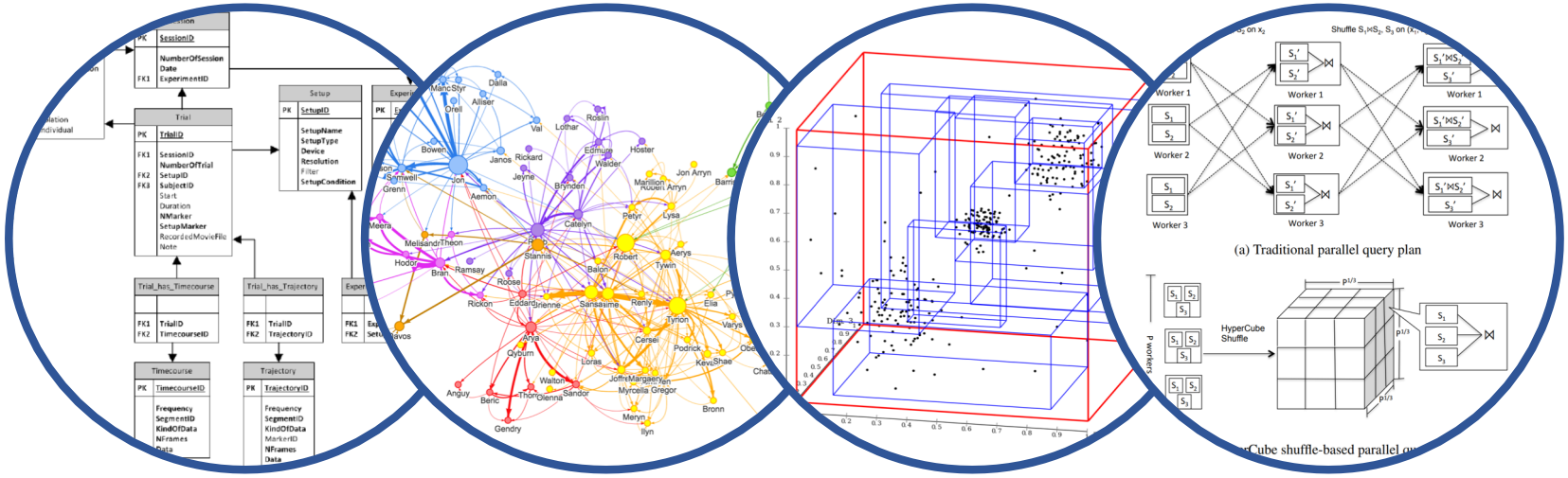



# Introduction to Data Management

## MapReduce and Spark

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

# Announcements

- Homework 5 due date extended to Monday 11pm

- Some notes about HW 5:
- `conn.setAutoCommit(true);` means every SQL statement you send to Azure is a transaction (i.e. is wrapped in an implicit BEGIN and COMMIT/ROLLBACK)
- `conn.setAutoCommit(false);` means you have to execute BEGIN and COMMIT/ROLLBACK, or else your transactions will never end

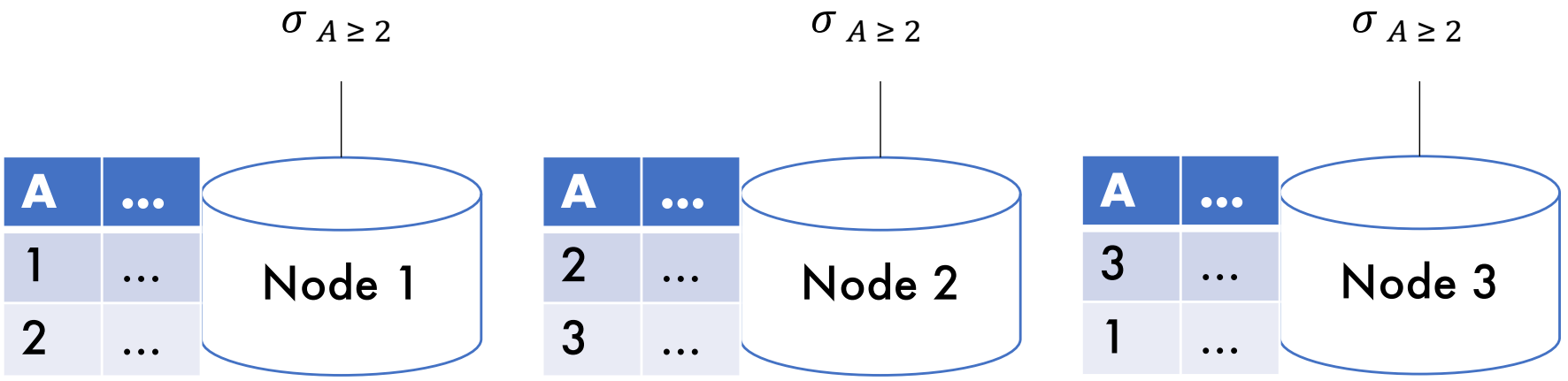
- Make sure to set this appropriately, methods that use only a single SQL query can usually just be run with autocommit true. To be safe, you could always have it as "false" and explicitly write each transaction begin and end.

