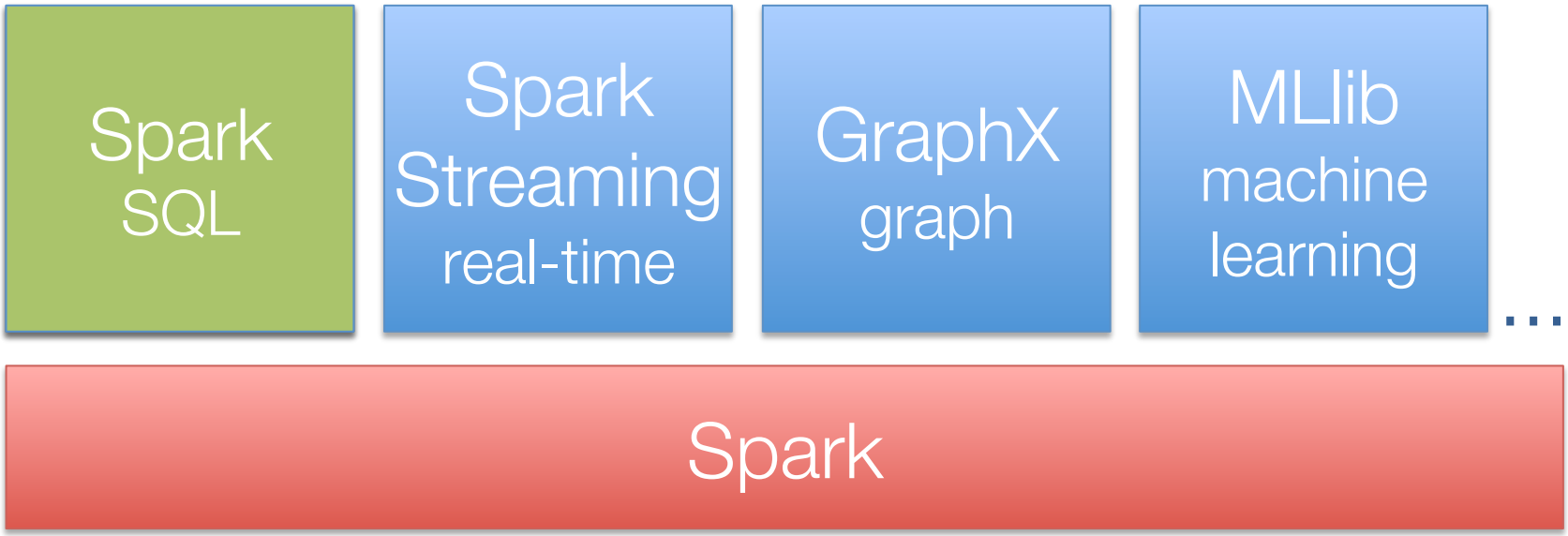
→ Up to 100× faster (2-10× on disk)

Improves usability through:

» Rich APIs in Scala, Java, Python
» Interactive shell

→ 2-5× less code

A General Stack



Spark Model

Write programs in terms of transformations on distributed datasets

Resilient Distributed Datasets (RDDs)

- » Collections of objects that can be stored in memory or disk across a cluster
- » Parallel functional transformations (map, filter, …)
- » Automatically rebuilt on failure

More than Map/Reduce

`map`	`reduce`	`sample`
`filter`	`count`	`take`
`groupBy`	`fold`	`first`
`sort`	`reduceByKey`	`partitionBy`
`union`	`groupByKey`	`mapWith`
`join`	`cogroup`	`pipe`
`leftOuterJoin`	`cross`	`save`
`rightOuterJoin`	`zip`	`...`

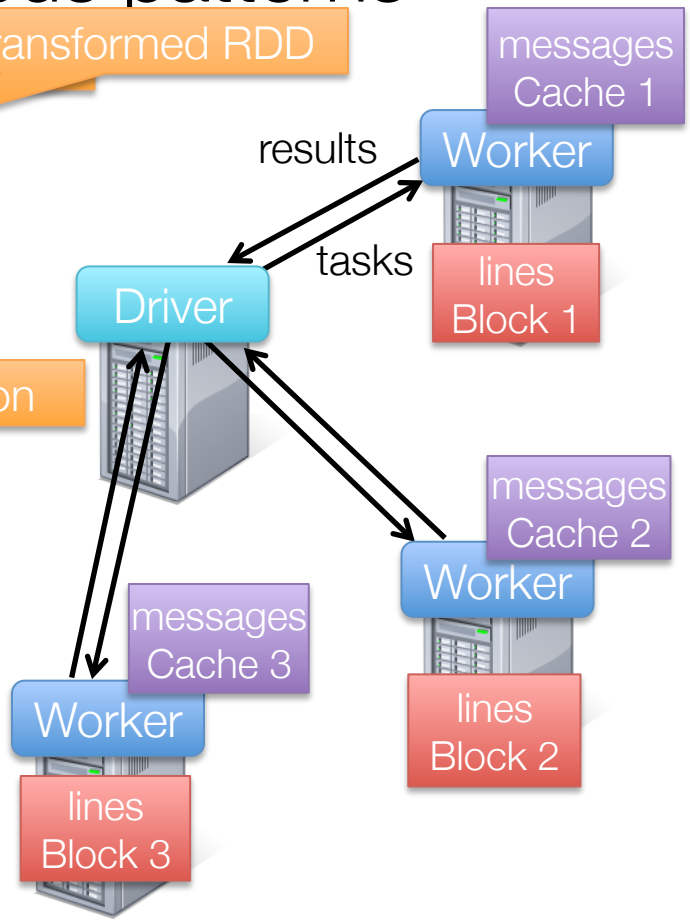
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile(“hdfs://...”)
val errors = lines.filter(_ startswith “ERROR”)
val messages = errors.map(_.split(“\t”)(2))
messages.cache()

messages.filter(_ contains “foo”).count()
messages.filter(_ contains “bar”).count()
. . .
```

Ba… Transformed RDD

Action

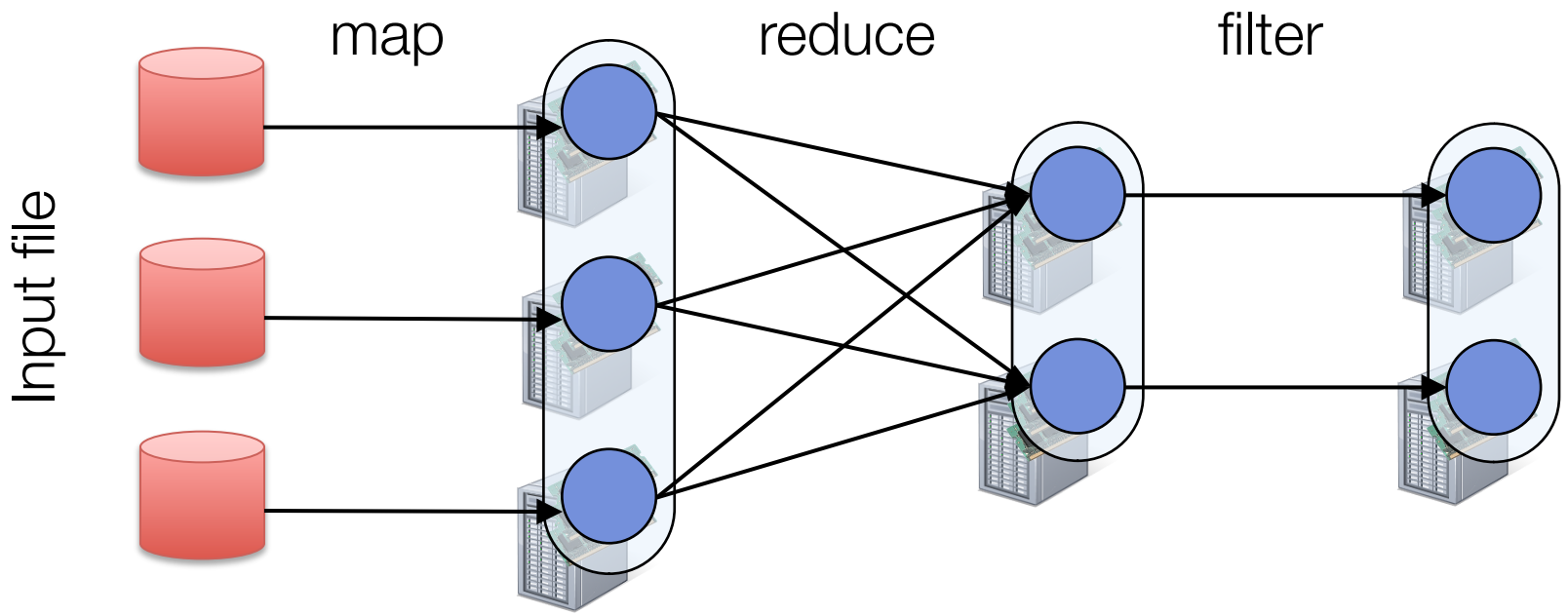


Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Fault Tolerance

RDDs track *lineage* info to rebuild lost data

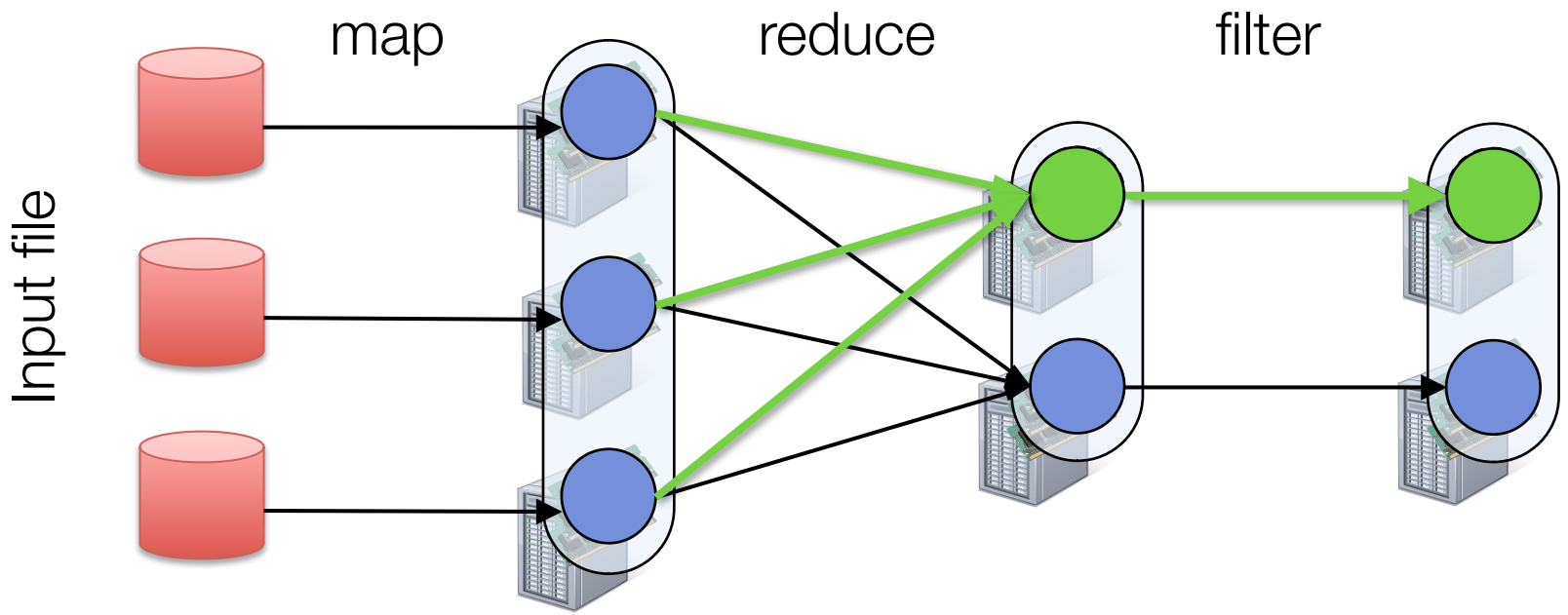
```
file.map(record => (record.tpe, 1))
    .reduceByKey(_ + _)
    .filter { case (_, count) => count > 10 }
```



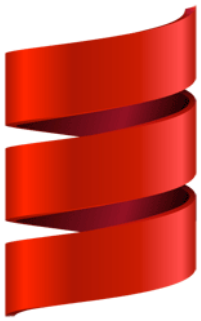
Fault Tolerance

RDDs track *lineage* info to rebuild lost data

