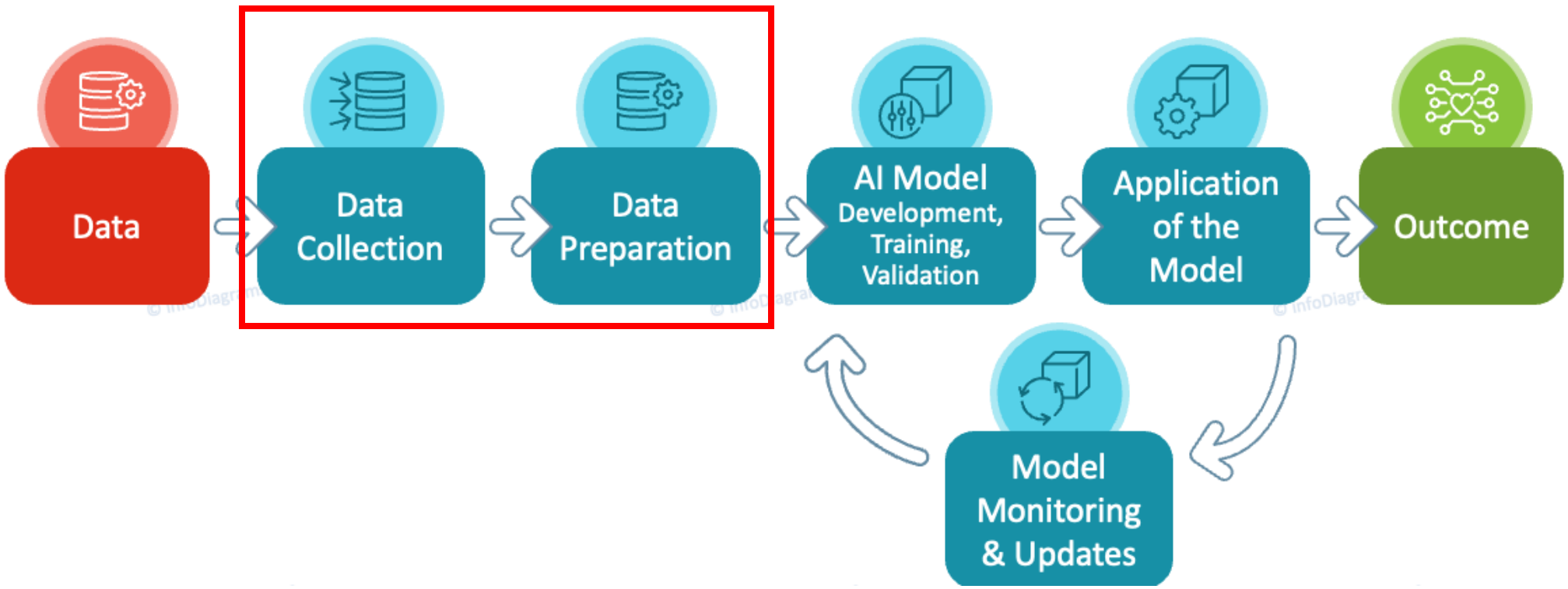
# IMEN891K/IDSC501 Data Engineering

# 13. **Big Data II**: Spark

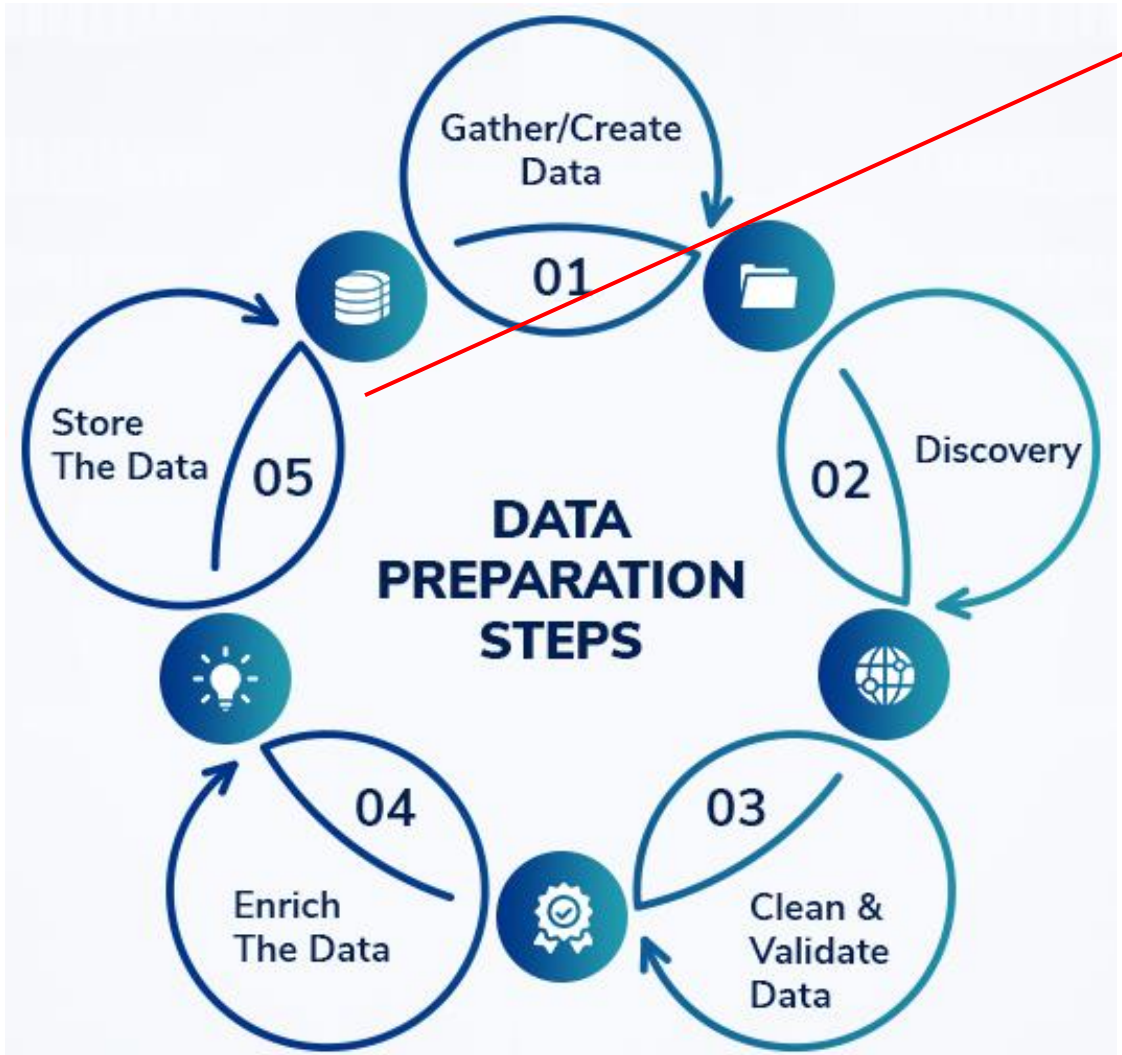
Fall 2025

Instructor: **Sangwoo Mo**

# Recap: Course Overview



# Recap: Course Overview




Lecture 7-15

# Disclaimer

The NoSQL part was heavily copy-pasted from Berkeley CS186

## Introduction to Database Systems

UC Berkeley, Fall 2024



**Alvin Cheung**
akcheung@cs.berkeley.edu

**Pronouns:** he/him/his

**OH:** Tues 11am-noon on zoom

Hi all! I'm an EECS faculty and my group works on data systems research. Looking forward to meeting you all!

