

Databricks

Notes

Session 1

Agenda

- 1 Databricks
- 2 Introduction to Spark
- 3 RDDs
- 4 DataFrames, Spark SQL

“Databricks”

DataBricks - Intro

- A cloud-based managed platform for running Apache Spark
- We'll use community.cloud.databricks.com for learning



Implementation of Spark to help reduce
complexity of setup and operation

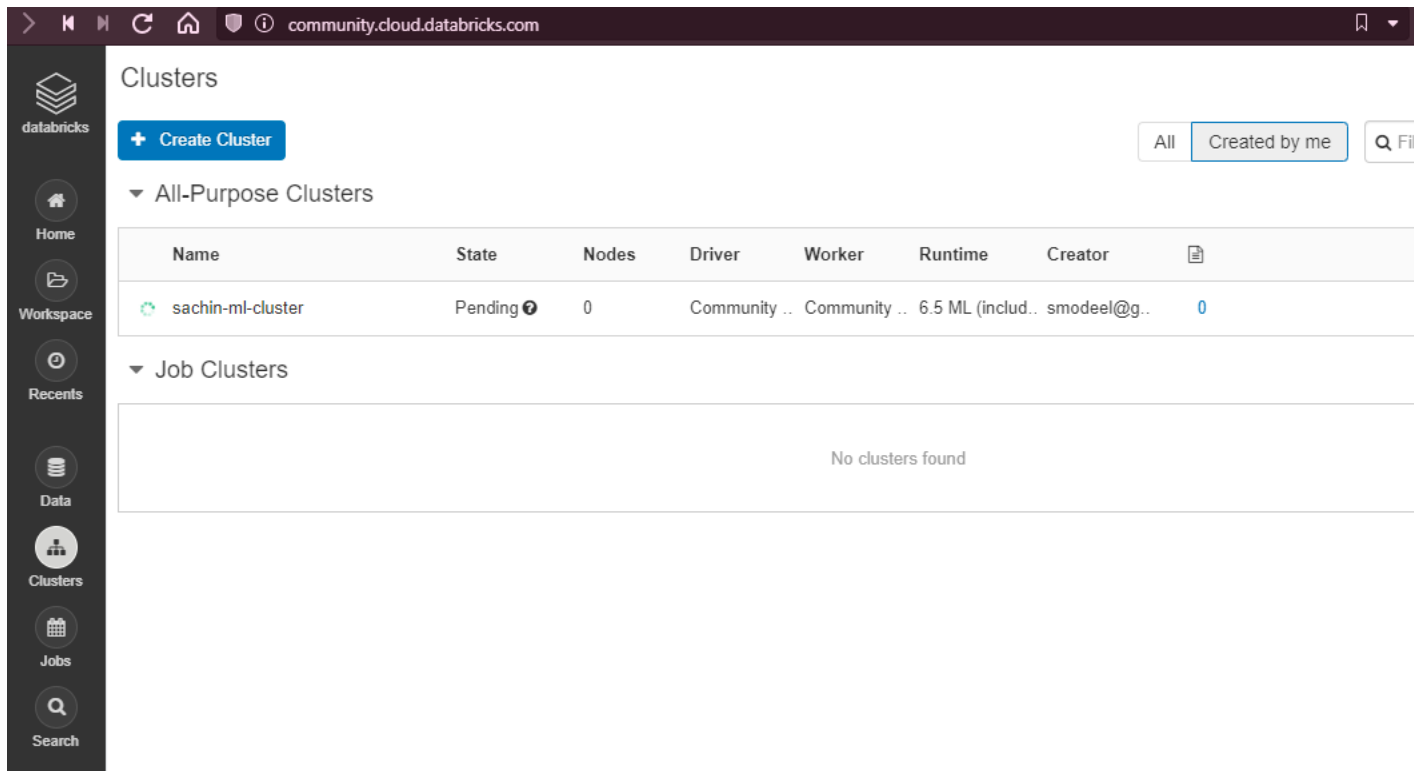


100% open-source platform for
distributed computing

Setting Up

→ community.cloud.databricks.com

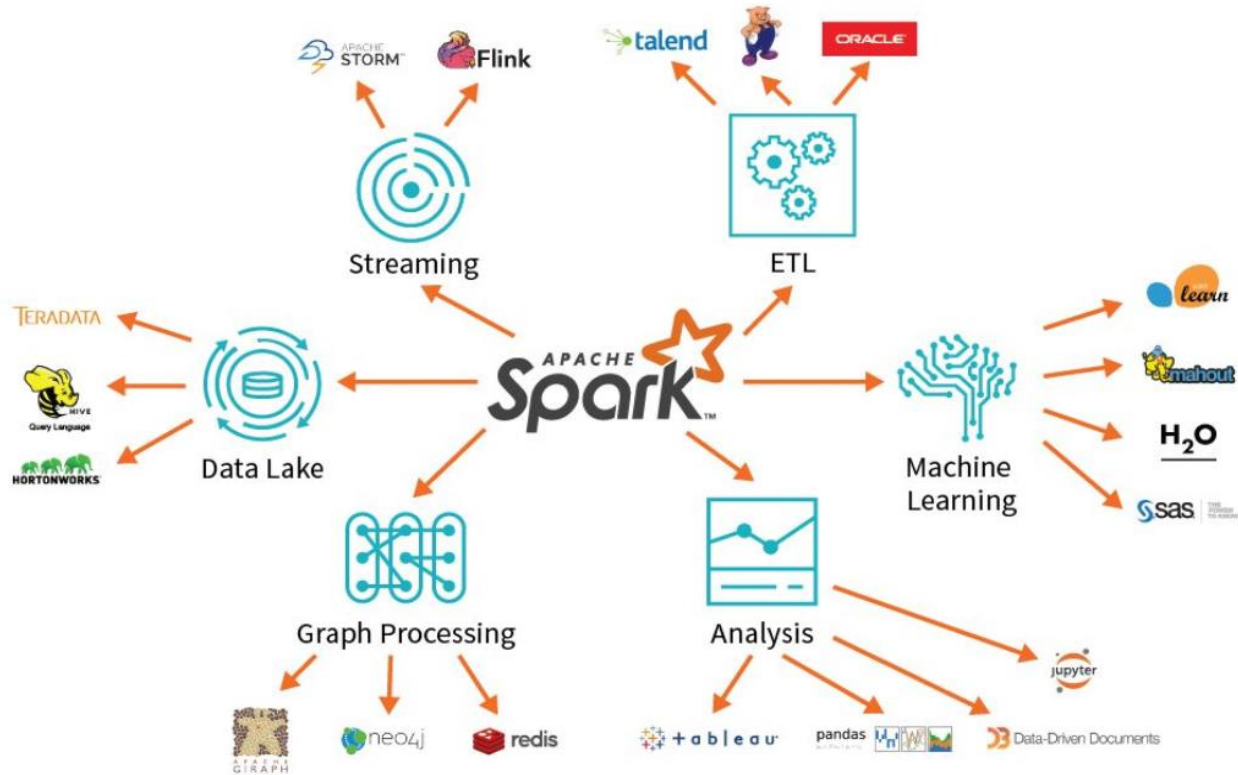
<https://help.databricks.com/s/>



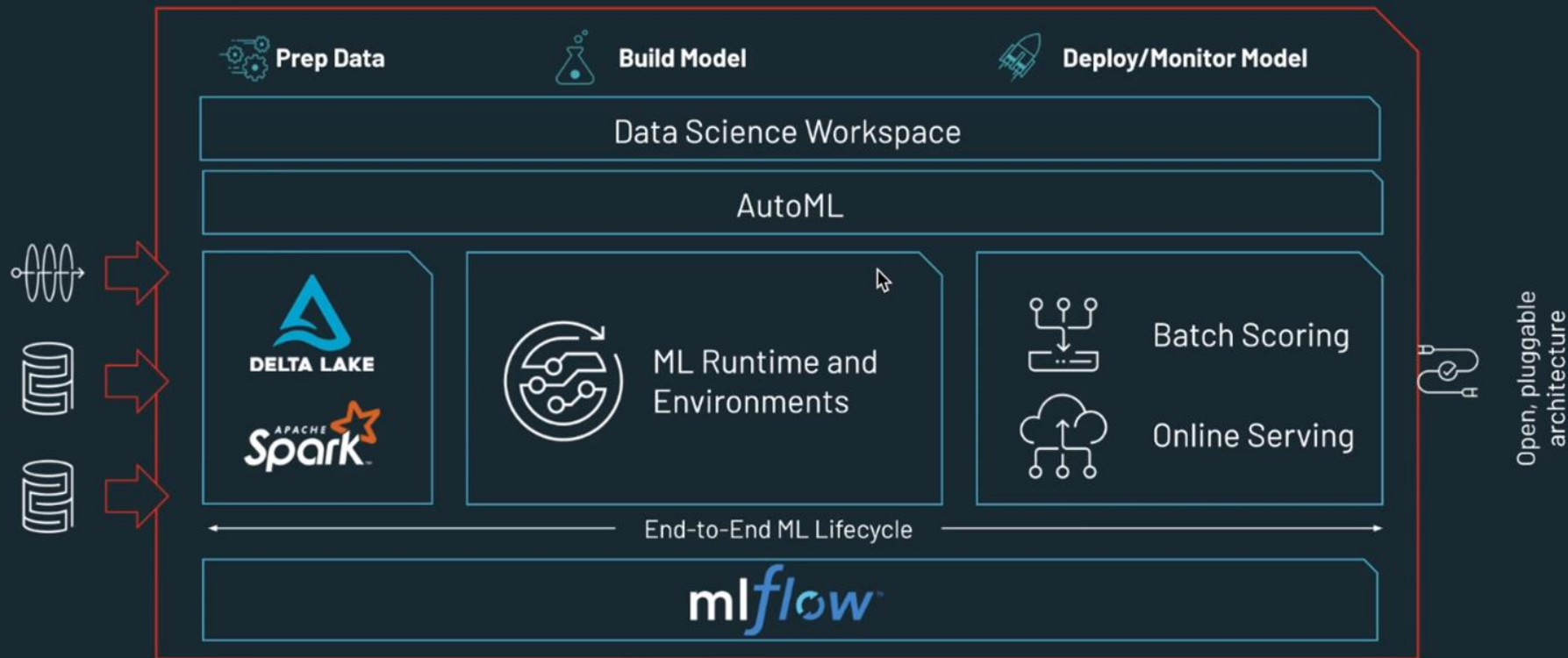
The screenshot shows the Databricks Clusters management interface. The browser address bar displays 'community.cloud.databricks.com'. The left sidebar contains navigation icons for Home, Workspace, Recents, Data, Clusters (which is highlighted), and Jobs, along with a Search icon at the bottom. The main content area is titled 'Clusters' and features a '+ Create Cluster' button. Below