```
file.map(record => (record.tpe, 1))
    .reduceByKey(_ + _)
    .filter { case (_, count) => count > 10 }
```



and

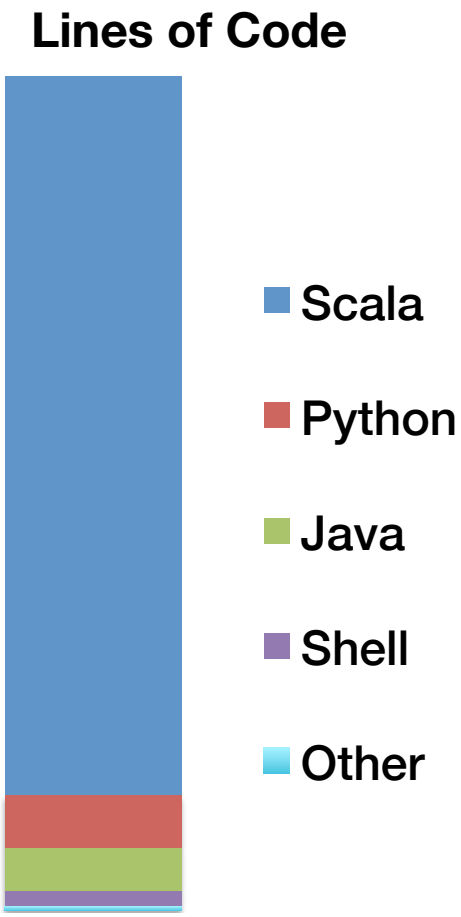


Scala

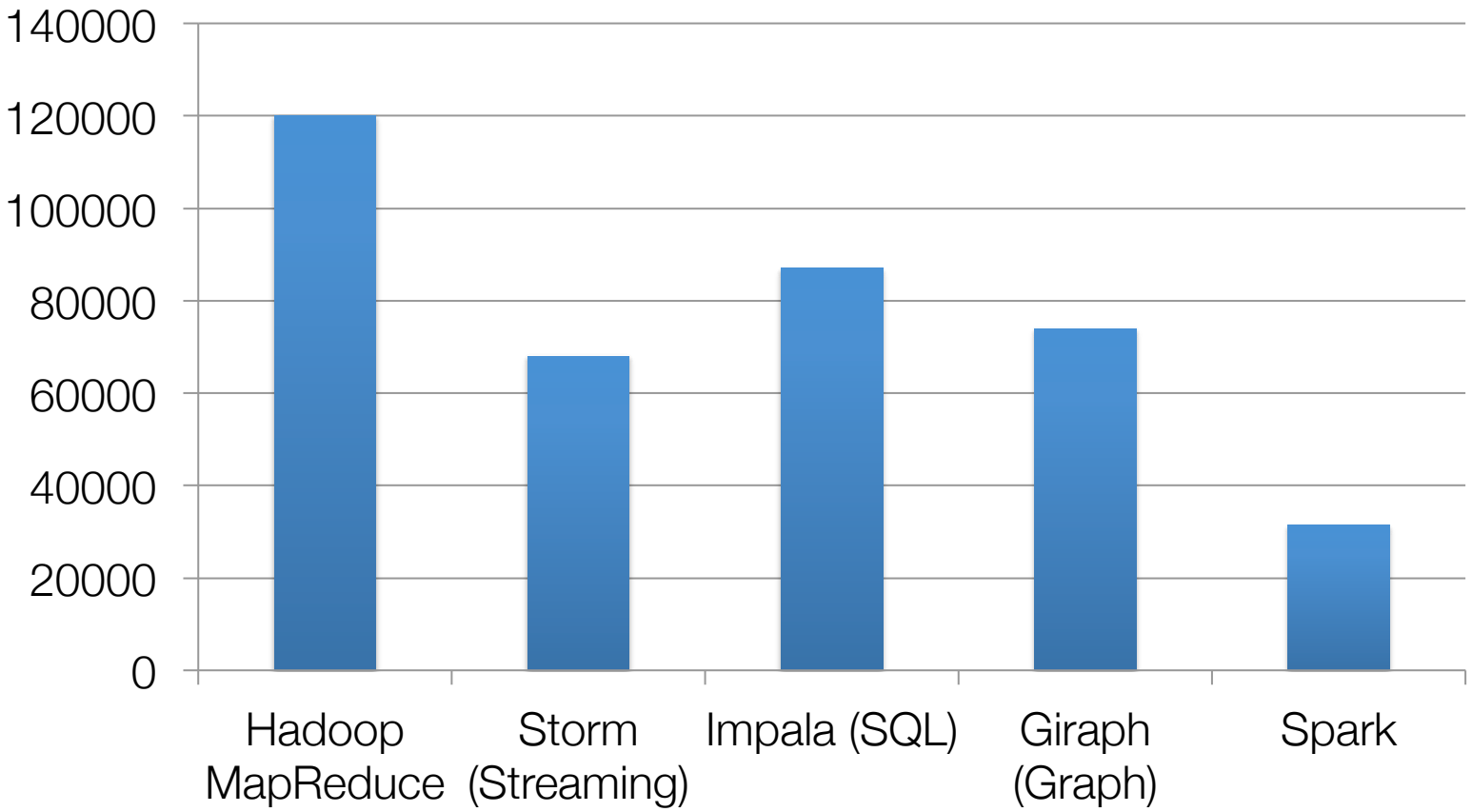
Scala Provides:

- Concise Serializable* Functions
- Easy interoperability with the Hadoop ecosystem
- Interactive REPL

* Made even better by Spores (5pm today)

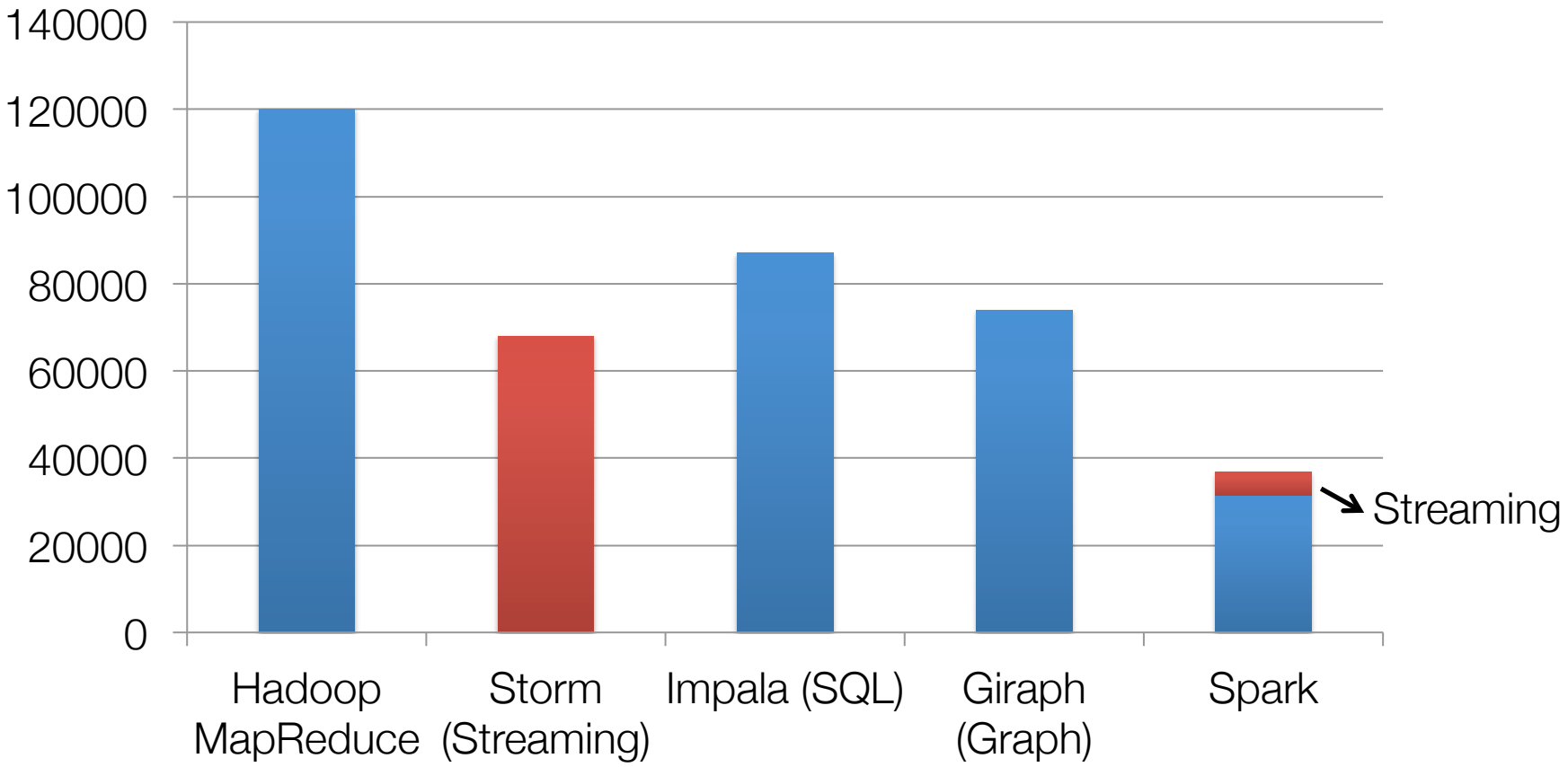


Reduced Developer Complexity



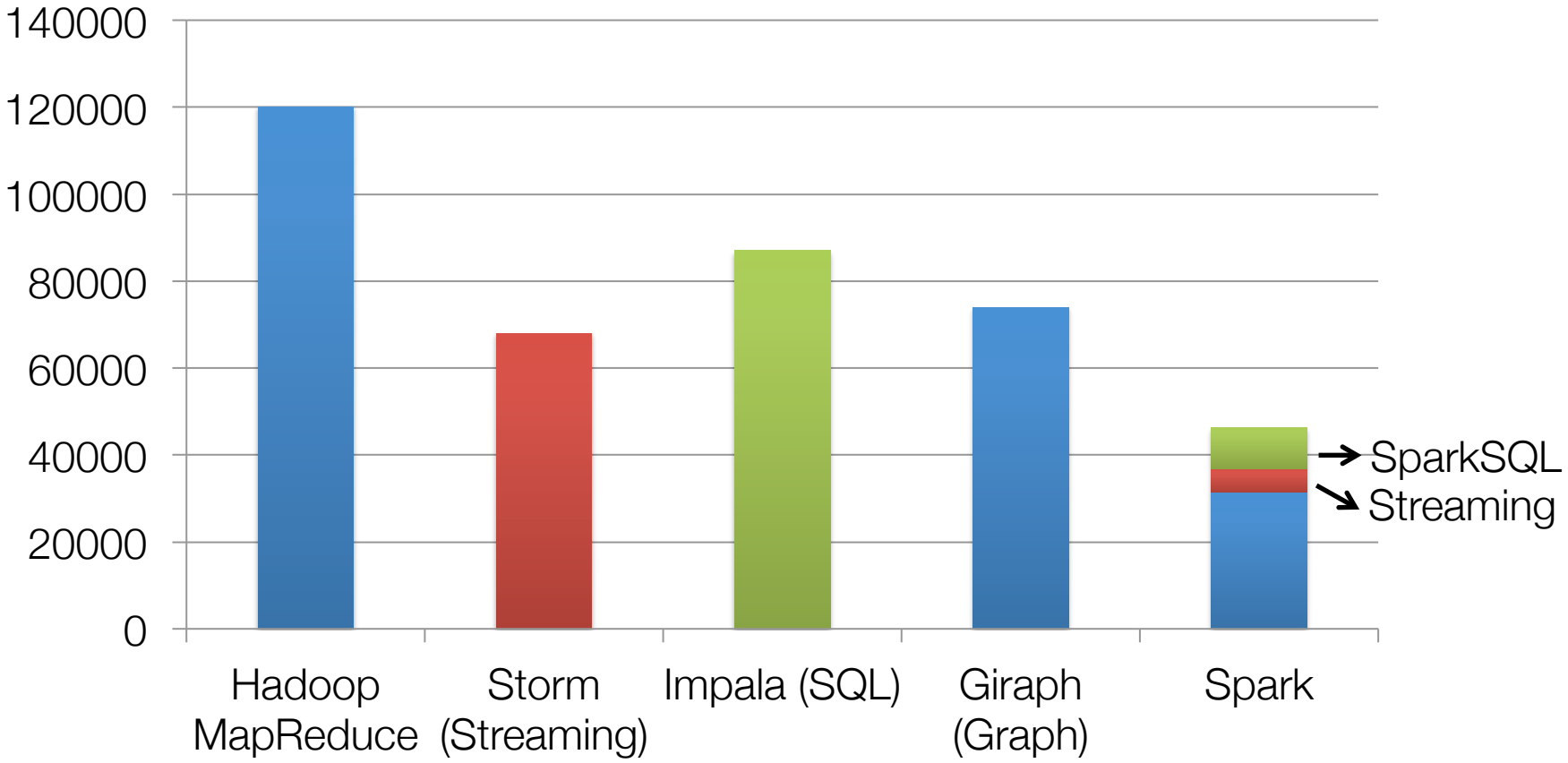
non-test, non-example source lines

Reduced Developer Complexity



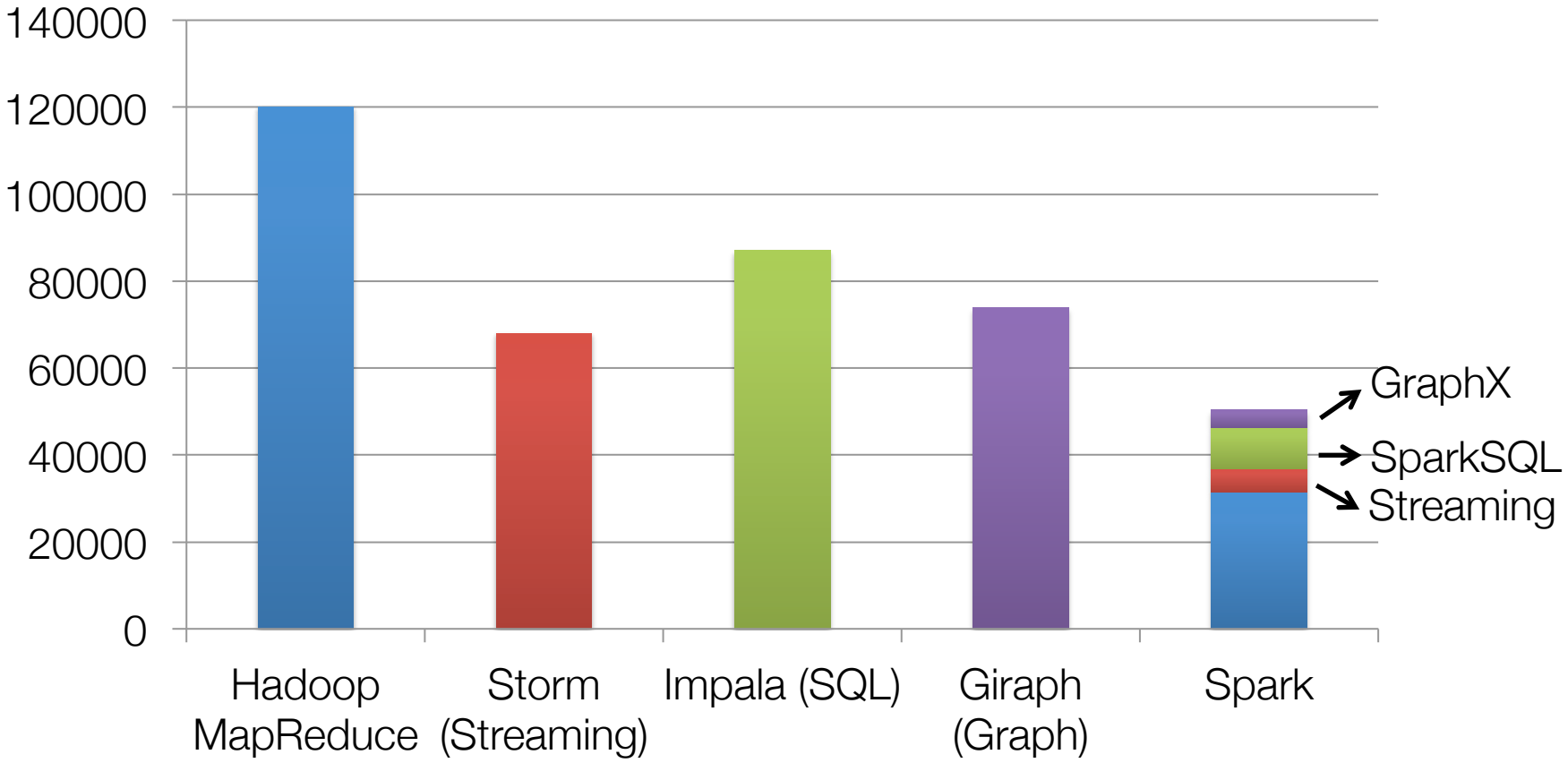
non-test, non-example source lines

Reduced Developer Complexity



non-test, non-example source lines

Reduced Developer Complexity



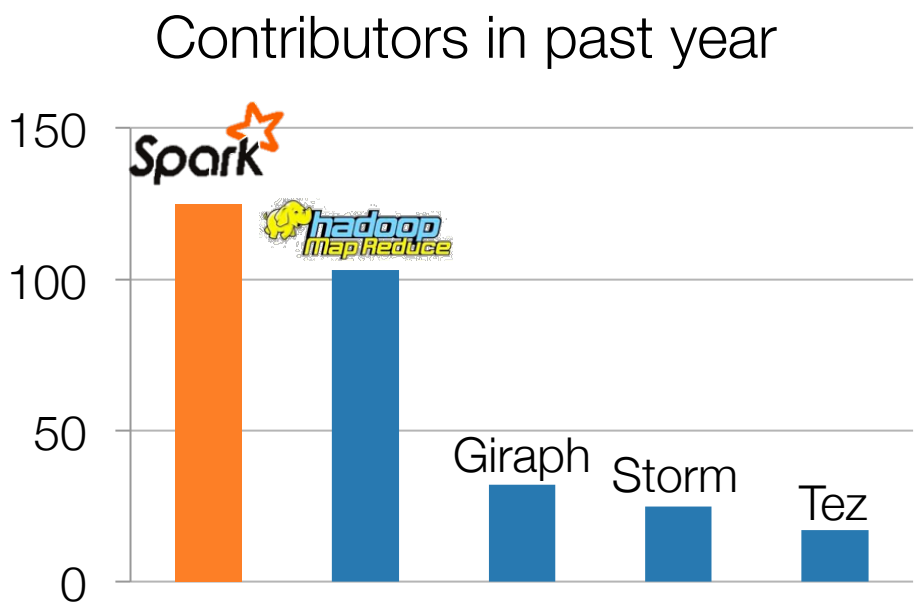