# Disclaimer

The Spark part was heavily copy-pasted from Reynold Xin's Talks



**Reynold Xin**
Co–founder and Chief Architect

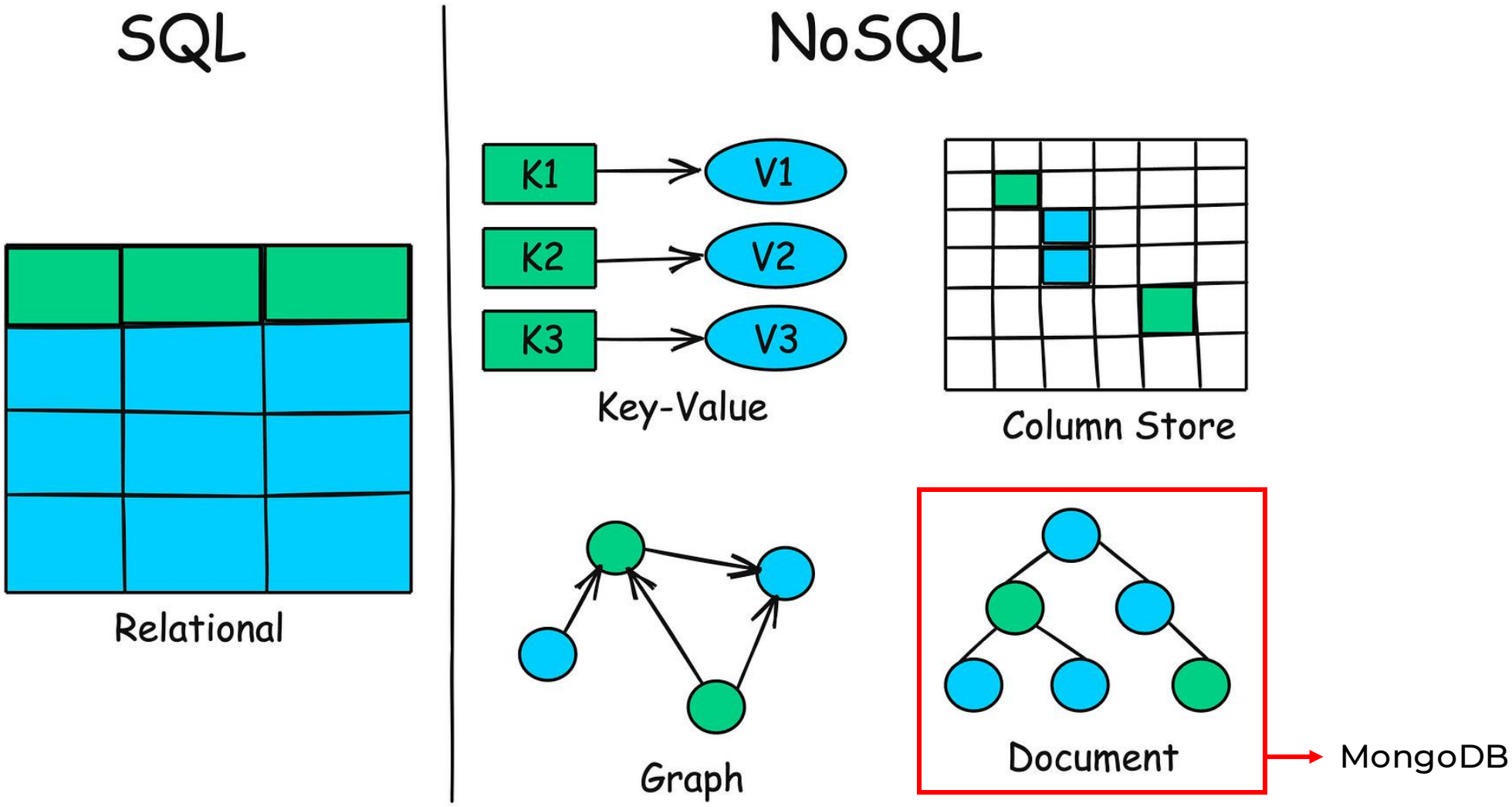
databricks

# Disclaimer

The PySpark part was heavily copy-pasted from Berkeley CS100.1x



# Recap: Database Systems




https://blog.algomaster.io/p/sql-vs-nosql-7-key-differences

# Recap: MapReduce

- Google: paper published 2004
- Free variant: Hadoop 


- MapReduce: high-level programming model and implementation for large-scale parallel data processing

# Recap: MapReduce

- Google: paper published 2004
- Free variant: Hadoop 


- MapReduce: high-level programming model and implementation for large-scale parallel data processing

# Recap: MapReduce

Data type: key-value *records*

Map function:

$$(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})$$

Reduce function:

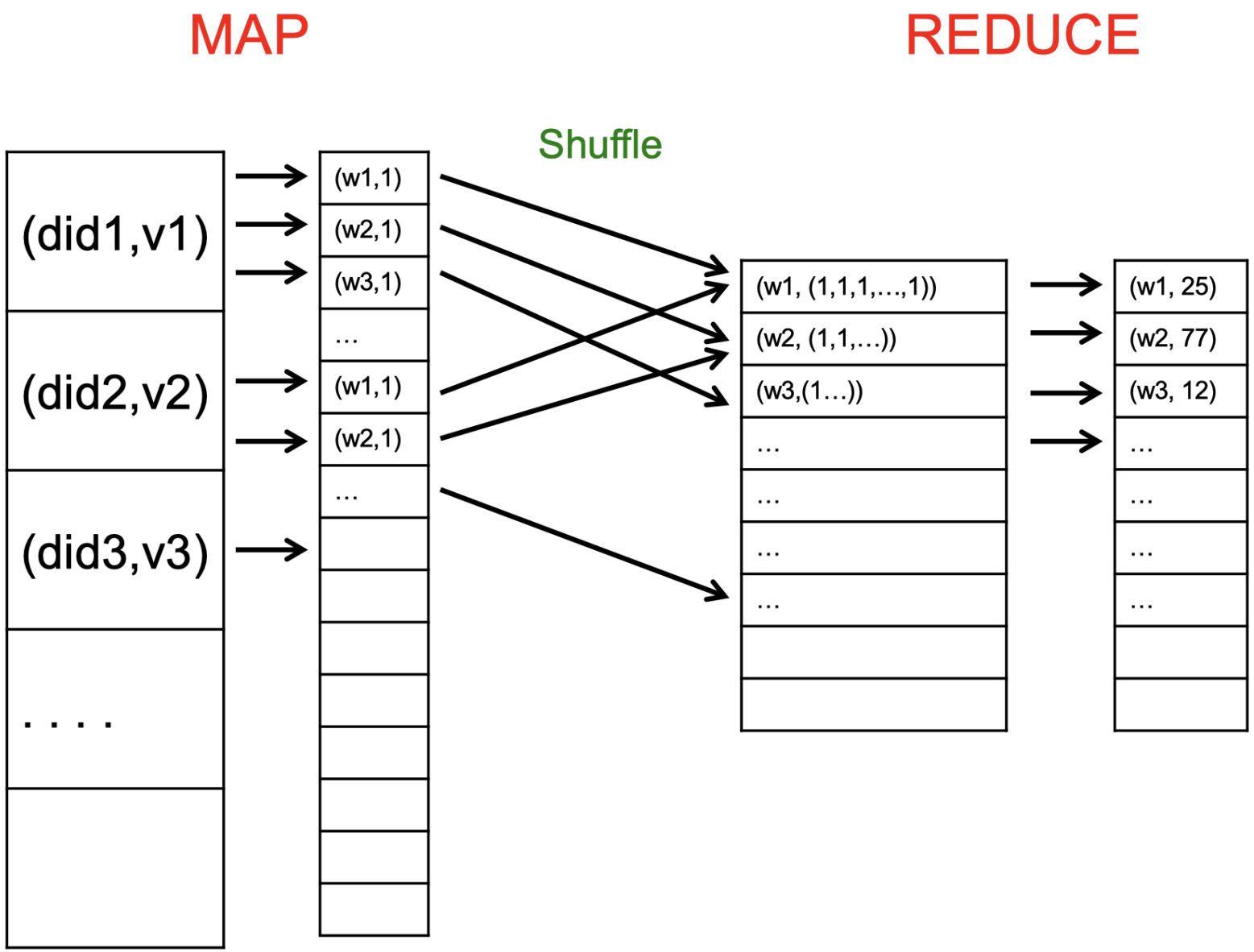
$$(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})$$

# Recap: MapReduce

- Counting the number of occurrences of each word in a large collection of documents
- Each Document
  - The key = document id (did)
  - The value = set of words (word)

```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
        emitIntermediate(w, “1”);
```