this, there are filter tabs for 'All' and 'Created by me', and a search input field. The 'All-Purpose Clusters' section is expanded, showing a table with one cluster: 'sachin-ml-cluster'. The table columns are Name, State, Nodes, Driver, Worker, Runtime, Creator, and an icon column. The cluster 'sachin-ml-cluster' is in a 'Pending' state with 0 nodes. Below the 'All-Purpose Clusters' section, the 'Job Clusters' section is collapsed and shows 'No clusters found'.

Name	State	Nodes	Driver	Worker	Runtime	Creator	
sachin-ml-cluster	Pending	0	Community ..	Community ..	6.5 ML (includ..	smodeel@g..	0

Unified Analytics Engine



End-to-End Data Science and ML on



“Introduction to Spark”

Spark Intro

“Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters”

One of the main motivations mentioned in the original 2010 Paper were the following:

Deficiencies with existing distributed computing frameworks around at that time:

Iterative Jobs:

- Tasks that take multiple passes at the same data set to optimize a parameter (**machine learning!**)

Interactive analytics or AdHoc querying:

- Doing ad-hoc exploratory queries on large datasets. (We see this everyday at our Banking customers.)

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-Merge [24] generalized the types of data flows supported. These systems achieve their scalability and fault tolerance by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention.

While this data flow programming model is useful for a large class of applications, there are applications that cannot be expressed efficiently as acyclic data flows. In this paper, we focus on one such class of applications: those that reuse a *working set* of data across multiple parallel operations. This includes two use cases where we have seen Hadoop users report that MapReduce is deficient:

- **Iterative jobs:** Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a

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.