non-test, non-example source lines

Spark Community

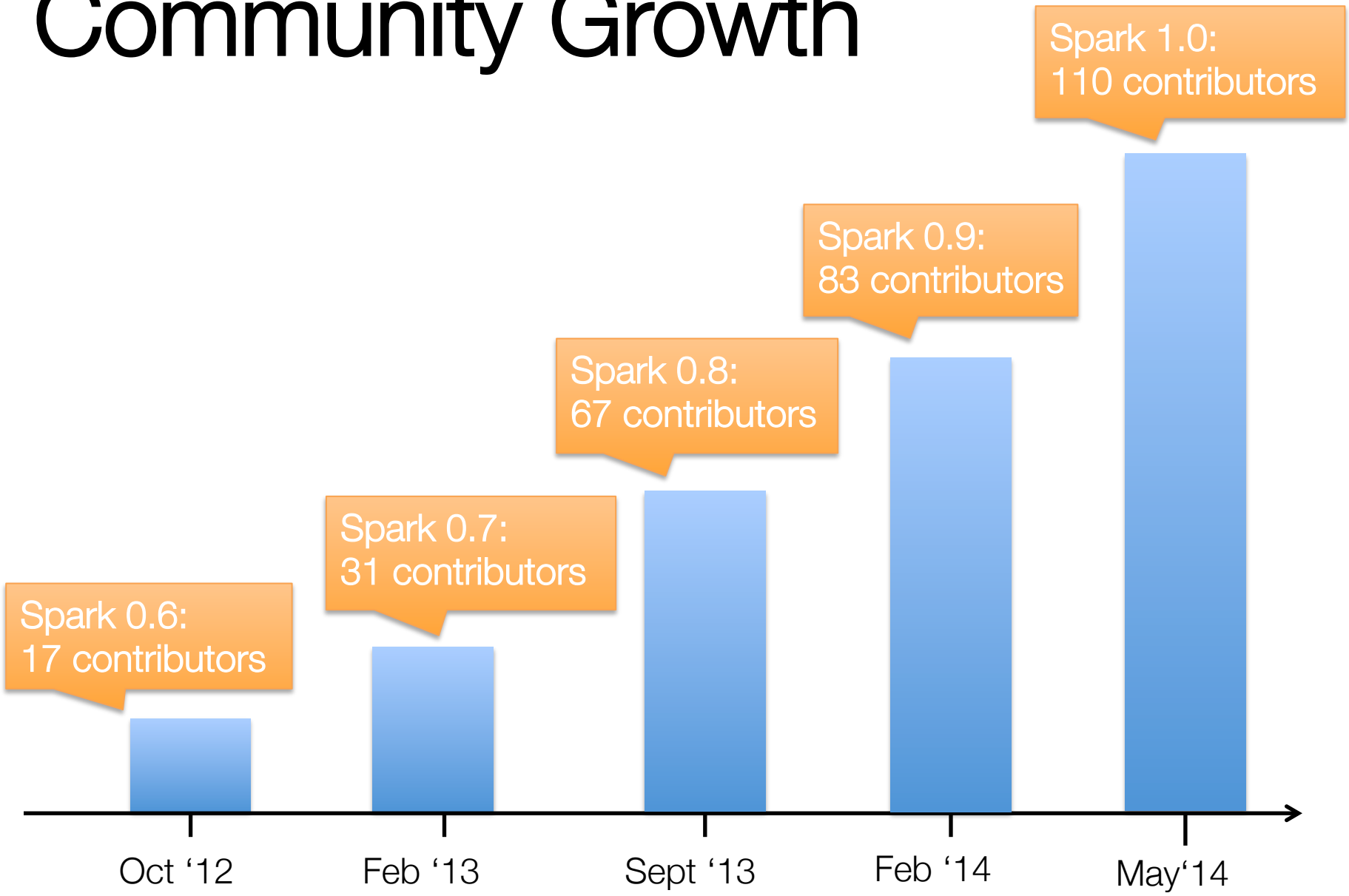
One of the largest open source projects in big data

150+ developers contributing

30+ companies contributing



Community Growth



With great power…

Strict project coding guidelines to make it easier for non-Scala users and contributors:

- Absolute imports only
- Minimize infix function use
- Java/Python friendly wrappers for user APIs
- …


Spark SQL

Relationship to 

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 the best parts of Shark:

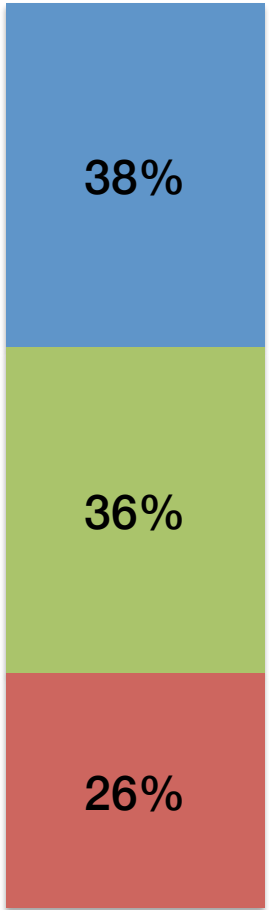
Borrows

- Hive data loading
- In-memory column store

Adds

- RDD-aware optimizer
- Rich language interfaces

Spark SQL Components

