Introduction to Data Management
CSE 344

Unit 5: Parallel Data Processing

Parallel RDBMS
MapReduce
Spark

(4 lectures)
Introduction to Data Management
CSE 344

Spark
Announcement

• HW6 posted
  – We use Amazon Web Services (AWS)
  – Urgent: please sign up for AWS credits (see instructions on the homework)

• No classes on Monday (Veterans’ Day)
Class Overview

- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
- Unit 3: Non-relational data
- Unit 4: RDMBS internals and query optimization
- Unit 5: Parallel query processing
  - Spark, Hadoop, parallel databases
- Unit 6: DBMS usability, conceptual design
- Unit 7: Transactions
- Unit 8: Advanced topics (time permitting)
Parallelism is of Increasing Importance

• Multi-cores:
  – Most processors have multiple cores
  – This trend will likely increase in the future

• Big data: too large to fit in main memory
  – Distributed query processing on 100x-1000x servers
  – Widely available now using cloud services
Performance Metrics for Parallel DBMSs

Nodes = processors, computers

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

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

Speedup

Ideal

# nodes (=P)

×1

×5

×10

×15

CSE 344 - 2018au
Linear v.s. Non-linear Scaleup

Batch Scaleup

# nodes (\(=P\)) AND data size

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

Ideal
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
Spark

A Case Study of the MapReduce Programming Paradigm
Parallel Data Processing @ 2010
Spark

• Open source system from UC Berkeley
• Distributed processing over HDFS
• Differences from MapReduce (CSE322):
  – Multiple steps, including iterations
  – Stores intermediate results in main memory
  – Closer to relational algebra (familiar to you)
• Details:
  http://spark.apache.org/examples.html
Spark

• Spark supports interfaces in Java, Scala, and Python
  – Scala: extension of Java with functions/closures

• We will illustrate use the Spark Java interface in this class

• Spark also supports a SQL interface (SparkSQL), and compiles SQL to its native Java interface
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
Example

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

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

```java
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

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

Given a large log file hdfs://logfile.log retrieve all lines that:
• Start with “ERROR”
• Contain the string “sqlite”

```java
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();
```

Transformation: Not executed yet...

Action: triggers execution of entire program
Example

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

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

is the same as:

```java
class FilterFn implements Function<Row, Boolean>{
    Boolean call (Row r)
    { return l.startsWith("ERROR"); }
}
errors = lines.filter(new FilterFn());
```
Example

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

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

```
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:

<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>

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

```
s = 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

• 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:
  – 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 and records its lineage
  – 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
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
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
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

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

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

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

Spark can recompute the result from errors

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

