SPARK: THE BASICS

CHARITY HILTON

DEVELOPER UNIVERSITY

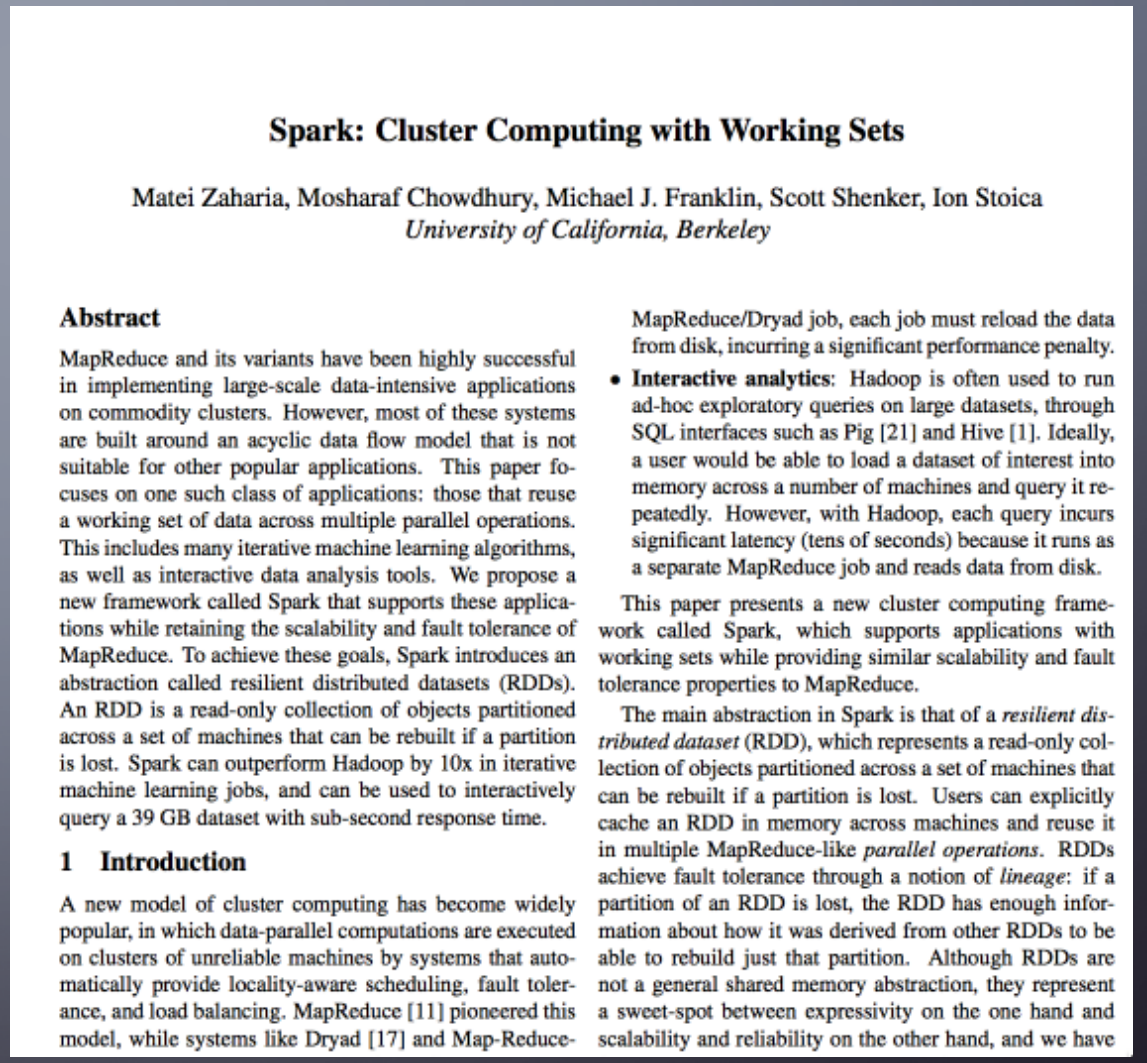
TOP 5 SPARK FACTS

1. Open source, Apache project
2. Fast, in-memory distributed data processing platform
3. Originated from Matei Zaharia at UC Berkeley's AMPLab
4. Written in Scala, but also supports Java, Python, R
5. Has outperformed Hadoop's MapReduce by 100 times on specific processing tasks

SPARK ORIGINS

SPARK ORIGINS

- Originally developed by Matei Zaharia, out of UC Berkeley's AMPLab, later donated to the Apache Foundation
- https://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf



Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

1 Introduction

A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. MapReduce [11] pioneered this model, while systems like Dryad [17] and Map-Reduce-MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.

- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *parallel operations*. RDDs achieve fault tolerance through a notion of *lineage*: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have

WHO USES SPARK?

SPARK USE CASES

Spark Streaming

Real-time data ETL

Capturing Real-time anomalies or trigger events

Run large scale analyses real-time

Spark MLLib

Provider user recommendations on real-time data (e.g. k-means or collaborative filtering)

Run algorithms like logistic regression out of the box

SPARK USE CASES

Spark SQL

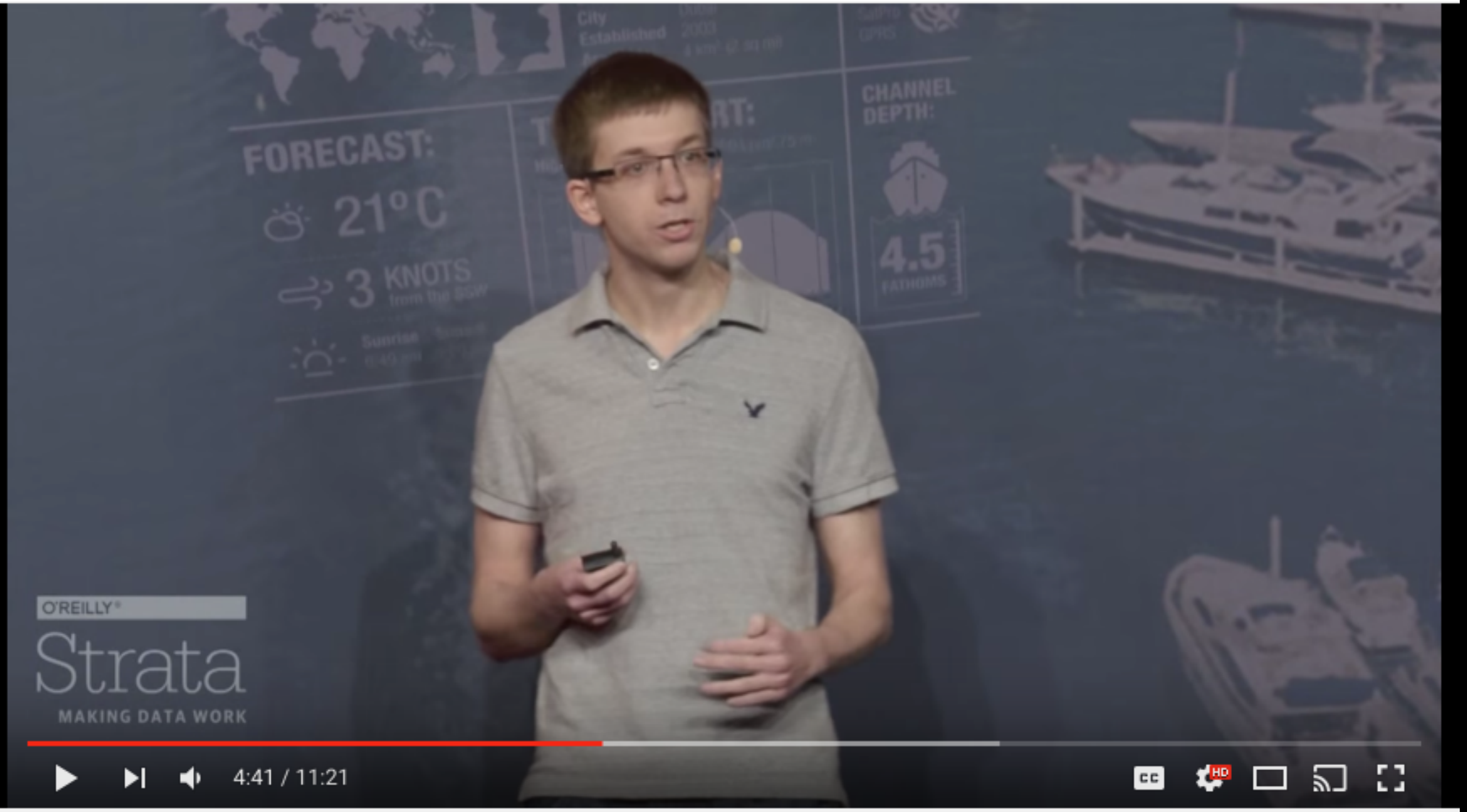
Move data from a relational database, analyze it, then move a big data store