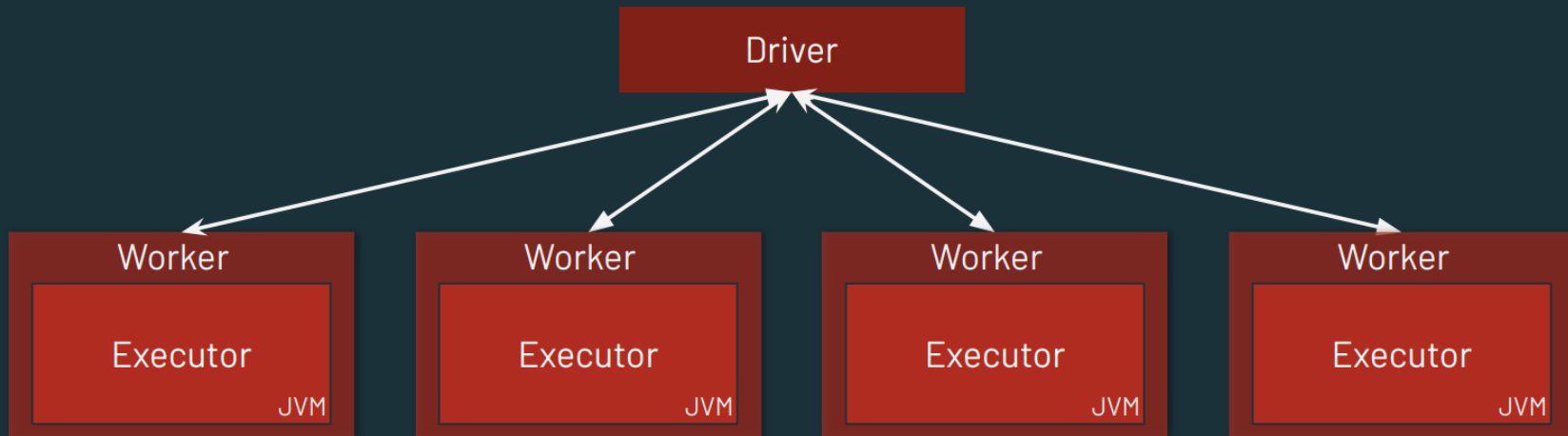
The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *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 found them well-suited for a variety of applications.

Spark is implemented in Scala [5], a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ [25]. In addition, Spark can be used interactively from a modified version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

Although our implementation of Spark is still a prototype, early experience with the system is encouraging. We show that Spark can outperform Hadoop by 10x in iterative machine learning workloads and can be used interactively to scan a 39 GB dataset with sub-second latency.

This paper is organized as follows. Section 2 describes

Spark Cluster



One Driver and many Executor
JVMs

Spark Intro – Spark Applications

“Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters”

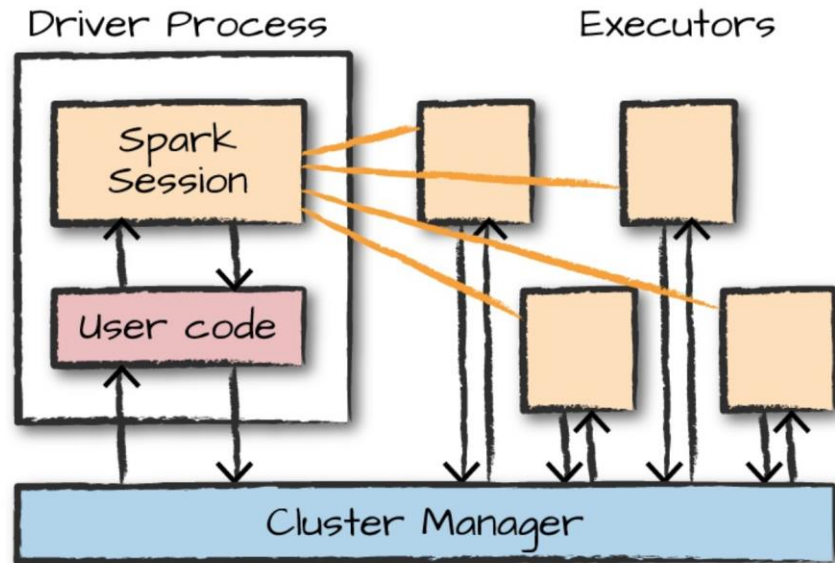
A Spark application consists of:

A Driver process

1. Maintaining info about spark application
2. Responding to user's i/p or program
3. Analyzing, distributing and scheduling works across slaves

A set of Executor processes

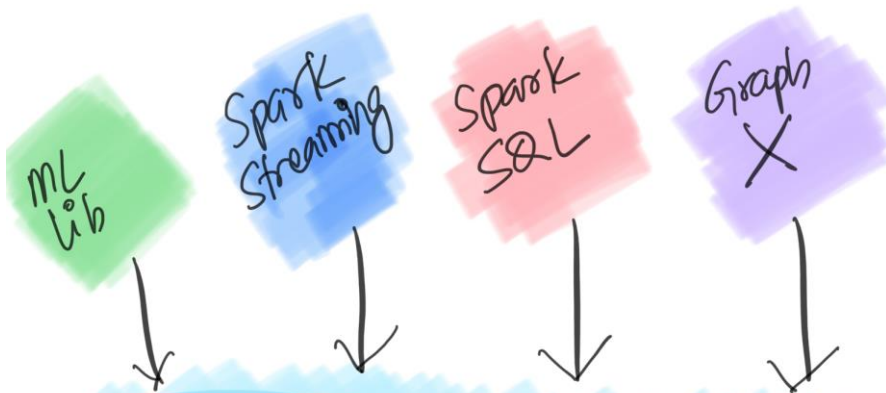
1. Executes code assigned to it by driver
2. Reporting the state of the computation back to driver node