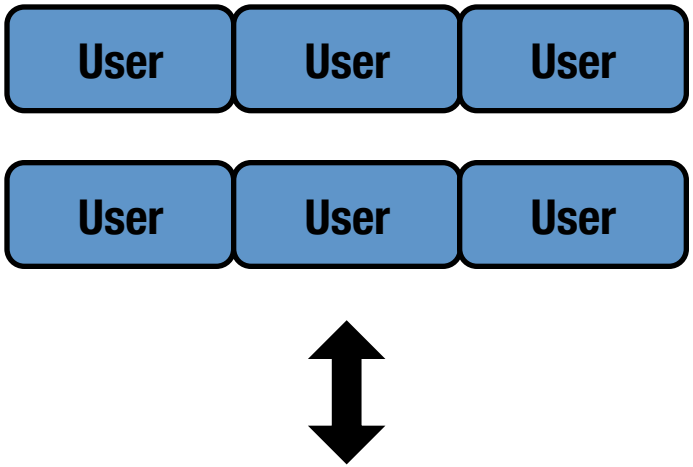


- Catalyst Optimizer
  - Relational algebra + expressions
  - Query optimization
- Spark SQL Core
  - Execution of queries as RDDs
  - Reading in Parquet, JSON …
- Hive Support
  - HQL, MetaStore, SerDes, UDFs

Adding Schema to RDDs

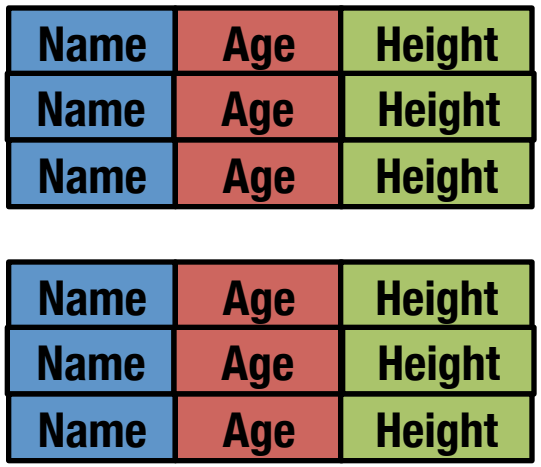
Spark + RDDs

Functional transformations on partitioned collections of **opaque** objects.



SQL + SchemaRDDs

Declarative transformations on partitioned collections of **tuples.**



Using Spark SQL

`SQLContext`

- Entry point for all SQL functionality
- Wraps/extends existing spark context

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// Importing the SQL context gives access to all the SQL
functions and conversions.
import sqlContext._
```

