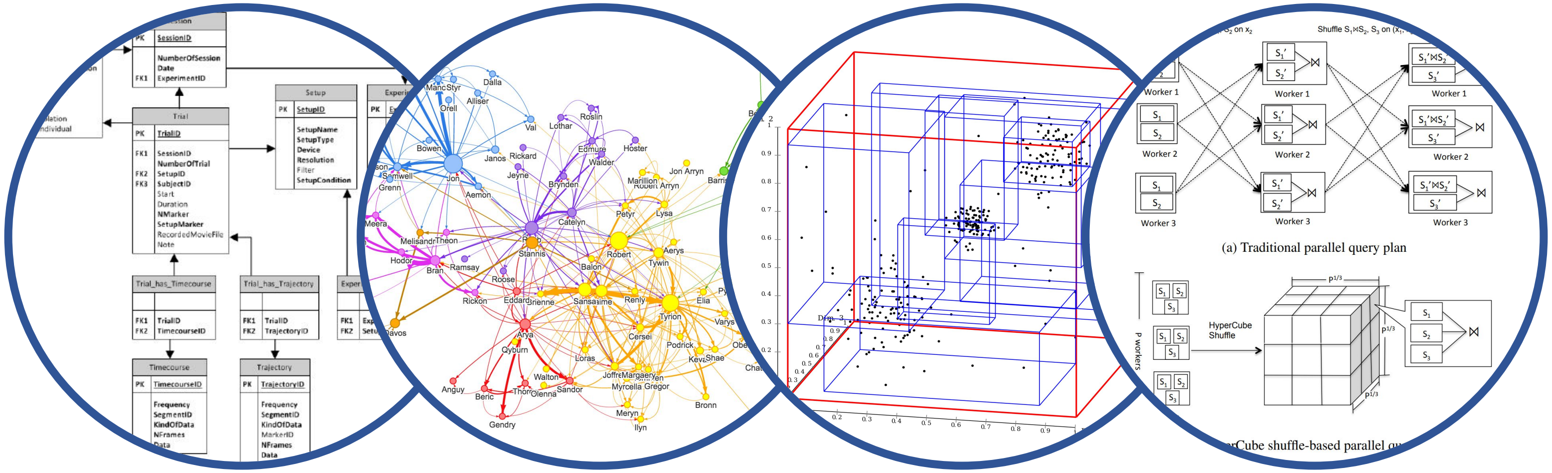


Introduction to Data Management

MapReduce and Spark

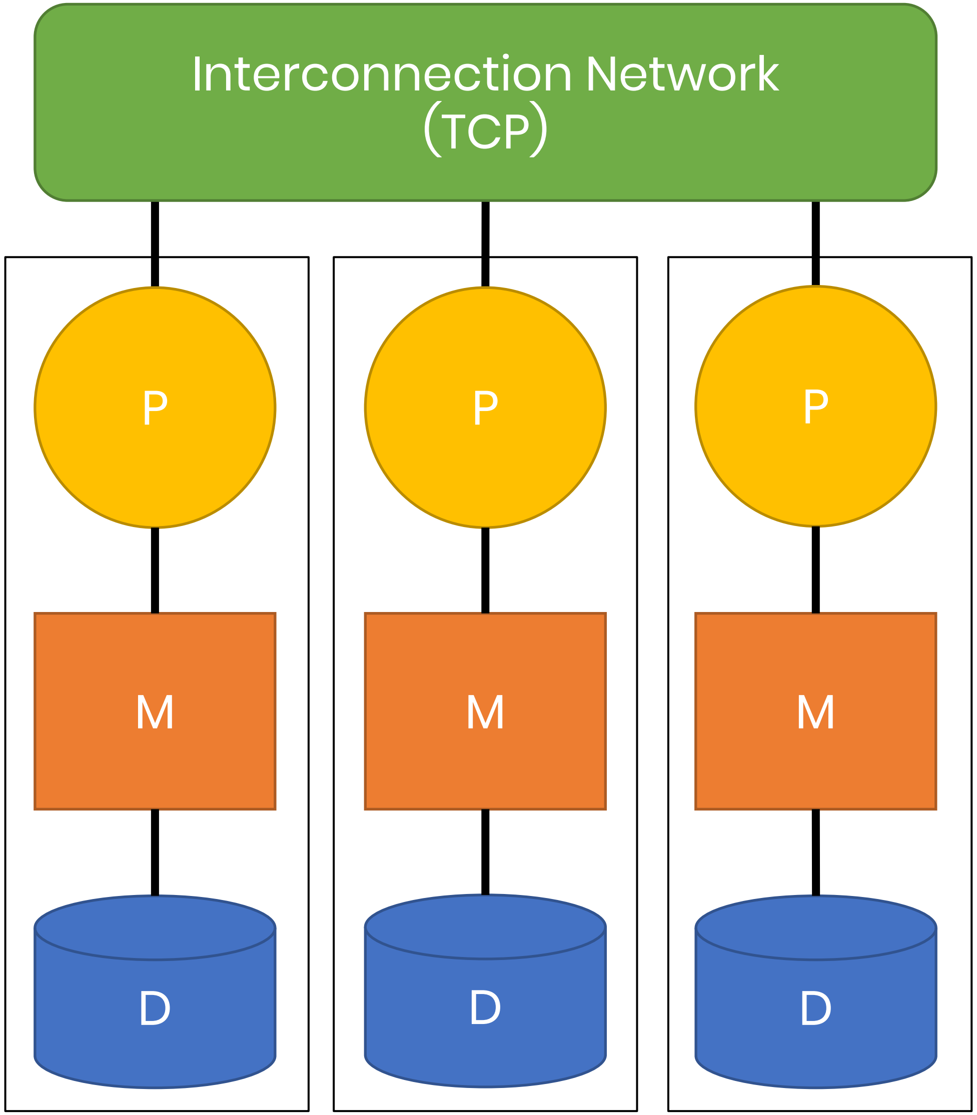
Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Announcements

- HW5 due tomorrow

Recap: Shared-Nothing Architecture



- Uses cheap, commodity hardware
- No contention for memory and high availability
- Theoretically can **scale infinitely**
- Hardest to implement on

teradata.

