

CSCI 381/780

Cloud Computing

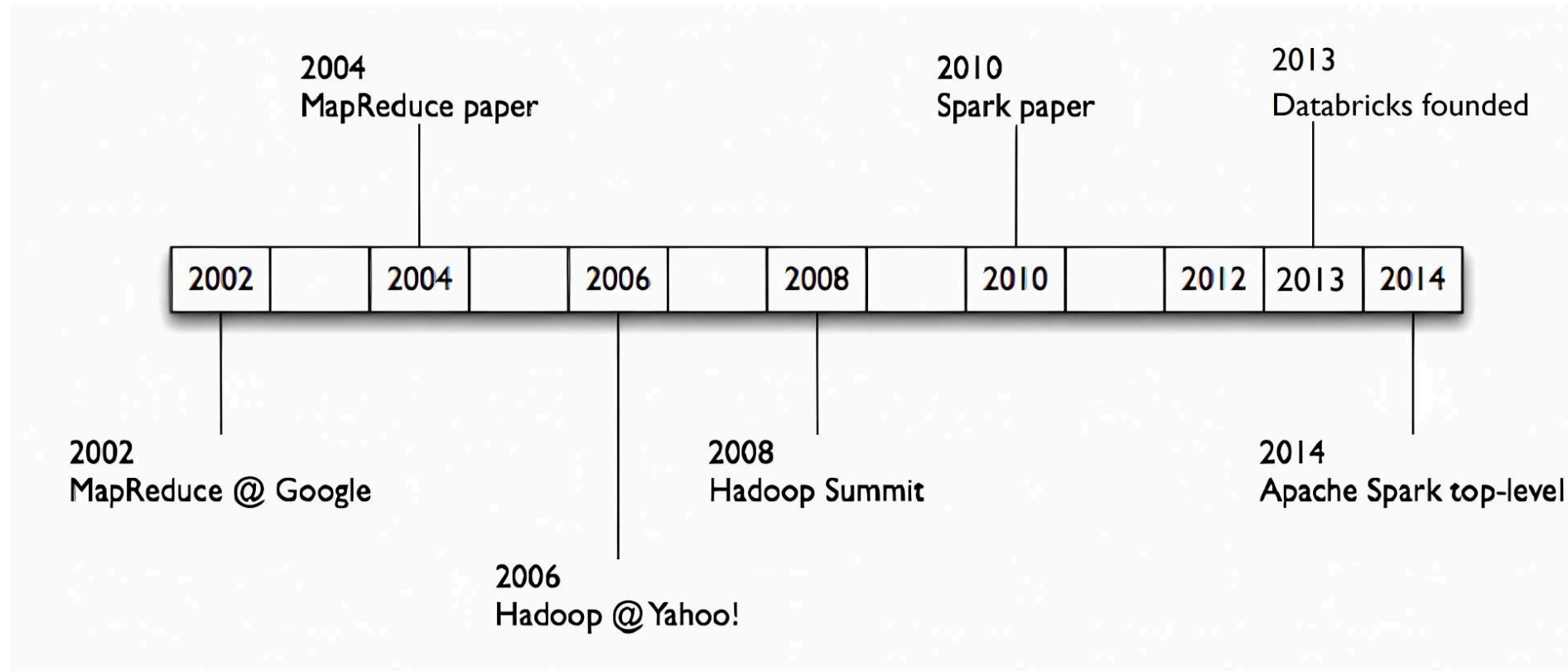
Spark

Jun Li
Queens College

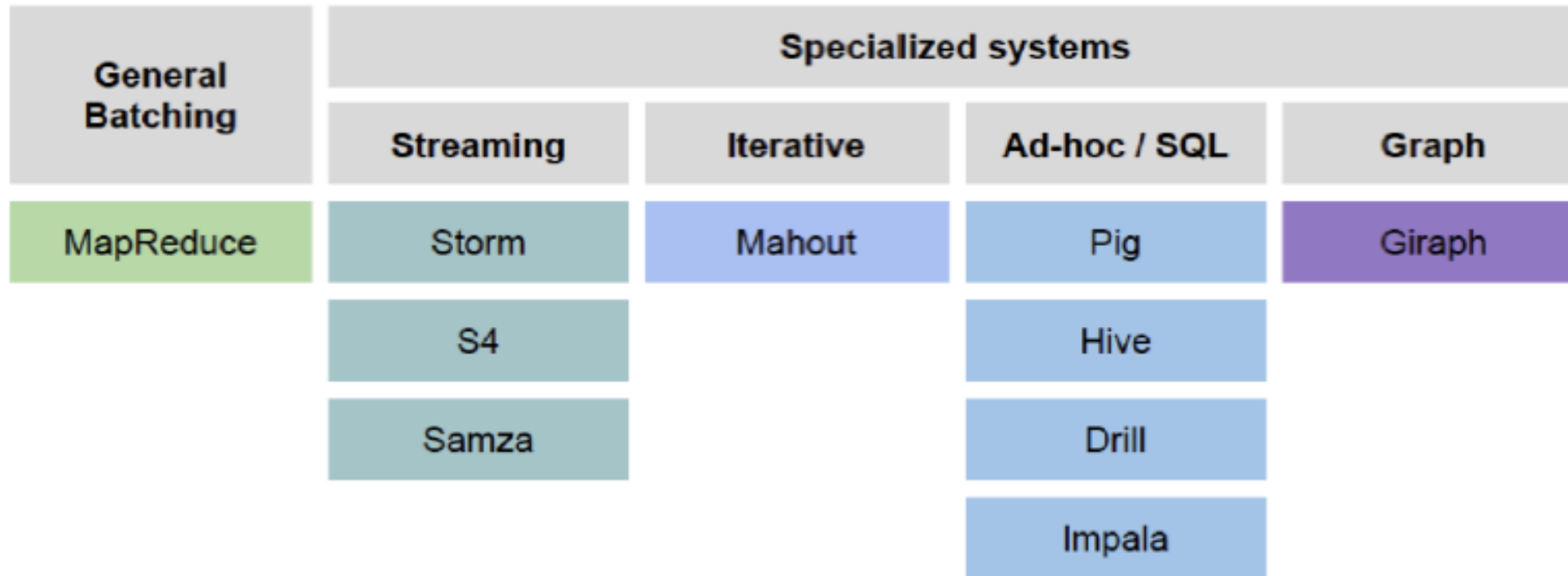
Today's Topics

- Motivation
- Spark Basics
- Spark Programming

History of Hadoop and Spark



Apache Hadoop Lacks Unified Vision



- Sparse Modules
- Diversity of APIs
- Higher Operational Costs

Key ideas

In Hadoop, each developer tends to invent his or her own style of work

With Spark, serious effort to standardize around the idea that people are writing parallel code that often runs for many “cycles” or “iterations” in which a lot of reuse of information occurs.

Spark centers on Resilient Distributed Dataset, RDDs, that capture the information being reused.