Example Dataset

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

```
Michael, 30

Andy, 31

Justin Bieber, 19

…
```

Turning an RDD into a Relation

```
// 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")
```

Querying Using SQL

```
// SQL statements are run with the sql method from sqlContext.
val teenagers = sql("""
    SELECT name FROM people WHERE age >= 13 AND age <= 19""")

// The results of SQL queries are SchemaRDDs but also
// support normal RDD operations.
// The columns of a row in the result are accessed by ordinal.
val nameList = teenagers.map(t => "Name: " + t(0)).collect()
```

Querying Using the Scala DSL

Express queries using functions, instead of SQL strings.

```
// The following is the same as:
//   SELECT name FROM people
//   WHERE age >= 10 AND age <= 19

val teenagers =
  people
    .where('age >= 10)
    .where('age <= 19)
    .select('name)
```

Caching Tables In-Memory

Spark SQL can cache tables using an in-memory columnar format:

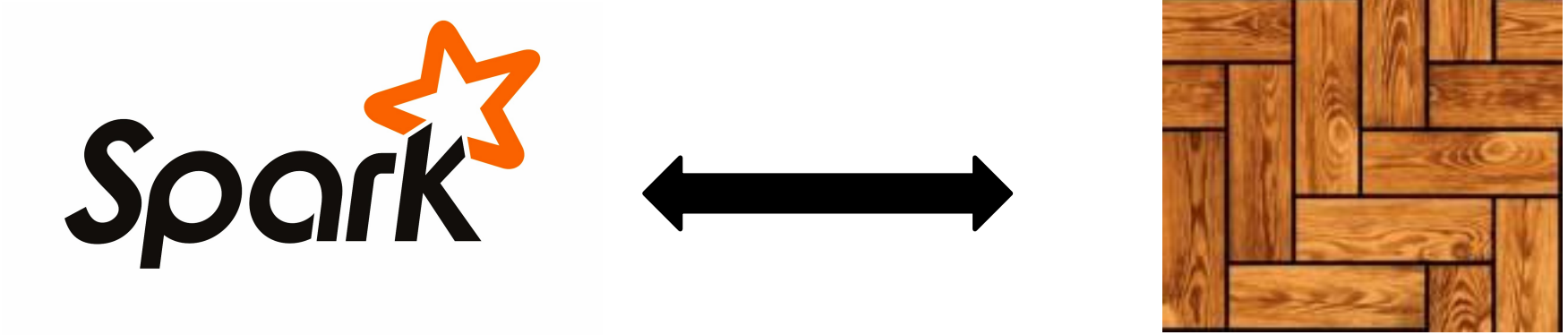
- Scan only required columns
- Fewer allocated objects (less GC)
- Automatically selects best compression

```
cacheTable("people")
```

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.



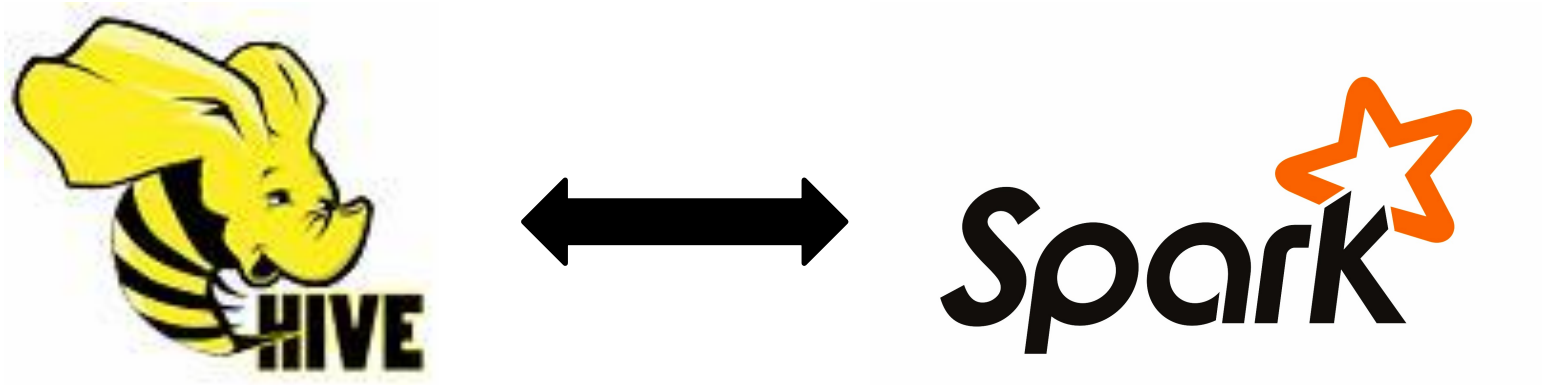
Using Parquet

```
// Any SchemaRDD can be stored as Parquet.
people.saveAsParquetFile("people.parquet")

// Parquet files are self-describing so the schema is preserved.
val parquetFile = sqlContext.parquetFile("people.parquet")

// Parquet files can also be registered as tables and then used
// in SQL statements.
parquetFile.registerAsTable("parquetFile”)
val teenagers = sql(
  "SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
```

Hive Compatibility

Interfaces to access data and code in the Hive ecosystem:

- Support for writing queries in HQL
- Catalog info from Hive MetaStore
- Tablescan operator that uses Hive SerDes
- Wrappers for Hive UDFs, UDAFs, UDTFs



Reading Data Stored In Hive

```
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)

import hiveContext._


hql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
hql("LOAD DATA LOCAL INPATH '.../kv1.txt' INTO TABLE src")


// Queries can be expressed in HiveQL.
hql("FROM src SELECT key, value")
```

SQL and Machine Learning

```
val trainingDataTable = sql("""
  SELECT e.action, u.age, u.latitude, u.logitude
    FROM Users  u
    JOIN Events e ON u.userId = e.userId""")


// SQL results are RDDs so can be used directly in Mllib.
val trainingData = trainingDataTable.map { row =>
  val features = Array[Double](row(1), row(2), row(3))
  LabeledPoint(row(0), features)
}
val model = new LogisticRegressionWithSGD().run(trainingData)
```

Supports Java Too!

```
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);
```

Supports Python Too!