APACHE Spark

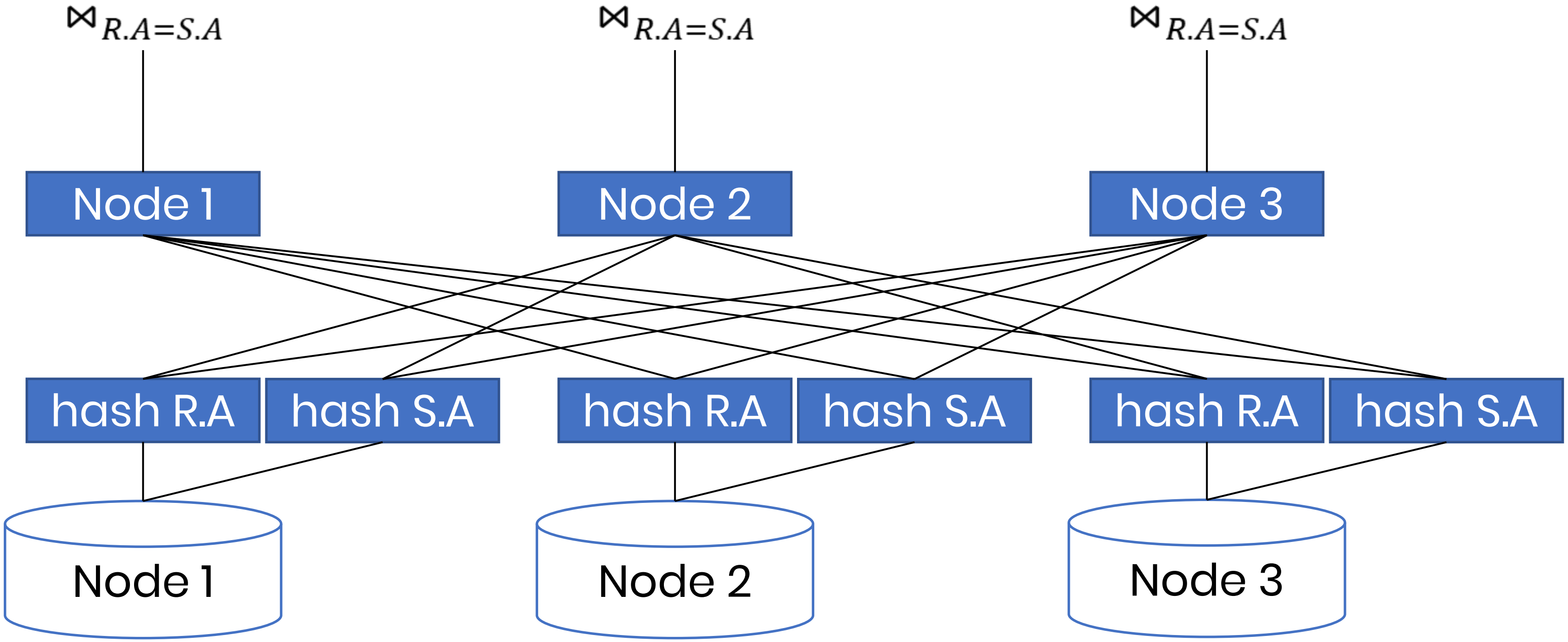
MySQL Cluster

Recap: Partitioned Hash Equijoin

1. Hash shuffle tuples on join attributes
2. Local join

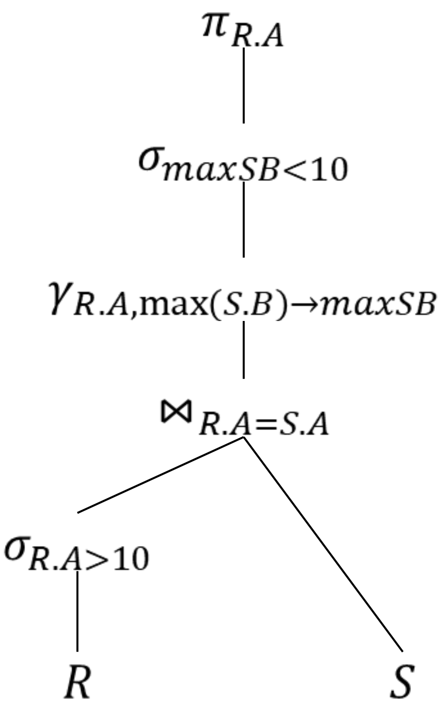
Assume:
R and S are block partitioned

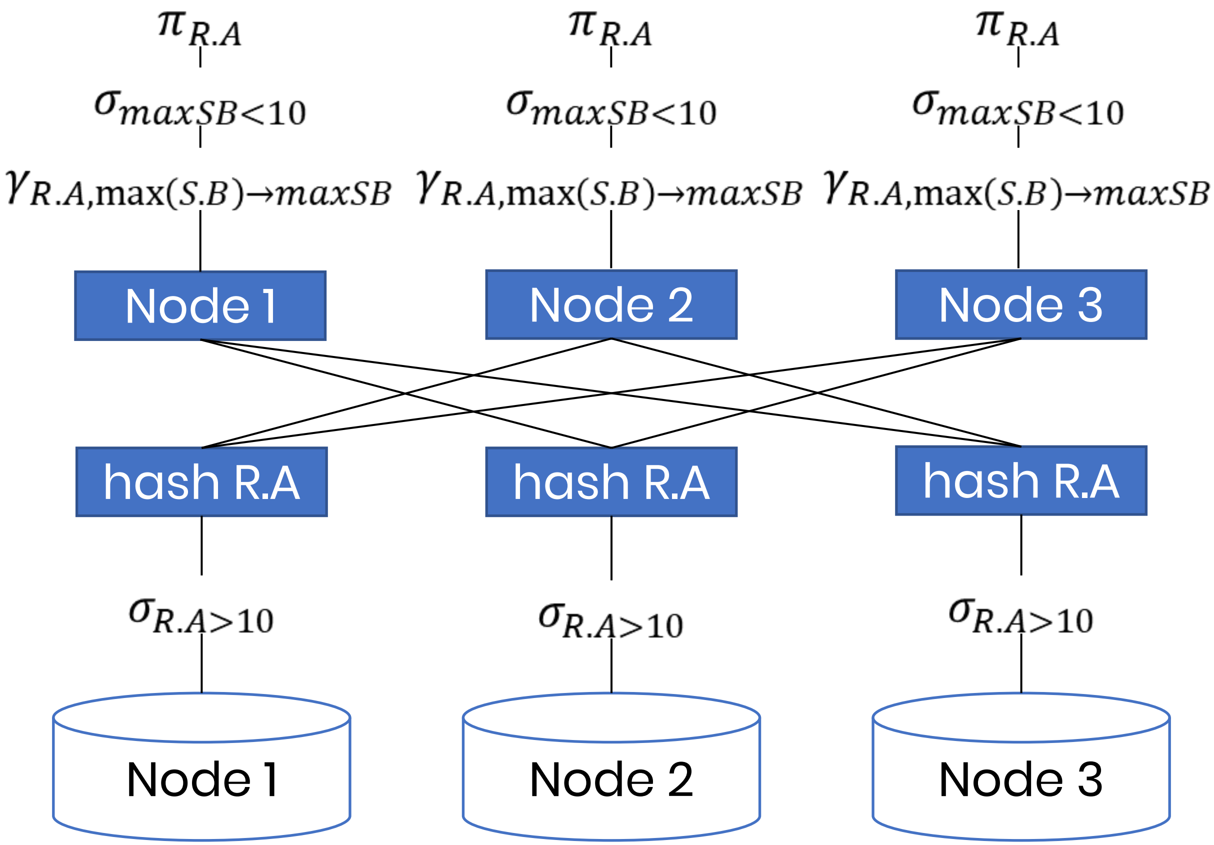
```
SELECT *
  FROM R, S
 WHERE R.A = S.A
```



Recap: Parallelize Any Query

Assume:
R is block partitioned
S is hash partitioned on A





What if..

….we don’t want to send the full data everywhere?
…we don’t want to express everything as SQL queries?

Apache Hadoop

- Two-part distributed system of data **storage** and **processing**
  - Storage: **Hadoop Distributed File System (HDFS)**
    - Originated from Google File System published 2003
  - Processing: **Hadoop MapReduce**
    - Algorithm originally published from Google circa 2004

Distributed File System (DFS)

- Purpose is to **store and manage access to large files** (tables) that are terabytes or petabytes large
- 10000-foot view of structure:
  - Files are partitioned into **chunks** (~64MB)
  - Each chunk is **replicated 3+ times** to provide **fault tolerance**
- Lots of different implementations:
  - HDFS
  - Google Cloud Storage (GCS)
  - Amazon S3
  - …

MapReduce

- MapReduce provided a fundamental and easy to use distributed **programming paradigm**
- Program pattern for MapReduce:
  - **Map**:
    - Read data from disks
    - Extract info from each tuple
    - Transform it into a useful key-value format
  - **Shuffle** key-value pairs into groups based on keys
  - **Reduce**:
    - Perform analysis on groups
    - Write results to disks
- Complex jobs need multiple map-then-reduce phases

MapReduce Data Model

- A file is a bag of (key, value) pairs
- A MapReduce program
  - Input: a bag of (input key, value) pairs
  - Output: a bag of (output key, value) pairs
- A map step
  - Input: a bag of (input key, value) pairs
  - Output: a bag of (intermediate key, value) pairs
- A (optional) combine step
  - Input: a bag of (intermediate key, value) pairs
  - Output: a bag of (intermediate key, value) pairs
- A reduce step
  - Input: a bag of (intermediate key, value) pairs
  - Output: a bag of (output key, value) pairs

MapReduce Counting Example

Count the number of times each word appears in a collection of text documents.

System will iterate through all data elements in parallel

```
Map(Document d):
  for each word w in d:
    emitIntermediate(w, 1)
```

