CSE544
Data Management

Lectures 16-17
Parallel Query Processing
Announcements

• Project proposal due on Friday
  – Please follow the suggestions on the Web: title/name, description, system, data, plan

• HW4 due on Tuesday
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, price)

Hint on HW4

```
SELECT x.sno, x.sname
FROM Supplier x
WHERE x.sstate = 'WA'
    and s.sno not in
    (SELECT y.sno
     FROM Supply y
     WHERE y.price > 100)
```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, price)

Hint on HW4

SELECT x.sno, x.sname
FROM Supplier x
WHERE x.sstate = 'WA'
and s.sno not in
(SELECT y.sno
FROM Supply y
WHERE y.price > 100)
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Hint on HW4

SELECT x.sno, x.sname
FROM Supplier x
WHERE x.sstate = ‘WA’
    and s.sno not in
    (SELECT y.sno
     FROM Supply y
     WHERE y.price > 100)
Hint on HW4

SELECT x.sno, x.sname
FROM Supplier x
WHERE x.sstate = 'WA'
    and s.sno not in
    (SELECT y.sno
     FROM Supply y
     WHERE y.price > 100)

Incorrect!

why?

How do we fix it?
Hint on HW4

```
SELECT x.sno, x.sname
FROM Supplier x
WHERE x.sstate = 'WA'
    and s.sno not in
       (SELECT y.sno
         FROM Supply y
         WHERE y.price > 100)
```
Hint on HW4

```
SELECT x.sno, x.sname
FROM Supplier x
WHERE x.sstate = 'WA'
    and s.sno not in
    (SELECT y.sno
     FROM Supply y
     WHERE y.price > 100)
```

Incorrect!

**why?**

Fixed

Also fixed
Outline

- MapReduce
- Spark
- Snowflake
References

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

• Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica: Spark: Cluster Computing with Working Sets. HotCloud 2010

• Dageville et al.: The Snowflake Elastic Data Warehouse, SIGMOD’2016
MapReduce
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

• 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
Data Model

Files!

A file = a bag of (key, value) pairs
Step 1: the MAP Phase

User provides the MAP-function:

• Input: (input key, value)
• Output: bag of (intermediate key, value)

System applies the map function in parallel to all (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");

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

$\text{Occurrence}(\text{docID, word})$

\[
\begin{align*}
\text{select} & \quad \text{word, count(*)} \\
\text{from} & \quad \text{Occurrence} \\
\text{group by} & \quad \text{word}
\end{align*}
\]
MAP

REDUCE

Shuffle

CSEP 544 - Spring 2021
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
MAP Tasks

REDUCE Tasks

Shuffle
Fault Tolerance

• If one server fails once every year... ... then a job on 10,000 servers fails in 1h

• MapReduce handles fault tolerance by writing intermediate files to disk:
  – Mappers write file to local disk
  – Reducers read the local files (=reshuffling);
  – If reducer fails, new work re-reads local files
MapReduce Execution Details

Map

Task

Reduce

(Shuffle)

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

File system: GFS or HDFS

Task

Data not necessarily local

File system: GFS or HDFS
MapReduce Phases
Joins in MR

• One MR job = GROUP-BY/AGGREGATE

• Question: How do we implement a join?

• Answer:
  – Map function: groups by the join attribute
  – Reduce function: cartesian product
Joins in MR

\[ R(A,B,C) \bowtie_{A=D} S(D,E) \]

**map(String \text{ value}):**

- case value of:
  - \( R(a,b,c) \): emit\_Intermediate(a, R(a,b,c))
  - \( S(d,e) \): emit\_Intermediate(d, S(d,e))

**reduce(String \text{ key}, Iterator values):**

local\_R = [], local\_S = []

for each \( v \) in values:
  - case \( R(a,b,c) \): local\_R += \( v \)
  - case \( S(d,e) \): local\_S += \( v \)

emit( local\_R x local\_S );
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?

*Original blog deleted, cached version still here; slightly longer paper here.
Outline

• MapReduce
  • Spark
• Snowflake
Spark
Spark

- Distributed processing over HDFS
- Multiple steps, including iterations
- Stores intermediate results in main memory
- Closer to relational algebra
Collections in Spark

• **RDD<T>** = an RDD collection of type T
  – Distributed
  – Not nested
  – Recoverable via lineage

• **Seq<T>** = a sequence
  – Local
  – May be nested
Programming in Spark

• Transformations (map, join…). Lazy

• Actions (count, reduce, save…). Eager
Example

Find in file hdfs://logfile.log the lines that:
• Start with “ERROR”
• Contain the string “sqlite”

```scala
s = SparkSession.builder().getOrCreate();
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l => l.startsWith("ERROR"));
sqlerrors = errors.filter(l => l.contains("sqlite"));
sqlerrors.collect();
```
Example

Find in file hdfs://logfile.log the lines that:

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

```java
s = SparkSession.builder()...getOrCreate();
sqlerrors = s.read().textFile("hdfs://logfile.log")
  .filter(l -> l.startsWith("ERROR"))
  .filter(l -> l.contains("sqlite"))
  .collect();
```