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

lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: {"name": p[0], "age": int(p[1])})

peopleTable = sqlCtx.applySchema(people)
peopleTable.registerAsTable("people")

teenagers = sqlCtx.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
teenNames = teenagers.map(lambda p: "Name: " + p.name)
```

Optimizing Queries with C  talyst

What is Query Optimization?

SQL is a *declarative* language:

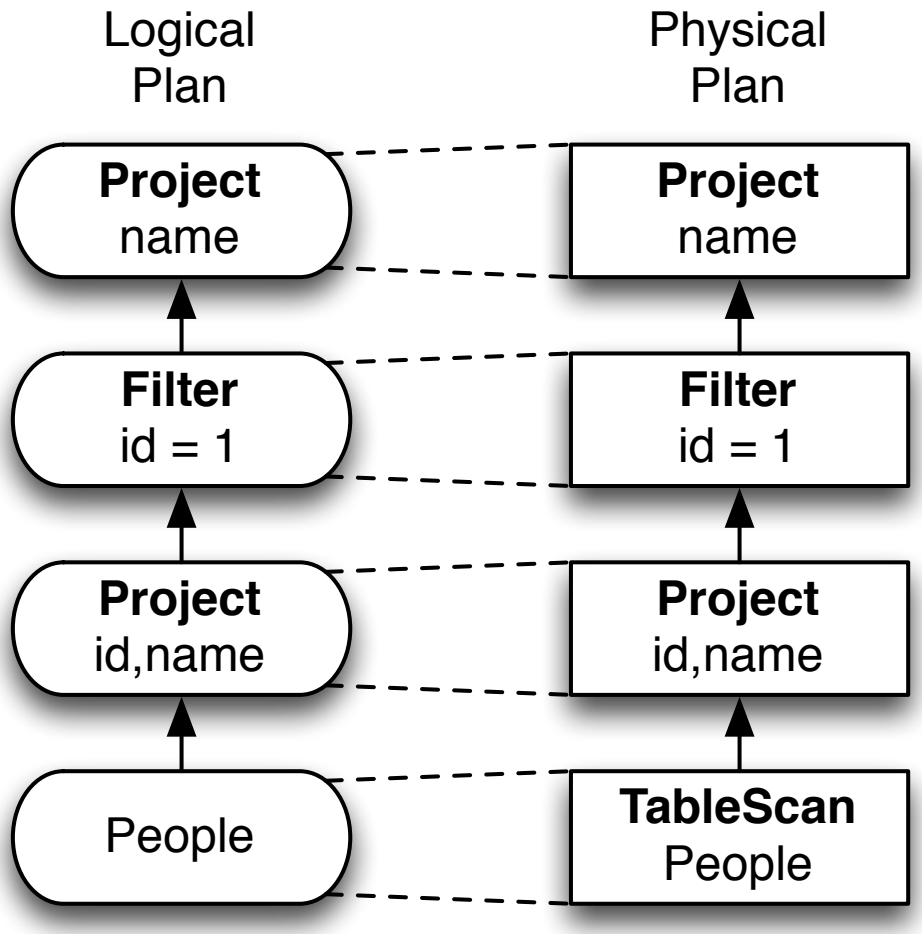
Queries express ***what*** data to retrieve,

not how to retrieve it.

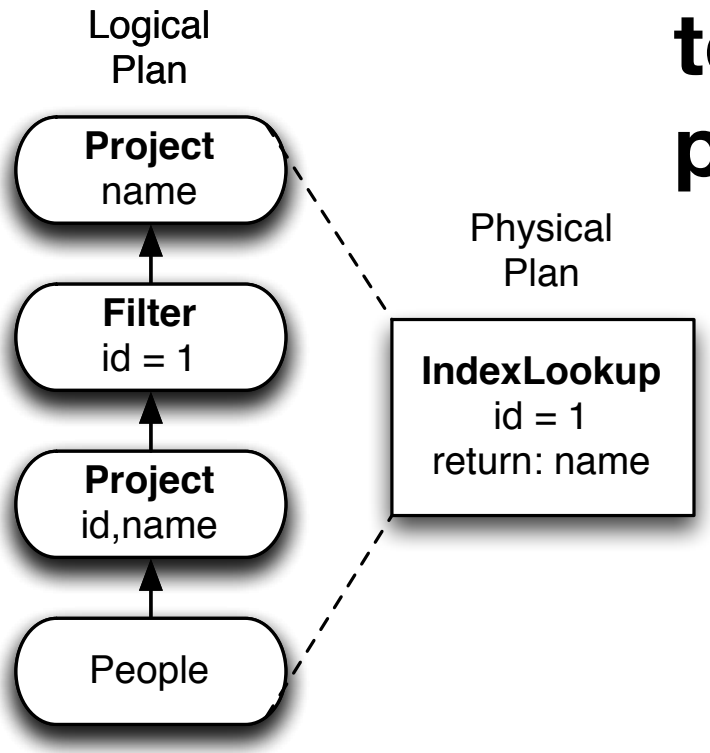
The database must pick the ‘best’ execution strategy through a process known as optimization.

Naïve Query Planning

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



Optimized Execution

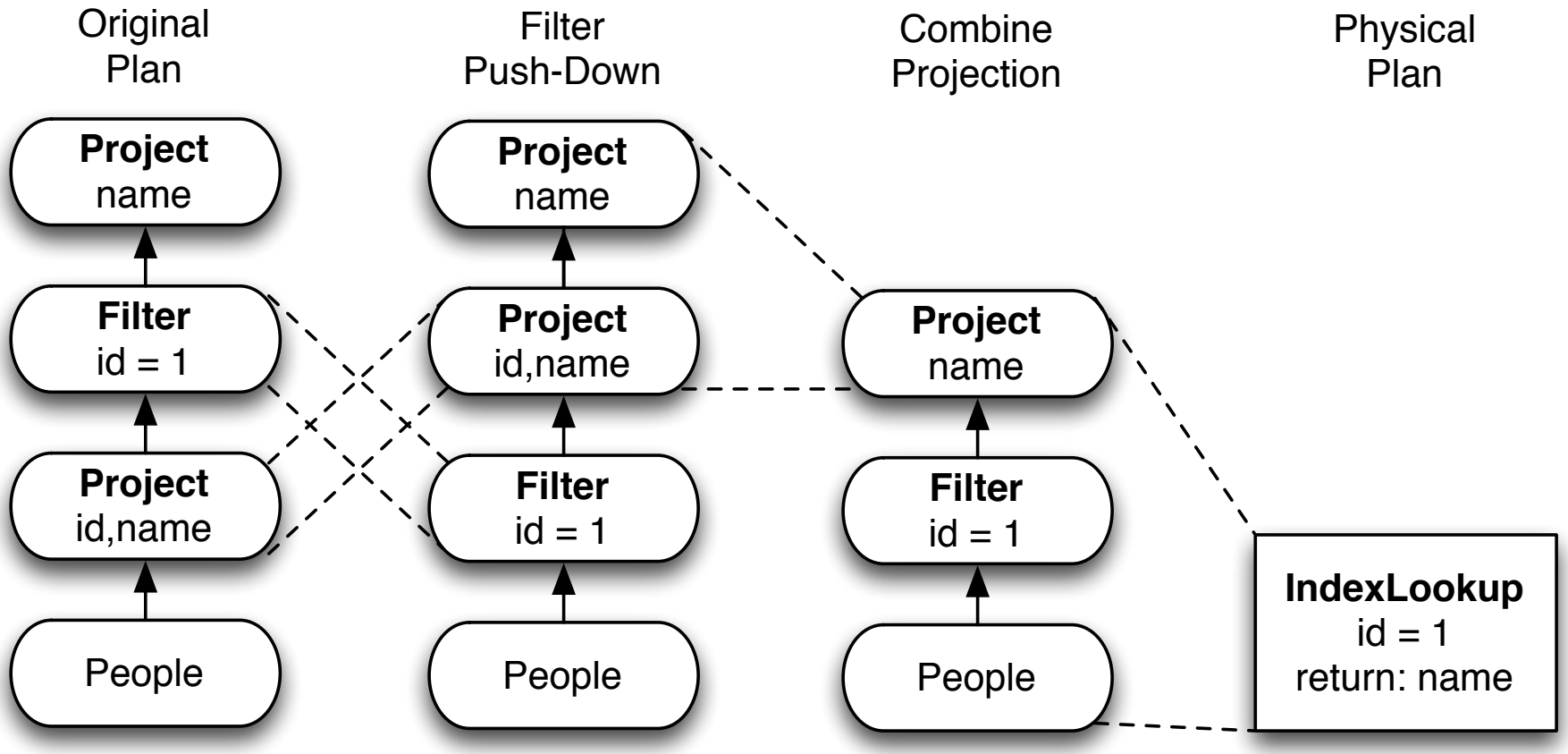


Writing imperative code to optimize all possible patterns is hard.

Instead write simple rules:

- Each rule makes one change
- Run many rules together to fixed point.

Optimizing with Rules



Prior Work: Optimizer Generators

Volcano / Cascades:

- Create a custom language for expressing rules that rewrite trees of relational operators.
- Build a compiler that generates executable code for these rules.

Cons: Developers need to learn this custom language. Language might not be powerful enough.

TreeNode Library

Easily transformable trees of operators

- Standard collection functionality - `foreach, map,collect,`etc.
- `transform` function – recursive modification of tree fragments that match a pattern.
- Debugging support – pretty printing, splicing, etc.

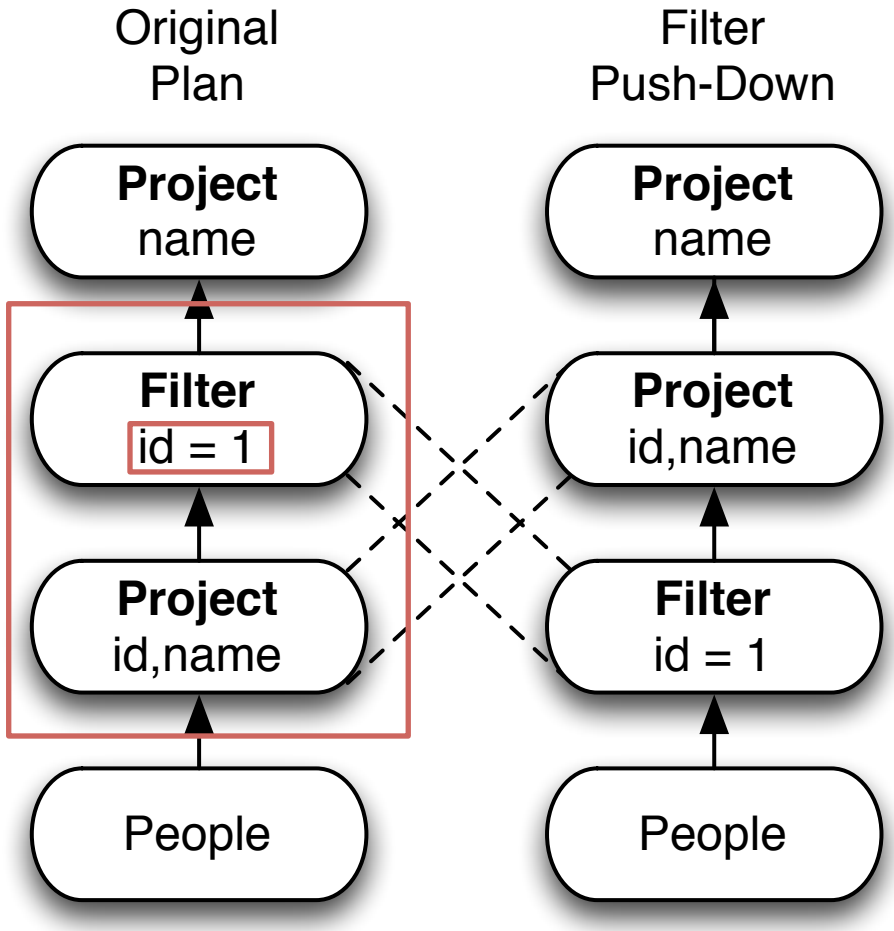
Tree Transformations

Developers express tree transformations as `PartialFunction[TreeType,TreeType]`

1. If the function **does apply** to an operator, that operator is **replaced** with the result.
2. When the function **does not apply** to an operator, that operator is **left unchanged**.
3. The transformation is **applied recursively** to all children.

Writing Rules as Tree Transformations

1. Find filters on top of projections.
2. Check that the filter can be evaluated without the result of the project.
3. If so, switch the operators.



Filter Push Down Transformation

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

Filter Push Down Transformation


