Announcements

• HW1 is graded and posted (thanks Kexuan!)

• Project proposals due this Friday!
  – Working in team? Only one of you submits

• HW2 (Spark) due on Monday
Distributed or Parallel Query Processing

• Clusters:
  – More servers $\rightarrow$ more in main memory
  – More servers $\rightarrow$ more computing power
  – Clusters are now cheaply available in the cloud
  – Distributed query processing

• Multicores:
  – The end of Moore’s law
  – Parallel query processing
Outline

• Spark

• MapReduce and critique

• Fault Tolerance

• Hive (short)

Next lecture: Parallel databases
Spark
Motivation

• Limitations of relational database systems:
  – Single server (at least traditionally)
  – SQL is a limited language (eg no iteration)

• Spark:
  – Distributed system
  – Functional language (Java/Scala) good for ML

• Implementation:
  – Extension of MapReduce
  – Distributed physical operators
Review: Single Client

E.g. data analytics
Review: Client-Server

E.g. accounting, banking, …

Connection: ODBC, JDBC
Review: Three-tier connection (ODBC, JDBC)

E.g. Web commerce
Review: Distributed Database

E.g. large-scale analytics or…

Sharded database

Spark, Snowflake

ODBC, JDBC

http

App server

…social networks
Programming in Spark

• A Spark program consists of:
  – Transformations (map, reduce, 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
Collections in Spark

RDD\langle T \rangle = an RDD collection of type T
  • Distributed on many servers, not nested
  • Operations are done in parallel
  • Recoverable via lineage; more later

Seq\langle T \rangle = a sequence
  • Local to one server, may be nested
  • Operations are done sequentially
// First line defines RDD backed by an HDFS file
lines = spark.textFile("hdfs://…")

// Now we create a new RDD from the first one
errors = lines.filter(x -> x.startsWith("Error"))

// Persist the RDD in memory for reuse later
errors.persist()
errors.collect()
errors.filter(x -> x.contains("MySQL")).count()
Example from paper, new syntax

Search logs stored in HDFS

// First line defines RDD backed by an HDFS file
lines = spark.textFile("hdfs://…")

// Now we create a new RDD from the first one
errors = lines.filter(x -> x.startsWith("Error"))

// Persist the RDD in memory for reuse later
errors.persist()
errors.collect()
errors.filter(x -> x.contains("MySQL")) . count()
// First line defines RDD backed by an HDFS file
lines = spark.textFile("hdfs://…")

// Now we create a new RDD from the first one
errors = lines.filter(x -> x.startsWith("Error"))

// Persist the RDD in memory for reuse later
errors.persist()
errors.collect()
errors.filter(x -> x.contains("MySQL")).count()}
Anonymous Functions

A.k.a. lambda expressions, starting in Java 8

```java
errors = lines.filter(x -> x.startsWith("Error"))
```
sqlerrors = spark.textFile("hdfs://...")
  .filter(x -> x.startsWith("ERROR"))
  .filter(x -> x.contains("sqlite"))
  .collect();
The RDDs:

```
sqlerrors = spark.textFile("hdfs://...")
  .filter(x -> x.startsWith("ERROR"))
  .filter(x -> x.contains("sqlite"))
  .collect();
```
The RDDs:

<table>
<thead>
<tr>
<th>Error...</th>
<th>Warning...</th>
<th>Warning...</th>
<th>Error...</th>
<th>Abort...</th>
<th>Abort...</th>
<th>Error...</th>
<th>Error...</th>
<th>Warning...</th>
<th>Error...</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter(&quot;ERROR&quot;)</td>
<td>filter(&quot;ERROR&quot;)</td>
<td>filter(&quot;ERROR&quot;)</td>
<td>filter(&quot;ERROR&quot;)</td>
<td>filter(&quot;ERROR&quot;)</td>
<td>filter(&quot;ERROR&quot;)</td>
<td>filter(&quot;ERROR&quot;)</td>
<td>filter(&quot;ERROR&quot;)</td>
<td>filter(&quot;ERROR&quot;)</td>
<td>filter(&quot;ERROR&quot;)</td>
</tr>
</tbody>
</table>

sqlerrors = spark.textFile("hdfs://...")
  .filter(x -> x.startsWith("ERROR"))
  .filter(x -> x.contains("sqlite"))
  .collect();
Example

The RDDs:

```
sqlerrors = spark.textFile("hdfs://...")
    .filter(x -> x.startsWith("ERROR"))
    .filter(x -> x.contains("sqlite"))
    .collect();
```
The RDDs:

```
sqlerrors = spark.textFile("hdfs://...")
  .filter(x -> x.startsWith("ERROR"))
  .filter(x -> x.contains("sqlite"))
  .collect();
```
More on Programming Interface

Large set of **pre-defined transformations:**
- Map, filter, flatMap, sample, groupByKey, reduceByKey, union, join, cogroup, crossProduct, ...