System will iterate through all intermediate keys with their respective values

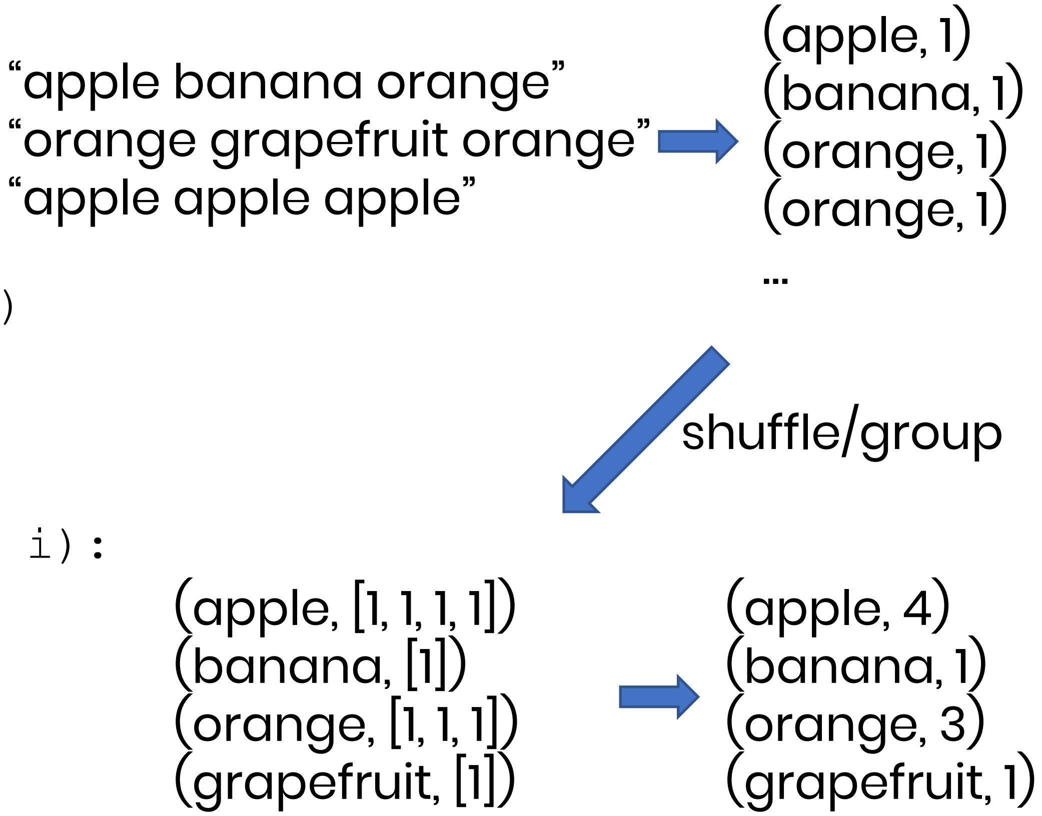
```
Reduce(String w, Seq<Int> i):
  count = 0
  for each int n in i:
    count++
  emit(w, count)
```

MapReduce Counting Example

Count the number of times each word appears in a collection of text documents.

```
Map(Document d):
   for each word w in d:
      emitIntermediate(w, 1)
```

```
Reduce(String w, Seq<Int> i):
   count = 0
   for each int n in i:
      count++
   emit(w, count)
```



“apple banana orange”
“orange grapefruit orange”
“apple apple apple”

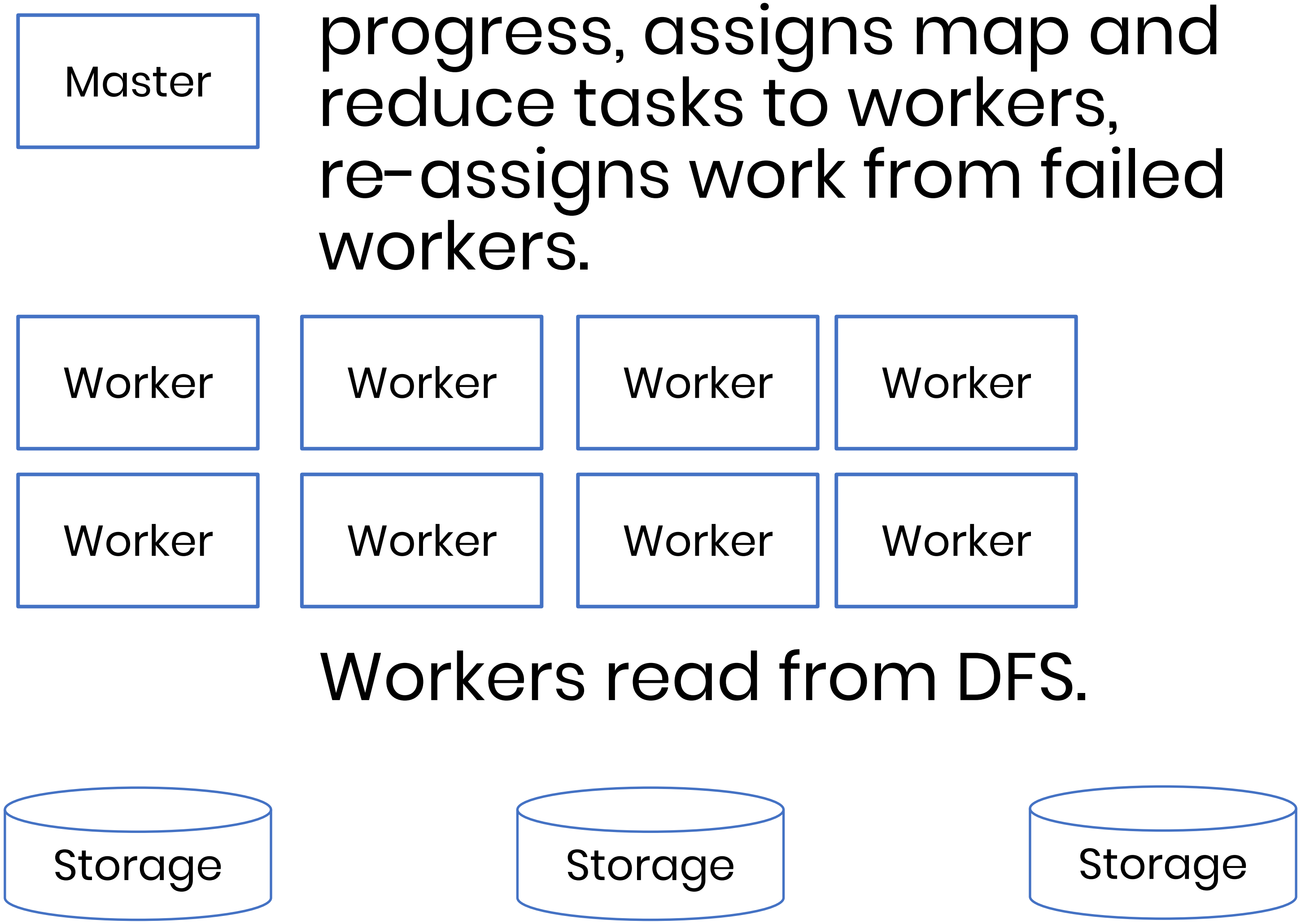
(apple, 1)
(banana, 1)
(orange, 1)
(orange, 1)
…

shuffle/group

(apple, [1, 1, 1, 1])
(banana, [1])
(orange, [1, 1, 1])
(grapefruit, [1])