'Everyone' knows SQL, bridges the gap between big data processing world and traditional data warehouses

Spark GraphX

Analyzing the fastest route to ship a package

Implementing a PageRank Algorithm for recommendations (evaluating the quality and quantity of links in web pages)



Strata 2014: Matei Zaharia, "How Companies are Using Spark, and Where the Edge in Big Data Will Be"

O'Reilly

Subscribed | 116,255

11,254 views

Up next Autoplay

"New Directions for Spark 2015" - Matei Zaharia (St

O'Reilly

6,072 views

Users

Distributors & Apps

USING SPARK WITH HEALTH DATA

PREDICTIVE ANALYTICS IN THE OR

- Beth Isreal Deaconess Medical Center + Databricks
- Improving Operating Room efficiency, $15-$20/minute for basic surgeries
- Built predictive model to determine OR availability with different services, emergencies, plan staffing resources, improve patient scheduling
- Originally built on SQL Server
- https://www.youtube.com/watch?v=ch2LXOsAkxE&index=6&list=PL-x35fyliRwhP52fwDqULJLOnqnrN5nDs

LARGE SCALE TEXT ANALYTICS

- Elsevier Labs and Databricks
- Data set contains TB of Text, and PB of images, videos, supplemental data
- Previously slow to integrate NLP systems with actual text, hard to use
- With Spark, flexibility of languages allows new users, easier to scale and update algorithms, easier to move data around
- http://www.textanalyticsworld.com/pdf/SF/2015/Day1_1405_Daniel.pdf

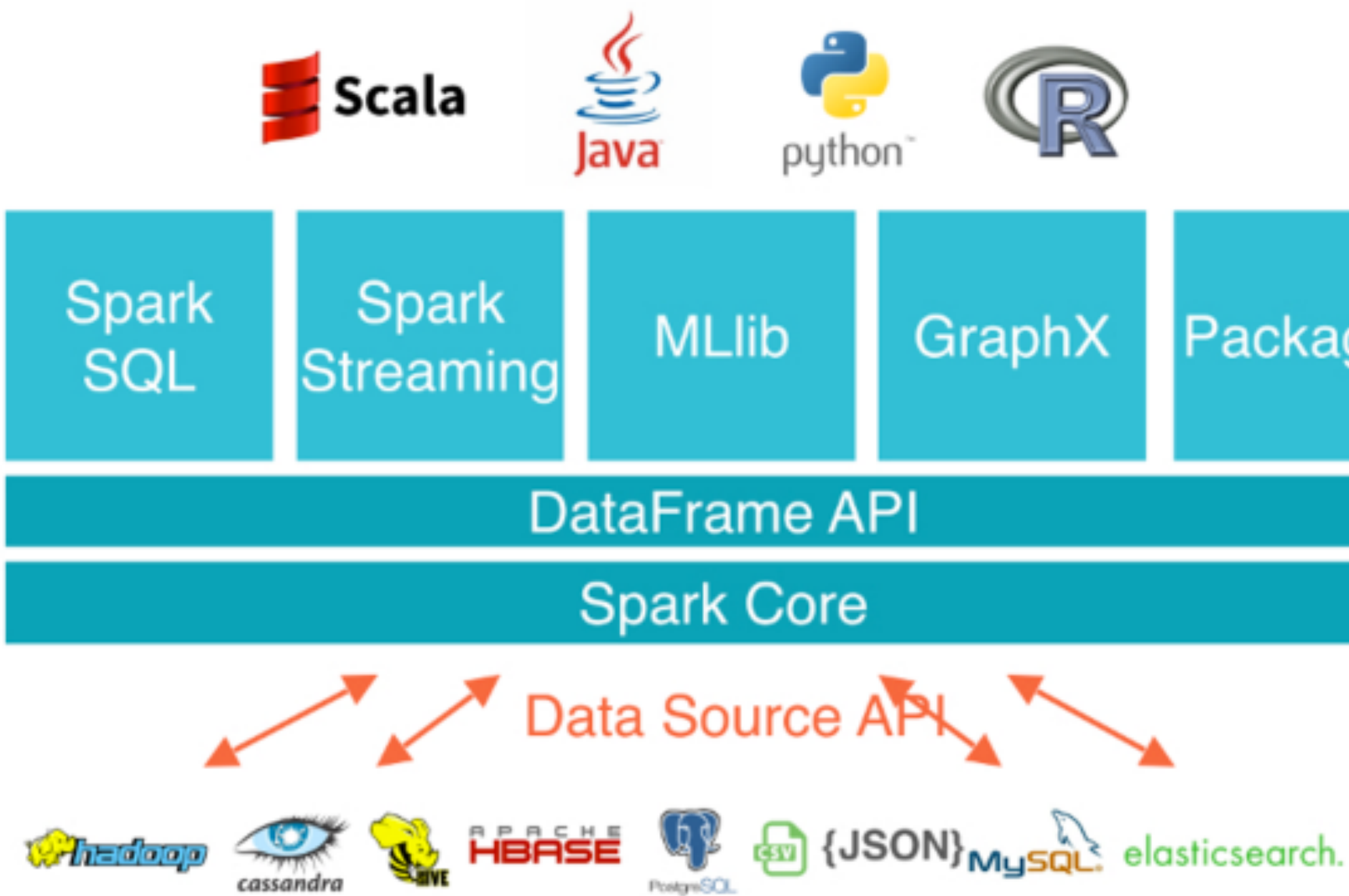
GENOME ANALYSIS

- Cloudera and Broad Institute
- Genome Analysis Toolkit (GATK) built on Spark
- Provide high performance gene traversal and iteration
- https://blog.cloudera.com/blog/2016/04/genome-analysis-toolkit-now-using-apache-spark-for-data-processing/
- https://www.broadinstitute.org/gatk/about/

WHY DO WE WANT TO USE SPARK?

WHY DO WE ~~WANT~~ NEED TO USE SPARK?