Small set of **pre-defined actions:**
- Count, collect, reduce, lookup, and save

Programming interface includes **iterations**
## Transformations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(f : T -&gt; U)</td>
<td>RDD&lt;T&gt; -&gt; RDD&lt;U&gt;</td>
</tr>
<tr>
<td>flatMap(f: T -&gt; Seq(U))</td>
<td>RDD&lt;T&gt; -&gt; RDD&lt;U&gt;</td>
</tr>
<tr>
<td>filter(f:T-&gt;Bool)</td>
<td>RDD&lt;T&gt; -&gt; RDD&lt;T&gt;</td>
</tr>
<tr>
<td>groupByKey()</td>
<td>RDD&lt;(K,V)&gt; -&gt; RDD&lt;(K,Seq[V])&gt;</td>
</tr>
<tr>
<td>reduceByKey(F:(V,V)-&gt; V)</td>
<td>RDD&lt;(K,V)&gt; -&gt; RDD&lt;(K,V)&gt;</td>
</tr>
<tr>
<td>union()</td>
<td>(RDD&lt;T&gt;,RDD&lt;T&gt;) -&gt; RDD&lt;T&gt;</td>
</tr>
<tr>
<td>join()</td>
<td>(RDD&lt;(K,V)&gt;,RDD&lt;(K,W)&gt;) -&gt; RDD&lt;(K,(V,W))&gt;</td>
</tr>
<tr>
<td>cogroup()</td>
<td>(RDD&lt;(K,V)&gt;,RDD&lt;(K,W)&gt;) -&gt; RDD&lt;(K,(Seq&lt;V&gt;,Seq&lt;W&gt;))&gt;</td>
</tr>
<tr>
<td>crossProduct()</td>
<td>(RDD&lt;T&gt;,RDD&lt;U&gt;) -&gt; RDD&lt;(T,U)&gt;</td>
</tr>
</tbody>
</table>

## Actions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>RDD&lt;T&gt; -&gt; Long</td>
</tr>
<tr>
<td>collect()</td>
<td>RDD&lt;T&gt; -&gt; Seq&lt;T&gt;</td>
</tr>
<tr>
<td>reduce(f:(T,T)-&gt;T)</td>
<td>RDD&lt;T&gt; -&gt; T</td>
</tr>
<tr>
<td>save(path:String)</td>
<td>Outputs RDD to a storage system e.g., HDFS</td>
</tr>
</tbody>
</table>
More Complex Example

val points = spark.textFile(...)  
  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
Spark Ecosystem Growth

- Spark SQL
- Spark Streaming
- MLlib (machine learning)
- GraphX (graph)

Image from: http://spark.apache.org/
Spark SQL vs Functional Prog. API

- Spark’s original functional programming API
  - General
  - But limited opportunities for automatic optimization

- Spark SQL simultaneously
  - Makes Spark accessible to more users
  - Improves opportunities for automatic optimizations
Three Java-Spark APIs

• **RDDs:** Syntax: `JavaRDD<T>`
  – `T` = anything, basically untyped
  – Distributed, main memory

• **Data frames:** `Dataset<Row>`
  – `<Row>` = a record, dynamically typed
  – Distributed, main memory or external (e.g. SQL)

• **Datasets:** `Dataset<Person>`
  – `<Person>` = user defined type
  – Distributed, main memory (not external)
DataFrames

• Like RDD: immutable distributed collection

• Organized into *named columns*
  – Just like a relation
  – Elements are untyped objects called *Row’s*

• Similar API as RDDs with additional methods
  – `people = spark.read().textFile(...)`;
    `ageCol = people.col("age")`;
    `ageCol.plus(10);` // creates a new DataFrame
Datasets

• Like DataFrames, but elements must be typed

• E.g.: Dataset<People> rather than Dataset<Row>

• Can detect errors during compilation time

• DataFrames are aliased as Dataset<Row> (as of Spark 2.0)
Datasets API: Sample Methods

• Functional API
  - `agg(Column expr, Column... exprs)`
    Aggregates on the entire Dataset without groups.
  - `groupBy(String col1, String... cols)`
    Groups the Dataset using the specified columns, so that we can run aggregation on them.
  - `join(Dataset<?> right)`
    Join with another DataFrame.
  - `orderBy(Column... sortExprs)`
    Returns a new Dataset sorted by the given expressions.
  - `select(Column... cols)`
    Selects a set of column based expressions.

