Introduction to Data Management
CSE 344

Lecture 25: MapReduce and Spark
Today

• MapReduce Review

• Spark
Review: Map Reduce Data Model

Started by Google in 2004

**Instance**: Files containing (key, value) pairs

**Schema**: None!
- just like other key-value data models

**Query language**: a MapReduce program:
- Input: a bag of (key, value) pairs
- Output: a bag of (key, value) pairs
- Implementation in Java (Hadoop), Python, Go, …
Review:
Lifecycle of a MR Program

1. Read a lot of data and parse into (key, value) pairs
2. Map: extract something you care about from each (key, value) pair
3. Shuffle output from mappers
   – done internally by implementation
4. Reduce: aggregate, summarize, filter, transform
5. Write the results to files

Paradigm stays the same,
change map and reduce functions for different problems
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");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of "1"s
  int result = 0;
  for each v in values:
    result += ParseInt(v);
    Emit(AsString(result));
```
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
Using MapReduce in Practice:
Implementing RA Operators in MR
Selection $\sigma_{A=42}(R)$

```java
map(String relationName, Tuple t):
    if t.A == 42:
        EmitIntermediate(relationName, t);
```

```java
reduce(String k, Iterator values):
    for each v in values:
        Emit(v);
```
Selection $\sigma_{A=42}(R)$

map(String relationName, Tuple t):
  if t.A == 42:
    EmitIntermediate(relationName, t);

reduce(String k, Iterator values):
  for each v in values:
    Emit(v);

- Reduce isn’t really needed
- But MR requires reduce functions
Group By $\gamma_{A,\text{sum}(B)}(R)$

map(String relationName, Tuple t):
    EmitIntermediate(t.A, t.B);

reduce(String k, Iterator values):
    s = 0
    for each v in values:
        s = s + v
    Emit(k, v);

Can’t use hashtable to map A’s to B’s
Implementing Join in MR

Two parallel join algorithms that we have seen:

• Partitioned hash-join

• Broadcast join
Parallel Execution of RA Operators: Partitioned Hash-Join

- **Data**: \( R(K1,A,B), S(K2,B,C) \)
- **Query**: \( R(K1,A,B) \bowtie S(K2,B,C) \)
  - Initially, both \( R \) and \( S \) are partitioned on \( K1 \) and \( K2 \)

Reshuffle \( R \) on \( R.B \) and \( S \) on \( S.B \)

Each server computes the join locally
\( R(A,B) \bowtie_{B=C} S(C,D) \)

**Partitioned Hash-Join in MR**

```java
map(String relationName, Tuple t):
    switch (relationName):
        case 'R': EmitIntermediate(t.B, IntKey('R', value))
        case 'S': EmitIntermediate(t.C, IntKey('S', value))

reduce(String k, Iterator values):
    R = [] S = []
    for each v in values:
        switch (v.relationName):
            'R': R.insert(v.value)
            'S': S.insert(v.value)
    for r in R, for s in S:
        Emit(Tuple(r,s))
```

Or call hash(t.B)

Relation name

value

All tuples here must join
Data: $R(A, B), S(C, D)$
Query: $R(A, B) \bowtie_{B=C} S(C, D)$

Broadcast Join

Broadcast $S$

Reshuffle $R$ on $R.B$
$R(A,B) \bowtie_{B=C} S(C,D)$

**Broadcast Join in MR**

```java
map(String relationName, Tuple [] rs):
    S = readFromNetwork()
    ht = new Hashtable()
    for each w in S:
        ht.insert(w.C, w)

    for each r in ts:
        for each s in ht.find(r.B):
            Emit(Tuple(r,s))
```

**reduce(...):**
/* empty: map-side only */
Issues with MapReduce

• Difficult to write complex queries
  – Nested queries
  – Correlated queries?

• Fault tolerance: only persists results between map / reduce

• Need multiple MR jobs: dramatically slows down because each job writes all results to disk
Parallel Data Processing @ 2010
Spark

• Open source system from Berkeley
• Distributed processing over HDFS
• Differences from MapReduce:
  – Not restricted to pairs of mapper and reducer
  – User decides when to persist results

• Details: [http://spark.apache.org/examples.html](http://spark.apache.org/examples.html)
Spark Data Model

**Instance**: Resilient Distributed Datasets (RDDs)

**Schema**: None! (just like MR)

**Query language**: a Spark program
- Implementation in Scala / Java / SQL
- Scala = extension of Java with functions/closures
RDD

• RDD = Resilient Distributed Datasets
  – A distributed relation, together with its lineage
  – Lineage: expression that says how that relation was computed

• 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
How does Spark store lineage?

- A Spark/Scala program consists of:
  - Transformations (map, reduce, join...). **Lazily evaluated**
  - Only record the function to invoke, actual work not done
  - Actions (count, reduce, save...). **Eagerly evaluated**
  - Really performs work
Example

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

- Start with “ERROR”
- Contain the string “sqlite”

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```
Example

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

• Start with “ERROR”
• Contain the string “sqlite”

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```
Persistence

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
result = sqlerrors.collect();
```

If any server fails before collect, then the entire job is restarted
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
errors.persist();
sqlerrors = errors.filter(_.contains("sqlite"));
result = sqlerrors.collect();

Spark can recompute the result from errors.
### Transformations:

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

### Actions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td><code>RDD[T] =&gt; Long</code></td>
</tr>
<tr>
<td><code>collect()</code></td>
<td><code>RDD[T] =&gt; Seq[T]</code></td>
</tr>
<tr>
<td><code>reduce(f:(T,T)=&gt;T)</code></td>
<td><code>RDD[T] =&gt; T</code></td>
</tr>
<tr>
<td><code>save(path:String)</code></td>
<td>Outputs RDD to a storage system e.g. HDFS</td>
</tr>
</tbody>
</table>
Conclusions

• Parallel databases
  – Predefined relational operators
  – Optimization using standard RA techniques
  – Transaction support is free

• MapReduce
  – User-defined map and reduce functions
  – Must implement/optimize manually relational ops
  – No updates/transactions

• Spark
  – Predefined relational operators
  – Must optimize manually
  – No updates/transactions