DATA516/CSED516
Scalable Data Systems and Algorithms

Lecture 3
MapReduce & Spark
Announcements

• HW2 is posted, and due on Nov. 2\textsuperscript{nd}

• Project proposals due on Oct. 30\textsuperscript{th}

• Three (!) paper reviews were due today!
  – Plus a blog…
  – We’ll discuss those topics in detail
Parallel Query Processing

• Clusters:
  – More servers → more likely to fit data in main memory
  – More servers → more computing power
  – Clusters are now cheaply available in the cloud
  – A.k.a. distributed query processing

• Multicores: the end of Moore’s law
Architectures for Parallel Databases

• Shared memory

• Shared disk

• Shared nothing
Shared Memory

- SMP = symmetric multiprocessor
- Nodes share RAM and disk
- 10x … 100x processors

- Example: SQL Server runs on a single machine and can leverage many threads to speed up a query

- Easy to use and program
- Expensive to scale
Shared Disk

- All nodes access same disks
- 10x processors
- Example: Oracle
- No more memory contention
- Harder to program
- Still hard to scale
Shared Nothing

- Cluster of commodity machines
- Called "clusters" or "blade servers"
- Each machine: own memory&disk
- Up to x1000-x10000 nodes
- Example: redshift, spark, etc, etc

Because all machines today have many cores and many disks, shared-nothing systems typically run many "nodes" on a single physical machine.

- Easy to maintain and scale
- Most difficult to administer and tune.
Performance Metrics

Nodes = processors = computers

• **Speedup:**
  – More nodes, same data → higher speed

• **Scaleup:**
  – More nodes, more data → same speed
Linear v.s. Non-linear Speedup

Ideal

Real

Speedup

# nodes (=P)

×1

×5

×10

×15

DATA516/CSED516 - Fall 2020
Linear v.s. Non-linear Scaleup

Batch Scaleup

# nodes (=P) AND data size

×1  ×5  ×10  ×15

Ideal

Real
Why Sub-linear?

• **Startup cost**
  – Cost of starting an operation on many nodes

• **Interference**
  – Contention for resources between nodes

• **Skew**
  – Slowest node becomes the bottleneck
Horizontal Data Partitioning

• Distribute the n data on the p servers, such that each server only needs to process n/p data items.

• Called *horizontal data partitioning*
Horizontal Data Partitioning

Data:

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Servers:

1 2 \ldots P
Horizontal Data Partitioning

Data:

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Servers:

1

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

2

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

...
Horizontal Data Partitioning

Data:

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Servers:

1

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

2

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

... 

P

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Which tuples go to what server?
Horizontal Data Partitioning

• **Block Partition, a.k.a. Round Robin:**
  – Partition tuples arbitrarily s.t. \( \text{size}(R_1) \approx \ldots \approx \text{size}(R_P) \)

• **Hash partitioned on attribute A:**
  – Tuple \( t \) goes to chunk \( i \), where \( i = h(t.A) \mod P + 1 \)

• **Range partitioned on attribute A:**
  – Partition the range of \( A \) into \( -\infty = v_0 < v_1 < \ldots < v_P = \infty \)
  – Tuple \( t \) goes to chunk \( i \), if \( v_{i-1} < t.A < v_i \)
Skew

- Skew means that one server runs much longer than the other servers

- Reasons:
  - Skew in data distribution: will discuss later in detail
  - Skew arising from computation: much harder to address, but less common during query processing
Brent’s Theorem

Suppose we can solve a problem in time $T$ using $P$ servers:
Brent’s Theorem

Suppose we can solve a problem in time $T$ using $P$ servers:

1 2 3 4 5 6 7 8 9 P

Then, with $P' < P$ servers, we can solve the problem in time $T \cdot \frac{P}{P'}$. 
Brent’s Theorem

Suppose we can solve a problem in time $T$ using $P$ servers:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Then, with $P' < P$ servers, we can solve the problem in time $T \cdot \frac{P}{P'}$.
Linear v.s. Non-linear Speedup

![Graph showing linear vs. non-linear speedup](image-url)

- **Ideal** line represents the theoretical speedup.
- **Real** line shows the actual performance, which is lower than ideal.