How this works

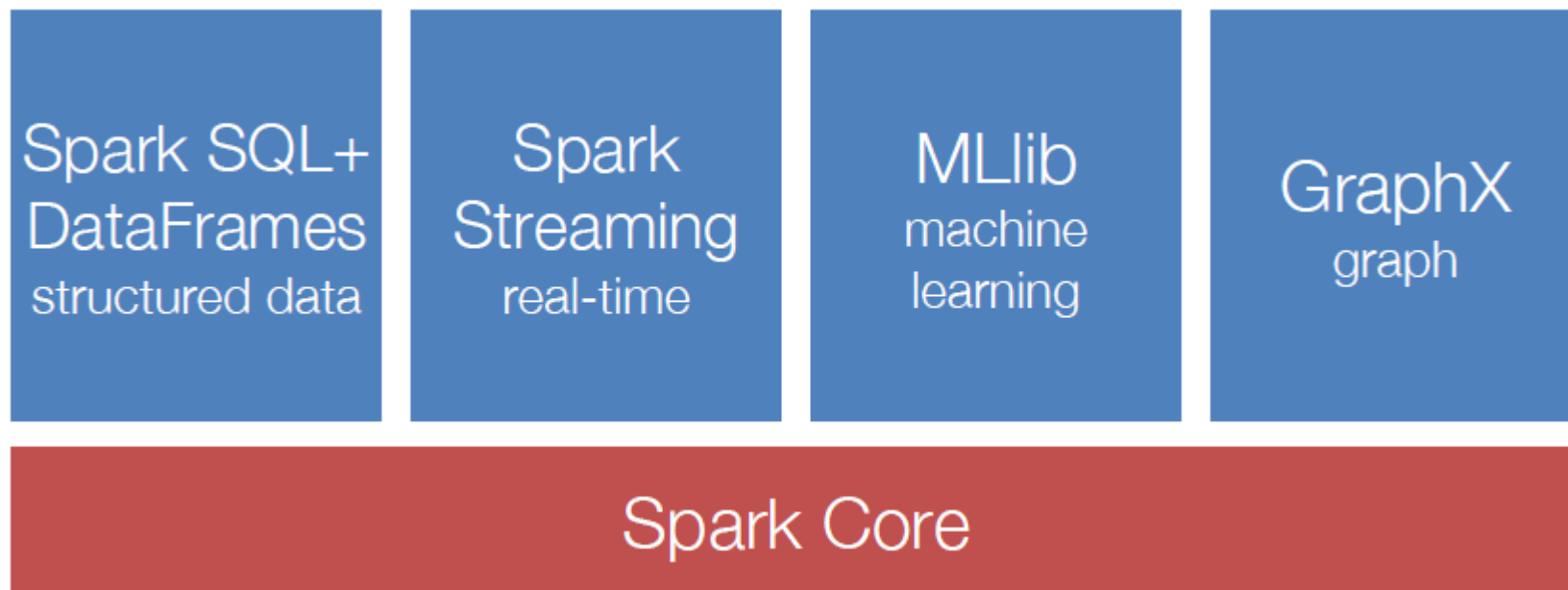
You express your application as a graph of RDDs.

The graph is only evaluated as needed, and they only compute the RDDs actually needed for the output you have requested.

Then Spark can be told to cache the reuseable information either in memory, in SSD storage or even on disk, based on *when* it will be needed again, *how big it is*, and *how costly it would be to recreate*.

You write the RDD logic and control all of this via hints

Spark Ecosystem: A Unified Pipeline



Motivation (1)

MapReduce: The original scalable, general, processing engine of the Hadoop ecosystem

- Disk-based data processing framework (HDFS files)
- Persists intermediate results to disk
- Data is reloaded from disk with every query → Costly I/O
 - Costly I/O → Not appropriate for iterative or stream processing workloads
 - Best for ETL like workloads (batch processing)

Motivation (2)

Spark: General purpose computational framework that substantially improves performance of MapReduce, but retains the basic model

- **Memory based data processing framework** → avoids costly I/O by keeping intermediate results in memory
- Leverages distributed memory
- Remembers operations applied to dataset
- Data locality based computation → High Performance
- Best for both iterative (or stream processing) and batch workloads

Today's Topics

- Motivation
- **Spark Basics**
- Spark Programming

Spark Basics(1)

Spark: Flexible, in-memory data processing framework written in Scala

Goals:

- Simplicity (Easier to use):
 - Rich APIs for Scala, Java, and Python
- Generality: APIs for different types of workloads
 - Batch, Streaming, Machine Learning, Graph
- Low Latency (Performance) : In-memory processing and caching
- Fault-tolerance: Faults shouldn't be special case

Spark Basics(2)

There are two ways to manipulate data in Spark

- Spark Shell:
 - Interactive – for learning or data exploration
 - Python or Scala
- Spark Applications
 - For large scale data processing
 - Python, Scala, or Java

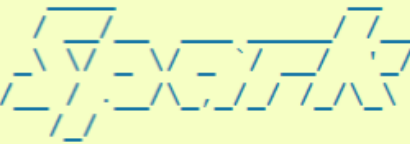
Spark Shell

The Spark Shell provides interactive data exploration (REPL)