(apple, 4)
(banana, 1)
(orange, 3)
(grapefruit, 1)

MapReduce architecture



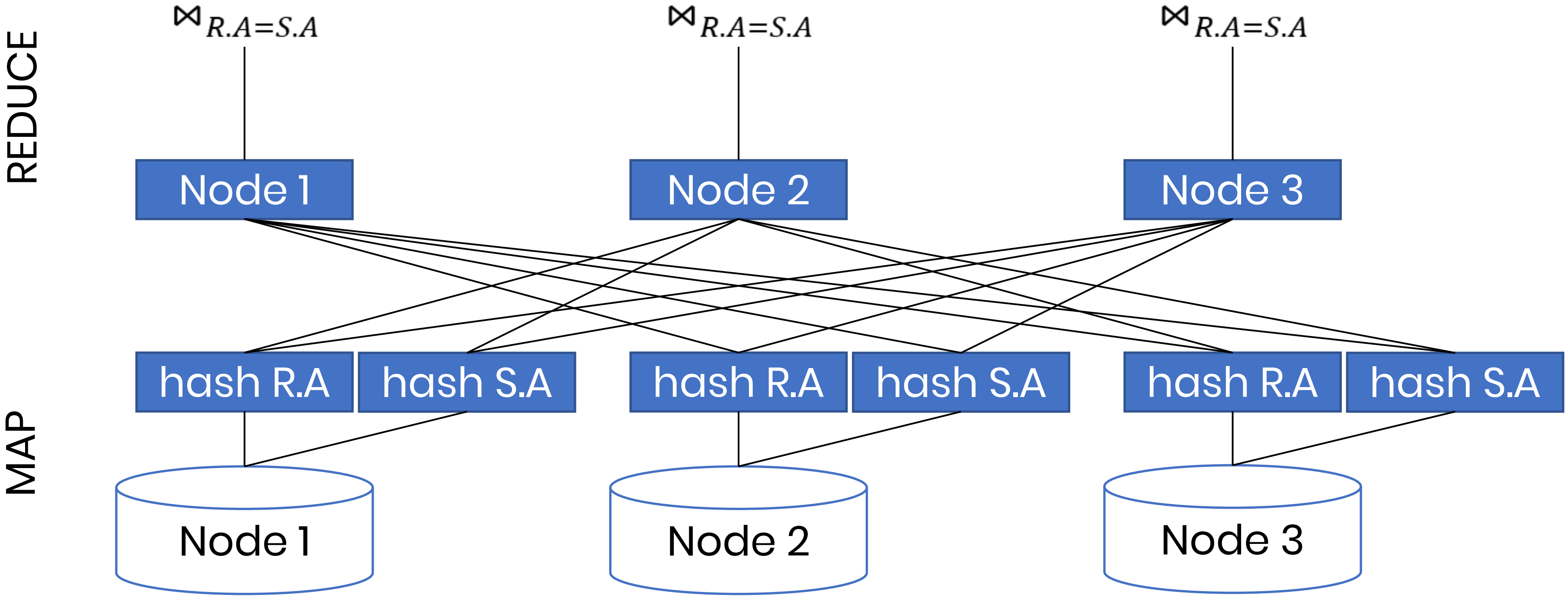
Master node keeps track of progress, assigns map and reduce tasks to workers, re-assigns work from failed workers.

Workers read from DFS.

MapReduce RA Example

Let’s say I wanted to implement Partitioned Hash Equijoin from last lecture…

```
SELECT *
  FROM R, S
 WHERE R.A = S.A
```



MapReduce RA Example

```
Map(Tuple t):
  if t is from R:
    emitIntermediate(t.A, (“R”, t))
  if t is from S:
    emitIntermediate(t.A, (“S”, t))
```

```
Reduce(String s, Seq<(String, Tuple)> ts):
  List<Tuple> rs
  List<Tuple> ss
  for each pair p in ts:
    if p.label = “R”: add p.t to rs
    if p.label = “S”: add p.t to ss
  for each tuple tr in rs, ts in ss:
    emit(tr, rs)
```

Fault Tolerance

- Distributed Systems 101: Something is gonna fail
  - A multi-petabyte job might run on ~10000 servers
  - Say a server fails once per year (~9000) hours
  - **We expect a server to fail** within an hour
- MapReduce implements fault tolerance via writing results to disk
  - Jobs are sloooooooowwww because of write IOs
  - Can we do something faster?

Apache Spark

Apache Spark

- Open source system from UC Berkeley
- **Fast distributed processing on top of HDFS**
  - Spark is not a DBMS
- Used frequently in:
  - ETL pipelines and data streams
  - Machine learning
  - Building other distributed systems (like databases)

Spark Fault Tolerance

- MapReduce is slow if there are intermediate results that must be written to disk
- Spark's solution: Don't use disk

Disk read/write cost = 0?!?