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
    for each v in values:
      result += ParseInt(v);
  emit(AsString(result));
```


MAP
REDUCE
Shuffle
(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(w1,1)
(w2,1)
(w3,1)
…
(w1,1)
(w2,1)
…
(w1, (1,1,1,…,1))
(w2, (1,1,…))
(w3,(1…))
…
…
…
…
(w1, 25)
(w2, 77)
(w3, 12)
…
…
…
…

# Problems with MapReduce

Most real applications require multiple MR steps

- Google indexing pipeline: 21 steps
- Analytics queries (e.g. count clicks & top K): 2 – 5 steps
- Iterative algorithms (e.g. PageRank): 10's of steps

Multi-step jobs create spaghetti code

- 21 MR steps -> 21 mapper and reducer classes
- Lots of boilerplate code per step

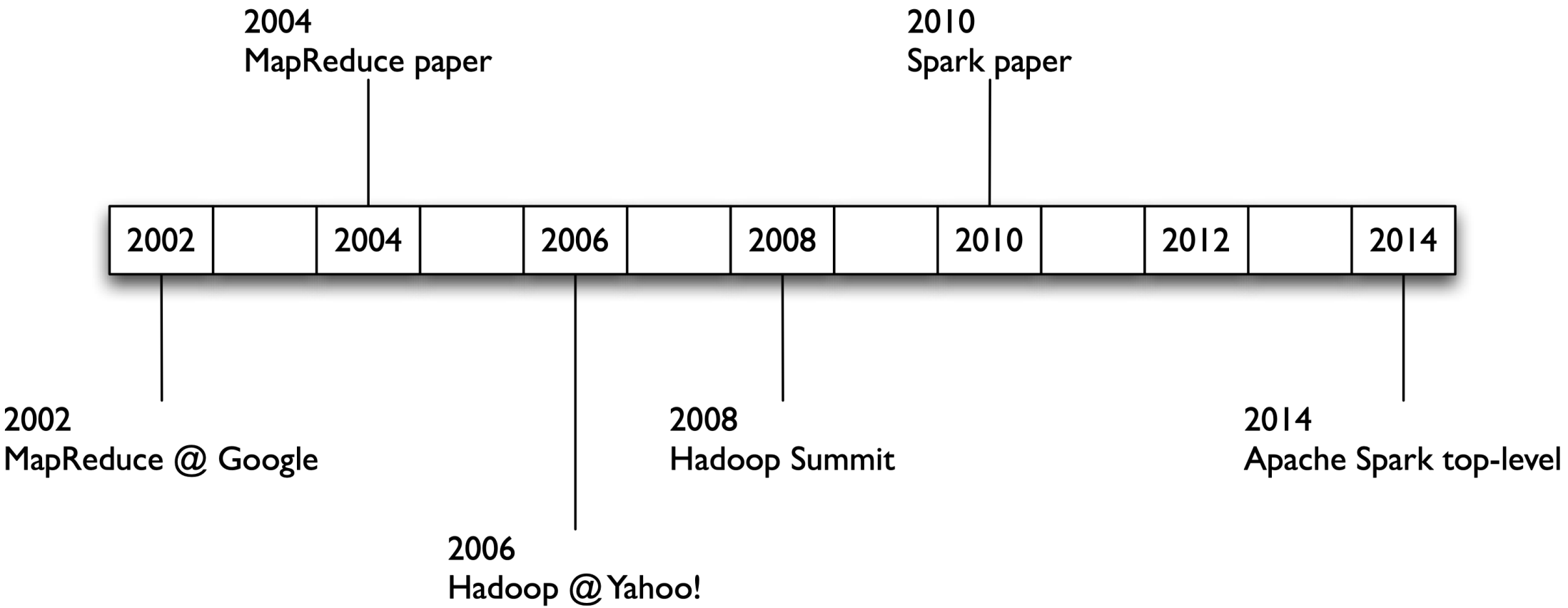
# Problems with MapReduce

MapReduce use cases showed two major limitations:

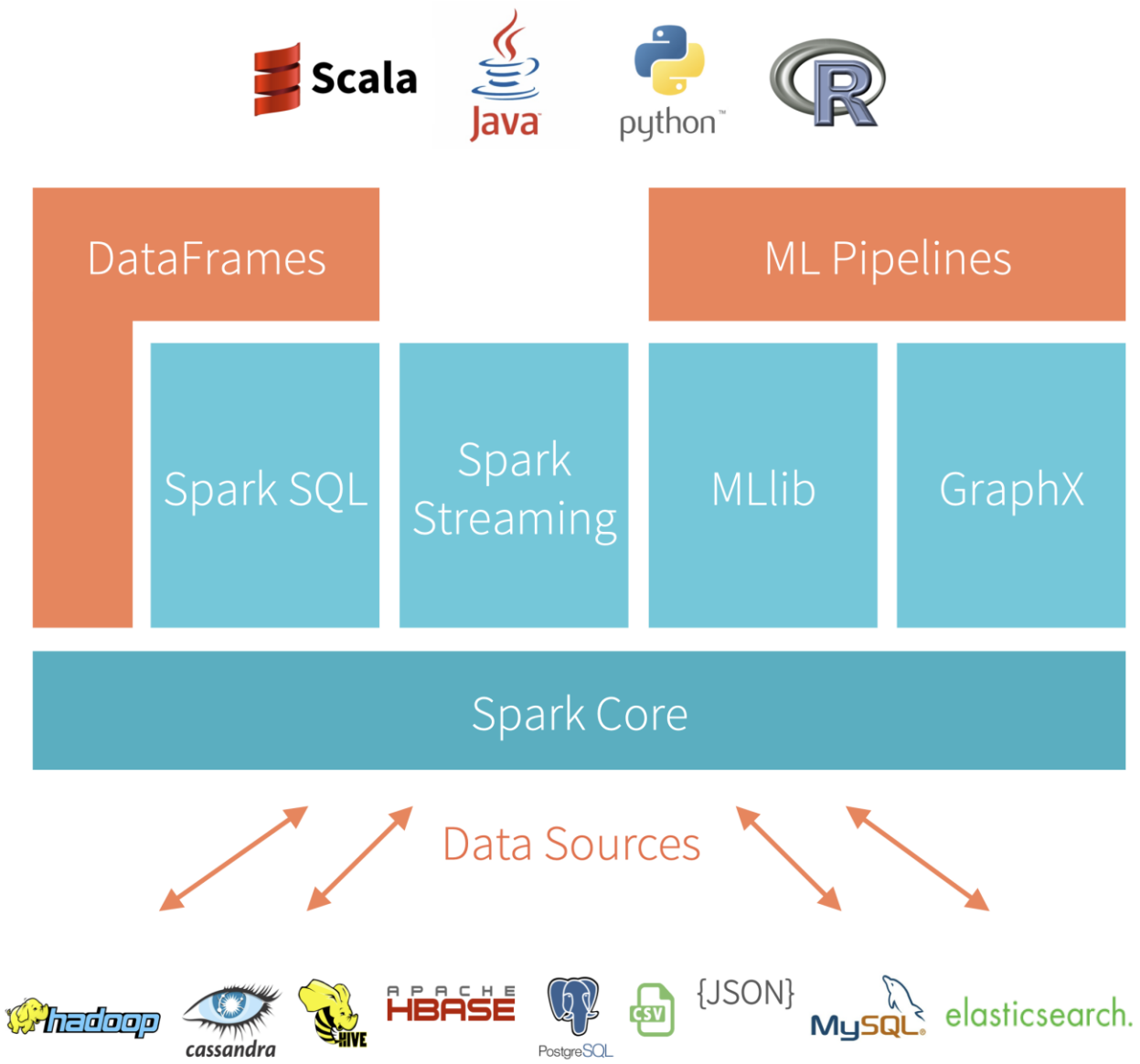
1. difficulty of programming directly in MR.
2. Performance bottlenecks

In short, MR doesn't compose well for large applications

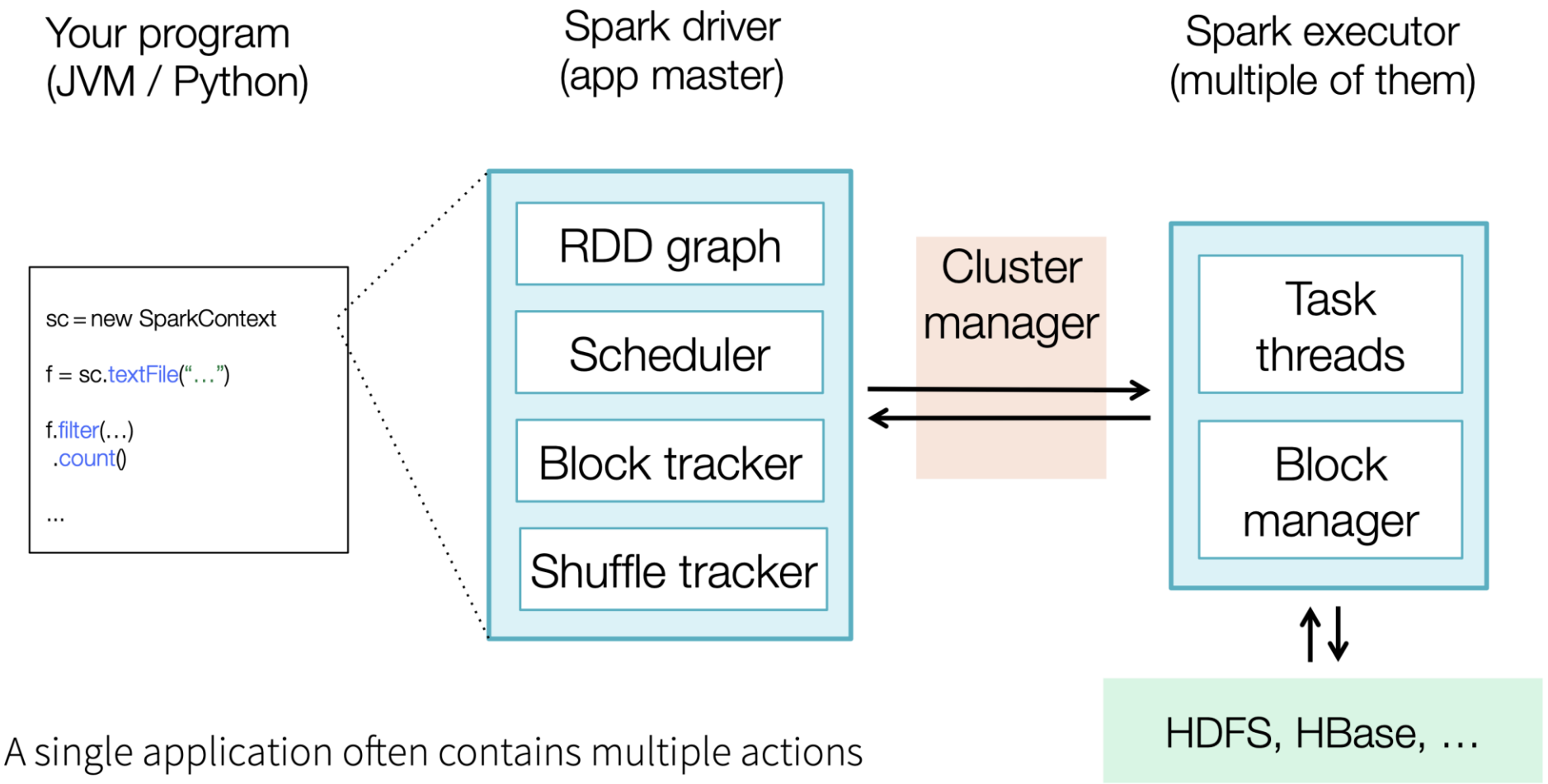
# Spark: A Brief History



# Spark Framework



# Spark Framework




A single application often contains multiple actions

# Why Spark? – Programmability