Python Shell: `pyspark`

```
$ pyspark

Welcome to

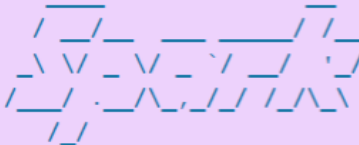
 version 1.3.0

Using Python version 2.7.8 (default, Aug 27
2015 05:23:36)
SparkContext available as sc, HiveContext
available as sqlCtx.
>>>
```

Scala Shell: `spark-shell`

```
$ spark-shell

Welcome to

 version 1.3.0

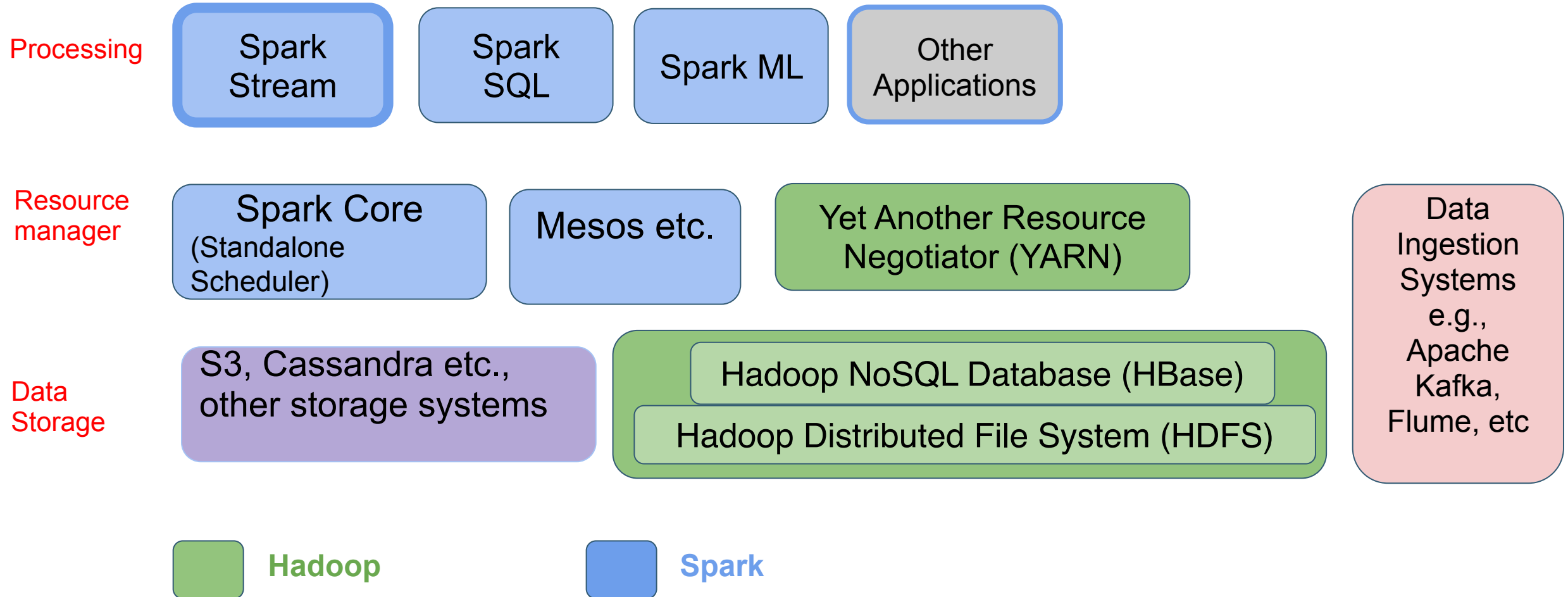
Using Scala version 2.10.4 (Java HotSpot(TM)
64-Bit Server VM, Java 1.7.0_67)
Created spark context..
Spark context available as sc.
SQL context available as sqlContext.

scala>
```