**# nodes (=P)**: The number of nodes increases as follows:
- ×1
- ×5
- ×10
- ×15

**Speedup**: The vertical axis represents the speedup achieved.
Linear v.s. Non-linear Speedup

Brent’s Theorem

Ideal

Real
Linear v.s. Non-linear Speedup

- **Ideal**
- **Real**

Brent’s Theorem

Speedup

\[
\text{Speedup} = \frac{\text{Ideal Speedup}}{\text{Real Speedup}} \times \left(\frac{\text{# nodes}}{1}\right)
\]

- Ideal:
  - \(\times 1\)
  - \(\times 5\)
  - \(\times 10\)
  - \(\times 15\)

- Real:
  - \(\times 1\)

# nodes (=P)

DATA516/CSED516 - Fall 2020
Linear v.s. Non-linear Speedup

- Ideal
- Real

# nodes (\(=P\))

\[\times 1, \times 5, \times 10, \times 15\]

Speedup

Brent’s Theorem
Linear v.s. Non-linear Speedup

![Graph showing linear vs. non-linear speedup with # nodes (P) on the x-axis and speedup on the y-axis. The graph includes a line labeled 'Ideal' and another line labeled 'Brent’s Theorem,' with markers indicating performance at different # nodes: x1, x5, x10, and x15. The real performance is shown as a dashed line below the ideal performance.]
Discussion: Virtual Servers

In practice, Brent’s theorem is used to justify designing algorithm \textit{virtual servers}:

- Design algorithm for \( P \) virtual servers
- Scale down to \( P' \ll P \) physical servers

Caveat: some advanced query algorithms are better design directly for physical servers.
Discussion: Skew Mitigation

The principle in Brent’s theorem is used to mitigate skew
Discussion: Skew Mitigation

The principle in Brent’s theorem is used to mitigate skew

• Design your parallel algorithm to work with \( P \) virtual servers; \( P \) is very large
  – Ideal runtime \( T_{\text{ideal}}(P) = \) very small
  – Skew causes \( T_{\text{real}}(P) = \) much bigger
Discussion: Skew Mitigation

The principle in Brent’s theorem is used to mitigate skew

• Design your parallel algorithm to work with $P$ virtual servers; $P$ is very large
  – Ideal runtime $T_{\text{ideal}}(P) =$ very small
  – Skew causes $T_{\text{real}}(P) =$ much bigger

• But we have $P' \ll P$ servers
  – Distribute load dynamically, balancing skew
  – $T_{\text{real}}(P') \approx P/P' * T_{\text{ideal}}(P)$
Discussion: Skew Mitigation

Skew!

1 2 3

1 2 ...
P'

1

2

P
Outline

Rest of this lecture:
• MapReduce (+ brief discussion of Hive)
• Spark

Next lecture(s):
• Brief discussion of Snowflake
• Parallel query evaluation algorithms
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
Data Model for MapReduce

Files!

A file = a bag of (key, value) pairs

A MapReduce program:
• Input: a bag of (inputkey, value) pairs
• Output: a bag of (outputkey, value) pairs
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)

```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));
```
MapReduce = GroupBy-Aggregate

Occurrence(docID, word)

\[
\text{select word, count(*) from Occurrence group by word}
\]

map = group by reduce = count(...) (or sum(...) or...)
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
<table>
<thead>
<tr>
<th>MAP Tasks</th>
<th>REDUCE Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(did1,v1)</td>
<td>(w1, (1,1,1,\ldots,1))</td>
</tr>
<tr>
<td>(did2,v2)</td>
<td>(w2, (1,1,\ldots))</td>
</tr>
<tr>
<td>(did3,v3)</td>
<td>(w3, (1\ldots))</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

**Shuffle**
Choosing Parameters in MR

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

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

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

Map

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

Task

Data not necessarily local

File system: GFS or HDFS
Discussion

Why doesn’t MR determine the number of reduce tasks R 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
- Split
- Record Reader
- Map
- Combine

Reduce Task
- Copy
- Sort
- Reduce

Local storage
HDFS

{P 1} {P 2} {P 3} {P 4} {P 5}
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
Interesting Implementation Details

Worker failure:

- Master pings workers periodically,
- If down then reassigns the task to another worker
Interesting Implementation Details

Backup tasks:

• *Straggler* = a machine that takes unusually long time to complete one of the last tasks.
  – Bad disk forces frequent correctable errors (30MB/s → 1MB/s)
  – The cluster scheduler has scheduled other tasks on that machine

• Stragglers are a main reason for slowdown

• Solution: *pre-emptive backup execution of the last few remaining in-progress tasks*
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?
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
Spark

A Case Study of the MapReduce Programming Paradigm
References


Motivation

Goal: Better use **distributed memory** in a cluster

- Modern data analytics requires **iterations**
- Users also want **interactive** data mining
- Both cases: want to **keep intermediate data in memory** and reuse it
- MapReduce does not do this well: requires writing data to disk between jobs
Spark v.s. Hive