```
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

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

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

WordCount in 50+ lines of Java MR

```
1  val f = sc.textFile(inputPath)
2  val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
3  w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

# Why Spark? – Performance

Time to sort 100TB

2013 Record: Hadoop

2100 machines



72 minutes



2014 Record: Spark

207 machines



23 minutes



# Resilient Distributed Datasets (RDD)

- RDD = Resilient Distributed Datasets
  - A distributed, immutable relation, together with its *lineage*
  - Lineage: expression that says how that relation was computed (e.g., a relational algebra plan)
- All inputs, outputs, and intermediates are stored as RDDs
- If a server crashes, its RDD in main memory is lost. However, the driver (=master node) knows the lineage, and will simply recompute the lost partition of the RDD

# Resilient Distributed Datasets (RDD)

> **RDD = main memory + lineage**

1. **Main memory** → Eliminates redundant disk I/O, enabling fast iterative computation (e.g., ML, graph algorithms).
2. **Lineage** → Stores only *how* data was derived, so Spark can recover lost data without checkpoints.

This design gave Spark:

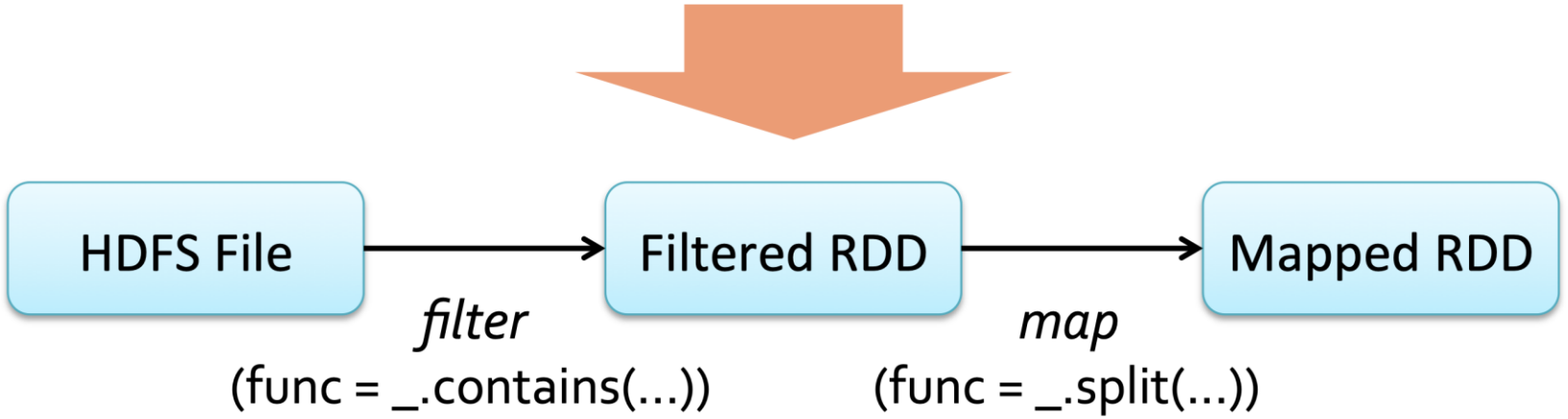
- ~10–100× faster performance than Hadoop
- The ability to unify **batch, streaming, and SQL** in one engine
- Fault tolerance without heavy disk overhead

# Resilient Distributed Datasets (RDD)

RDDs track *lineage* information that can be used to efficiently reconstruct lost partitions

Ex:

```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```



# MapReduce vs RDD

> Hadoop: "**Write to disk after every step**"
> Spark: "**Keep everything in memory and execute as one logical plan.**"

**Example:**

- Hadoop:

```scss
Map() → write to HDFS → Reduce() → write to HDFS → next job...
```

- Spark:

