



# Introduction to Distributed Computing with Spark

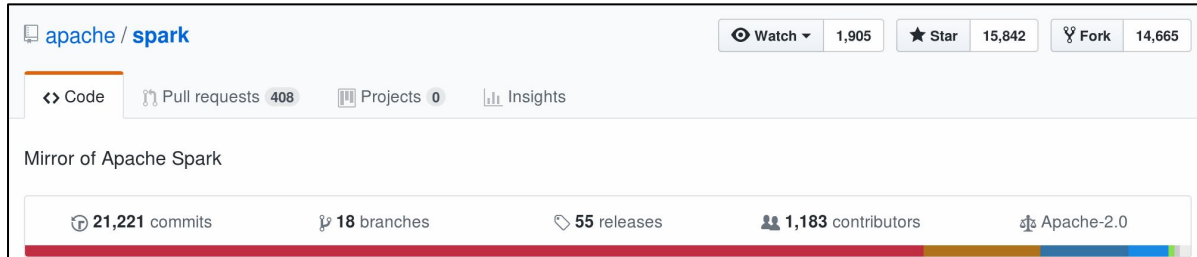


**Gina Cody School of Engineering  
and Computer Science**

Department of Computer Science and Software Engineering

# Scope

- Most active open-source project in Big Data Analytics



- Used in thousands organizations from diverse sectors:
  - Technology, banking, retail, biotech, astronomy
- Deployments with up to 8,000 nodes have been announced



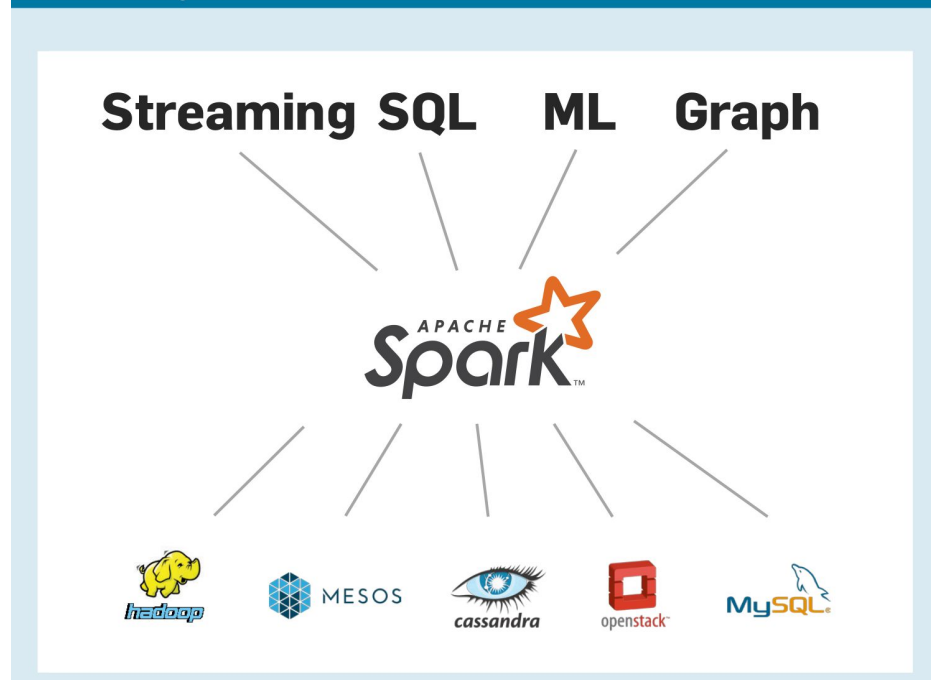
# Introduction

- **We need cluster computing to analyse Big Data**
- **There has been an explosion of specialized computing models**
  - MapReduce
  - Google's Dremel (SQL) and Pregel (graphs)
  - Hadoop's Storm, Impala, etc
- **Big Data applications need to combine many types of processing**
  - To handle data variety
  - Users need to stitch systems together

## Introduction (2)

- The profusion of systems is detrimental to
  - Usability
  - Performance
- Spark provides a uniform framework for Big Data Analytics

Figure 1. Apache Spark software stack, with specialized processing libraries implemented over the core engine.





# Highlights

- **Spark concepts**
  - Built around a *data structure*: Resilient Distributed Datasets (RDDs)
  - Programming model is MapReduce + extensions
- **Added value compared to MapReduce**
  - Easier to program because API is unified
  - Interactive explorations of Big Data are possible (pyspark)
  - More efficient to combine multiple tasks (workflows) due to lazy evaluation.