# Recap of Intra-Operator Parallelism

"Data parallel":
Each node can do operation
In-place

We are running this selection on every node:

```
SELECT *
  FROM R
 WHERE R.A >= 2
```

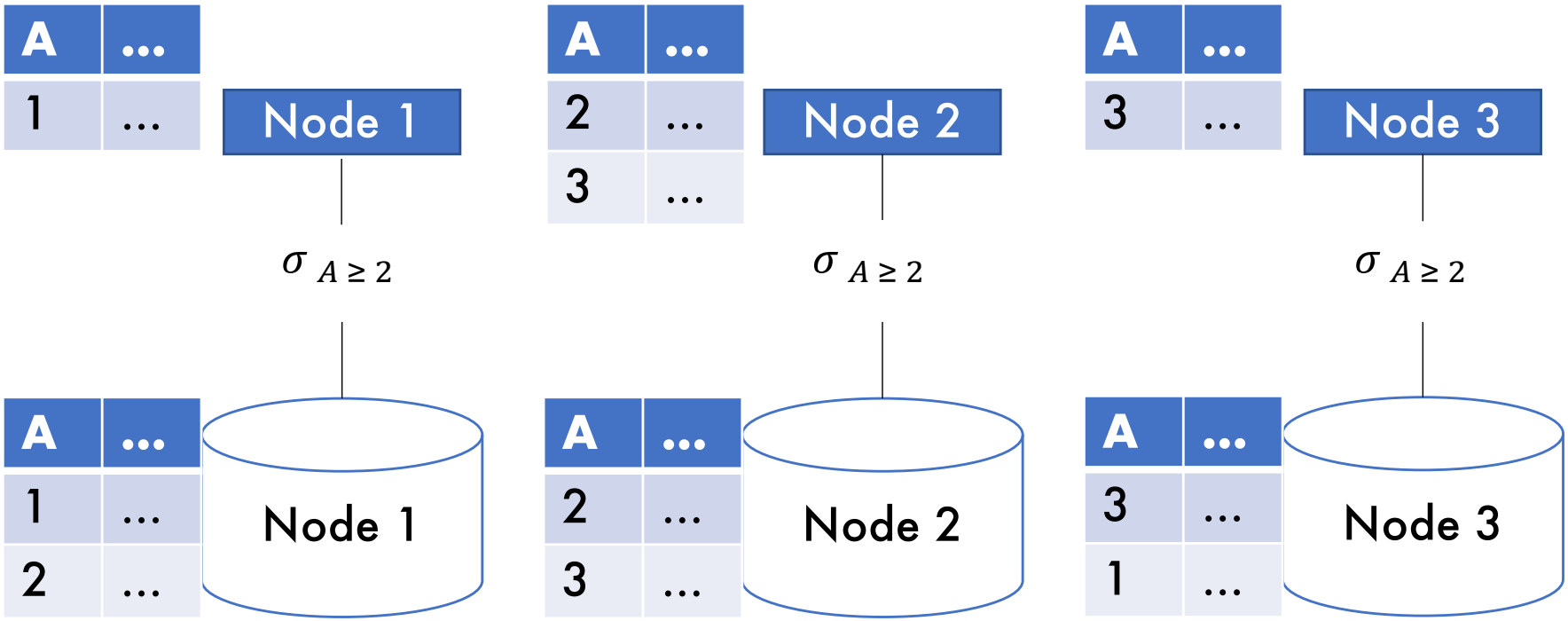


# Recap of Intra-Operator Parallelism

"Data parallel":
Each node can do operation
In-place

We are running this selection on every node:

```
SELECT *
  FROM R
 WHERE R.A >= 2
```