```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

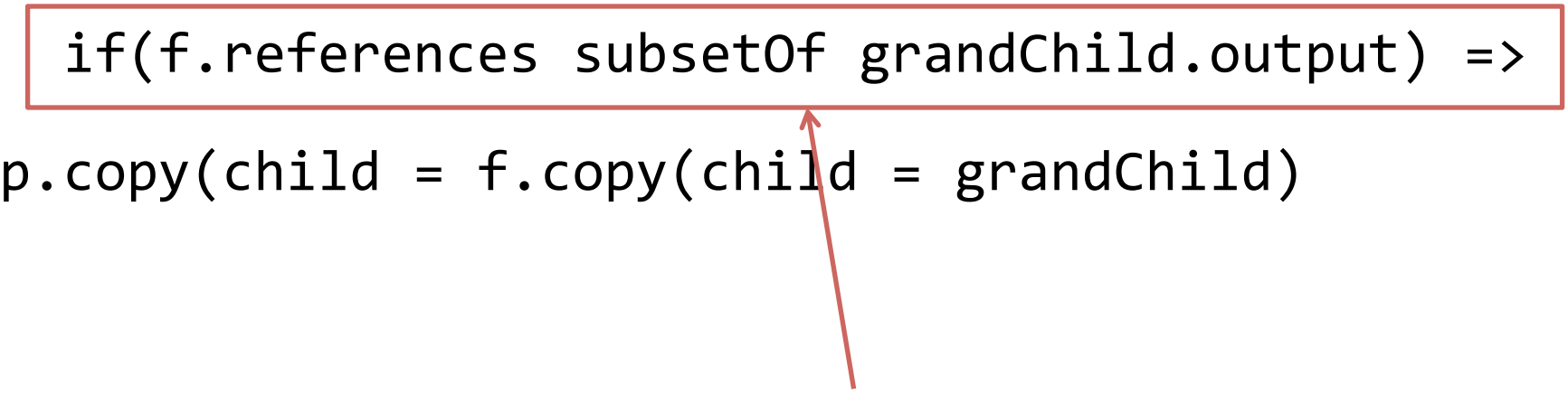
Filter Push Down Transformation

Find Filter on Project

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

Filter Push Down Transformation

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

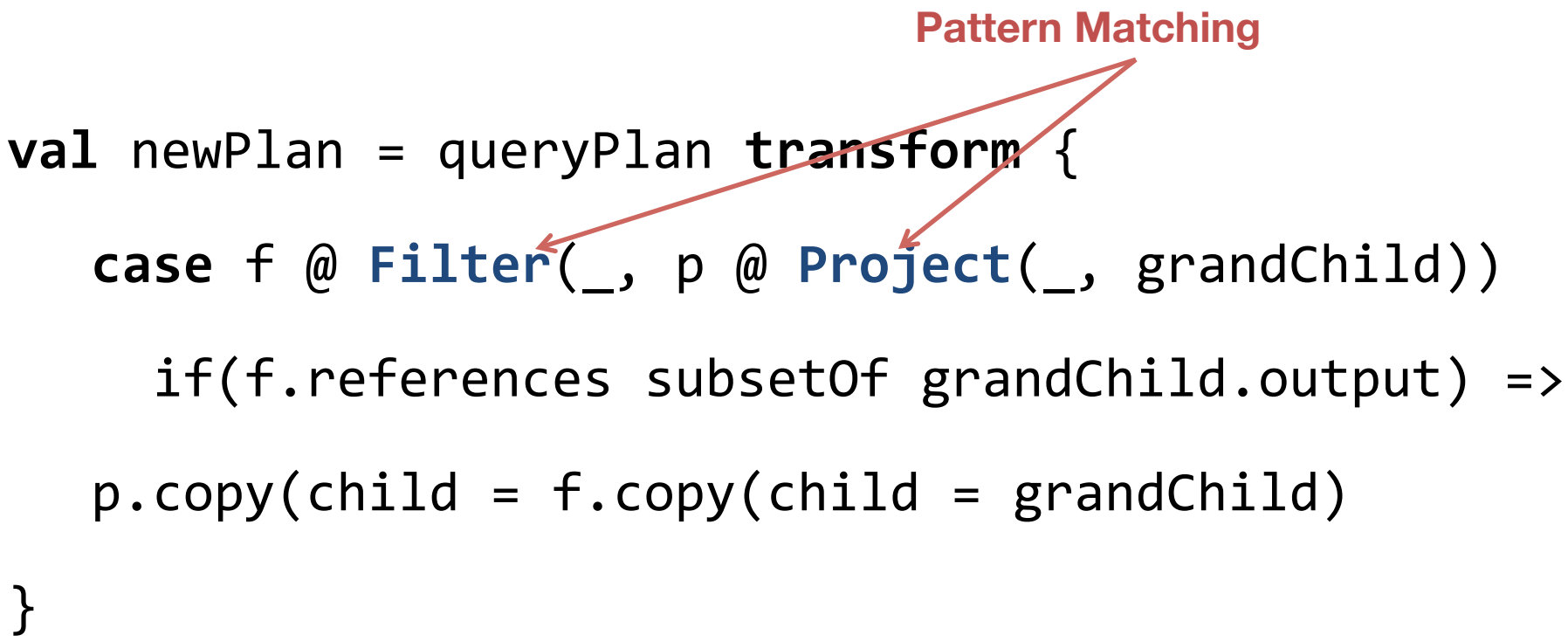


Filter Push Down Transformation

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild))
}
```

If so, switch the order.

Filter Push Down Transformation



Filter Push Down Transformation

Collections library

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

Filter Push Down Transformation

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

Copy Constructors

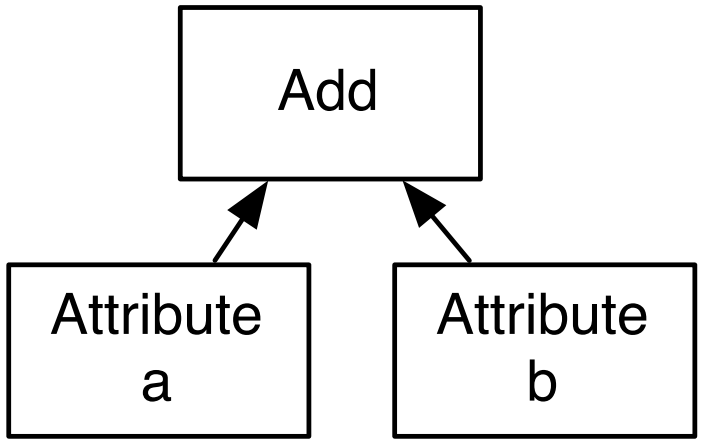
Efficient Expression Evaluation

Interpreting expressions (e.g., ‘a + b’) can very expensive on the JVM:

- Virtual function calls
- Branches based on expression type
- Object creation due to primitive boxing
- Memory consumption by boxed primitive objects

Interpreting “a+b”

1. Virtual call to Add.eval()
2. Virtual call to a.eval()
3. Return boxed Int
4. Virtual call to b.eval()
5. Return boxed Int
6. Integer addition
7. Return boxed result



Using Runtime Reflection

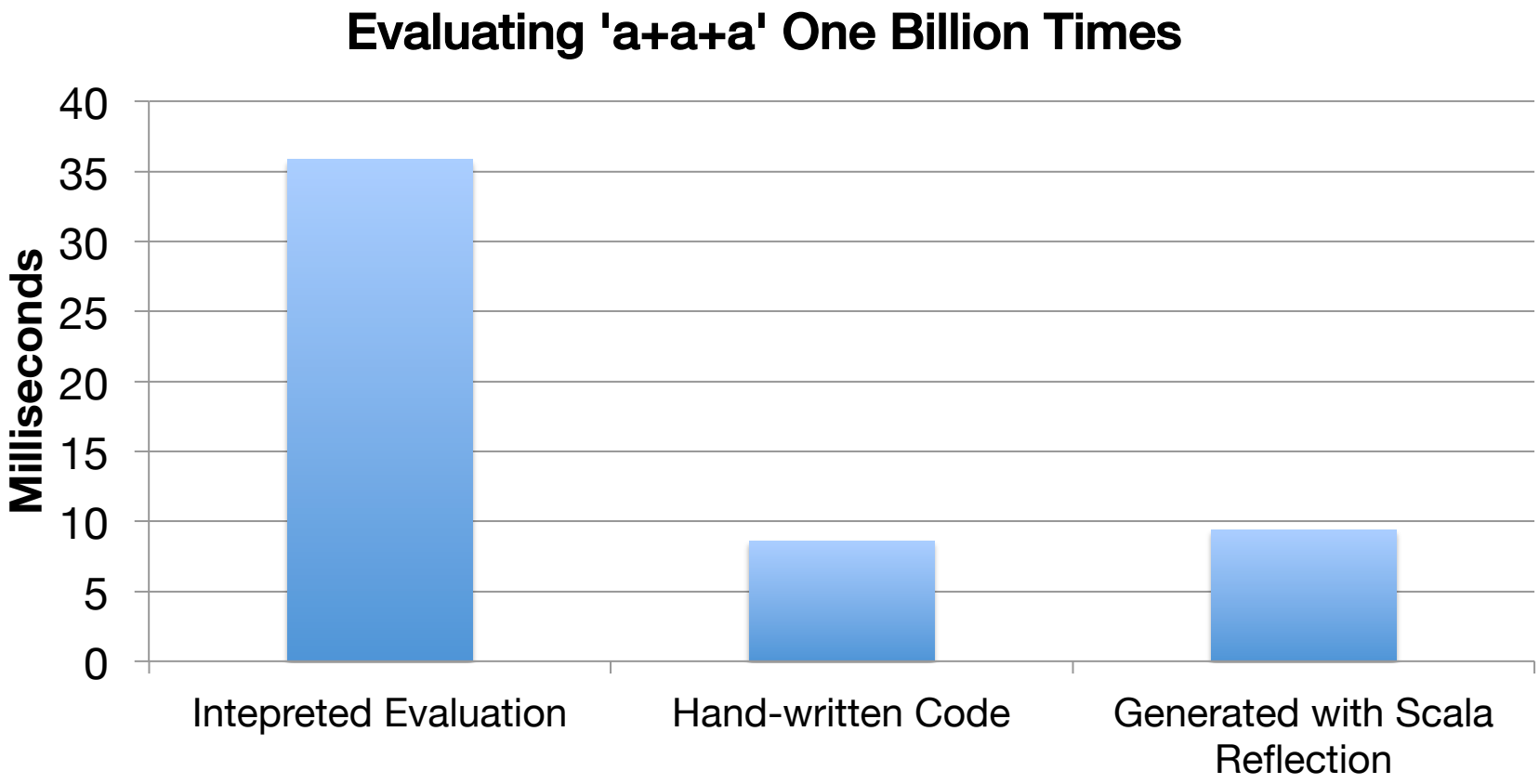
```
def generateCode(e: Expression): Tree = e match {
  case Attribute(ordinal) =>
    q"inputRow.getInt($ordinal)"
  case Add(left, right) =>
    q"""
      {
        val leftResult = ${generateCode(left)}
        val rightResult = ${generateCode(right)}
        leftResult + rightResult
      }
    """
}
```

Executing “a + b”

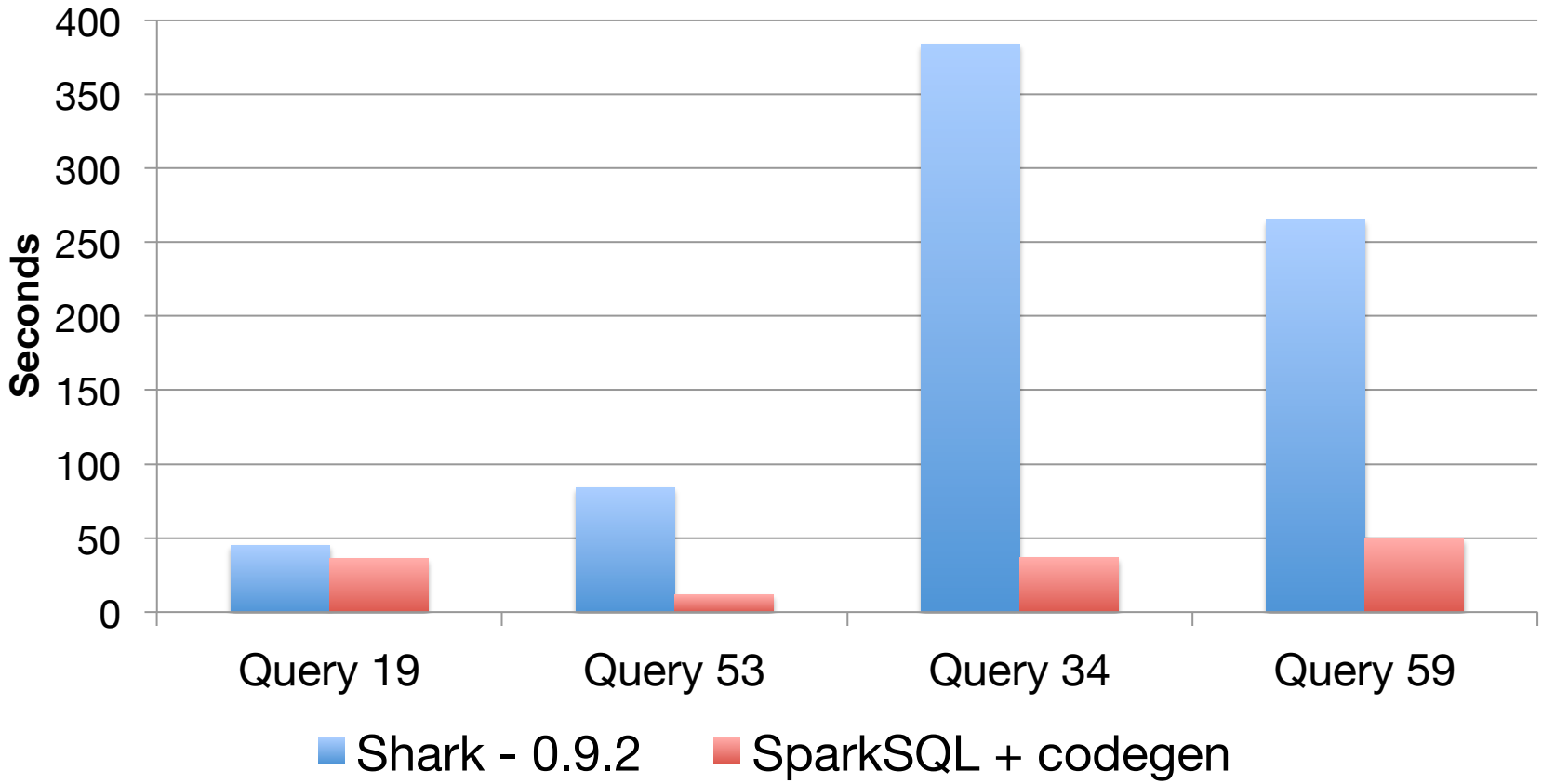
```
val left: Int = inputRow.getInt(0)
val right: Int = inputRow.getInt(1)