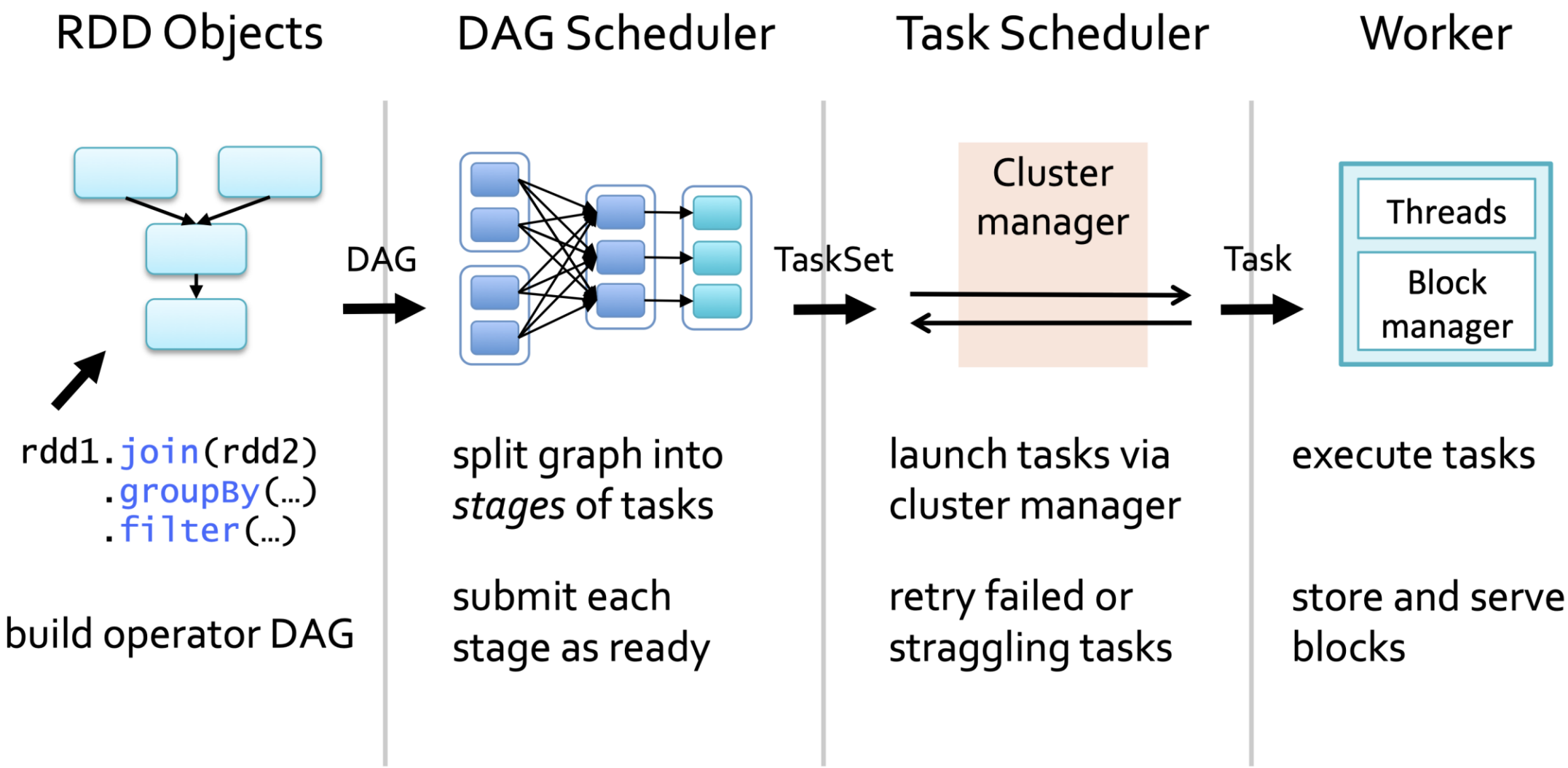
```python
rdd.map(...).filter(...).groupBy(...).collect()
```

→ Spark builds a **DAG** internally and executes it efficiently in memory.

# MapReduce vs RDD

| Category | Hadoop MapReduce | Spark (RDD) |
|---|---|---|
| **Core Idea** | Disk-based **batch execution** | In-memory **resilient computation** |
| **Data Storage** | Writes intermediate results to **HDFS** after every stage | Keeps data in **memory (RDD)**, reconstructs via **lineage** if lost |
| **Fault Tolerance** | **Re-executes tasks** on failure (checkpoint-based) | **Recomputes** lost partitions from lineage graph |
| **Performance** | High disk I/O overhead (read/write between every step) | 10–100× faster via in-memory execution |
| **Iteration / Pipeline** | Each iteration = new MapReduce job | Builds a **DAG** of transformations executed together |
| **Programming Model** | Limited to `map` / `reduce`, Java-heavy | Rich functional API (`map`, `filter`, `join`, `reduceByKey`), Python/Scala-friendly |
| **Flexibility** | Batch only | Unified engine for batch, streaming, and SQL |

# RDD Execution Process



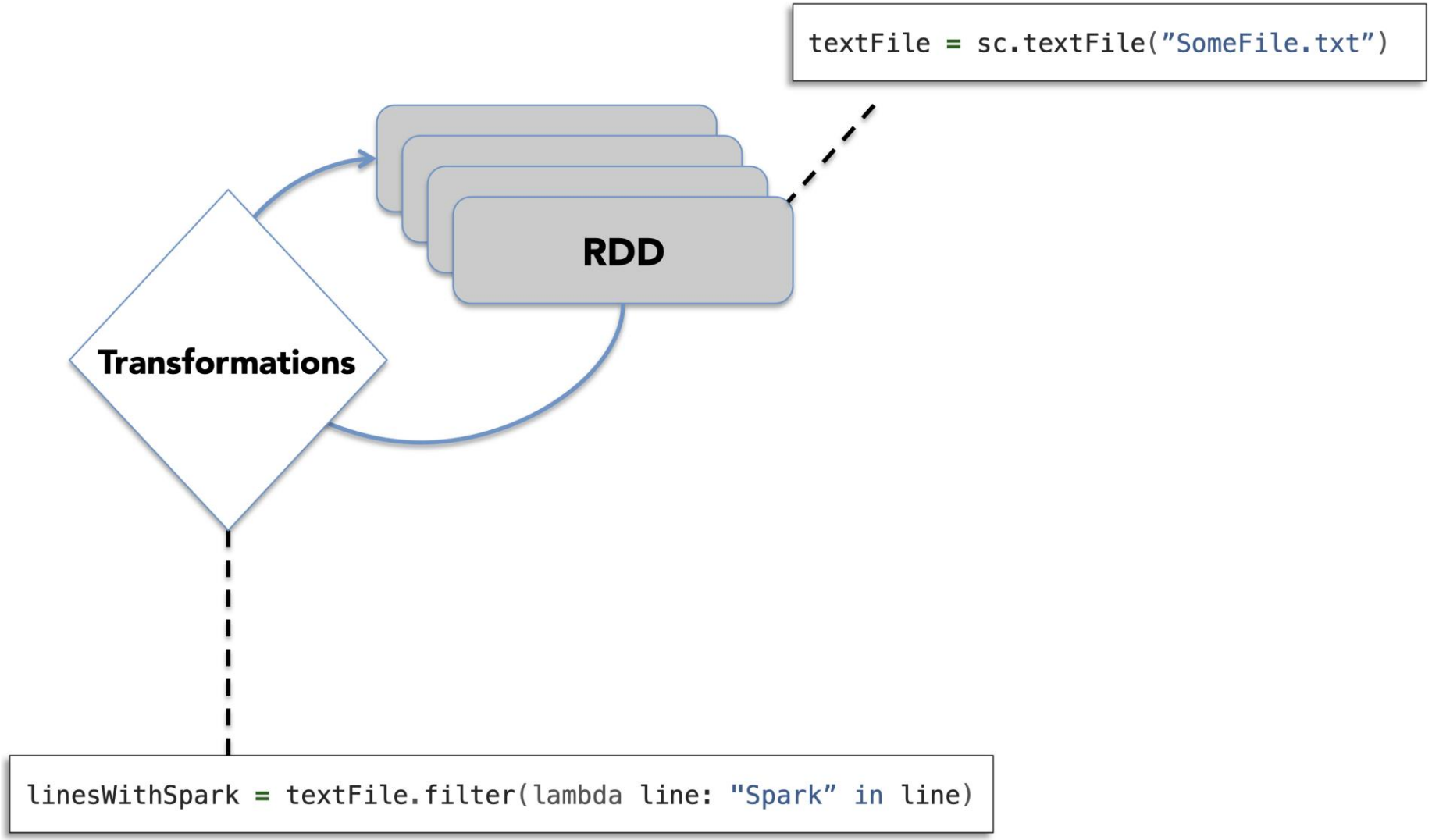
# RDD Programming

- A Spark program consists of:
  - Transformations (map, reduceByKey, join…). Lazy
  - Actions (count, reduce, save...). Eager
- Eager: operators are executed immediately
- Lazy: operators are not executed immediately
  - A *operator tree* is constructed in memory instead
  - Similar to a relational algebra tree