Spark Fault Tolerance

- Spark’s solution: **Compute everything in memory but keep track of how it is computed**
- RAM is expensive but getting cheaper
- **Resilient Distributed Dataset (RDD)**
    - A distributed, immutable **relation** and a **lineage**
    - A lineage is essentially an RA plan
- Recovery is easy, fast, and efficient:
    - Failed worker will lose its data
    - Master node detects failure
    - Master node has new worker recompute exactly what was lost by looking at the lineage

Spark Fault Tolerance

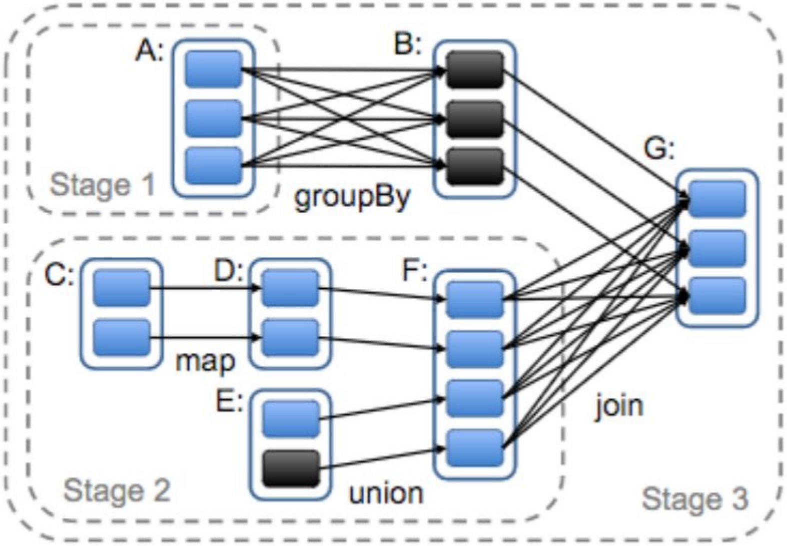


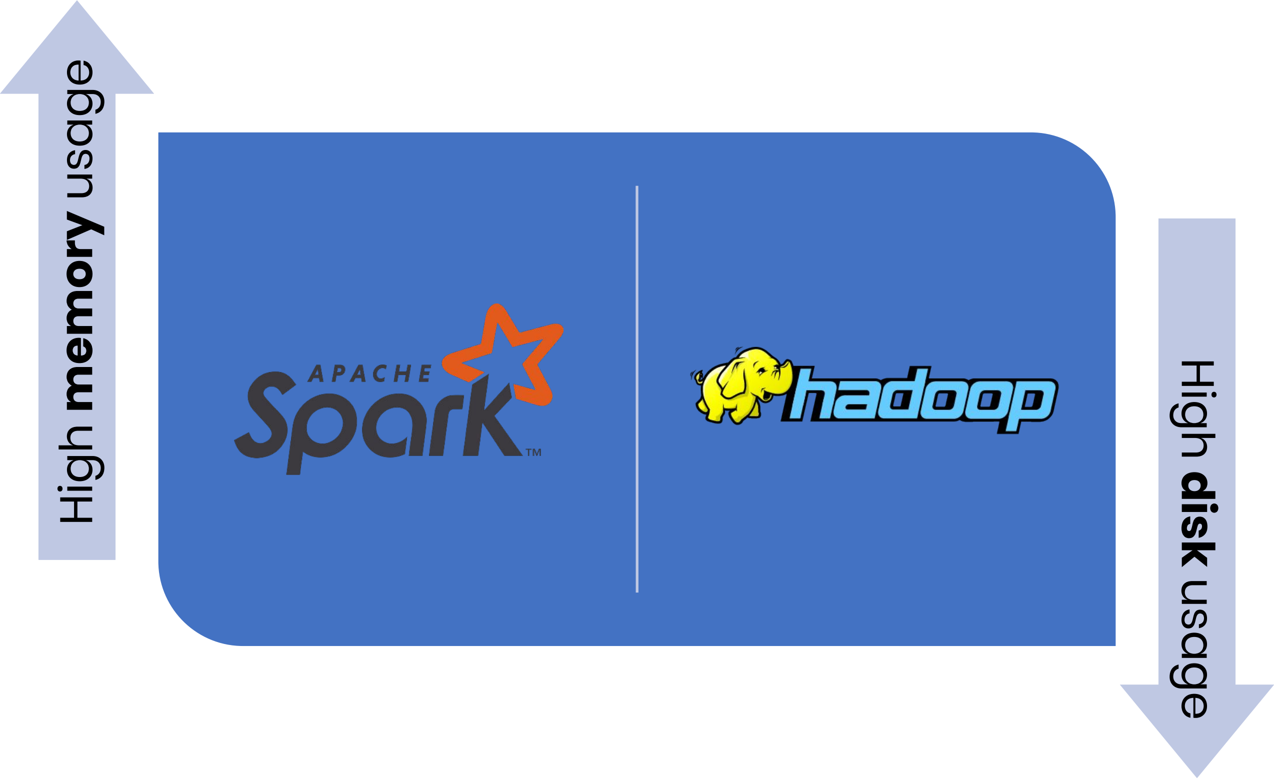
Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

Image from [Zahariah 12]

Each blue-outlined box is an RDD, which contains partitions of data. The partitions store their lineage, i.e. edges back to the original file from which data came.

Upon node failure, any RDD in the query plan can be recomputed based on the lineage from previous nodes.

Spark vs Hadoop MapReduce



High **memory** usage

High **disk** usage

Using Spark

- Latest version of Spark: 2.4.5 (Feb 5, 2020)
- Spark APIs for Java, Scala, and Python
- Spark has a SQL interface called SparkSQL

Using Spark

- We will stick to tuple and key-value processing in this class.
- Java objects we'll look at:
  - SparkSession
  - Row
  - Dataset<Row>
  - JavaRDD<T>

Setting Up Spark

- Local execution configurations:
  1. Install Spark
  2. Profit.