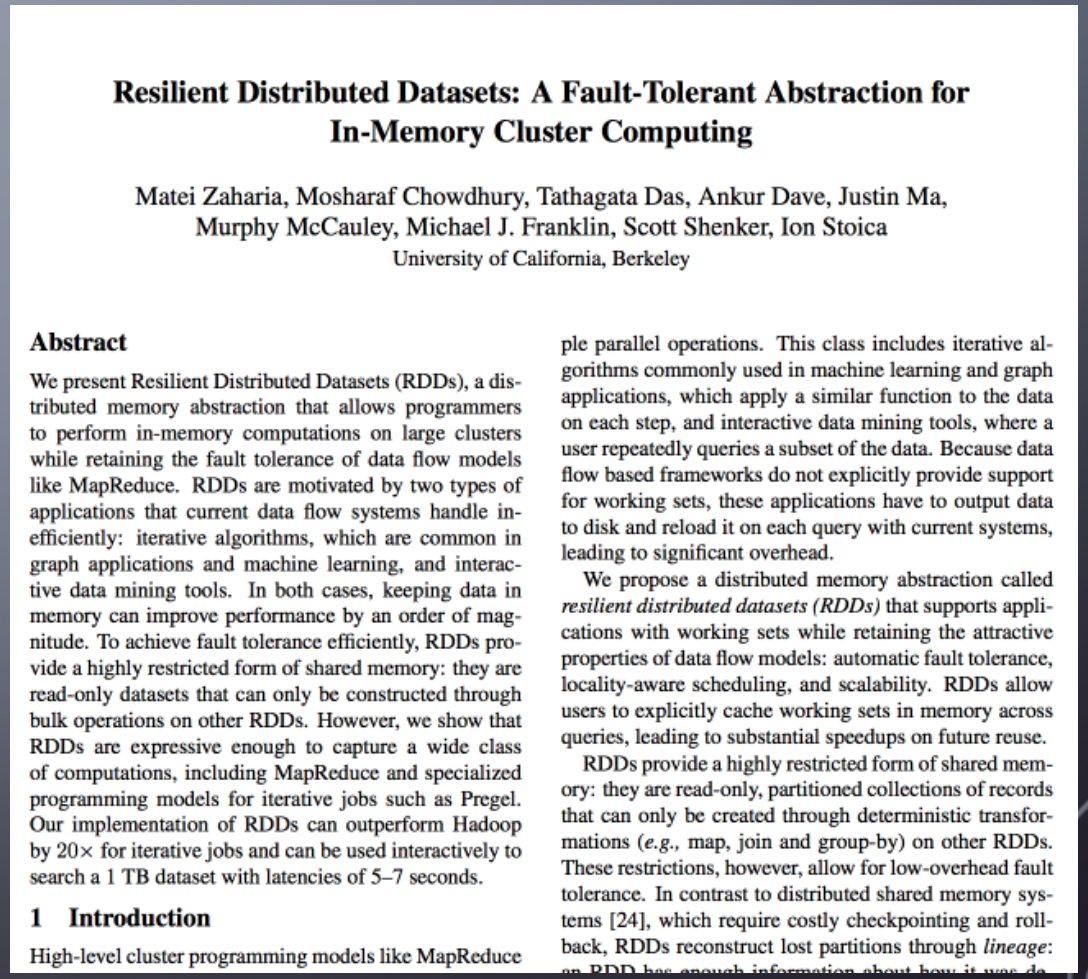
- Faster alternative to MapReduce when doing large amounts of data processing
- Easily scalable
- Compatible with HDFS, YARN, HBASE
- SparkSQL to connect to relational DBs
- Stable Java APIs
- Machine Learning libraries
- Streaming hooks to analyze real-time data
- Memory grows cheaper and cheaper, and Spark can take full advantage of this
- Spark provides a layer of abstraction on distributed computing, developers don't have to worry so much about it
- There's no real efficient way to analyze large data sets without distributed systems
- All the cool kids are using Spark


Scala
Java
python
R
Spark SQL
Spark Streaming
MLlib
GraphX
Packages
DataFrame API
Spark Core
Data Source API
hadoop
cassandra
HIVE
APACHE HBASE
PostgreSQL
CSV
{JSON}
MySQL
elasticsearch.

HOW DOES SPARK WORK?

RESILIENT DISTRIBUTED DATASETS (RDD)

- Primary abstraction in Spark
- Represents a collection of elements (think of a List in Java, but it has additional features
  - Immutable (read-only)
  - Resilient (fault-tolerent)
  - Distributed (data is spread out across a cluster)
- RDDs are distributed on separate workers across your cluster (or locally)
- You can specify RDD partitions if you want more parallellism
- https://people.eecs.berkeley.edu/~matei/papers/2011/tr_spark.pdf



Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that allows programmers to perform in-memory computations on large clusters while retaining the fault tolerance of data flow models like MapReduce. RDDs are motivated by two types of applications that current data flow systems handle inefficiently: iterative algorithms, which are common in graph applications and machine learning, and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a highly restricted form of shared memory: they are read-only datasets that can only be constructed through bulk operations on other RDDs. However, we show that RDDs are expressive enough to capture a wide class of computations, including MapReduce and specialized programming models for iterative jobs such as Pregel. Our implementation of RDDs can outperform Hadoop by 20× for iterative jobs and can be used interactively to search a 1 TB dataset with latencies of 5–7 seconds.

1 Introduction

High-level cluster programming models like MapReduce

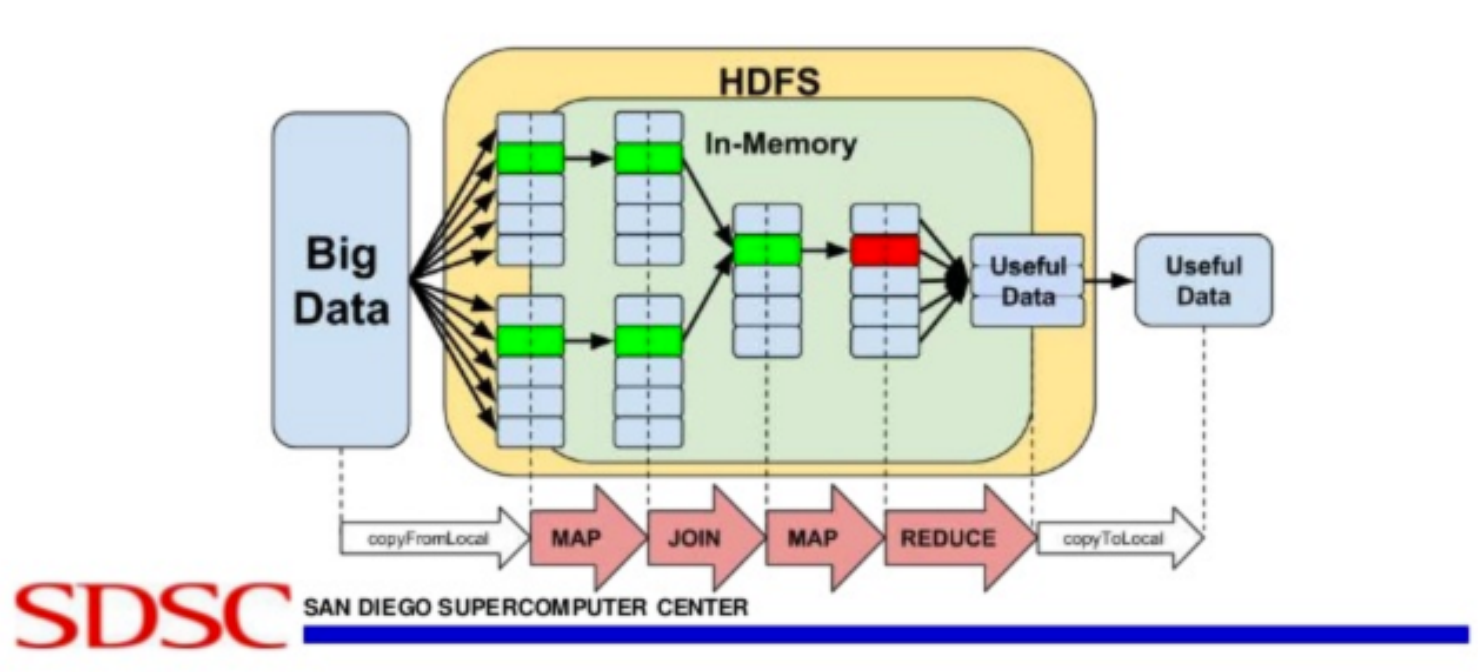
ple parallel operations. This class includes iterative algorithms commonly used in machine learning and graph applications, which apply a similar function to the data on each step, and interactive data mining tools, where a user repeatedly queries a subset of the data. Because data flow based frameworks do not explicitly provide support for working sets, these applications have to output data to disk and reload it on each query with current systems, leading to significant overhead.