val result: Int = left + right
resultRow.setInt(0, result)
```

- Fewer function calls
- No boxing of primitives

Performance Comparison



TPC-DS Results



Code Generation Made Simple

- Other Systems (Impala, Drill, etc) do code generation.
- **Scala Reflection + Quasiquotes** made our implementation an experiment done over a few weekends instead of a major system overhaul.

Initial Version ~1000 LOC

Future Work: Typesafe Results

Currently:

```
people.registerAsTable(“people”)
val results = sql(“SELECT name FROM people”)
results.map(r => r.getString(0))
```

Joint work with: Heather Miller, Vojin Jovanovic, Hubert Plociniczak

Future Work: Typesafe Results

Currently:

```
people.registerAsTable(“people”)
val results = sql(“SELECT name FROM people”)
results.map(r => r.getString(0))
```

Joint work with: Heather Miller, Vojin Jovanovic, Hubert Plociniczak

Future Work: Typesafe Results

Currently:

```
people.registerAsTable(“people”)
val results = sql(“SELECT name FROM people”)
results.map(r => r.getString(0))
```

What we want:

```
val results = sql“SELECT name FROM $people”
results.map(_.name)
```

Joint work with: Heather Miller, Vojin Jovanovic, Hubert Plociniczak

Get Started

Visit spark.apache.org for videos & tutorials

Download Spark bundle for CDH

Easy to run on just your laptop

Free training talks and hands-on exercises: spark-summit.org

Conclusion

Big data analytics is evolving to include:

- More **complex** analytics (e.g. machine learning)
- More **interactive** ad-hoc queries, including SQL
- More **real-time** stream processing

Spark is a fast platform that *unifies* these apps

Join us at Spark Summit 2014!

June 30-July 2, San Francisco