Setting Up Spark

- Cluster execution configurations:
  - More involved but still managed by Spark
    1. Install Spark on all nodes
    2. Have all nodes know about each other (via hosts file)
    3. Make sure the master node can SSH into slave nodes
    4. Use provided Spark scripts to spin up the cluster
  - All automatic on cloud services!
    - Amazon Elastic MapReduce (EMR)
    - Google Dataproc
    - Azure Databricks

Cloud Computing

- Up to this point we have only used software-as-a-service (SaaS)
  - Azure Database (SQL Server)
  - Google Dataprep (Trifacta)
- HW7 will use AWS EMR which is classified as platform-as-a-service (PaaS)

Cloud Computing

- Typical categories of products you might see on cloud platforms:
  - **SaaS** – Managed end-use software
    - Databases, Google Drive, Slack, Pre-trained AI
  - **PaaS** – Managed application development platform
    - Autoscaled application hosting, managed clusters
  - **IaaS/HaaS** – Managed hardware
    - Infrastructure-as-a-Service/Hardware-as-a-Service
    - Servers, FPGAs, quantum computers

Creating a Spark Java Application

- Step 1 is to go from building a Java program to building a Spark program
- The **SparkSession** object is an interface that lets us issue commands in Spark.

```
SparkSession sparkCluster = SparkSession.builder()
                                        .appName("MyApp")
                                        .getOrCreate();
SparkSession sparkLocal = SparkSession.builder()
                                      .appName("MyApp")
                                      .config("spark.master",
"local")
                                      .getOrCreate();
```

Creating a Spark Java Application

- Step 1 is to go from building a Java program to building a Spark program
- The **SparkSession** object is an interface that lets us issue commands in Spa

Get the SparkSession builder (constructor)

```
SparkSession sparkCluster = SparkSession.builder()
                                        .appName("MyApp")
                                        .getOrCreate();
SparkSession sparkLocal = SparkSession.builder()
                                      .appName("MyApp")
                                      .config("spark.master",
"local")
                                      .getOrCreate();
```

Creating a Spark Java Application

- Step 1 is to go from building a Java program to building a Spark program
- The **SparkSession** object is an interface that lets us issue commands in Spark.

Give the SparkSession a name

```
SparkSession sparkCluster = SparkSession.builder()
                                        .appName("MyApp")
                                        .getOrCreate();
SparkSession sparkLocal = SparkSession.builder()
                                      .appName("MyApp")
                                      .config("spark.master",
"local")
                                      .getOrCreate();
```

Creating a Spark Java Application

- Step 1 is to go from building a Java program to building a Spark program
- The **SparkSession** object is an interface that lets us issue commands in Spark.

Create the new SparkSession

```
SparkSession sparkCluster = SparkSession.builder()
                                        .appName("MyApp")
                                        .getOrCreate();
SparkSession sparkLocal = SparkSession.builder()
                                      .appName("MyApp")
                                      .config("spark.master",
"local")
                                      .getOrCreate();
```

Creating a Spark Java Application

- Step 1 is to go from building a Java program to building a Spark program
- The **SparkSession** object is an interface that lets us issue commands in Spark.

```
SparkSession sparkCluster = SparkSession.builder()
                                        .appName("MyApp")
                                        .getOrCreate();
SparkSession sparkLocal = SparkSession.builder()
                                      .appName("MyApp")
                                      .config("spark.master",
"l

                                      .getOrCreate();
```

If we don’t need cluster management, we just say so

Reading Parquet Data

- Apache Parquet is a data storage format for Hadoop-based systems
- Interpreting Parquet data is prebuilt into Spark

```
Dataset<Row> d =
sparkCluster.read().parquet(“some/file.path”);
```

SparkSQL

- We can label Datasets as tables and then query using SQL directly through Spark

```
Dataset<Row> d = sparkCluster.read().parquet(“some/file.path”);
d.createOrReplaceTempView(“myTable”);
Dataset<Row> r = sparkCluster.sql(“SELECT * FROM myTable WHERE attr =
‘’”);
```

Won’t be checked at compile time

Will be optimized at runtime

Spark Functional APIs

- JavaRDD<Row> = Collection of data
- Functional API is essentially RA and some extra operators:
  - **`agg`** (…)
  - **`groupBy`** (…)
  - **`join`** (…)
  - **`orderBy`** (…)
  - **`select`** (…)
  - **`map`** (…)
  - **`reduce`** (…)
  - …

Transformations and Actions

- Spark functional API:
  - Transformations (map, reduceByKey, join, filter, …)
    - Lazy evaluation
    - JavaRDD JavaRDD
  - Actions (count, reduce, save, foreach, …)
    - Eager evaluation
    - JavaRDD Non-Spark format