• “SQL” API
  - `SparkSession.sql("select * from R");`

• Look familiar?
Recap: Programming in Spark

• A Spark/Scala program consists of:
  – Transformations (map, reduce, join…). Lazy
  – Actions (count, reduce, save…). Eager

• RDD<T> = an RDD collection of type T
  – Partitioned, recoverable (through lineage), not nested

• Seq<T> = a sequence
  – Local to a server, may be nested
Outline

• Spark
  • MapReduce and critique
• Fault Tolerance
• Hive (short)

Next lecture: Parallel databases
MapReduce: References

• Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters. OSDI’04

MapReduce

- Google:
  - Started around 2000
  - Paper published 2004
  - Discontinued September 2019

- Free variant: Hadoop

- MapReduce = high-level programming model and implementation for large-scale parallel data processing
Distributed File System (DFS)

• For very large files: TBs, PBs
• Each file partitioned into chunks (64MB)
• Each chunk replicated (≥3 times) – why?

• Implementations:
  – Google’s DFS: GFS, proprietary
  – Hadoop’s DFS: HDFS, open source
MapReduce

• Describe the **input** and **output** to map reduce

• Describe the **Map** function

• Describe the **Reduce** function
MapReduce

- Describe the **input** and **output** to map reduce
  - Input: a bag of \((\text{inputkey}, \text{value})\) pairs
  - Output: a bag of \((\text{outputkey}, \text{value})\) pairs
- Describe the **Map** function
- Describe the **Reduce** function
MapReduce

• Describe the **input** and **output** to map reduce
  – Input: a bag of \((\text{input key}, \text{value})\) pairs
  – Output: a bag of \((\text{output key}, \text{value})\) pairs

• Describe the **Map** function
  – Input: \((\text{input key}, \text{value})\)
  – Output: bag of \((\text{intermediate key}, \text{value})\)

• Describe the **Reduce** function
MapReduce

• Describe the input and output to map reduce
  – Input: a bag of (inputkey, value) pairs
  – Output: a bag of (outputkey, value) pairs

• Describe the Map function
  – Input: (input key, value)
  – Output: bag of (intermediate key, value)

• Describe the Reduce function
  – Input: (intermediate key, bag of values)
  – Output: bag of output (values)
Step 1: the MAP Phase

User provides the MAP-function:

- Input: \((\text{input\ key, value})\)
- Output: bag of \((\text{intermediate\ key, value})\)

System applies the map function in parallel to all \((\text{input\ key, value})\) pairs in input file
Step 2: the REDUCE Phase

User provides the REDUCE function:
• Input: (intermediate key, bag of values)
• Output: bag of output (values)

System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function
Example

• 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**)
Example

• 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)

```java
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
    EmitIntermediate(w, "1");
```
Example

- 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)

```java
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
    EmitIntermediate(w, "1");
```

```java
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));
```
Think "Relational"!

Documents: did1  did2 ...

Relation

Hive – A Petabyte Scale Data Warehouse Using Hadoop

1. Introduction
A petabyte scale data warehouse using Hadoop. In addition, HiveQL has been core to the implementation of popular query languages like SQL and as a result users ended up spending hours (if not days) to write programs for data warehousing solution built on top of Hadoop, while still maintaining the extensibility and features our requirements at Facebook. However, using Hadoop was not easy for end users, like our implementation of HiveQL to analyze this data more productively, we had to improve the SQL and as a result users ended up spending hours (if not days) to write programs for commercial RDBMS. The data that we were generating was so inadequate that some daily data processing needs of users is what gives us an idea that in order to really empower the company to reduce. End users had to write map-reduce jobs like Facebook's Lexicon product that contains tens of thousands of jobs executed on commodity hardware was also used for various applications. In thousands of jobs were run on a system. But the overall query time increased...
Think “Relational”!