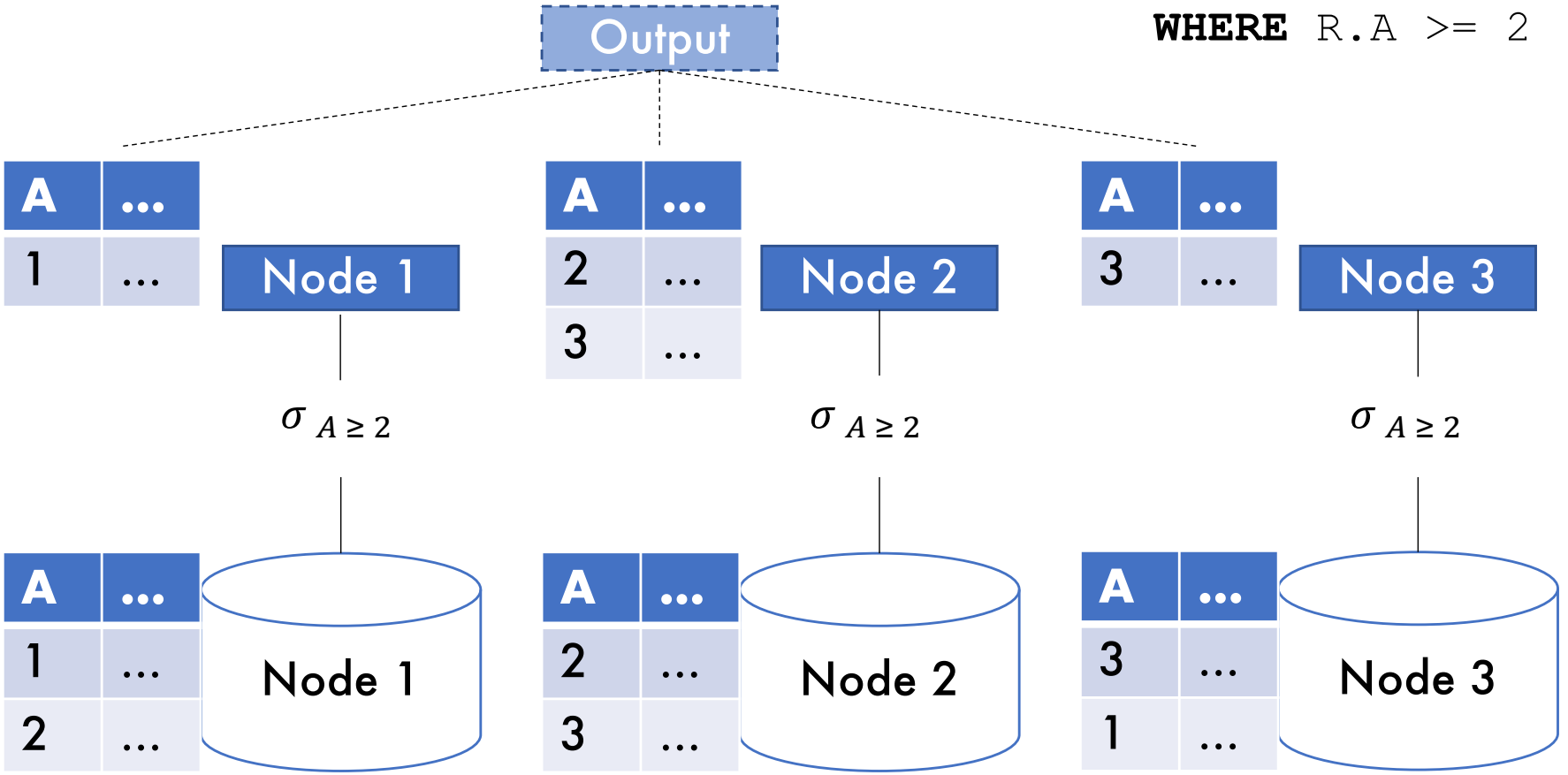


# Recap of Intra-Operator Parallelism

"Data parallel":
Each node can do operation
In-place

We are running this selection on every node:

```
SELECT *
  FROM R
 WHERE R.A >= 2
```