We propose a distributed memory abstraction called *resilient distributed datasets (RDDs)* that supports applications with working sets while retaining the attractive properties of data flow models: automatic fault tolerance, locality-aware scheduling, and scalability. RDDs allow users to explicitly cache working sets in memory across queries, leading to substantial speedups on future reuse.

RDDs provide a highly restricted form of shared memory: they are read-only, partitioned collections of records that can only be created through deterministic transformations (*e.g.,* map, join and group-by) on other RDDs. These restrictions, however, allow for low-overhead fault tolerance. In contrast to distributed shared memory systems [24], which require costly checkpointing and rollback, RDDs reconstruct lost partitions through *lineage*:

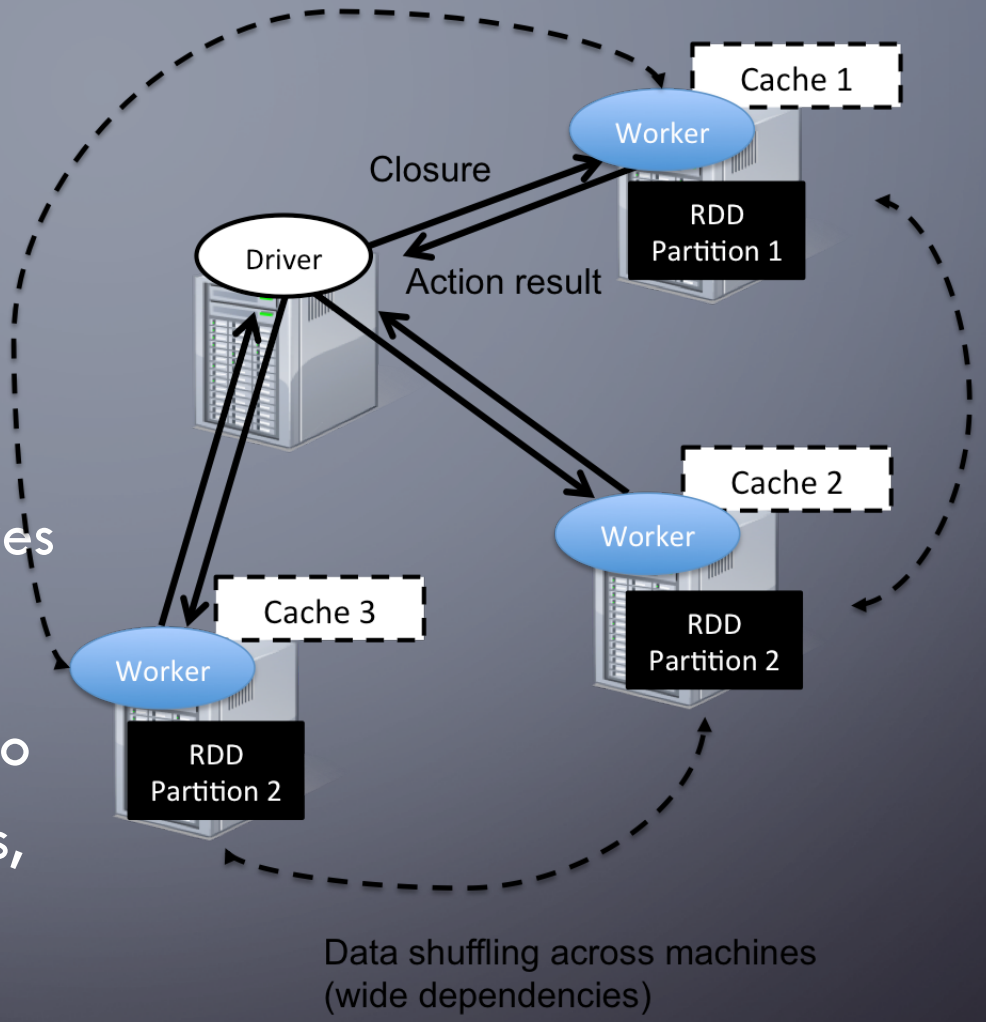
RESILIENT

- Spark has built in fault tolerance
- Spark maintains a lineage graph of the data through different transformations, so any missing data, or errors can be recovered from



DISTRIBUTED

- All of Spark's datasets will be distributed, that could be on different machines or different cores
- However, this is known up front, so we don't have to worry about it too much when we're doing operations, like map, sort, reduce, etc.



IMMUTABLE

- While there are lots of data operations we can do in Spark, they will result in a new RDD
- If objects are mutable, they are more likely to introduce bugs and complexity, immutability helps Spark ensure data integrity in a distributed environment

```
scala> val lines = sc.textFile("/home/spark/spark-dev-u/dat
a/sherlock.txt")
lines: org.apache.spark.rdd.RDD[String] = /home/spark/spark
-dev-u/data/sherlock.txt MapPartitionsRDD[1] at textFile at
 <console>:27
```

```
scala> val data = 1 to 10
data: scala.collection.immutable.Range.Inclusive = Range(1,
 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val distData = sc.parallelize(data)
distData: org.apache.spark.rdd.RDD[Int] = ParallelCollectio
nRDD[2] at parallelize at <console>:29
```

```
final JavaRDD<String> dent = sc.textFile("testfiles/dent.txt");
```

RDD OPERATIONS

ACTIONS

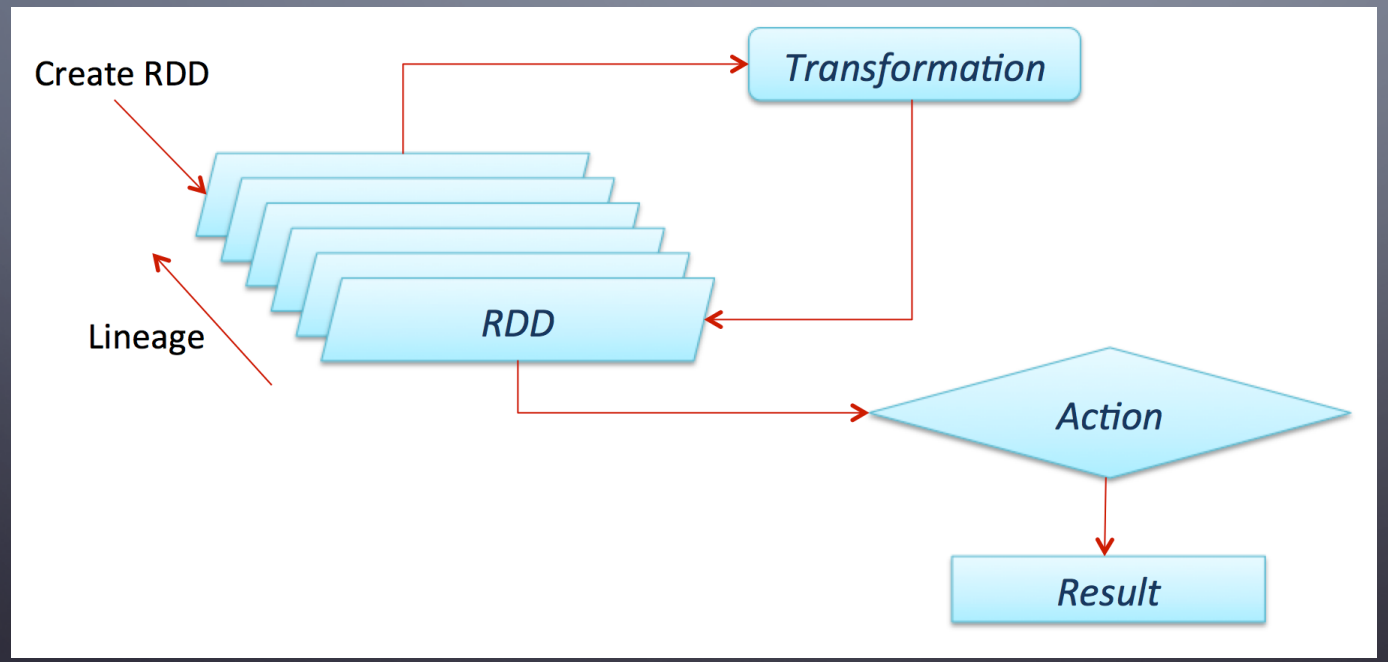
- Trigger transformations to be computed, or returned
- One common action, is ‘collect()’, which takes the RDD and returns it to the shell or variable, just don’t use on big data sets, try ‘take(n)’