# In-memory processing

- **Spark keeps data in memory**
  - Reduces disk I/O
  - Spills to disk if not enough space
- **MapReduce lacks abstraction to leverage distributed memory**
  - Inefficient for applications that *reuse* intermediate results
  - For instance *iterative* algorithms: kmeans, PageRank



## Programming model: RDDs

RDDs are **fault-tolerant** collections of objects **partitioned** **across a cluster** that can be **manipulated in parallel**.



# RDD abstraction

- **An RDD is a read-only, partitioned collection of records.**
  - RDD is fault-tolerant, hence it is read-only.
- **Can be created through operations on:**
  - Data in stable storage (e.g., `sc.textFile`)
  - Other RDDs: transformations
  - Existing collections (`sc.parallelize`)





# Functional programming API

- Transformations:

- Return RDDs
- Are parametrized by user-provided functions

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(
    s => s.startsWith("ERROR"))
println("Total errors: "+errors.count())
```

- Actions return results to application or storage system

# RDD transformations and actions

<b>Transformations</b> Return an RDD	$map(f : T \Rightarrow U) : RDD[T] \Rightarrow RDD[U]$ $filter(f : T \Rightarrow Bool) : RDD[T] \Rightarrow RDD[T]$ $flatMap(f : T \Rightarrow Seq[U]) : RDD[T] \Rightarrow RDD[U]$ $sample(fraction : Float) : RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling) $groupByKey() : RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$ $reduceByKey(f : (V, V) \Rightarrow V) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $union() : (RDD[T], RDD[T]) \Rightarrow RDD[T]$ $join() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$ $cogroup() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$ $crossProduct() : (RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$ $mapValues(f : V \Rightarrow W) : RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning) $sort(c : Comparator[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $partitionBy(p : Partitioner[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$
<b>Actions</b> Return something else	$count() : RDD[T] \Rightarrow Long$ $collect() : RDD[T] \Rightarrow Seq[T]$ $reduce(f : (T, T) \Rightarrow T) : RDD[T] \Rightarrow T$ $lookup(k : K) : RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs) $save(path : String) : \text{Outputs RDD to a storage system, e.g., HDFS}$

(K, V) is a key-value pair  
MapReduce = flatMap +  
reduceByKey

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.



# Lazy evaluation

- **Transformations do not compute RDDs immediately**
  - The returned RDD object is a representation of the result
- **Actions trigger computations**
- **This allows optimization in the execution plan**
  - Multiple map operations are fused in one pass
  - While preserving modularity

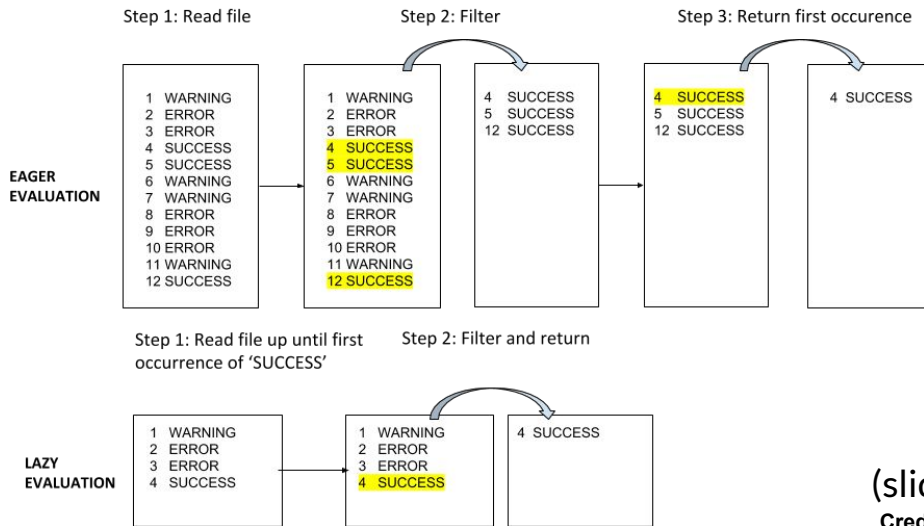
# Lazy evaluation (2)

Computations are costly and memory is limited → Compute only what is necessary

PIPELINE: Read file → Filter lines containing 'SUCCESS' → Return first line containing 'SUCCESS'

**Note:**

With *eager evaluation*, the entire file must be read and filtered. *Lazy evaluation* limits the amount of data that needs to be processed



(slide from V. Hayot-Sasson)  
Credit: Tristan Glatard - Big Data Analytics



## Persistence (or caching)

- By default RDDs are ephemeral
- They might be evicted from memory
- Users can persist RDDs in memory or on disk, by calling `.persist()`
- `persist()` accepts priorities for spilling to disk

# Example: Log Mining

Reads from HDFS

Nothing has executed so far (**lazy**)

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

Triggers execution

```
errors.count()
```

Other uses of errors won't reload data from the text file.

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()
```

```
// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
    .map(_.split('\t')(3))
    .collect()
```

Lines is never stored in memory!