A Key Difference

• Hive used the MR runtime

• Spark built its own runtime
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
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<T> = an RDD collection of type T
• Distributed on many servers, not nested
• Operations are done in parallel
• Recoverable via lineage; more later

Seq<T> = a sequence
• Local to one server, may be nested
• Operations are done sequentially
Search logs stored in HDFS

```java
// 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()
```
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
events = lines.filter(x -> x.startsWith("Error"))

// Persist the RDD in memory for reuse later
events.persist()
events.collect()
events.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();
Example

The RDD s:

<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>
</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();
```

Parallel step 1
Example

The RDDs:

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

The RDDs:

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

Parallel step 1

Parallel step 2
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><code>map(f : T -&gt; U)</code></td>
<td>RDD&lt;T&gt; -&gt; RDD&lt;U&gt;</td>
</tr>
<tr>
<td><code>flatMap(f: T -&gt; Seq(U))</code></td>
<td>RDD&lt;T&gt; -&gt; RDD&lt;U&gt;</td>
</tr>
<tr>
<td><code>filter(f:T-&gt;Bool)</code></td>
<td>RDD&lt;T&gt; -&gt; RDD&lt;T&gt;</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>RDD&lt;(K,V)&gt; -&gt; RDD&lt;(K,Seq[V])&gt;</td>
</tr>
<tr>
<td><code>reduceByKey(F:(V,V)-&gt; V)</code></td>
<td>RDD&lt;(K,V)&gt; -&gt; RDD&lt;(K,V)&gt;</td>
</tr>
<tr>
<td><code>union()</code></td>
<td>(RDD&lt;T&gt;,RDD&lt;T&gt;) -&gt; RDD&lt;T&gt;</td>
</tr>
<tr>
<td><code>join()</code></td>
<td>(RDD&lt;(K,V)&gt;,RDD&lt;(K,W)&gt; ) -&gt; RDD&lt;(K,(V,W))&gt;</td>
</tr>
<tr>
<td><code>cogroup()</code></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><code>crossProduct()</code></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><code>count()</code></td>
<td>RDD&lt;T&gt; -&gt; Long</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>RDD&lt;T&gt; -&gt; Seq&lt;T&gt;</td>
</tr>
<tr>
<td><code>reduce(f:(T,T)-&gt;T)</code></td>
<td>RDD&lt;T&gt; -&gt; T</td>
</tr>
<tr>
<td><code>save(path: String)</code></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
}

[From Zaharia12]
Fault Tolerance

• When a job is executed on x100 or x1000 servers, the probability of a failure is high

• Example: if a server fails once/year, then a job with 10000 servers fails once/hour

• Different solutions:
  – RDBMS: Restart!
  – MapReduce: write everything to disk, redo. Slow.
  – Spark: redo only what is needed. Better.
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.
Digression: Lineage

*Lineage* or *provenance* means “information telling us how the data was derived”

- **Fine grain lineage**: complete query plan to reconstruct the data, e.g. RDD
- **Coarse grain lineage**:
  - “this dataset was computed on 2017.03.14 using PROG733 version 4.13, from the dataset D3225.dat collected on 2017.02.01 from the field”
There is no such thing as “data pedigree” 😊
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

```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

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

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

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
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 = strm.read().textFile("R.csv").map(parseRecord).persist();
S = strm.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();
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
1. Input data in HDFS or other Hadoop input source
2. User writes driver program, which includes a SparkContext
   1. SparkContext connects to cluster manager (e.g., YARN)
   2. Spark acquires executors (= procs) through cluster manager
   3. Ships code to workers, which cache data & execute tasks
3. Each app is independent set of procs (no sharing across apps)
Query Execution Details

• Lazy evaluation
  – RDDs are not evaluated until an action is called

• In memory caching
  – Spark workers are long-lived processes
  – RDDs can be materialized in memory in workers
  – Base data is not cached in memory
Key Challenge

• How to provide fault-tolerance efficiently?
Fault-Tolerance Through Lineage

Represent RDD with 5 pieces of information

• A set of partitions
• A set of dependencies on parent partitions
  – Distinguishes between narrow (one-to-one)
  – And wide dependencies (one-to-many)
• Function to compute dataset based on parent
• Metadata about partitioning scheme and data placement

RDD = Distributed relation + lineage
More Details on Execution

Scheduler builds a DAG of stages based on lineage graph of desired RDD.

Pipelined execution within stages

Synchronization barrier with materialization before shuffles

If a task fails, re-run it

Can checkpoint RDDs to disk

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 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]
Spark Ecosystem Growth

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
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