Documents:

<table>
<thead>
<tr>
<th>did1</th>
<th>did2</th>
<th>...</th>
</tr>
</thead>
</table>

Relation

<table>
<thead>
<tr>
<th>Did</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>did1</td>
<td>Scalable</td>
</tr>
<tr>
<td>did1</td>
<td>analysis</td>
</tr>
<tr>
<td>did1</td>
<td>on</td>
</tr>
<tr>
<td>did1</td>
<td>large</td>
</tr>
<tr>
<td>did1</td>
<td>...</td>
</tr>
<tr>
<td>did2</td>
<td>system</td>
</tr>
<tr>
<td>did2</td>
<td>with</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Think “Relational”!

```
select word, count(*)
from Data
group by word
```

<table>
<thead>
<tr>
<th>Did</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>did1</td>
<td>Scalable</td>
</tr>
<tr>
<td>did1</td>
<td>analysis</td>
</tr>
<tr>
<td>did1</td>
<td>on</td>
</tr>
<tr>
<td>did1</td>
<td>large</td>
</tr>
<tr>
<td>did1</td>
<td>...</td>
</tr>
<tr>
<td>did2</td>
<td>system</td>
</tr>
<tr>
<td>did2</td>
<td>with</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Think “Relational”!

```
select word, count(*)
from Data
group by word
```

map = group by
reduce = count(...) (or sum(...) or...)

<table>
<thead>
<tr>
<th>Did</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>did1</td>
<td>Scalable</td>
</tr>
<tr>
<td>did1</td>
<td>analysis</td>
</tr>
<tr>
<td>did1</td>
<td>on</td>
</tr>
<tr>
<td>did1</td>
<td>large</td>
</tr>
<tr>
<td>did1</td>
<td>...</td>
</tr>
<tr>
<td>did2</td>
<td>system</td>
</tr>
<tr>
<td>did2</td>
<td>with</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Think “Relational”!

```
select word, count(*)
from Data
group by word
```

Relation

<table>
<thead>
<tr>
<th>Did</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>did1</td>
<td>Scalable</td>
</tr>
<tr>
<td>did1</td>
<td>analysis</td>
</tr>
<tr>
<td>did1</td>
<td>on</td>
</tr>
<tr>
<td>did1</td>
<td>large</td>
</tr>
<tr>
<td>did1</td>
<td>...</td>
</tr>
<tr>
<td>did2</td>
<td>system</td>
</tr>
<tr>
<td>did2</td>
<td>with</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

map = group by
reduce = count(…) (or sum(…) or…)

MapReduce = Group-by-aggregate
### MAP

<table>
<thead>
<tr>
<th>(did1, v1)</th>
<th>(w1, 1)</th>
<th>(did2, v2)</th>
<th>(w2, 1)</th>
<th>(did3, v3)</th>
<th>(w3, 1)</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### REDUCE

Shuffle

<table>
<thead>
<tr>
<th>(w1, 1)</th>
<th>(w2, 1)</th>
<th>(w3, 1)</th>
<th>(w1, (1,1,1,...,1))</th>
<th>(w2, (1,1,...))</th>
<th>(w3, (1...))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|     |     |     | (w1, 25)       | (w2, 77)       | (w3, 12)      |
|     |     |     |               |               |              |
|     |     |     |               |               |              |
|     |     |     |               |               |              |
|     |     |     |               |               |              |

DATA516/CSED516 - Fall 2021
Examples from the paper

Discuss in class how to implement in MR

• Distributed grep

• Count URL access frequency: (URL, count)

• Reverse web-link graph: (URL, (list of URLs))

• Inverted index: (word, (list of URLs))
Jobs v.s. Tasks

• A MapReduce Job
  – One simple “query”, e.g. count words in docs
  – Complex queries may require many jobs

• A Map Task, or a Reduce Task
  – A group of instantiations of the map-, or reduce-function, to be scheduled on a single worker
Workers

• A worker is a process that executes one task at a time

• Typically there is one worker per processor, hence 4 or 8 per node
Fault Tolerance

• If one server fails once every year…
  … then a job with 10,000 servers will fail in less than one hour

• MapReduce handles fault tolerance by writing intermediate files to disk:
  – Mappers write file to local disk
  – Reducers read the files (=reshuffling); if the server fails, the reduce task is restarted on another server
MAP Tasks

(did1, v1)

(did2, v2)

(did3, v3)

...

REDUCE Tasks

Shuffle

(w1, 1)

(w2, 1)

(w3, 1)

...

(w1, (1, 1, 1, ..., 1))

(w2, (1, 1, ..., 1))

(w3, (1, 1, ..., 1))

...

(w1, 25)

(w2, 77)

(w3, 12)

...

...

...

...

...

...

...

...