Note: There is also a **local mode**, where you can run all this on a single machine

Spark Intro - Spark Components

Libraries



Core

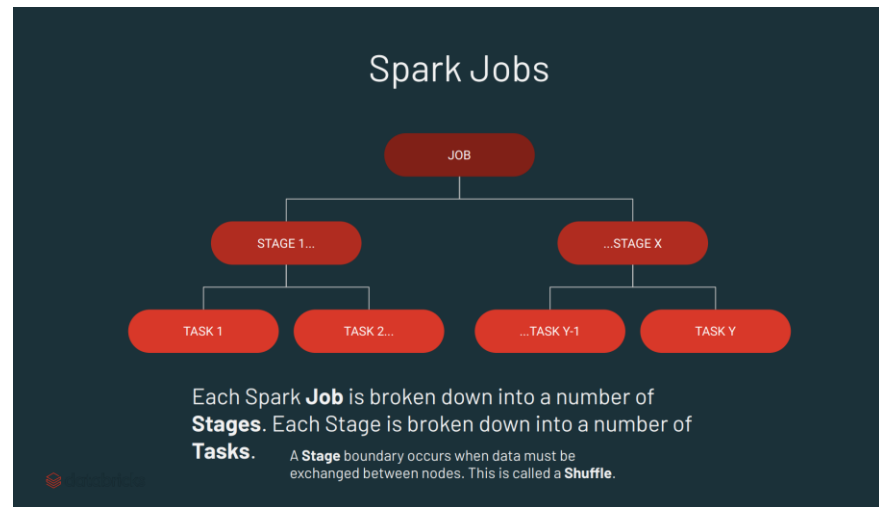
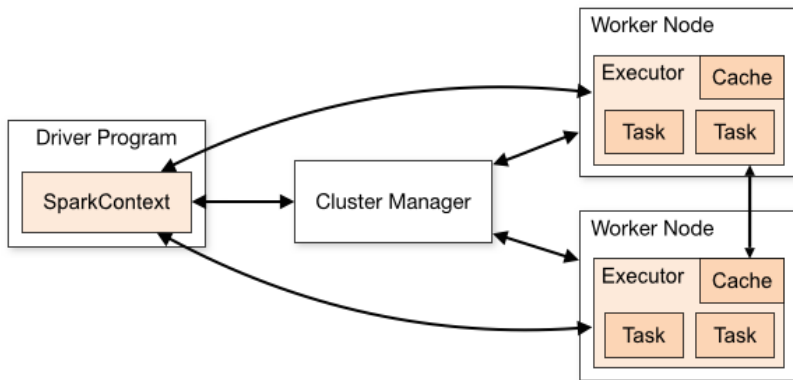


Cluster Managers



Spark Intro - Cluster overview

Standalone, Mesos, YARN, Kubernetes*



Spark Intro – SparkContext vs SparkSession

class pyspark.sql.SparkSession:

- The entry point to programming Spark with the Dataset and DataFrame API.
- A SparkSession can be used **create DataFrame**, register **DataFrame** as tables, **execute SQL** over tables, cache tables, and **read parquet files**
- **>>>spark**

class pyspark.SparkContext

- Main entry point for Spark functionality.
- A SparkContext represents the connection to a Spark cluster, and can be used to **create RDD** and **broadcast variables** on that cluster.
- **>>>spark.sparkContext**
- **>>>sc**

Scala:

```
scala> sc  
res: spark.SparkContext = spark.SparkContext@470d1f30
```

Python:

```
>>> sc  
<pyspark.context.SparkContext object at 0x7f7570783350>
```



Notebook

Spark Intro - SparkContext

The `master` parameter for a `SparkContext` determines which cluster to use

<i>master</i>	<i>description</i>
local	run Spark locally with one worker thread (no parallelism)
local[K]	run Spark locally with K worker threads (ideally set to # cores)
spark://HOST:PORT	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
mesos://HOST:PORT	connect to a Mesos cluster; PORT depends on config (5050 by default)