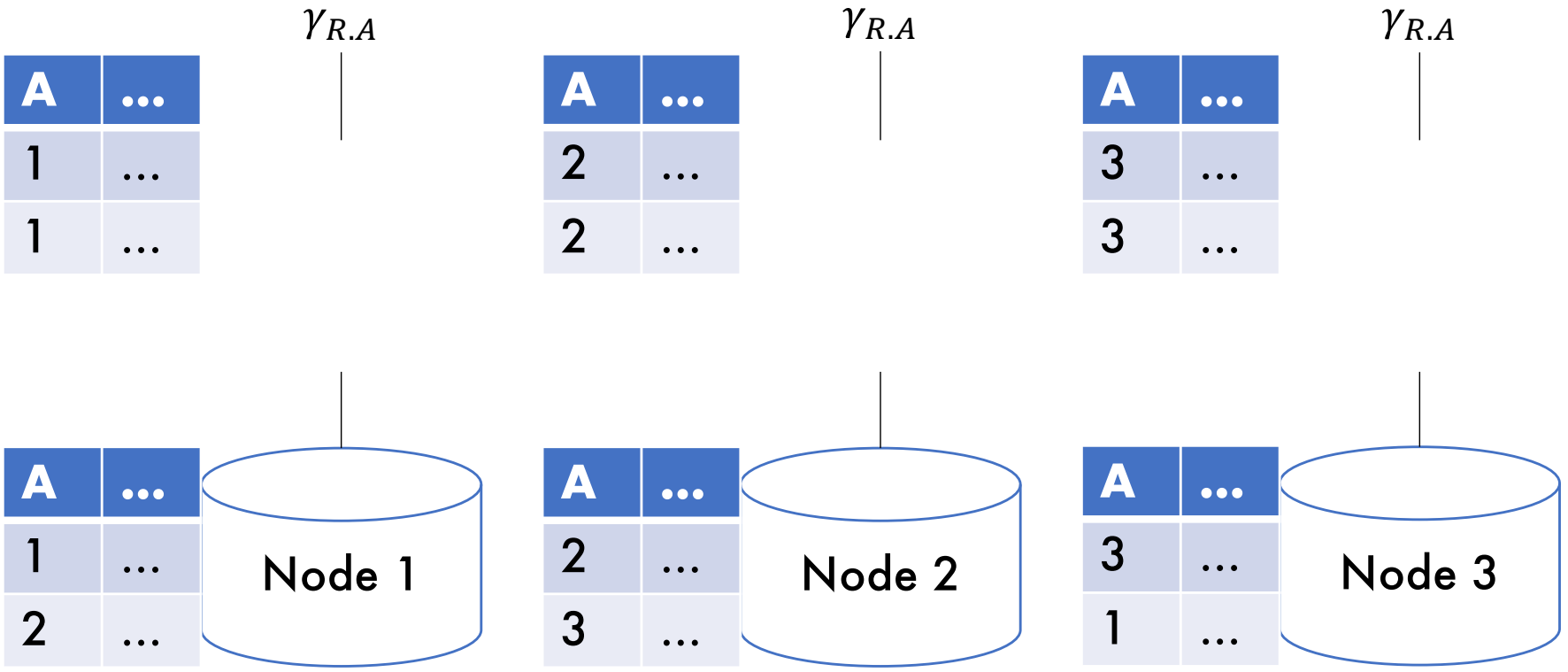


# Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned

```
SELECT *
  FROM R
 GROUP BY R.A
```