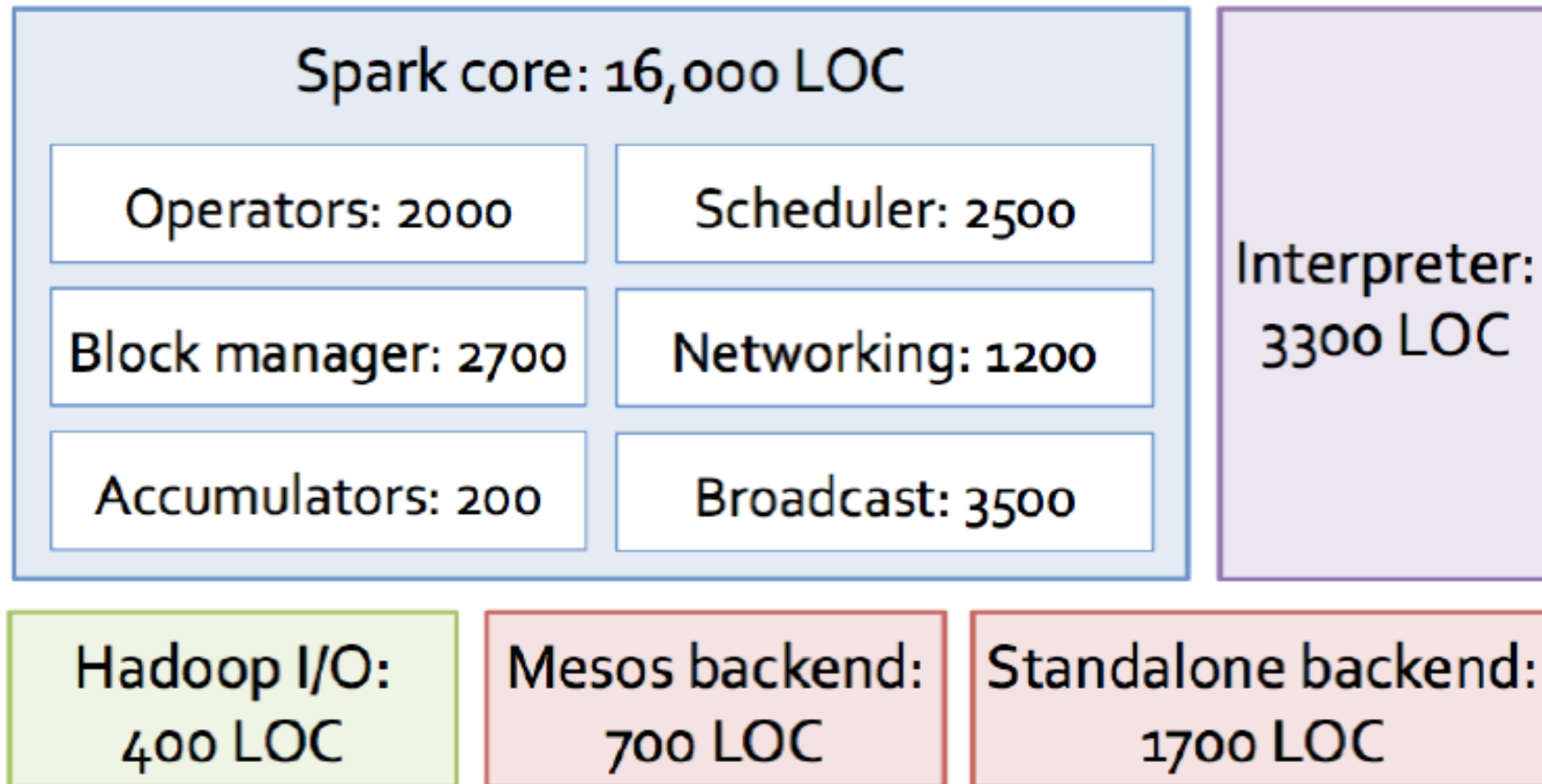
REPL: Repeat/Evaluate/Print Loop

Apache Spark

** Spark can connect to several types of *cluster managers* (either Spark's own standalone cluster manager, Mesos or YARN)



Spark Core: Code Base (2012)



Spark Fundamentals

Example of an application:

```
val sc = new SparkContext("spark://...", "MyJob", home, jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is an RDD

errors.cache()

errors.count() // This is an action
```

- **Spark Context**
- **Resilient Distributed Data**
- **Transformations**
- **Actions**

Spark Context (1)

- Every Spark application requires a *spark context*: the main entry point to the Spark API
- Spark Shell provides a preconfigured Spark Context called “sc”