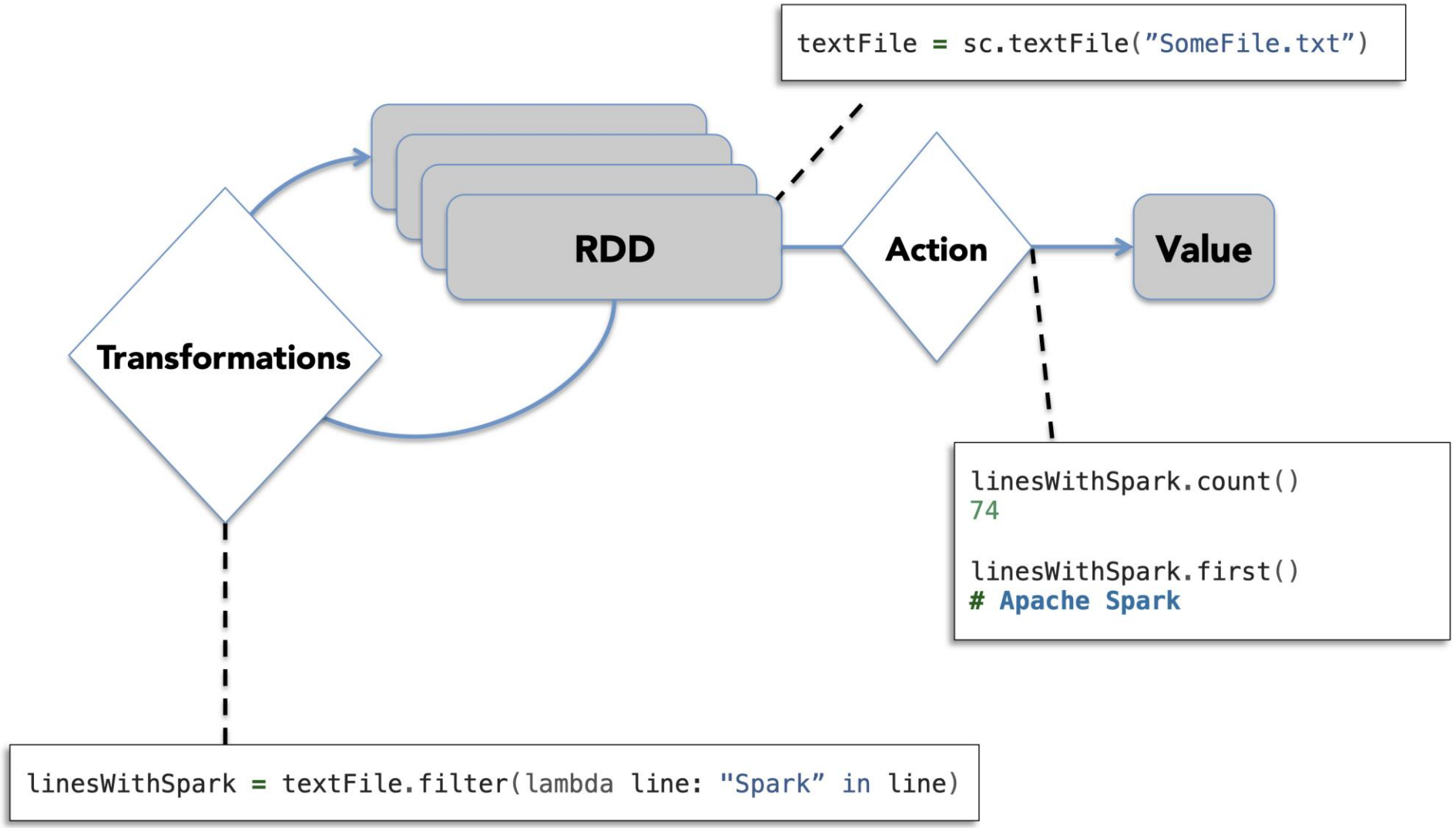
# RDD Programming



# RDD Programming



# RDD Programming



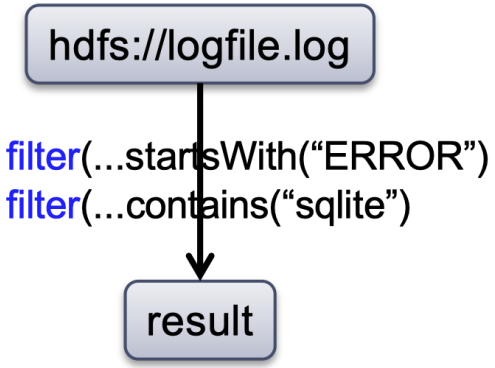
# RDD Programming

| Transformations: | |
|---|---|
| map(f : T -> U): | RDD<T> -> RDD<U> |
| flatMap(f: T -> Seq(U)): | RDD<T> -> RDD<U> |
| filter(f:T->Bool): | RDD<T> -> RDD<T> |
| groupByKey(): | RDD<(K,V)> -> RDD<(K,Seq[V])> |
| reduceByKey(F:(V,V)-> V): | RDD<(K,V)> -> RDD<(K,V)> |
| union(): | (RDD<T>,RDD<T>) -> RDD<T> |
| join(): | (RDD<(K,V)>,RDD<(K,W)>) -> RDD<(K,(V,W))> |
| cogroup(): | (RDD<(K,V)>,RDD<(K,W)>)-><br>RDD<(K,(Seq<V>,Seq<W>))> |
| crossProduct(): | (RDD<T>,RDD<U>) -> RDD<(T,U)> |

| Actions: | |
|---|---|
| count(): | RDD<T> -> Long |
| collect(): | RDD<T> -> Seq<T> |
| reduce(f:(T,T)->T): | RDD<T> -> T |
| save(path:String): | Outputs RDD to a storage system e.g., HDFS |

# RDD Programming – Persistence (a.k.a. Cache)

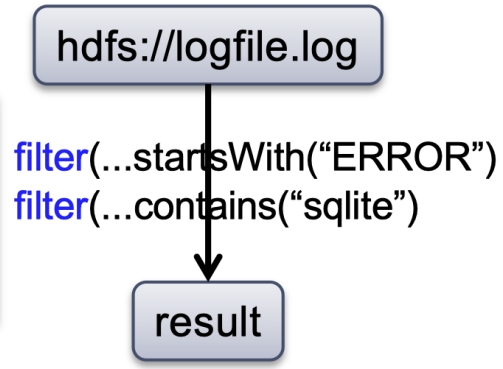
```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect();
```




If any server fails before the end, then Spark must restart

# RDD Programming – Persistence (a.k.a. Cache)

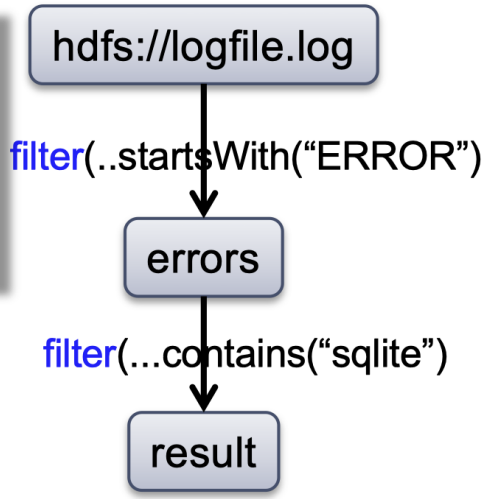
```
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect();
```




If any server fails before the end, then Spark must restart

```
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
errors.persist();
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect()
```

New RDD




Spark can recompute the result from errors

# RDD Programming – Example

R(A,B)
S(A,C)

```
SELECT count(*)  FROM R, S
WHERE R.B > 200 and S.C < 100  and R.A = S.A
```

```
R = s.read().textFile("R.csv").map(parseRecord).persist();
S = s.read().textFile("S.csv").map(parseRecord).persist();
```