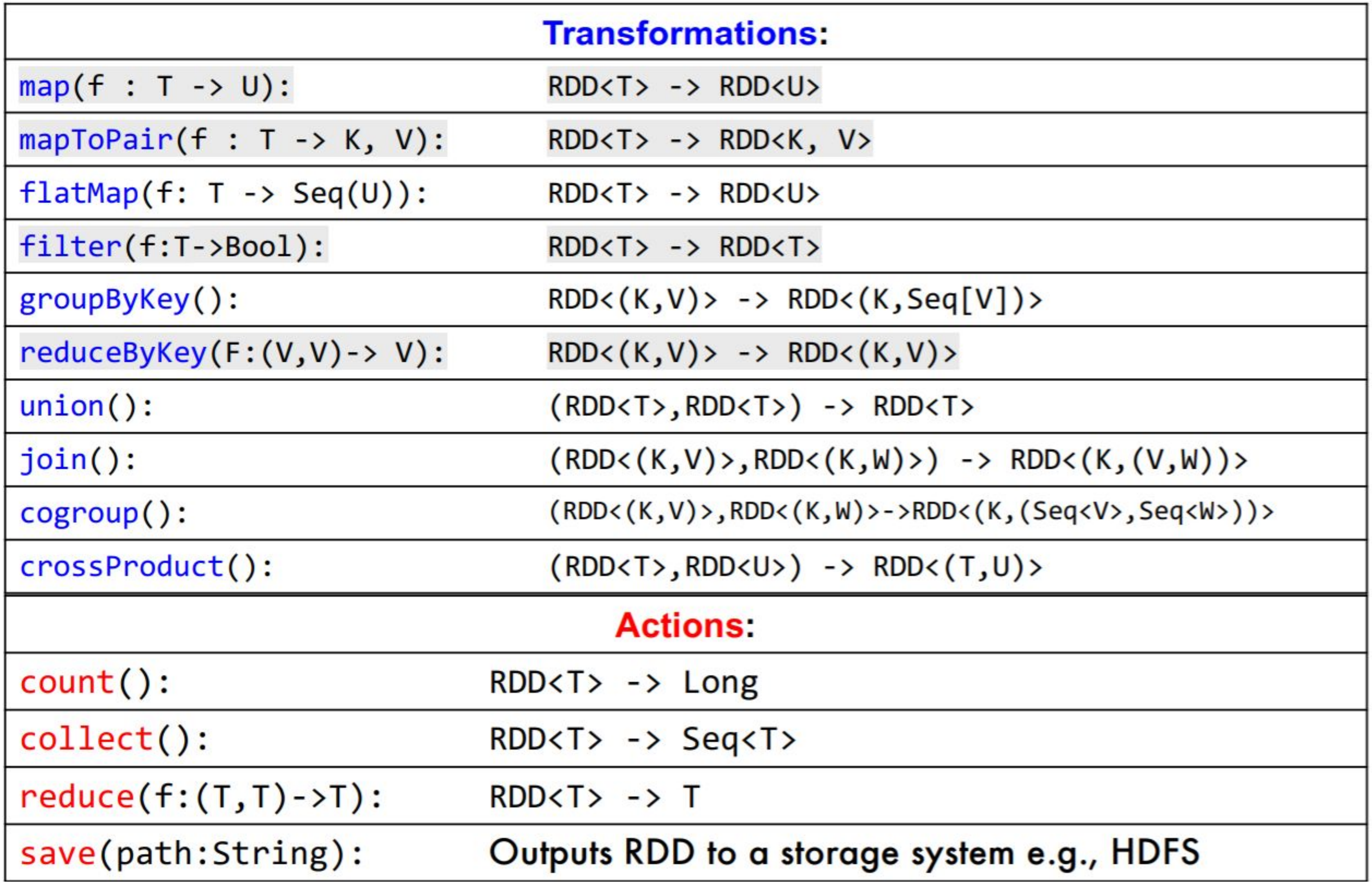
Lazy versus Eager Evaluation

- **Eager operators** are **executed immediately**
- **Lazy operators wait** for an eager operator to trigger execution
  - Chained lazy operators will make an **operator tree**
  - Operator tree is the "lineage" we talked about earlier
  - Spark will optimize the operator tree before execution

```
Dataset<Row> d = sparkCluster.read().parquet("some file path");
d.join(…)        // lazy
  .crossJoin(…)  // lazy
  .filter(…)     // lazy
  .groupBy(…)    // lazy
  .filter(…)     // lazy
  .foreach(…);   // eager!
```

Optimize like RA

JavaRDD



Transformations:	
map(f : T -> U):	RDD<T> -> RDD<U>
mapToPair(f : T -> K, V):	RDD<T> -> RDD<K, V>
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)>->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

JavaRDD and Java 8 Lambdas

JavaRDDs are built like collections so lambdas are used in transformations rather than strongly-typed Dataset objects

```
Dataset<Row> d = sparkCluster.read().parquet(“some/file.path”);
JavaRDD<Row> r = d.javaRDD();
JavaRDD<Row> f = r.filter(row -> row.getString(4).equals(“hello there”));
```

Java 8 Lambdas

Recall: Java 8 introduced anonymous functions (lambda expressions).

```
errors = lines.filter(l -> l.startsWith(“ERROR”));
```

is the same as:

```
class FilterFn implements Function<Row, Boolean> {
    Boolean call(Row l) {
        return l.startsWith(“ERROR”);
    }
 }
 errors = lines.filter(new FilterFn());
```

Full Spark Example - RDD

```
// Get all lines of a HDFS log file that start with “ERROR”
// and contain “sqlite”

// Create an interface to Spark
SparkSession sparkCluster = SparkSession.builder()
                                        .appName("MyApp")
                                        … // any other configs
                                        .getOrCreate();

// Acquire data (textFile will delimit by newline)
JavaRDD<String> textFile = sparkCluster.textFile("hdfs://...");

// Form and execute query
List<String> f = r.filter(line -> line.startsWith(“ERROR”))
                  .filter(line -> line.contains(“sqlite”))
                  .collect();
```

Spark 2.0: Dataframes

- Dataframe are implemented on top of RDDs
  - Dataframe = Table
  - JavaRDD = Collection
- Elements are Rows
- Pretty much the same functional API as RDDs
- Adds a schema with named columns
  - compile-time query checking!
- Queries are optimized

Spark 2.0: Datasets

- Dataset<T> is similar to Dataframe
    - Dataframe = Dataset<Row>
- Elements are types you define
    - compile-time type checking!
- Also optimizes queries

Transformations and Actions

- Spark functional API:
  - Transformations (map, reduceByKey, join, filter, …)
    - Lazy evaluation
    - Dataset □ Dataset
  - Actions (count, reduce, save, foreach, …)
    - Eager evaluation
    - Dataset□ Non-Spark format

Full Spark Example - DataFrame

```
// Creates a DataFrame having a single column named "line"
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaRDD<Row> rowRDD = textFile.map(RowFactory::create);
List<StructField> fields = Arrays.asList(
  DataTypes.createStructField("severity",
DataTypes.StringType, true)
  DataTypes.createStructField("line",
DataTypes.StringType, true));
StructType schema = DataTypes.createStructType(fields);
DataFrame df = sqlContext.createDataFrame(rowRDD, schema);

DataFrame errors =
df.filter(col("severity").equalTo("ERROR"));
// Fetches the MySQL errors as an array of strings
errors.filter(col("line").like("%MySQL%")).collect();
```