TRANSFORMATIONS

- Operations that perform data manipulations, producing a new RDD by transforming the original one
- Lazily computed, not invoked until an action is called

RDD OPERATIONS

- Operations can be chained together in a directed acyclic graph or DAG, and nothing will be evaluated until an Action is invoked



https://dzone.com/refcardz/apache-spark

COMMON TRANSFORMATIONS

- **map** - Return a new distributed dataset formed by passing each element of the source through a function *func*.
- **filter** - Return a new dataset formed by selecting those elements of the source on which *func* returns true.
- **flatMap** - Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
- **distinct** - Return a new dataset that contains the distinct elements of the source dataset.
- **groupByKey** - When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
- **reduceByKey -** When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

TRANSFORMATIONS

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- sample
- union
- intersection
- distinct
- groupByKey
- reduceByKey
- aggregateByKey
- sortByKey
- join
- cogroup
- cartesian
- pipe
- coalesce
- repartition
- repartitionAndSortWithinPartitions

COMMON ACTIONS

- **reduce** - Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- **collect** - Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- **take** - Return an array with the first *n* elements of the dataset.
- **saveAsTextFile** – (Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.)
- **foreach -** Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems.

ACTIONS

- reduce
- collect
- count
- first
- take
- takeSample
- takeOrdered
- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- countByKey
- foreach

RDD PERSISTENCE

- Spark supports both on memory and disk persistence, default is in memory only
- One of Spark's big speed advantages is its ability to make use of distributed in-memory caching
- You can tag RDDs to persist using persist() or cache(), storing am RDD in cache will store it in memory, and be much faster when you need to evaluate it
- If you've got enough memory, it's most CPU efficient to leave in memory to run as fast as possible (MEMORY_ONLY, default)
- If you don't have enough, try MEMORY_ONLY_SER, which stores RDDs in serialized Java objects
- Avoid writing to disk, unless you need to for massive computations

WHY IN MEMORY MATTERS

In-Memory Can Make a Big Difference

- Two iterative Machine Learning algorithms:

Berkeley CS105x
https://www.youtube.com/watch?v=Ddq3Gua2QFg

SPARK CONTEXT

- Created with every Spark program
- Tells Spark how and where to access the cluster
- You can also think of the Spark Context as the driver, and runs where you execute your Spark job
- Spark Context is created automatically if you're using PySpark or Spark Shell, but is easily invoked with a constructor in Java and other platforms

```
final SparkConf sparkConf = new SparkConf().setAppName("SparkLabeledLogisticRegression")
        .setMaster("local[2]");
final JavaSparkContext jsc = new JavaSparkContext(sparkConf);
```

SPARK CONTEXT MASTER

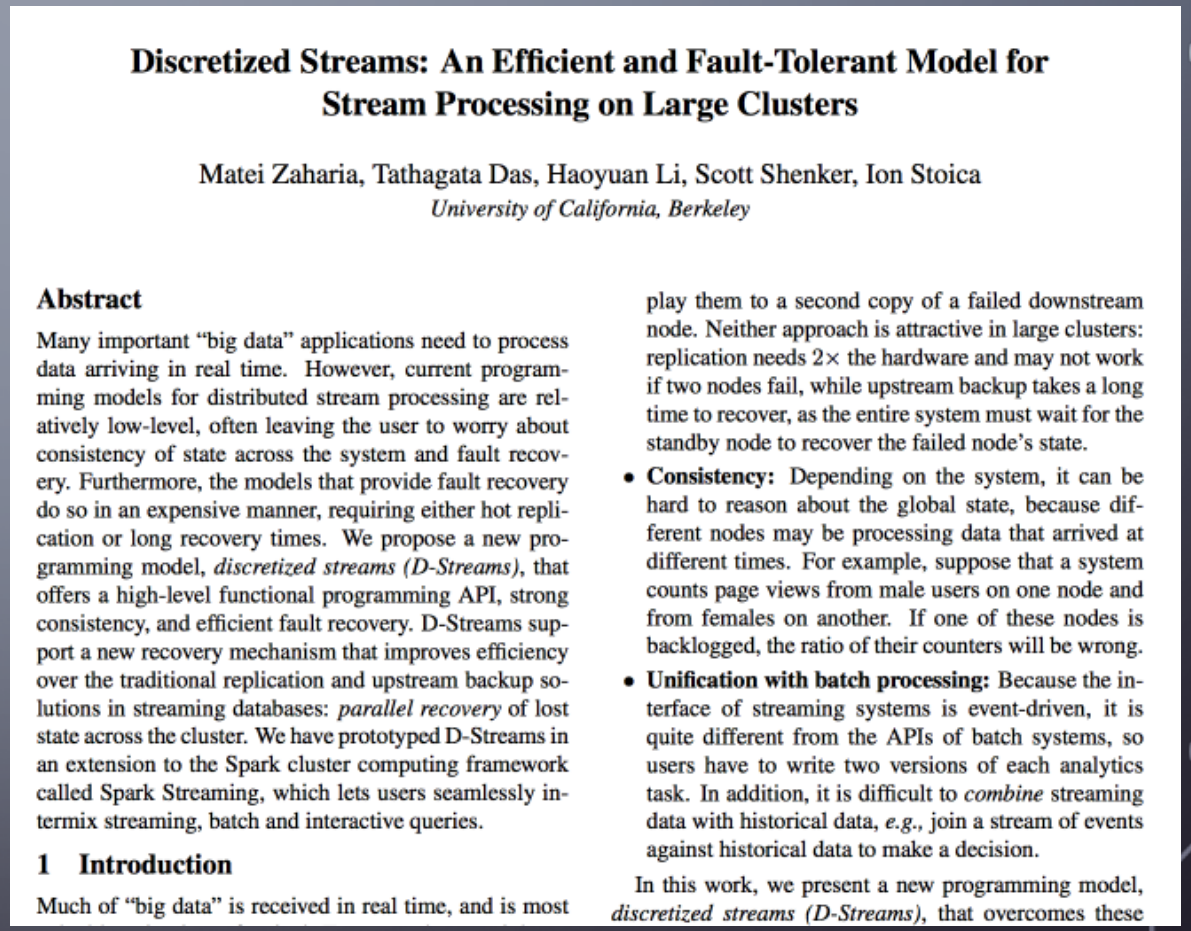
- local - run local on one thread
- local[*] – run local on all cores
- local[n] – run local on n cores
- spark://host:port – connect to a Spark cluster
- mesos://host:port – connect to a Mesos cluster
- If running on Hadoop, you can also set master to Yarn, setting the –master parameter on Spark shell or Spark submit

SPARK COMPONENTS

SPARK STREAMING

SPARK STREAMING

- High throughput stream processing, built on top of Spark
- Read from Kafka, Twitter, Flume, and input streams, process in Spark, then output to HDFS, databases, etc.
- http://people.eecs.berkeley.edu/~haoyuan/papers/2012_hotcloud_spark_streaming.pdf



Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters

Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

Many important "big data" applications need to process data arriving in real time. However, current programming models for distributed stream processing are relatively low-level, often leaving the user to worry about consistency of state across the system and fault recovery. Furthermore, the models that provide fault recovery do so in an expensive manner, requiring either hot replication or long recovery times. We propose a new programming model, *discretized streams (D-Streams)*, that offers a high-level functional programming API, strong consistency, and efficient fault recovery. D-Streams support a new recovery mechanism that improves efficiency over the traditional replication and upstream backup solutions in streaming databases: *parallel recovery* of lost state across the cluster. We have prototyped D-Streams in an extension to the Spark cluster computing framework called Spark Streaming, which lets users seamlessly intermix streaming, batch and interactive queries.