Python

```
Using Python version 2.7.8 (default, Aug 27 2015 05:23:36)
SparkContext available as sc, HiveContext available as sqlCtx.

>>> sc.appName
u'PySparkShell'
```

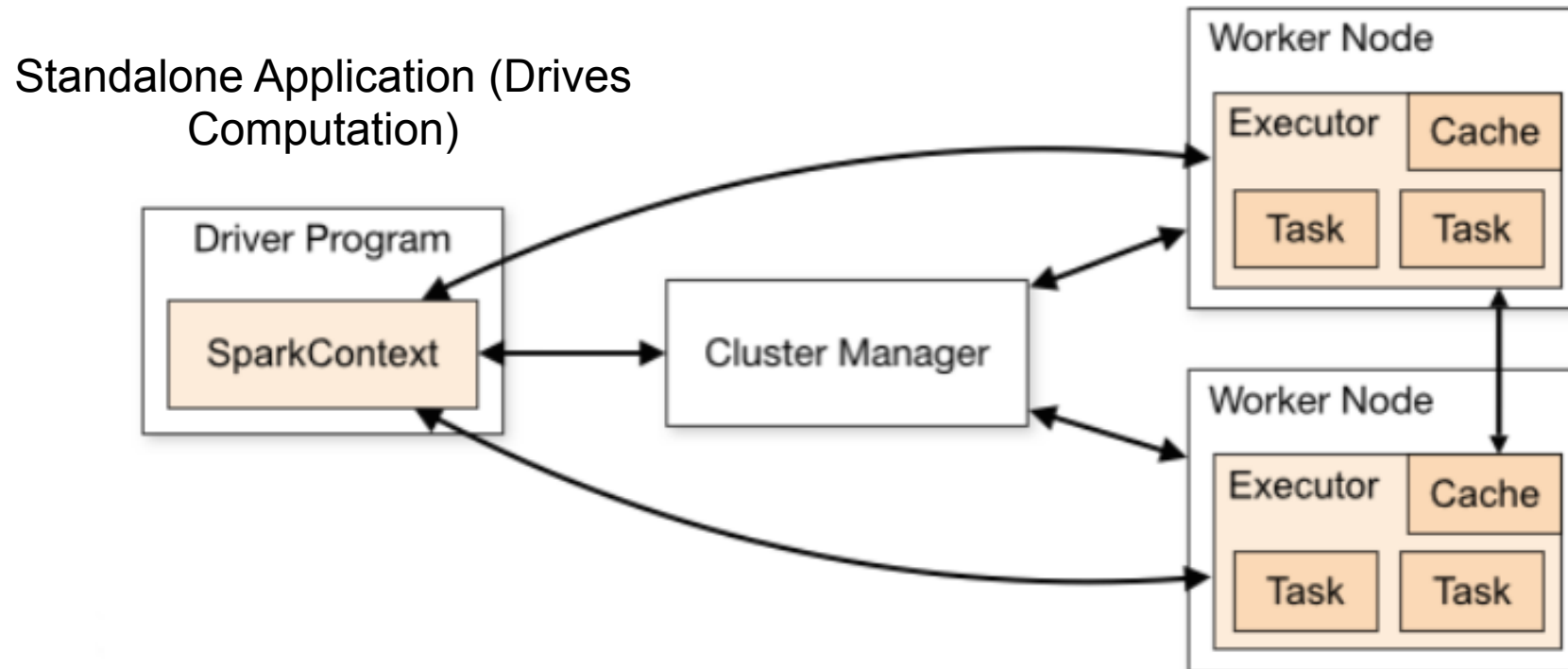
Scala

```
...
Spark context available as sc.
SQL context available as sqlContext.

scala> sc.appName
res0: String = Spark shell
```