Choosing Parameters in MR

• Number of map tasks (M):
  – Default: one map task per chunk
  – E.g. data = 64TB, chunk = 64MB $\Rightarrow M = 10^6$

• Number of reduce tasks (R):
  – No good default; set manually $R \ll M$
  – E.g. $R = 500$ or $5000$

• In general, MapReduce had very many parameters that required expertise to tune
MapReduce Execution Details

Map

Task

Reduce

(Task)

Intermediate data goes to local disk: $M \times R$ files (why?)

Data not necessarily local

File system: GFS or HDFS

Output to GFS or HDFS

Data not necessarily local

File system: GFS or HDFS

Intermediate data goes to local disk: $M \times R$ files (why?)
Discussion

Why doesn’t MR determine the number of reduce tasks dynamically, after all map tasks finish?
Discussion

Why doesn’t MR determine the number of reduce tasks $R$ dynamically, after all map tasks finish?

Because each map tasks needs to write its output into $R$ file; so $R$ must be known before the map tasks start.
MapReduce Phases

Map Task

{P 1} Split → Record Reader → Map → Combine

Reduce Task

{P 3} Copy → Sort → Reduce

Local storage

HDFS

file

HDFS

{P 4} {P 5}
Riddle

• The combiner function performs an optimization that you already know
• Which one?
Riddle

• The combiner function performs an optimization that you already know

• Which one?

• Pushing aggregates down
Riddle

• The combiner function performs an optimization that you already know

• Which one?

• Pushing aggregates down:
  – Each mapper groups by word

\[ \text{Temp} = \]
\[
\begin{align*}
\text{select} & \quad \text{server, word, count(\ast) as c} \\
\text{from} & \quad \text{Data} \\
\text{group by} & \quad \text{server, word}
\end{align*}
\]
Riddle

• The combiner function performs an optimization that you already know

• Which one?

• Pushing aggregates down:
  – Each mapper groups by word
  – Reducers perform final group-by

\[
\text{Temp} = \begin{align*}
\text{select} & \quad \text{server, word, count}(\ast) \text{ as } c \\
\text{from} & \quad \text{Data} \\
\text{group by} & \quad \text{server, word}
\end{align*}
\]

\[
\text{Output} = \begin{align*}
\text{select} & \quad \text{word, sum}(c) \\
\text{from} & \quad \text{Temp} \\
\text{group by} & \quad \text{word}
\end{align*}
\]
Implementation

- There is one master node
- Master partitions input file into $M$ splits, by key
- Master assigns workers (=servers) to the $M$ map tasks, keeps track of their progress
- Workers write their output to local disk, partition into $R$ regions
- Master assigns workers to the $R$ reduce tasks
- Reduce workers read regions from the map workers’ local disks
MapReduce v.s. Databases

Blog by DeWitt and Stonebraker
MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

• “Schemas are good”
MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

• “Schemas are good”
• “Indexes”
MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- “Schemas are good”
- “Indexes”
- “Skew” (MR mitigates it somewhat, how?)
MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

• “Schemas are good”
• “Indexes”
• “Skew” (MR mitigates it somewhat, how?)
• The M * R problem – what is it?
MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- “Schemas are good”
- “Indexes”
- “Skew” (MR mitigates it somewhat, how?)
- The M * R problem – what is it?
- “Parallel databases uses push (to sockets) instead of pull” – what’s the point?
Outline

• Spark

• MapReduce and critique

  • Fault Tolerance

• Hive (short)

Next lecture: Parallel databases
Fault Tolerance
Fault Tolerance

• Traditional RDBMs:
  – Major concern: recover after failure
  – FT: not a concern

• Massively distributed systems:
  – Probability of failure increases w/ no. of workers and length of job
Fault Tolerance

Example:

• if a server fails once/year…

• … a job with 10000 servers fails once/hour
Fault Tolerance

How is fault tolerance handled in each system?

• **MapReduce**: if a worker fails then

• **Spark**: 
Fault Tolerance

How is fault tolerance handled in each system?

• **MapReduce**: if a worker fails then
  – All its completed map tasks need re-executed
  – Its in-progress reduce task needs re-executed

• **Spark**: 
Fault Tolerance

How is fault tolerance handled in each system?

- **MapReduce**: if a worker fails then
  - All its completed map tasks need re-executed
  - Its in-progress reduce task needs re-executed: this is possible because the map tasks still have intermediate data on their local disks

- **Spark**:

DATA516/CSED516 - Fall 2021
Fault Tolerance

How is fault tolerance handled in each system?