1 Introduction

Much of "big data" is received in real time, and is most

play them to a second copy of a failed downstream node. Neither approach is attractive in large clusters: replication needs 2× the hardware and may not work if two nodes fail, while upstream backup takes a long time to recover, as the entire system must wait for the standby node to recover the failed node's state.

- **Consistency:** Depending on the system, it can be hard to reason about the global state, because different nodes may be processing data that arrived at different times. For example, suppose that a system counts page views from male users on one node and from females on another. If one of these nodes is backlogged, the ratio of their counters will be wrong.
- **Unification with batch processing:** Because the interface of streaming systems is event-driven, it is quite different from the APIs of batch systems, so users have to write two versions of each analytics task. In addition, it is difficult to *combine* streaming data with historical data, *e.g.*, join a stream of events against historical data to make a decision.

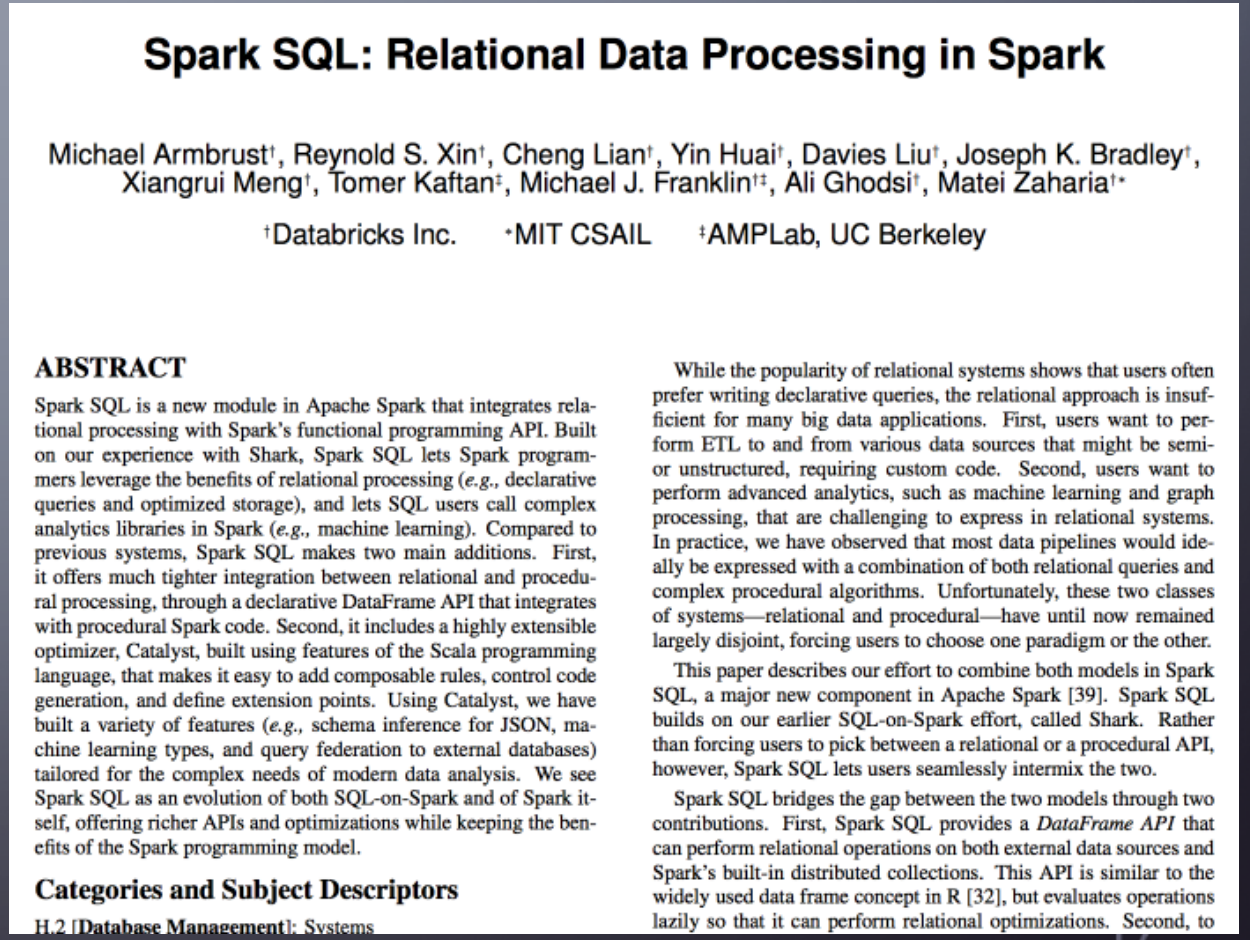
In this work, we present a new programming model, *discretized streams (D-Streams)*, that overcomes these

SPARK COMPONENTS

SPARK SQL AND DATAFRAMES

SPARK SQL

- Built-in module to integrate with structured data sources
- Provides SQL interface to any data source in Spark
- https://web.eecs.umich.edu/~prabal/teaching/resources/eecs582/armbrust15sparksql.pdf



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 on our experience with Shark, Spark SQL lets Spark programmers leverage the benefits of relational processing (*e.g.,* declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (*e.g.,* machine learning). Compared to previous systems, Spark SQL makes two main additions. First, it offers much tighter integration between relational and procedural processing, through a declarative DataFrame API that integrates with procedural Spark code. Second, it includes a highly extensible optimizer, Catalyst, built using features of the Scala programming language, that makes it easy to add composable rules, control code generation, and define extension points. Using Catalyst, we have built a variety of features (*e.g.,* schema inference for JSON, machine learning types, and query federation to external databases) tailored for the complex needs of modern data analysis. We see Spark SQL as an evolution of both SQL-on-Spark and of Spark itself, offering richer APIs and optimizations while keeping the benefits of the Spark programming model.

Categories and Subject Descriptors

H.2 [Database Management]: Systems

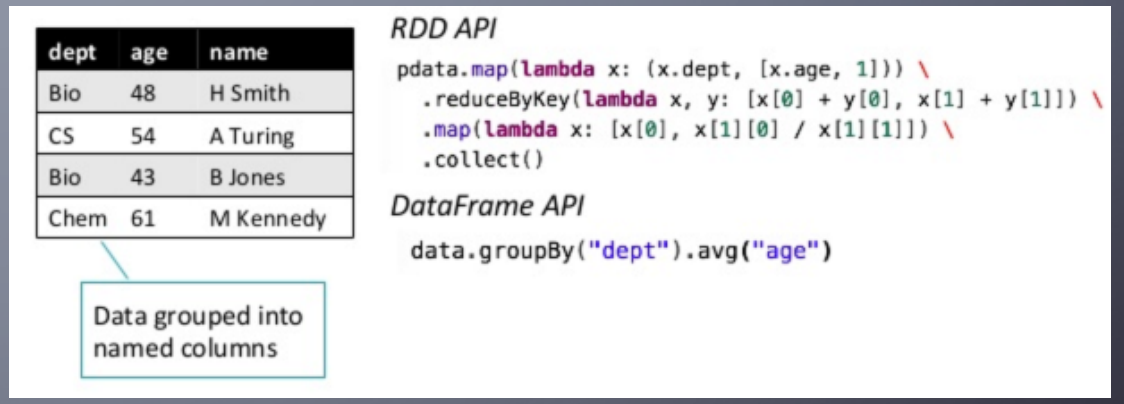
While the popularity of relational systems shows that users often prefer writing declarative queries, the relational approach is insufficient for many big data applications. First, users want to perform ETL to and from various data sources that might be semi- or unstructured, requiring custom code. Second, users want to perform advanced analytics, such as machine learning and graph processing, that are challenging to express in relational systems. In practice, we have observed that most data pipelines would ideally be expressed with a combination of both relational queries and complex procedural algorithms. Unfortunately, these two classes of systems—relational and procedural—have until now remained largely disjoint, forcing users to choose one paradigm or the other.