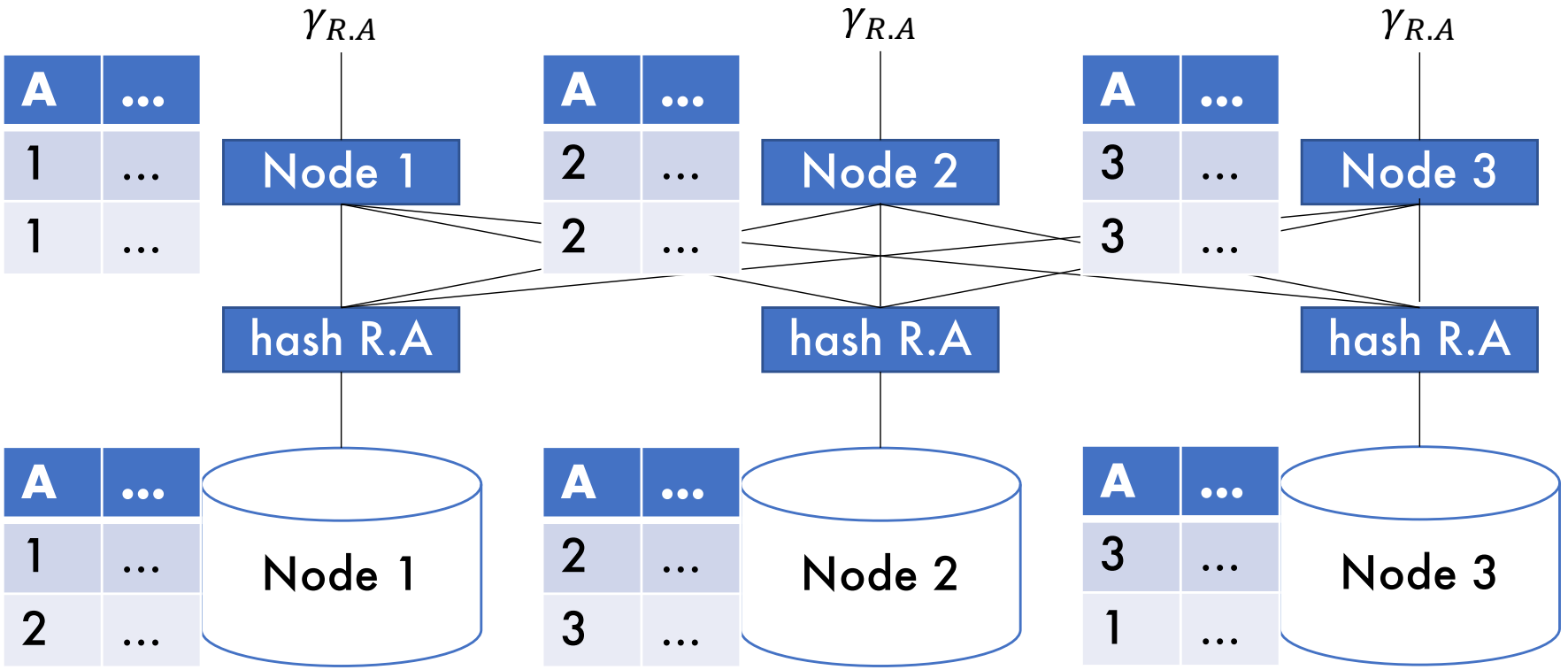


# Partitioned Aggregation

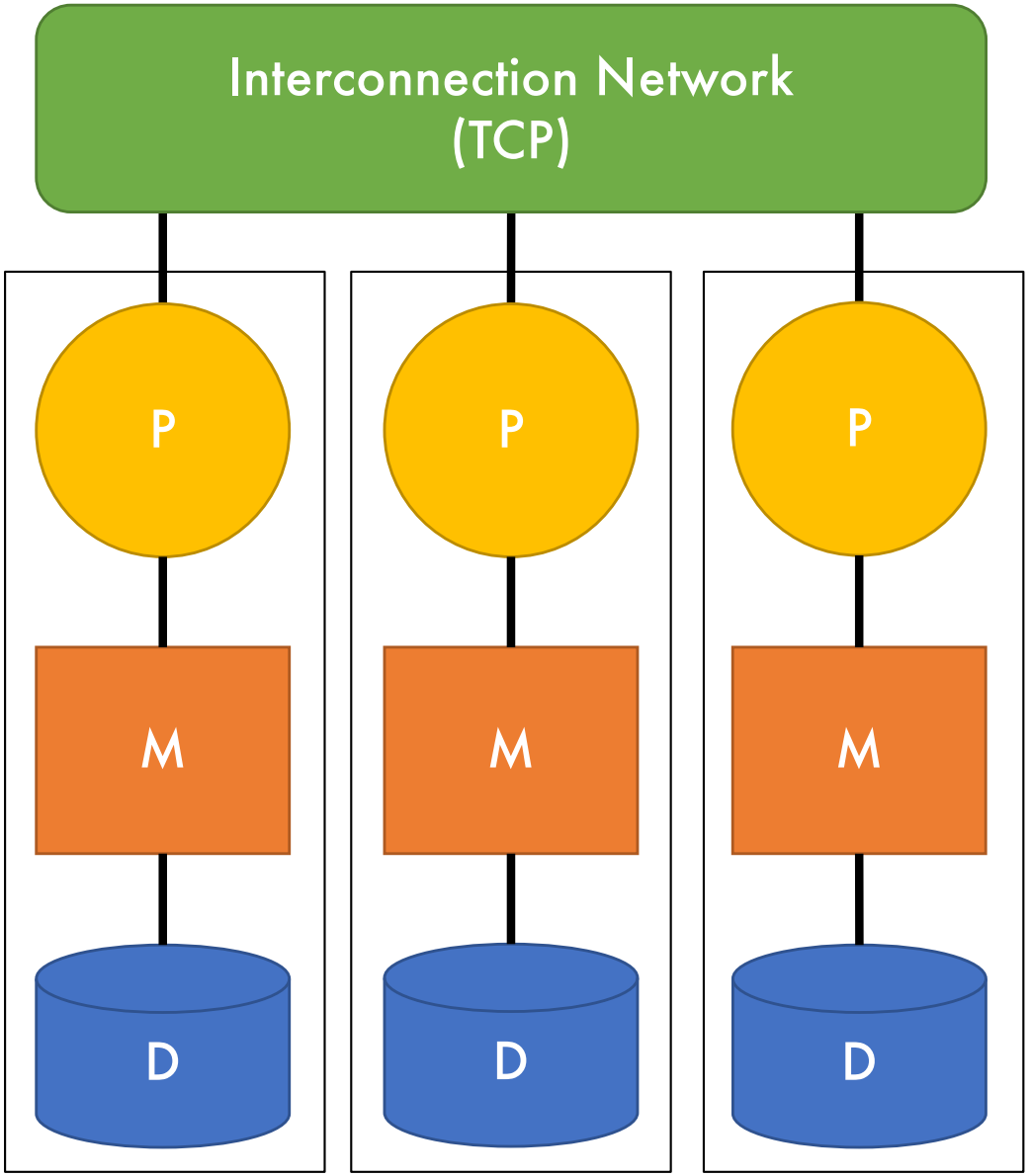
1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned

```
SELECT *
  FROM R
 GROUP BY R.A
```



# 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

# The Apache Software Foundation

- Decentralized organization of developers
- Projects are free and open source software
  - Apache Spark
  - Apache Maven (build tool)
  - Apache Kafka (streams)
  - Apache Parquet (distributed files)
  - Apache AsterixDB (NoSQL database)
  - Apache SystemML (declarative machine learning)
  - …

# Apache Hadoop

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

# 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
  - …


# Hadoop MapReduce

- MapReduce (2004) 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
    - This is usually "data parallel" operations like a filter
  - **Shuffle** key-value pairs into groups based on keys
  - **Reduce**:
    - Perform analysis on groups collected from other nodes
    - Write results to disks
- Complex jobs need multiple map-then-reduce phases