“Call chaining” style
Example

The RDD s:

| Error... | Warning... | Warning... | Error... | Abort... | Abort... | Error... | Error... | Warning... | Error... |

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

sqlerrors = s.read().textFile("hdfs://logfile.log")
  .filter(l -> l.startsWith("ERROR"))
  .filter(l -> l.contains("sqlite"))
  .collect();
```
Example

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>

\[
s = \text{SparkSession.builder()...getOrCreate();}
\]

\[
sqlerrors = s.read().textFile("hdfs://logfile.log")
    .filter(l -> l.startsWith("ERROR"))
    .filter(l -> l.contains("sqlite"))
    .collect();
\]
Example

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>

Error...

Error...

Error...

Error...

Error...

Error...

Error...

Error...

Error...

Error...

\[
s = \text{SparkSession.builder()...getOrCreate();}
\]

\[
sqlerrors = s.read().textFile("hdfs://logfile.log")
 .filter(l -> l.startsWith("ERROR"))
 .filter(l -> l.contains("sqlite"))
 .collect();
\]
s = SparkSession.builder()...getOrCreate();

sqlerrors = s.read().textFile("hdfs://logfile.log")
  .filter(l -> l.startsWith("ERROR"))
  .filter(l -> l.contains("sqlite"))
  .collect();
Fault Tolerance

• Parallel database systems: restart. Expensive.

• MapReduce: write everything to disk, redo. Slow.

• Spark: redo only what is needed. Efficient.
Resilient Distributed Datasets

RDD = Resilient Distributed Dataset

• Distributed, immutable *lineage*

• Lineage = a relational algebra plan

• If a server crashes, recompute the lost partition of the RDD using the lineage
Persistence

```scala
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect();
```

If any server fails before the end, then Spark must restart
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect();

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

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

```scala
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
sqllerrors = errors.filter(l->l.contains("sqlite"));
sqllerrors.collect();
```

Spark can recompute the result from errors
Persistence

If any server fails before the end, then Spark must restart the result from errors. If errors are filtered where they start with "ERROR" and contain "sqlite", then Spark can recalculate the result.
Example

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

R(A,B)  
S(A,C)  

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

Parses each line into an object  
persist
Example

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();

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

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

### Actions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Input Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>RDD&lt;T&gt;</td>
<td>Long</td>
</tr>
<tr>
<td>collect()</td>
<td>RDD&lt;T&gt;</td>
<td>Seq&lt;T&gt;</td>
</tr>
<tr>
<td>reduce(f : (T,T)→T)</td>
<td>RDD&lt;T&gt;</td>
<td>T</td>
</tr>
<tr>
<td>save(path: String)</td>
<td>Outputs RDD to a storage system e.g., HDFS</td>
<td></td>
</tr>
</tbody>
</table>
Spark 2.0

• DataFrames
  – Records, dynamically typed

• Datasets
  – Records, statically typed
Outline

• MapReduce

• Spark

• Snowflake
Snowflake
Snowflake

• It is an SaaS – what is this? Give other examples of types of cloud services…
Snowflake

• It is an SaaS – what is this? Give other examples of types of cloud services…
• SaaS = software as a service
• Other examples:
  – Platform as a service (PaaS): e.g. Amazon’s EC
  – Infrastructure as a service (virtual machines)
  – Software as a Service
  – Function as a Service: Amazon’s Lambda
Snowflake

• Describe Snowflake’s Data Storage
Snowflake

• Describe Snowflake’s Data Storage

In class:
• S3: PUT/GET/DELETE
• Table → horizontal partition in files
• Blobs+PAX
• Temp storage → S3

Figure 1: Multi-Cluster, Shared Data Architecture
Snowflake

• Describe Elasticity in Snowflake

• Describe failure handling in Snowflake
Snowflake

• Describe Elasticity in Snowflake
  – Virtual Warehouse (VW) serves one user
  – T-Shirt sizes: X-Small … XX-Large
  – Small query may run on subset of VW

• Describe failure handling in Snowflake
Snowflake

• Describe Elasticity in Snowflake
  – Virtual Warehouse (VW) serves one user
  – T-Shirt sizes: X-Small … XX-Large
  – Small query may run on subset of VW

• Describe failure handling in Snowflake
  – Restart the query
  – No partial retries (like MapReduce or Spark)
Snowflake

• Describe its execution engine
Snowflake

• Describe its execution engine

• Column-oriented (in class)

• Vectorized (“tuple batches” – in class)

• Push-based (in class)
Snowflake

• What does Snowflake use instead of indexes?
Snowflake

• What does Snowflake use instead of indexes?

• “Pruning”: for each file (recall: this is a horizontal partition of a table) and each attribute, it stores the min/max values in that column in that file; may skip files when not needed.
Conclusion

• Distributed data processing:
  – Spread the data to fit in main memory
  – Take advantage of parallelism
• “SQL is embarrassingly parallel”
  – Relational algebra: easy to parallelize
  – Hash-based algorithm suffer from skew