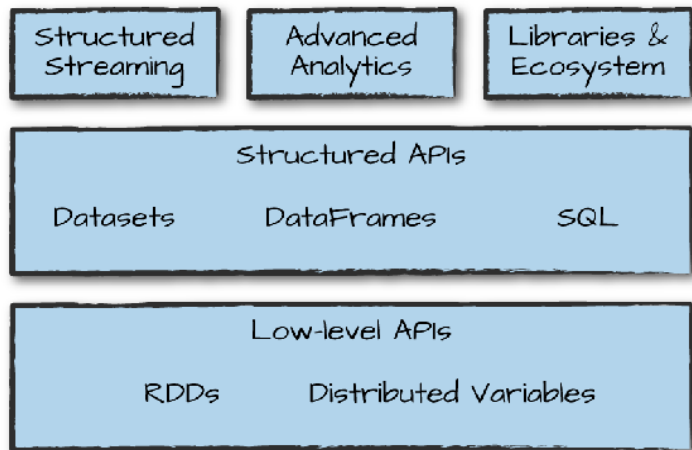
```
Cmd 13
1 | sc

Out[18]:

SparkContext
Spark UI
Version
  v2.4.5
Master
  local[8]
AppName
  Databricks Shell
```

Notebook

Spark Intro - components and Libraries



Spark Intro - RDD

The primary abstraction in Spark

“RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.”

Is a read-only partition collection of records

Resilient - can be created if data in memory is lost

- How? Because RDD keeps its lineage information
→ it can be recreated from parent RDDs

Distributed

Datasets

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 lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms 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 restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (e.g., between two MapReduce jobs) is to write it to an external stable storage system, e.g., a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serializa-

tion, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (e.g., looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, e.g., to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (e.g., cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (e.g., map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.¹ If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

¹Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

Spark Intro – Transformations and Actions

Two types of operations on Data Abstractions in spark. **Transformations** and **Actions**.

Transformations create a new dataset from an existing one.

All transformations in Spark are **lazy**: they do not compute their results right away - instead they remember the transformations applied to some base dataset

- Optimize the required calculations
- Recover from lost data partitions

Actions are **eager to execute**. And typically involve returning a value from the RDD.

Actions

show

count

collect

save

Transformation (L)