# Hadoop MapReduce

- Format → Implicitly Group → Analyze

Map

Reduce

# MapReduce Counting Example

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

Implicitly iterate through all data elements

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

Implicitly iterate through all key groups with respective sequence of 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)
```

"apple banana orange"
"orange grapefruit orange"
"apple apple apple"

→

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

(MapReduce shuffle/group)

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

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

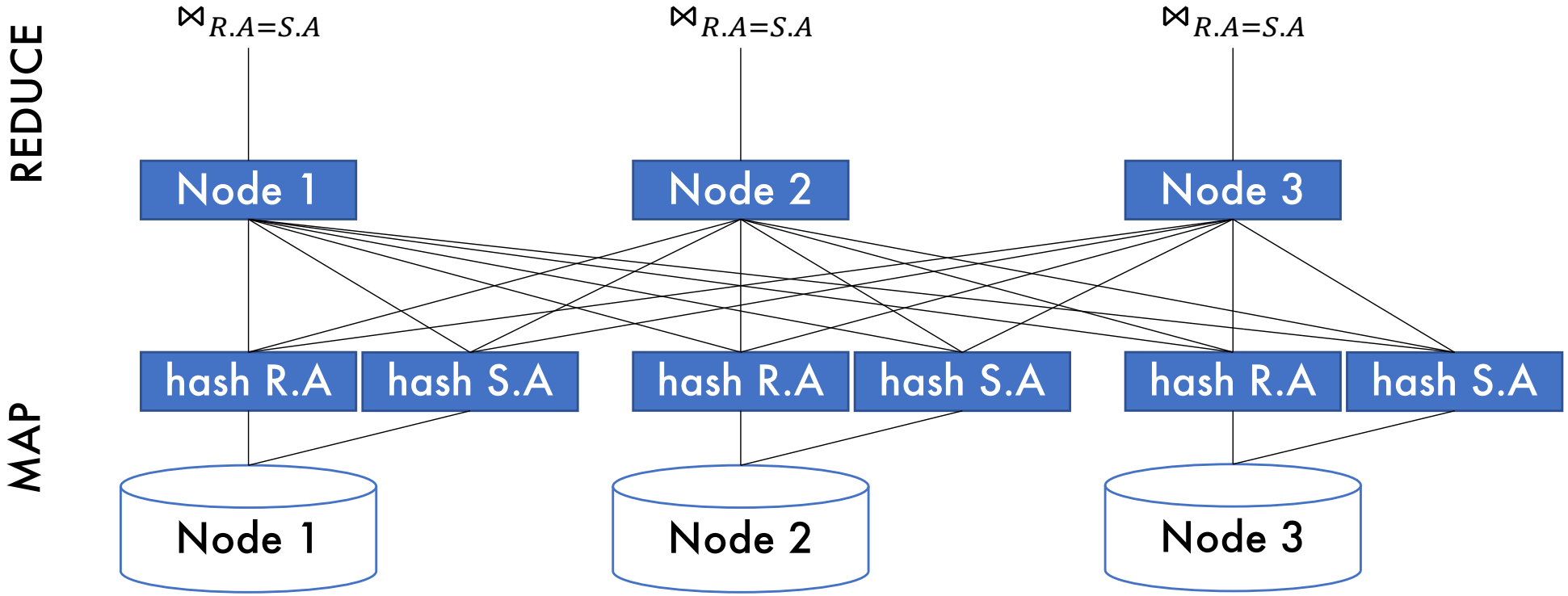
→

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

# MapReduce RA Example

Let's say I wanted to implement Partitioned Hash Join 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 going to 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 slooooooowwww because of write IOs
  - Can we do something faster?

# 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 = zero ?!

# 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 everything
  - Master node detects failure
  - Master node issues commands by looking at the lineage to recompute exactly what partition was lost

# Spark vs Hadoop MapReduce

# Using Spark

- Latest version of Spark: 2.4.2 (April 23, 2019)
- **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

- 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

# 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)
- HW 6 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-user 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 Spark.

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 an awesome 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.


```
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", "local")
                                      .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");
```

# Dataset and Row

- Dataset<Row> = DataFrame = Table
- Dataset functional API is essentially RA and some extra operators:
  - `agg(...)`
  - `groupBy(...)`
  - `join(...)`
  - `orderBy(...)`
  - `select(...)`
  - `map(...)`
  - `reduce(...)`
  - `...`

# SparkSQL – easiest way to use Spark

- 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 = ''");
```

# Transformations and Actions

- SparkSQL queries must be optimized at runtime
- Java native Spark queries can be **checked and optimized** at compile time (better)
- Spark functional API:
  - **Transformations** (map, reduce, join, filter, …)
    - Lazy evaluation
    - Dataset → Dataset
  - **Actions** (count, reduce, save, foreach, …)
    - Eager evaluation
    - Dataset → 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