# Lineage graph

An RDD maintains information on how it was derived from other datasets.

From the lineage, the RDD can be completely (re-)computed.

This is used for **fault-tolerance**.

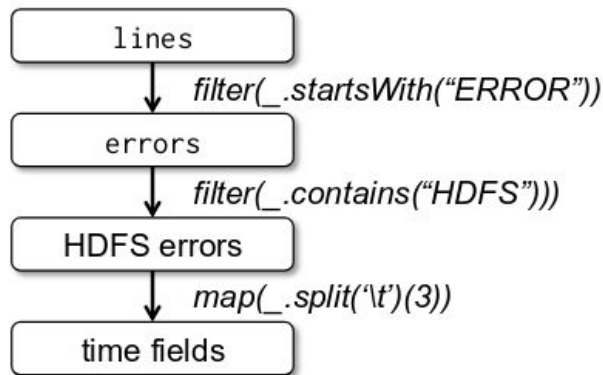


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

# Spark programming interface

- **Driver**
  - Runs user program
  - Defines RDDs and invokes actions
  - Tracks the lineage
- **Workers**
  - Long-lived processes
  - Store RDD partitions in RAM across operations

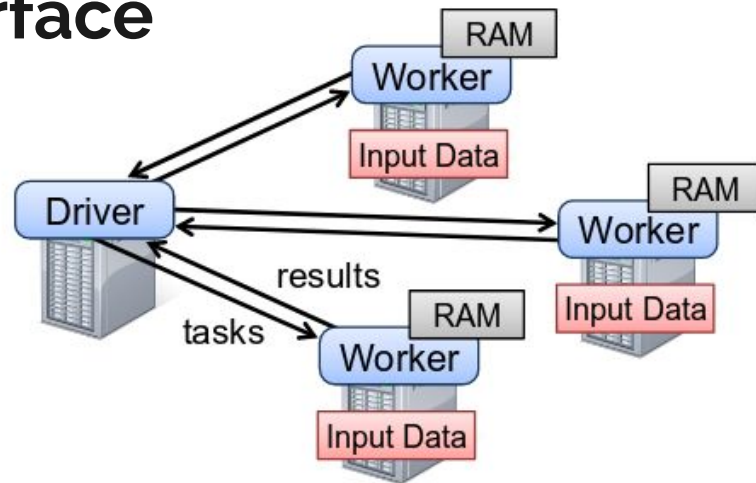


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.





# Fault tolerance

- **When a task fail**
  - If parent partitions are available: re-submit to another node
  - Otherwise: resubmit tasks to recompute missing partitions
- **Checkpointing**
  - Lineage-based recovery of RDDs may be time-consuming
  - RDDs can be checkpointed to stable storage, through a flag to `.persist()`

# Spark DataFrames

- A DataFrame is a set of records with a known schema
- A DataFrame is an RDD
- DataFrame operations map to the SQL engine

```
>>> df.printSchema()
root
|-- Nom_arrond: string (nullable = true)
|-- Invent: string (nullable = true)
|-- no_civiq: string (nullable = true)
|-- Rue: string (nullable = true)
|-- Rue_De: string (nullable = true)
|-- Rue_A: string (nullable = true)
|-- Nom_parc: string (nullable = true)
|-- Sigle: string (nullable = true)
|-- Injections: string (nullable = true)
|-- x: string (nullable = true)
|-- y: string (nullable = true)
|-- longitude: string (nullable = true)
|-- latitude: string (nullable = true)
```

```
>>> df.where(df.Nom_arrond=="Outremont").select(df.Rue)
nt().orderBy(df.Rue).show(100)
+-----+-----+
|                Rue|count|
+-----+-----+
|    Ainslie, avenue|    1|
|Antonine-Maillet,...|    1|
|    Beloeil, avenue|   26|
|    Bloomfield, avenue|    6|
|Brunante, avenue ...|    1|
|    Cambrai, place|    1|
```



## Recap

**General framework** (increases usability and performance)

**Rich programming model** (transformations, actions, libraries)

**Data locality** (as in MapReduce)

**RDDs remain in-memory** (reduces disk I/O)

**Lazy RDD evaluations** (allow for merging transformations)

**Fault-tolerance through lineage** (better than data replication)



# Installing PySpark

- Make sure Java (JDK) is installed.
- Install PySpark:

```
pip install pyspark
```

- For Windows, make sure environment variables are set correctly and winutils.exe is installed. For a step-by-step guide, see [here](#) .

# PySpark Hello World



```
import pyspark
sc = pyspark.SparkContext('local[*]')

txt = sc.textFile('sample.log')

error_lines = txt.filter(lambda line: 'error' in line)

print(error_lines.count())
```