This paper describes our effort to combine both models in Spark SQL, a major new component in Apache Spark [39]. Spark SQL builds on our earlier SQL-on-Spark effort, called Shark. Rather than forcing users to pick between a relational or a procedural API, however, Spark SQL lets users seamlessly intermix the two.

Spark SQL bridges the gap between the two models through two contributions. First, Spark SQL provides a *DataFrame API* that can perform relational operations on both external data sources and Spark's built-in distributed collections. This API is similar to the widely used data frame concept in R [32], but evaluates operations lazily so that it can perform relational optimizations. Second, to

DATAFRAMES

- Loads data into a organized collection, with named, accessible columns, similar to what how data is organized in a structured database
- Can be sourced from an external database, a Hive data store, or an existing RDD, as long as you define a structure.
- Access via the SQLContext, or the HiveContext, an extension of the SparkContext
- Built in support for parquet, JSON, JDBC. Use additional library spark-csv for CSVContext



http://www.slideshare.net/databricks/dataframes-and-pipelines

DATAFRAMES API

- select - Selects a set of column based expressions.
- join – join with another DataFrame
- filter – Filter rows using the expresion
- groupBy – Groups the DataFrame by the specified columns
- write – Interface to write to external storage
- sort – returns a new DataFrame sorted by the specified column
- map – Returns an RDD mapped against this function
- flatMap – Returns an RDD, mapping the data against the function, then flattening the results
- explain – Prints the plan to the console

SPARK COMPONENTS

SPARK MLLIB

MLLIB

- Machine learning API, interoperates with NumPy and any Spark data source
- Supports most common machine learning algorithms that work well in a distributed environment
- Supports basic stats libraries, classification and regression, collaborative filtering, clustering, dimensionality reduction, feature extraction, pattern mining
- Logistic regression is 100x faster than MapReduce
- https://amplab.cs.berkeley.edu/wp-content/uploads/2016/04/15-237.pdf

MLlib: Machine Learning in Apache Spark

MENG ET AL.

Abstract

Apache Spark is a popular open-source platform for large-scale data processing that is well-suited for iterative machine learning tasks. In this paper we present MLLIB, Spark's open-source distributed machine learning library. MLLIB provides efficient functionality for a wide range of learning settings and includes several underlying statistical, optimization, and linear algebra primitives. Shipped with Spark, MLLIB supports several languages and provides a high-level API that leverages Spark's rich ecosystem to simplify the development of end-to-end machine learning pipelines. MLLIB has experienced a rapid growth due to its vibrant open-source community of over 140 contributors, and includes extensive documentation to support further growth and to let users quickly get up to speed.

Keywords: scalable machine learning, distributed algorithms, apache spark

MLLIB - ALGORITHMS

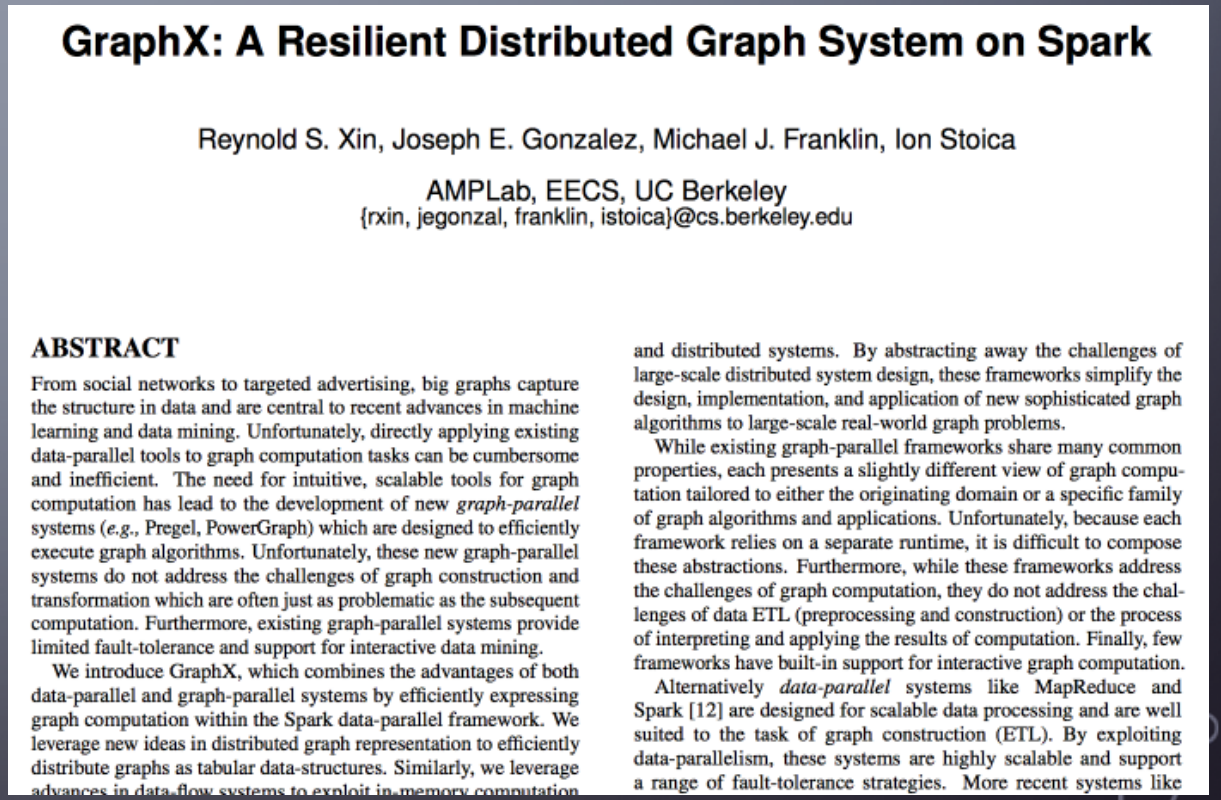
- logistic regression and linear support vector machine (SVM)
- classification and regression tree
- random forest and gradient-boosted trees
- recommendation via alternating least squares (ALS)
- clustering via k-means, bisecting k-means, Gaussian mixtures (GMM), and power iteration clustering
- topic modeling via latent Dirichlet allocation (LDA)
- survival analysis via accelerated failure time model
- singular value decomposition (SVD) and QR decomposition
- linear regression with L_1, L_2, and elastic-net regularization
- isotonic regression
- multinomial/binomial naive Bayes
- frequent itemset mining via FP-growth and association rules
- sequential pattern mining via PrefixSpan
- summary statistics and hypothesis testing
- feature transformations
- model evaluation and hyper-parameter tuning

SPARK COMPONENTS

SPARK GRAPHX

GRAPHX

- Graph API used for ETL, analysis and iterative computation
- Supports common graph algorithms:
  - PageRank
  - Connected components
  - Label propagation
  - SVD++
  - Strongly connected components
  - Triangle count
- https://amplab.cs.berkeley.edu/wp-content/uploads/2013/05/grades-graphx_with_fonts.pdf



GraphX: A Resilient Distributed Graph System on Spark

Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica

AMPLab, EECS, UC Berkeley
{rxin, jegonzal, franklin, istoica}@cs.berkeley.edu

ABSTRACT

From social networks to targeted advertising, big graphs capture the structure in data and are central to recent advances in machine learning and data mining. Unfortunately, directly applying existing data-parallel tools to graph computation tasks can be cumbersome and inefficient. The need for intuitive, scalable tools for graph computation has lead to the development of new *graph-parallel* systems (*e.g.,* Pregel, PowerGraph) which are designed to efficiently execute graph algorithms. Unfortunately, these new graph-parallel systems do not address the challenges of graph construction and transformation which are often just as problematic as the subsequent computation. Furthermore, existing graph-parallel systems provide limited fault-tolerance and support for interactive data mining.

