Apache Spark: What’s New, What’s Coming

Patrick Wendell - @pwendell
May 2015 – Strata London
About Me

Founding committer of Spark at Berkeley AMPLab

Co-founder of Databricks, now manage Spark roadmap and community

Release manager for Spark 1.3 and 1.4
About Databricks

Founded by Apache Spark creators

Largest contributor to Spark project, committed to keeping Spark 100% open source

End-to-End hosted platform, Databricks Cloud
Show of Hands

A. New to Spark

B. Played around with Spark API in training or examples

C. Have written a full application on Spark (POC or production)
What Is Spark?

A fast and general execution engine for big data processing

Fast to write code
High level API’s in Python, Java, and Scala
Multi paradigm (streaming, batch, and interactive)

Fast to run code
Low overhead scheduling
Optimized engine
Can exploit in-memory when available
What is Spark?

Spark SQL  Spark Streaming  MLlib  GraphX  Packages

Spark Core

YARN  Standalone  Mesos  Databricks
### spark-packages.org

<table>
<thead>
<tr>
<th>API Extensions</th>
<th>Data Sources</th>
<th>Deployment Utilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clojure API</td>
<td>Avro</td>
<td>Google Compute</td>
</tr>
<tr>
<td>Spark Kernel</td>
<td>CSV</td>
<td>Microsoft Azure</td>
</tr>
<tr>
<td>Zeplin Notebook</td>
<td>Elastic Search</td>
<td>Spark Jobserver</td>
</tr>
<tr>
<td>Indexed RDD</td>
<td>MongoDB</td>
<td></td>
</tr>
</tbody>
</table>

> ./bin/spark-shell --packages databricks/spark-avro:0.2
Spark Today

Embraced by Hadoop community…

But also beyond Hadoop…
Contributors per Month to Spark

Most active project at Apache,
More than 500 known production deployments
What’s New In Spark?
Some Spark 1.4 and 1.5 Initiatives

**MLlib**
- Pipelines API for machine learning
- Dozens of new algorithms/utils

**Streaming**
- Metric viz and monitoring
- Deeper Kafka integrations

**Spark SQL**
- Support all Hive metastore versions
- HQL/SQL coverage (window functions, etc)

**DataFrames**
- Storage integrations
- Math and stats functions
- Code generation

**Core Runtime**
- Managed memory
- Cache aware data structures
public static class WordCountMapClass extends MapReduceBase
implements Mapper<LongWritable, Text, Text, IntWritable> {

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

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
}

public static class WordCountReduce extends MapReduceBase
implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
                        OutputCollector<Text, IntWritable> output,
                        Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}

val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
RDD API

Most data is structured (JSON, CSV, Avro, Parquet, Hive …)
  • Programming RDDs inevitably ends up with a lot of tuples (_1, _2, …)

Functional transformations (e.g. map/reduce) are not as intuitive

Memory management with arbitrary Java objects is doable, but challenging
pdata.map(lambda x: (x.dept, [x.age, 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()

data.groupBy("dept").avg("age")
DataFrames in Spark

Distributed collection of data grouped into named columns (i.e. RDD with schema)

Domain-specific functions designed for common tasks

- Metadata
- Sampling
- Project, filter, aggregation, join, …
- UDFs

Available in Python, Scala, Java, and R
Demo (Time Permitting)
From DataFrames Spring…

1. A *Dataframe* model for expressive and concise programs
2. A pluggable *Datasource API* API for reading and writing data frames while minimizing IO
3. The *Catalyst* logical optimizer for speeding up data frame operations
4. *Project Tungsten* – optimized physical execution throughout Spark
Spark’s Data Source API can read and write DataFrames using a variety of formats.

**Built-In**
- { JSON }
- JDBC
- Parquet
- Hive
- MySQL
- PostgreSQL
- HDFS
- Amazon S3
- H2

**Packages**
- Avro
- CSV
- dBase
- elasticsearch
- Cassandra
- Amazon Redshift

and more...
Read Less Data

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

DataFrames can help you read less data automatically:

- Columnar formats can skip fields (i.e. parquet)
- Using partitioning (i.e., /year=2014/month=02/...)\(^1\)
- Skipping data using statistics (i.e., min, max)\(^2\)
- Pushing predicates into storage systems (i.e., JDBC)

\(^1\)Only supported for Parquet and Hive, more support coming in Spark 1.4 - \(^2\)Turned off by default in Spark 1.3
Plan Optimization & Execution

DataFrames and SQL share the same optimization/execution pipeline
Optimization happens as late as possible, therefore Spark SQL can optimize even across functions.
def add_demographics(events):
    u = sqlCtx.table("users")
    events \
        .join(u, events.user_id == u.user_id) \
        .withColumn("city", zipToCity(u.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()
def add_demographics(events):
    u = sqlCtx.table("users")
    events =
        .join(u, events.user_id == u.user_id)
        .withColumn("city", zipToCity(u.zip))
    # Load partitioned Hive table
    # Join on user_id
    # 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()
Physical Execution: Unified Across Languages

Time to Aggregate 10 million int pairs (secs)
Physical Execution: Fully Managed Memory

Spark’s core API uses raw Java objects and Java GC for aggregations and joins.

DataFrame’s will use a custom binary format and off-heap managed memory.

Both faster computationally and “GC-free”
Physical Execution:
CPU Efficient Data Structures

Keep data closure to CPU cache
Physical Execution:
CPU Efficient Data Structures
Other Optimizations

**Code Generation**
- Avoid interpretation overhead on records
- Already used in Spark, but will expand

**Vectorized record processing**
- Process in small batches to avoid function call overhead
- In some cases can exploit GPU’s
Spark Tomorrow vs. Today

**Today**
- RDD API
- Hadoop InputFormat
- Java objects for aggregations
- Java GC

**Tomorrow**
- DataFrame API
- Spark Data Source
- Spark binary format
- Spark managed memory
Learn More About Spark

Docs:  
http://spark.apache.org/docs/latest/

Site:  
http://spark.apache.org/

Databricks Cloud:  
http://go.databricks.com/register-for-dbc
Thank You!

Questions?