• **MapReduce**: if a worker fails then
  – All its completed map tasks need re-executed
  – Its in-progress reduce task needs re-executed: this is possible because the map tasks still have intermediate data on their local disks

• **Spark**: will discuss next
Approach

New abstraction: Resilient Distributed Datasets

RDD properties
• Parallel data structure
• Can be persisted in memory
• Fault-tolerant
• Users can manipulate RDDs with rich set of operators
Resilient Distributed Datasets

• RDD = Resilient Distributed Dataset
  – Distributed, immutable.
  – Records lineage = expression that says how that relation was computed = a relational algebra plan

• Spark stores intermediate results as RDD

• 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
RDDs

```scala
lines = spark.textFile("hdfs://...")
result = lines.filter(l -> l.startsWith("ERROR"))
  .filter(l -> l.contains("sqlite"))
result.collect();
```

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

```
lines   = spark.textFile("hdfs://…")
result = lines.filter(l -> l.startsWith("ERROR"))
       .filter(l -> l.contains("sqlite"))
result.collect();
```

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

```
lines   = spark.textFile("hdfs://…")
errors = lines.filter(l -> l.startsWith("ERROR"))
result = errors.filter(l -> l.contains("sqlite"))
result.collect();
```
lines   = spark.textFile("hdfs://...")
result = lines.filter(l -> l.startsWith("ERROR"))
         .filter(l -> l.contains("sqlite"))
result.collect();

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

Spark can recompute the result from errors
Example

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

Parses each line into an object persisting in memory or on disk
SELECT count(*) FROM R, S
WHERE R.B > 200 and S.C < 100 and R.A = S.A
RDD Details

• An RDD is a partitioned collection of records
  – RDD’s are typed: RDD[Int] is an RDD of integers
  – Records are Java/Python objects
• An RDD is read only
  – This means no updates to individual records
  – This is to contrast with in-memory key-value stores
• To create an RDD
  – Execute a deterministic operation on another RDD
  – Or on data in stable storage
  – Example operations: map, filter, and join
RDD Materialization

• Users control persistence and partitioning

• Persistence
  – Materialize this RDD in memory

• Partitioning
  – Users can specify key for partitioning an RDD
Outline

• Spark

• MapReduce and critique

• Fault Tolerance

• Hive (short)

Next lecture: Parallel databases
Hive

• Facebook’s implementation of SQL over MR
• Supports subset of SQL
• Uses MapReduce runtime (pros/cons?)
  – Note: this is similar to Google’s FlumeJava
Hive

• Facebook’s implementation of SQL over MR
• Supports subset of SQL
• Uses MapReduce runtime (pros/cons?)
  – Note: this is similar to Google’s FlumeJava
• Optimizations:
Hive

- Facebook’s implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google’s FlumeJava
- Optimizations:
  - Column pruning
Hive

• Facebook’s implementation of SQL over MR
• Supports subset of SQL
• Uses MapReduce runtime (pros/cons?)
  – Note: this is similar to Google’s FlumeJava
• Optimizations:
  – Column pruning
  – Predicate push-down
Hive

- Facebook’s implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google’s FlumeJava

- Optimizations:
  - Column pruning
  - Predicate push-down
  - Partition pruning
Hive

- Facebook’s implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google’s FlumeJava
- Optimizations:
  - Column pruning
  - Predicate push-down
  - Partition pruning
  - Map-side join = "broadcast join" (discuss in class)
Hive

- Facebook’s implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google’s FlumeJava
- Optimizations:
  - Column pruning
  - Predicate push-down
  - Partition pruning
  - Map-side join = ”broadcast join” (discuss in class)
  - Join reordering
Discussion

- Parallel database systems: since the 80s
  - Will discuss next lecture
- MapReduce: around 2000
- Hive: built on MapReduce
- Spark: “better” MapReduce around 2010
- Snowflake, Aurora: cloud, parallel databases; around 2015 (next lecture)

Quick comparison (next slides)
MapReduce v.s. Spark

- Job = Map+Reduce
- Language = Java
- Data = untyped
- Optimization = no
- Job = any query
- Language ≈ RA
- Data = has schema
- Optimization = yes but limited: missing stats on base data
Spark v.s. RDBMS (e.g. Snowflake)

- Query language = its own proprietary
- Optimizer = limited
- Runtime = its own proprietary
- External functions = yes; very useful in ML

- Query language = SQL
- Optimizer = full scale
- Runtime = efficient SQL query engine
- External functions = no