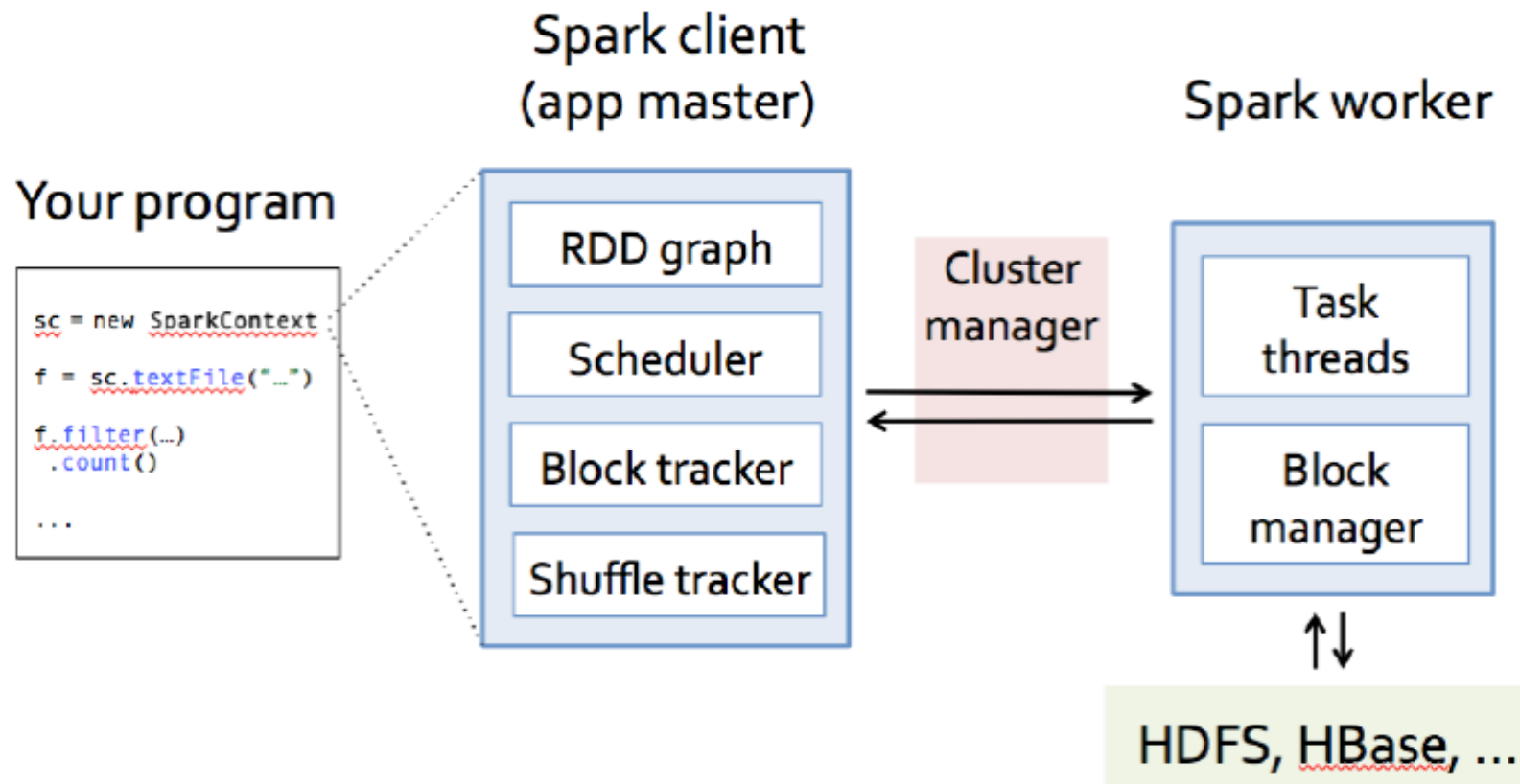
Spark Context (2)

- Standalone applications → Driver code → Spark Context
- Spark Context holds configuration information and represents connection to a Spark cluster



Spark Context (3)

Spark context works as a client and represents connection to a Spark cluster



Spark Fundamentals

Example of an application:

```
val sc = new SparkContext("spark://...", "MyJob", home, jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is an RDD

errors.cache()

errors.count() // This is an action
```

- Spark Context
- **Resilient Distributed Data**
- Transformations
- Actions

Resilient Distributed Dataset (RDD)

The RDD (Resilient Distributed Dataset) is the fundamental unit of data in Spark: An *Immutable* collection of objects (or records, or elements) that can be operated on “in parallel” (spread across a cluster)

Resilient -- if data in memory is lost, it can be recreated

- Recover from node failures
- An RDD keeps its lineage information → it can be recreated from parent RDDs

Distributed -- processed across the cluster

- Each RDD is composed of one or more partitions → (more partitions – more parallelism)

Dataset -- initial data can come from a file or be created

RDDs

Key Idea: Write applications in terms of transformations on distributed datasets. One RDD per transformation.