Parses each line into an object

persisting on disk

# RDD Programming – Example

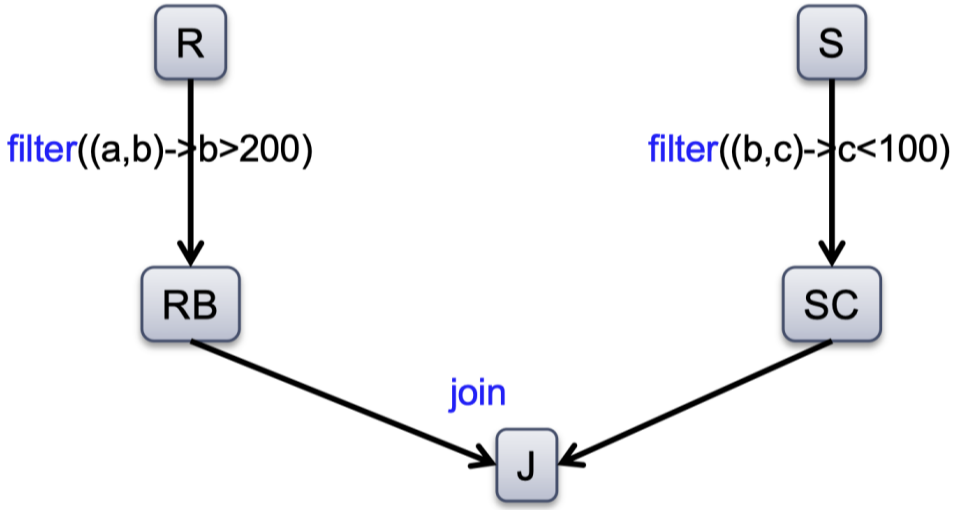
R(A,B)
S(A,C)

```
SELECT count(*)  FROM R, S
WHERE R.B > 200 and S.C < 100  and R.A = S.A
```

```
R = s.read().textFile("R.csv").map(parseRecord).persist();
S = s.read().textFile("S.csv").map(parseRecord).persist();
RB = R.filter(t -> t.b > 200).persist();
SC = S.filter(t -> t.c < 100).persist();
J = RB.join(SC).persist();
J.count();
```

transformations

action



# DataFrames in Spark

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

DSL designed for common tasks

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

Available in Python, Scala, Java, and R (via SparkR)

# DataFrames in Spark

DataFrames hold rows with a known schema and offer relational operations on them through a DSL

```
val users = spark.sql("select * from users")

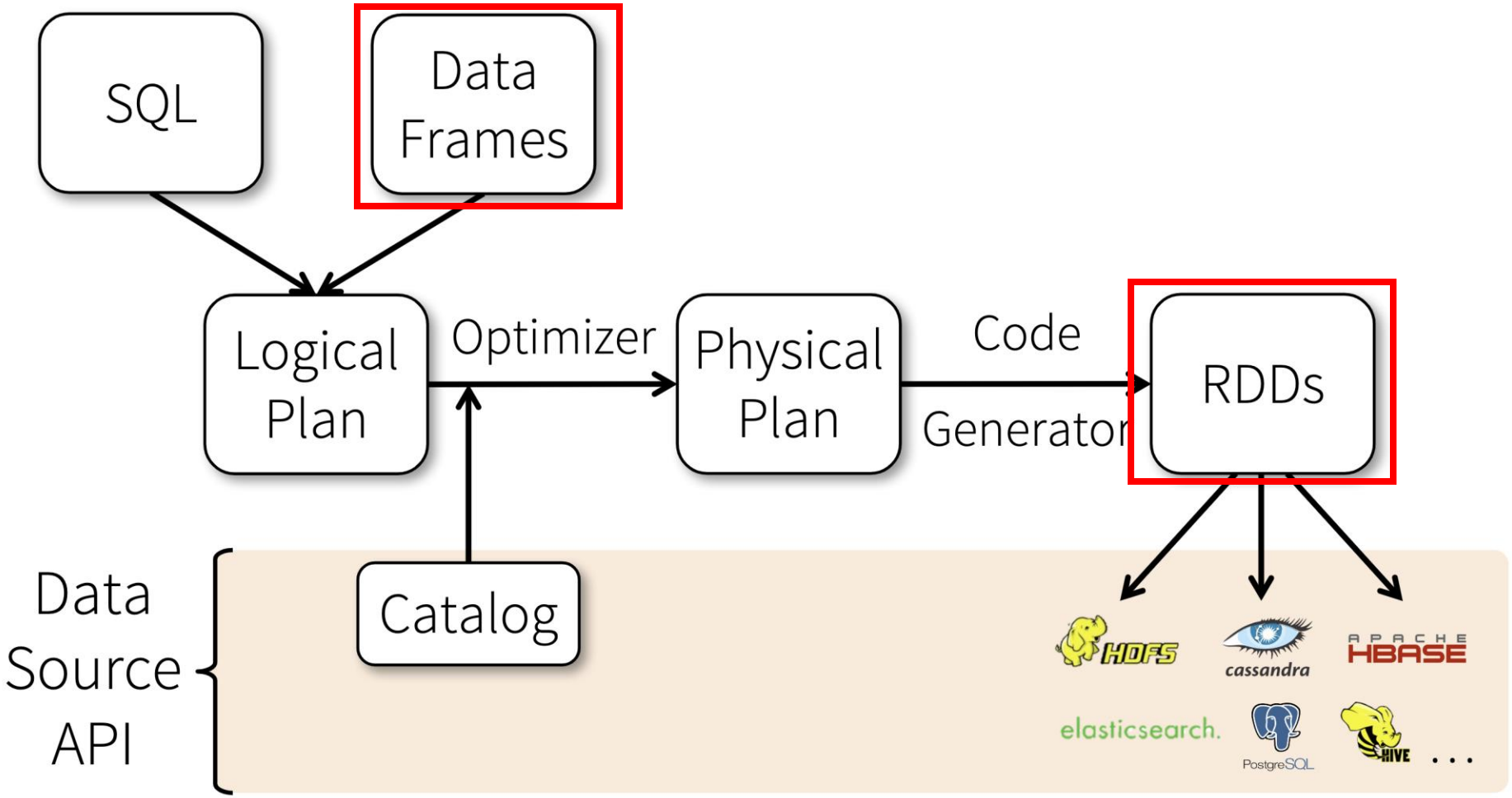
val massUsers = users('country === "Canada")

massUsers.count()

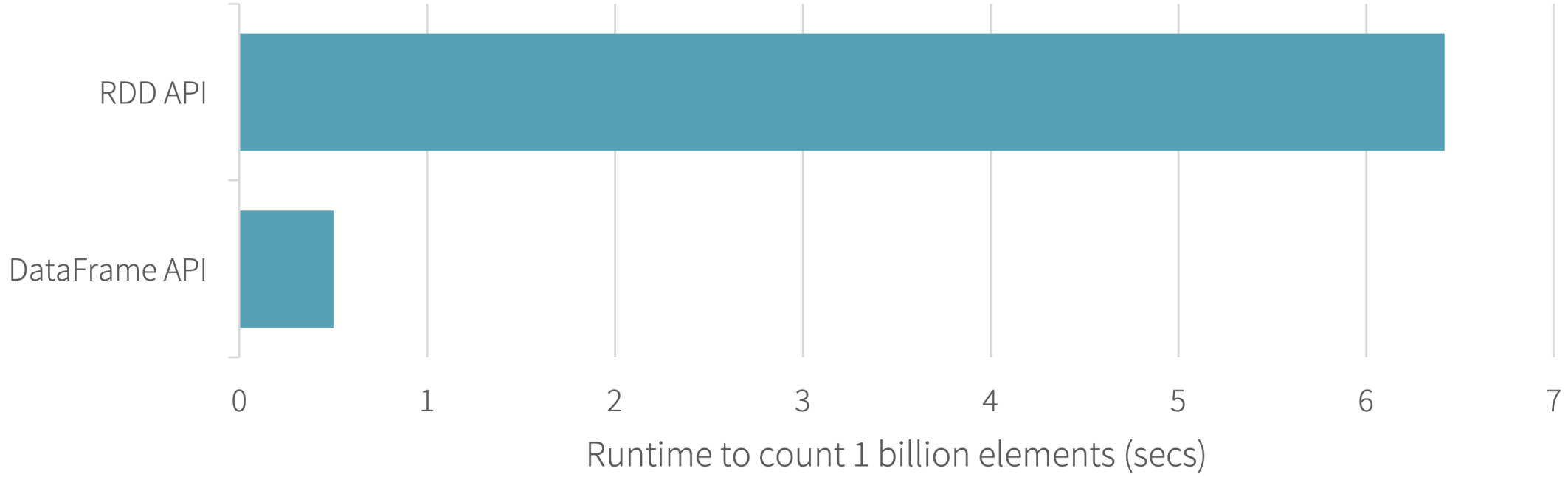
massUsers.groupBy("name").avg("age")
```