We introduce GraphX, which combines the advantages of both data-parallel and graph-parallel systems by efficiently expressing graph computation within the Spark data-parallel framework. We leverage new ideas in distributed graph representation to efficiently distribute graphs as tabular data-structures. Similarly, we leverage advances in data-flow systems to exploit in-memory computation and distributed systems. By abstracting away the challenges of large-scale distributed system design, these frameworks simplify the design, implementation, and application of new sophisticated graph algorithms to large-scale real-world graph problems.

While existing graph-parallel frameworks share many common properties, each presents a slightly different view of graph computation tailored to either the originating domain or a specific family of graph algorithms and applications. Unfortunately, because each framework relies on a separate runtime, it is difficult to compose these abstractions. Furthermore, while these frameworks address the challenges of graph computation, they do not address the challenges of data ETL (preprocessing and construction) or the process of interpreting and applying the results of computation. Finally, few frameworks have built-in support for interactive graph computation.

Alternatively *data-parallel* systems like MapReduce and Spark [12] are designed for scalable data processing and are well suited to the task of graph construction (ETL). By exploiting data-parallelism, these systems are highly scalable and support a range of fault-tolerance strategies. More recent systems like

RUNNING SPARK

RUNNING SPARK - LOCAL

- As you can see, the main engine in Spark is built in Scala, but you can also run Java, R or Python
- In Scala, R and Python, you can run in a REPL (Read-eval-print-loop) which is easy for prototyping code
  - Scala - ./bin/spark-shell
  - R - ./bin/sparkR
  - Python - ./bin/PySpark
- In Java, you can run similarly by setting your master to local (including the number of cores you wish to utilize)

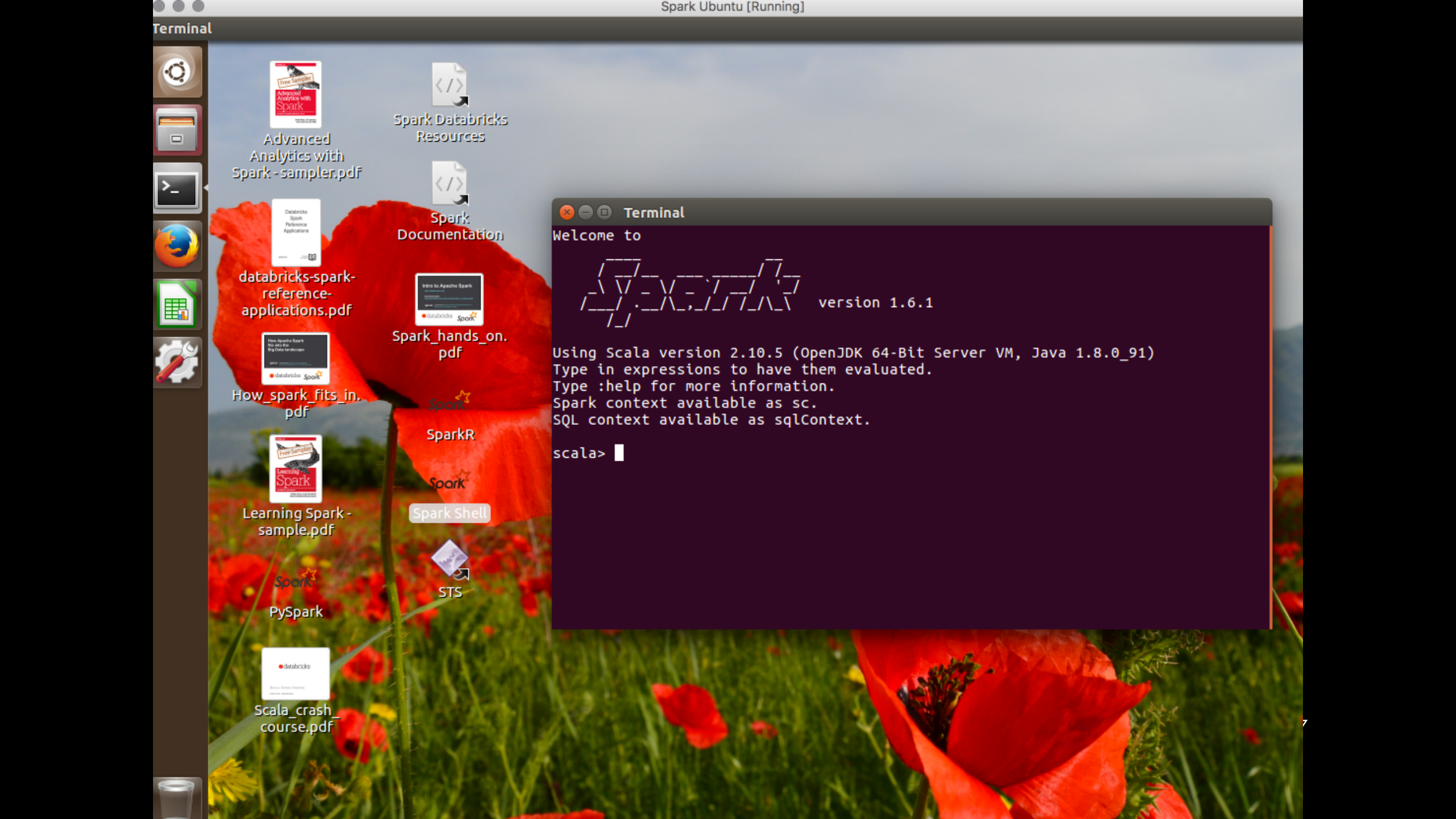
```
/** Local Spark Context **/
final SparkConf sparkConf = new SparkConf()
    .setAppName("SparkLogisticRegression")
    .setMaster("local[4]");
```

RUNNING SPARK – CLUSTER

- Spark can run on Amazon EC2, Mesos or Hadoop YARN
- Spark can also run in a standalone mode
  - Start the Spark master
  - Start as many slaves as you need, referencing the Spark master
  - View running jobs/status at http://host:8080

DEMO

START UP YOUR VIRTUALBOX IMAGE (OR JAVA PROJECT)!


Spark Ubuntu [Running]
Terminal
Advanced Analytics with Spark - sampler.pdf
Spark Databricks Resources
Spark Documentation
databricks-spark-reference-applications.pdf
Spark_hands_on.pdf
How_spark_fits_in.pdf
SparkR
Learning Spark - sample.pdf
Spark Shell
STS
PySpark
Scala_crash_course.pdf
Terminal
Welcome to
version 1.6.1
Using Scala version 2.10.5 (OpenJDK 64-Bit Server VM, Java 1.8.0_91)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.
scala>

LEARNING RESOURCES

WHERE TO LEARN MORE ABOUT SPARK

- Docs - https://spark.apache.org/docs/latest/index.html
- Spark in Action - https://www.manning.com/books/spark-in-action
- Spark MOOCs (start June 15) - https://www.edx.org/course/introduction-apache-spark-uc-berkeleyx-cs105x
- Databricks Dev Resources - https://sparkhub.databricks.com/resources/
- Paco Nathan course - http://shop.oreilly.com/product/0636920036807.do
- YouTube - https://www.youtube.com/user/TheApacheSpark/playlists