select

distinct

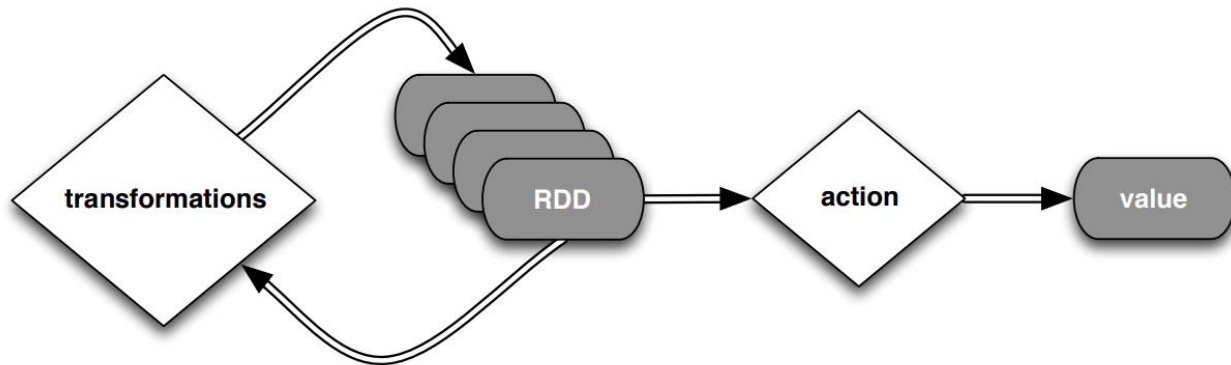
groupBy

sum

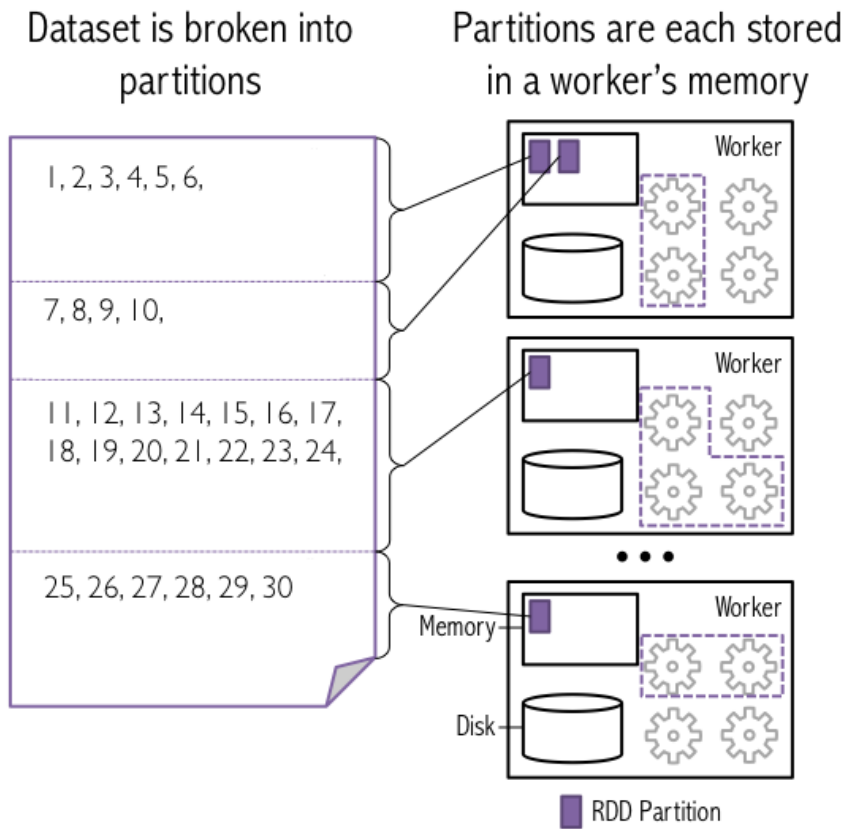
orderBy

filter

limit



Spark Intro - RDD Partitions



Notebook

“DataFrames and Spark SQL”

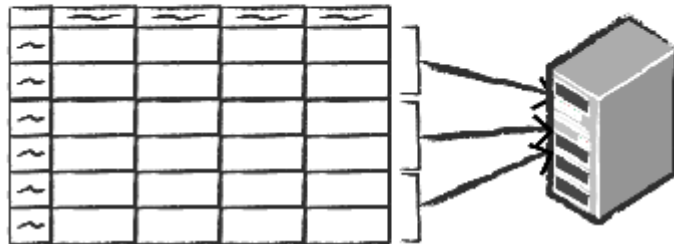
Spark Intro - Structured APIs - DataFrames

The DataFrame concept is not unique to Spark. R and Python both have similar concepts. However, Python/R DataFrames (with some exceptions) exist on one machine rather than multiple machines.

Spreadsheet on
a single machine



Table or Data Frame
partitioned across servers
in a data center



Notebook

Fundamentals First.

RDD

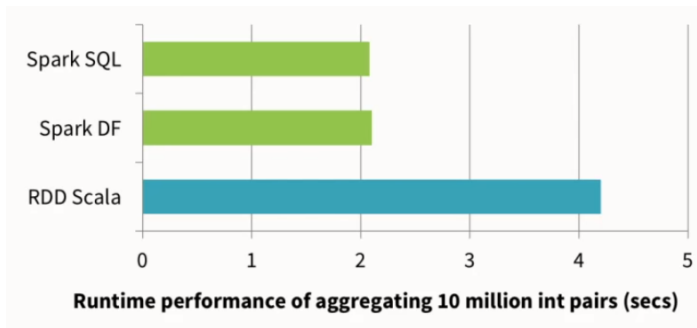
```
pdata.map { case (dpt, age) => dpt -> (age, 1) }  
  .reduceByKey { case ((a1, c1), (a2, c2)) => (a1 + a2, c1 + c2) }  
  .map { case (dpt, (age, c)) => dpt -> age / c }
```