Expression AST

# DataFrames in Spark



# DataFrames in Spark



# RDD vs DataFrame

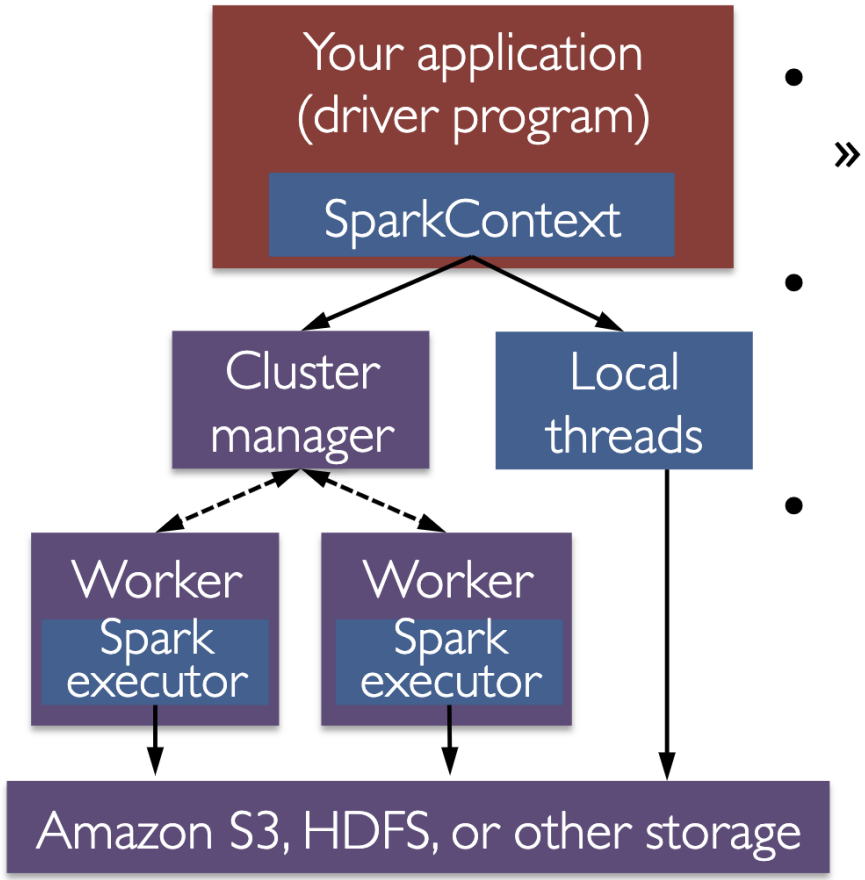
| Aspect | RDD | DataFrame |
| --- | --- | --- |
| **Level** | Low-level API (manual control) | High-level API (table abstraction) |
| **Optimization** | No automatic optimization | Catalyst Optimizer (automatic) |
| **Performance** | Slower (object-based) | Faster (columnar, Tungsten) |
| **Schema** | Unstructured | Schema-aware |
| **Ease of Use** | Complex functional code | Simple SQL-like syntax |
| **When to Use** | Custom logic, unstructured data | ETL, analytics, ML pipelines |

# pySpark

- We are using the Python programming interface to Spark (pySpark)
- pySpark provides an easy-to-use programming abstraction and parallel runtime:
  - » "Here's an operation, run it on all of the data"

- http://spark.apache.org/docs/latest/programming-guide.html
- http://spark.apache.org/docs/latest/api/python/index.html

# pySpark – Spark Driver and Workers




- A Spark program is two programs:
  - » A driver program and a workers program
- Worker programs run on cluster nodes or in local threads
- RDDs are distributed across workers

# pySpark – Spark Context

- A Spark program first creates a **`SparkContext`** object
  - » Tells Spark how and where to access a cluster
  - » pySpark shell and Databricks Cloud automatically create the **`sc`** variable
  - » iPython and programs must use a constructor to create a new **`SparkContext`**
- Use **`SparkContext`** to create RDDs

# pySpark – Master Parameter

- The **`master`** parameter for a **`SparkContext`** determines which type and size of cluster to use

| Master Parameter | Description |
|---|---|
| `local` | run Spark locally with one worker thread (no parallelism) |
| `local[K]` | run Spark locally with K worker threads (ideally set to number of 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) |

# pySpark – Creating RDDs

- Create RDDs from Python collections (lists)

```
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> rDD = sc.parallelize(data, 4)
>>> rDD
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

No computation occurs with **`sc.parallelize()`**

- Spark only records how to create the RDD with four partitions

# pySpark – Creating RDDs

- From HDFS, text files, Hypertable, Amazon S3, Apache Hbase, SequenceFiles, any other Hadoop **`InputFormat`**, and directory or glob wildcard: **`/data/201404*`**

```
>>> distFile = sc.textFile("README.md", 4)

>>> distFile

MappedRDD[2] at textFile at

   NativeMethodAccessorImpl.java:-2
```

# Recap: RDD Programming

- A Spark program consists of:
  - Transformations (map, reduceByKey, join…). Lazy
  - Actions (count, reduce, save...). Eager
- Eager: operators are executed immediately
- Lazy: operators are not executed immediately
  - A *operator tree* is constructed in memory instead
  - Similar to a relational algebra tree

# pySpark – RDD Transformations

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.map(lambda x: x * 2)
RDD: [1, 2, 3, 4] → [2, 4, 6, 8]

>>> rdd.filter(lambda x: x % 2 == 0)
RDD: [1, 2, 3, 4] → [2, 4]

>>> rdd2 = sc.parallelize([1, 4, 2, 2, 3])
>>> rdd2.distinct()
RDD: [1, 4, 2, 2, 3] → [1, 4, 2, 3]
```

Function literals (green) are closures automatically passed to workers

# pySpark – RDD Transformations

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.Map(lambda x: [x, x+5])
RDD: [1, 2, 3] → [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(lambda x: [x, x+5])
RDD: [1, 2, 3] → [1, 6, 2, 7, 3, 8]
```

# pySpark – RDD Transformations

```
>>> rdd = sc.parallelize([(1,2), (3,4), (3,6)])
>>> rdd.reduceByKey(lambda a, b: a + b)
RDD: [(1,2), (3,4), (3,6)] → [(1,2), (3,10)]

>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.sortByKey()
RDD: [(1,'a'), (2,'c'), (1,'b')] →
                [(1,'a'), (1,'b'), (2,'c')]
```

# pySpark – RDD Transformations

```
>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.groupByKey()
RDD: [(1,'a'), (1,'b'), (2,'c')] →
                [(1,['a','b']), (2,['c'])]
```

Be careful using **groupByKey()** as it can cause a lot of data movement across the network and create large Iterables at workers

# pySpark – RDD Actions

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.reduce(lambda a, b: a * b)
Value: 6

>>> rdd.take(2)
Value: [1,2] # as list

>>> rdd.collect()
Value: [1,2,3] # as list
```

# Recap: DataFrames in Spark

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

DSL designed for common tasks

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

Available in Python, Scala, Java, and R (via SparkR)

# pySpark – DataFrames

```
# Convert Spark DataFrame to Pandas
pandas_df = spark_df.toPandas()

# Create a Spark DataFrame from Pandas
spark_df = context.createDataFrame(pandas_df)
```

# pySpark – DataFrames

- SparkSQL and Spark DataFrames `join()` supports:
  - » inner, outer, left outer, right outer, semijoin

- For Pair RDDs, pySpark supports:
  - » inner join(), leftOuterJoin(), rightOuterJoin(), fullOuterJoin()

# pySpark – DataFrames

- X.join(Y)
  - » Return RDD of all pairs of elements with matching keys in X and Y
  - » Each pair is (k, (v1, v2)) tuple, where (k, v1) is in X and (k, v2) is in Y

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> sorted(x.join(y).collect())

Value: [('a', (1, 2)), ('a', (1, 3))]
```