- Organize the RDDs into a DAG showing how data flows.
- RDD can be saved and reused or recomputed. Spark can save it to disk if the dataset does not fit in memory
- Built through parallel transformations (map, filter, group-by, join, etc). Automatically rebuilt on failure
- Controllable persistence (e.g. caching in RAM)

RDDs are designed to be “immutable”

- Create once, then reuse without changes. Spark knows lineage → can be recreated at any time → Fault-tolerance
- Avoids data inconsistency problems (no simultaneous updates) → Correctness
- Easily live in memory as on disk → Caching → Safe to share across processes/tasks → Improves performance
- Tradeoff: (**Fault-tolerance & Correctness**) vs (**Disk Memory & CPU**)

Creating a RDD

Three ways to create a RDD

- From a file or set of files
- From data in memory
- From another RDD

Example: A File-based RDD

```
> val mydata = sc.textFile("purplecow.txt")
...
15/01/29 06:20:37 INFO storage.MemoryStore:
  Block broadcast_0 stored as values to
  memory (estimated size 151.4 KB, free 296.8
  MB)

> mydata.count()

...
15/01/29 06:27:37 INFO spark.SparkContext: Job
  finished: take at <stdin>:1, took
  0.160482078 s
```

4

File: purplecow.txt

I've never seen a purple
cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

RDD: mydata

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

Spark Fundamentals

Example of an application:

```
val sc = new SparkContext("spark://...", "MyJob", home, jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is an RDD

errors.cache()

errors.count() // This is an action
```

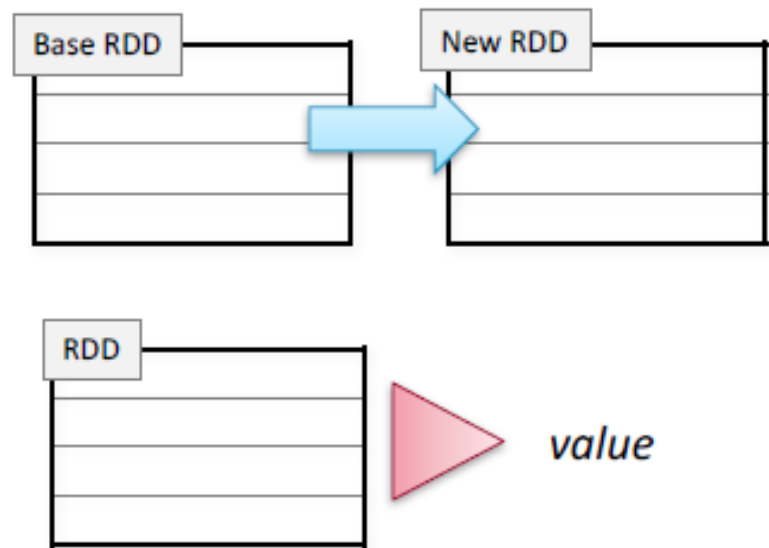
- Spark Context
- Resilient Distributed Data
- Transformations
- Actions

RDD Operations

Two types of operations

Transformations: Define a new RDD based on current RDD(s)

Actions: return values



```
val sc = new SparkContext("spark://...", "MyJob", home, jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is an RDD

errors.cache()

errors.count() // This is an action
```

RDD Transformations

- Set of operations on a RDD that define how they should be transformed
- As in relational algebra, the application of a transformation to an RDD yields a new RDD (because RDD are immutable)
- Transformations are lazily evaluated, which allow for optimizations to take place before execution
- Examples: `map()`, `filter()`, `groupByKey()`, `sortByKey()`, etc.