# RDDs as nodes in query plan

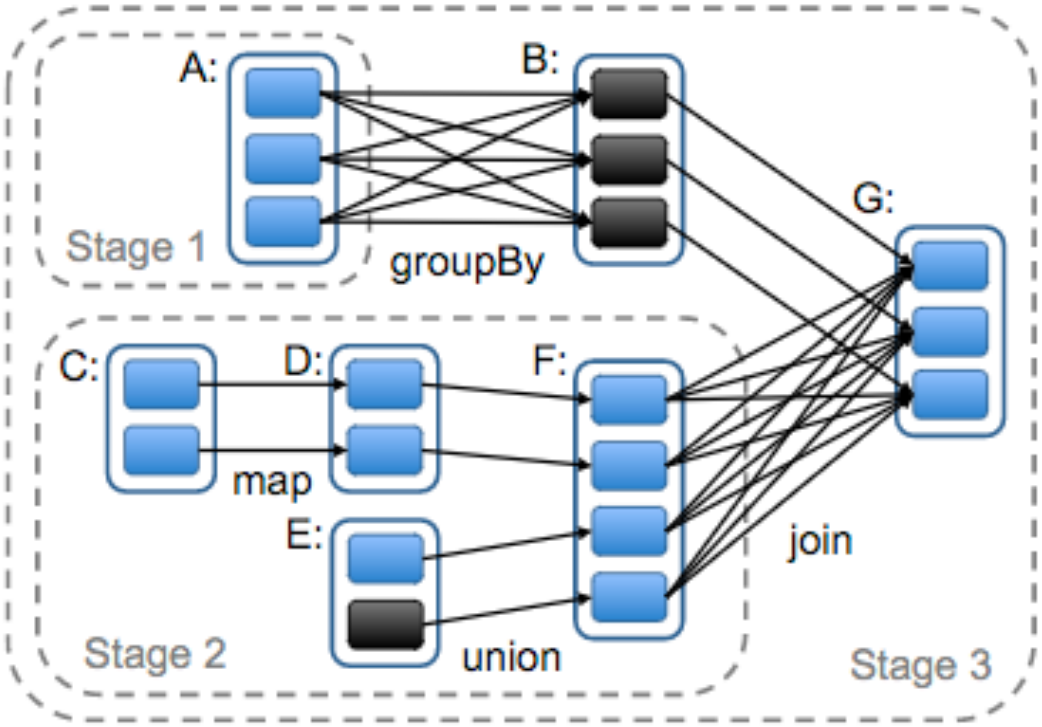



Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case,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

[From Zaharia12]

In this figure from the book, each small box in an RDD partition of the later file.

RDDs store information about their lineage, i.e. edges back to the original file.

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

# Using JavaRDD

- JavaRDD is more general purpose than Dataset
  - Dataset = Table
  - JavaRDD = Collection
- Very similar functional API to Dataset:
  - Transformations (map, reduce, join, filter, …)
    - Lazy evaluation
    - JavaRDD → JavaRDD
  - Actions (count, reduce, save, foreach, …)
    - Eager evaluation
    - JavaRDD → Non-Spark format

# Many RDD operators

| 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 |

# Example

Given a large log file hdfs://logfile.log retrieve all lines that:

- Start with "ERROR"
- Contain the string "sqlite"

```
s = SparkSession.builder()...getOrCreate();

lines = s.read().parquet("hdfs://logfile.log");

errors = lines.filter(l -> l.startsWith("ERROR"));

sqlerrors = errors.filter(l -> l.contains("sqlite"));

sqlerrors.collect();
```

# Example

Recall: anonymous functions
(lambda expressions) starting in Java 8

```
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());
```

# JavaRDD and Java 8 Lambdas

JavaRDDs are built like collections so lambdas are used in transformations rather than special built objects in Datasets

```
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"));
```

# Full Spark Example

```
// 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)
Dataset<String> d = sparkCluster.read().textFile("hdfs://logfile.log");

// Convert Dataset into JavaRDD
JavaRDD<String> r = d.javaRDD();

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