Dataframe

```
data.groupBy("dept").avg("age")
```

SQL

```
select dept, avg(age) from data group by 1
```

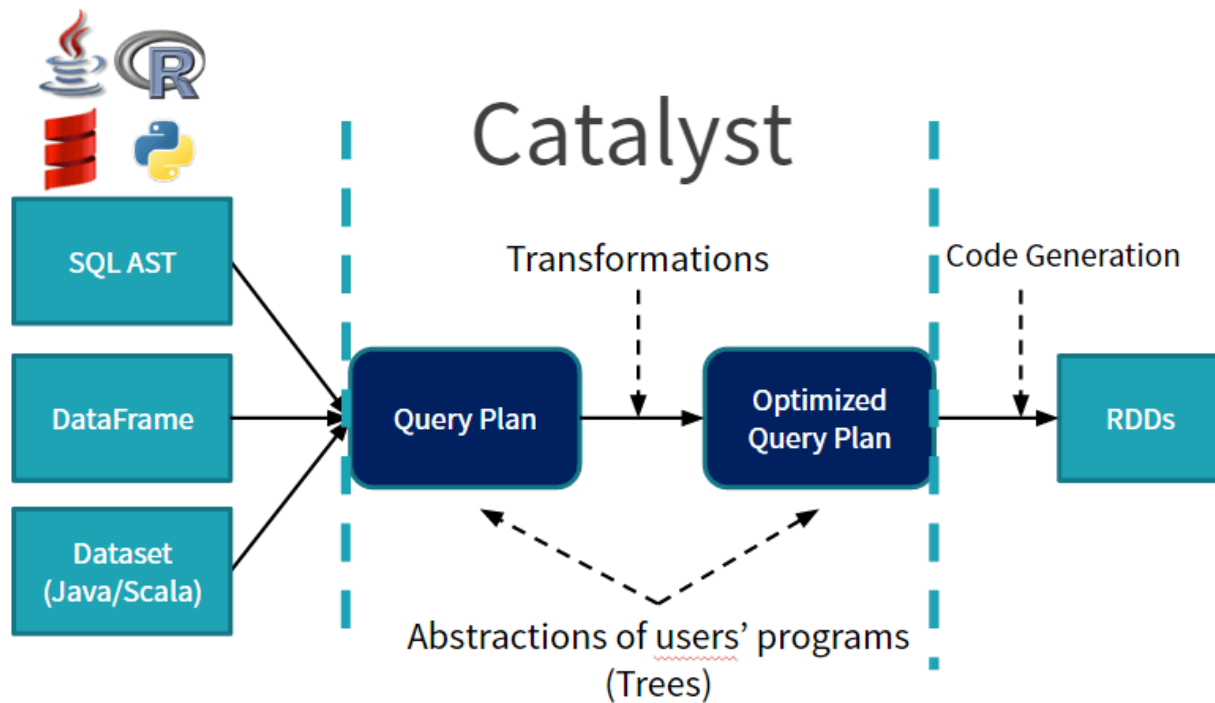


Spark Intro - Notebook



Launch Databricks Notebook shared for Demo

How Catalyst Works: An Overview

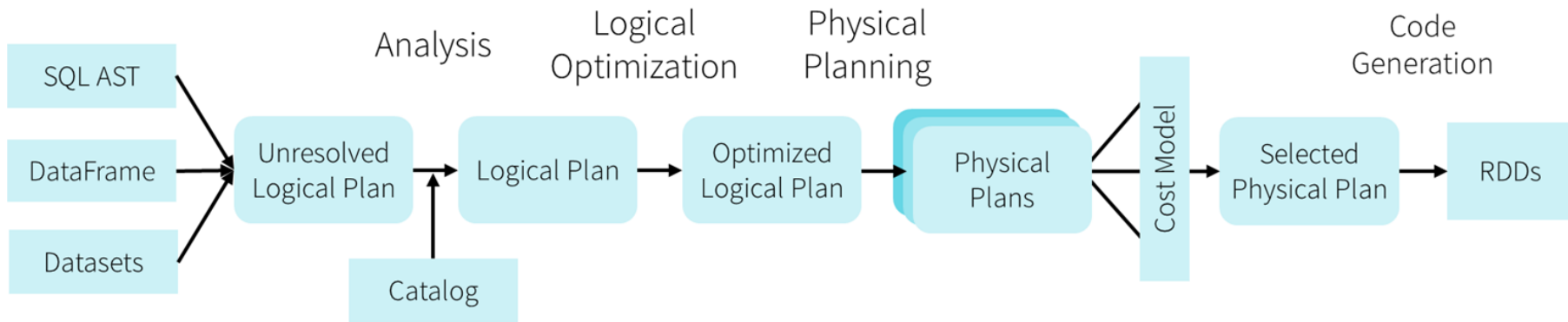


Catalyst Optimizer

Because our API is declarative a large number of optimizations are available to us.

Some of the examples include:

- Optimizing data type for storage
- Rewriting queries for performance
- Predicate push downs



Thank You