```java
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 on disk
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();
```
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
## 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[W])))</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>
Spark 2.0

The DataFrame and Dataset Interfaces
DataFrames

• Like RDD, also an immutable distributed collection of data

• Organized into *named columns* rather than individual objects
  – 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

• Similar to DataFrames, except that elements must be typed objects

• 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)

• You will use both Datasets and RDD APIs in HW6
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?
Introduction to Data Management
CSE 344

Parallel Databases
Architectures for Parallel Databases

• Shared memory

• Shared disk

• Shared nothing
Shared Memory

- Nodes share both RAM and disk
- Dozens to hundreds of processors

Example: SQL Server runs on a single machine and can leverage many threads to speed up a query
- check your HW3 query plans

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

- All nodes access the same disks
- Found in the largest "single-box" (non-cluster) multiprocessors

**Example: Oracle**

- No more memory contention
- Harder to program
- Still hard to scale: existing deployments typically have fewer than 10 machines
Shared Nothing

- Cluster of commodity machines on high-speed network
- Called "clusters" or "blade servers"
- Each machine has its own memory and disk: lowest contention.

Example: Spark

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.

We discuss only Shared Nothing in class
Approaches to Parallel Query Evaluation

• **Inter-query parallelism**
  – Transaction per node
  – Good for transactional workloads

• **Inter-operator parallelism**
  – Operator per node
  – Good for analytical workloads

• **Intra-operator parallelism**
  – Operator on multiple nodes
  – Good for both?

**We study only intra-operator parallelism: most scalable**
Single Node Query Processing (Review)

Given relations R(A,B) and S(B, C), no indexes:

• **Selection**: $\sigma_{A=123}(R)$  
  – Scan file R, select records with A=123

• **Group-by**: $\gamma_{A,\text{sum}(B)}(R)$  
  – Scan file R, insert into a hash table using A as key  
  – When a new key is equal to an existing one, add B to the value

• **Join**: $R \bowtie S$  
  – Scan file S, insert into a hash table using B as key  
  – Scan file R, probe the hash table using B
Distributed Query Processing

• Data is horizontally partitioned on servers

• Operators may require data reshuffling
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  ...  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:

<table>
<thead>
<tr>
<th>1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...  

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

Which tuples go to what server?
Horizontal Data Partitioning

- **Block Partition:**
  - 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 \( i = h(t.A) \mod P + 1 \)
  - Recall: calling hash fn’s is free in this class

- **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 \( v_{i-1} < t.A < v_i \)
Uniform Data v.s. Skewed Data

Let $R(K,A,B,C)$; which of the following partition methods may result in skewed partitions?

- **Block partition**
  - Uniform

- **Hash-partition**
  - On the key $K$
  - On the attribute $A$

Uniform may be skewed

Assuming a good hash function

E.g. when all records have the same value of the attribute $A$, then all records end up in the same partition

Keep this in mind in the next few slides
Parallel Execution of RA Operators: Grouping

**Data**: \( R(K,A,B,C) \)
**Query**: \( \gamma_{A,\text{sum}(C)}(R) \)

How to compute group by if:

- \( R \) is hash-partitioned on \( A \)?
- \( R \) is block-partitioned?
- \( R \) is hash-partitioned on \( K \)?
Parallel Execution of RA Operators: Grouping

**Data:** $R(K, A, B, C)$

**Query:** $\gamma_{A, \text{sum}(C)}(R)$

- $R$ is block-partitioned or hash-partitioned on $K$

Diagram:

- Reshuffle $R$ on attribute $A$
- Run grouping on reshuffled partitions
Speedup and Scaleup

• Consider:
  – Query: $\gamma_{A,\sum(C)}(R)$
  – Runtime: only consider I/O costs

• If we double the number of nodes $P$, what is the new running time?
  – Half (each server holds $\frac{1}{2}$ as many records)

• If we double both $P$ and the size of $R$, what is the new running time?
  – Same (each server holds the same # of records)

But only if the data is without skew!
Skewed Data

• $R(K,A,B,C)$
• Informally: we say that the data is skewed if one server holds much more data that the average
• E.g. we hash-partition on A, and some value of A occurs very many times (“Justin Bieber”)  
• Then the server holding that value will be skewed
Parallel Execution of RA Operators: Partitioned Hash-Join

- **Data**: \( R(K_1, A, B), S(K_2, B, C) \)
- **Query**: \( R(K_1, A, B) \bowtie S(K_2, B, C) \)
  - Initially, both \( R \) and \( S \) are partitioned on \( K_1 \) and \( K_2 \)

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

Each server computes the join locally
Data: \( R(K1, A, B), S(K2, B, C) \)
Query: \( R(K1, A, B) \bowtie S(K2, B, C) \)
Data: R(A, B), S(C, D)
Query: R(A, B) \bowtie_{B=C} S(C, D)

Broadcast Join

Why would you want to do this?
Putting it Together: Example Parallel Query Plan

Find all orders from today, along with the items ordered

```
SELECT *
FROM Order o, Line i
WHERE o.item = i.item
AND o.date = today()
```
Example Parallel Query Plan

Node 1
- hash: h(o.item)
- select: date=today()
- scan: Order o

Node 2
- hash: h(o.item)
- select: date=today()
- scan: Order o

Node 3
- hash: h(o.item)
- select: date=today()
- scan: Order o

Order(oid, item, date), Line(item, …)
Example Parallel Query Plan

Node 1

hash

h(i.item)

scan

Item i

Node 2

hash

h(i.item)

scan

Item i

Node 3

hash

h(i.item)

scan

Item i

Order(oid, item, date), Line(item, ...)

date = today()

join

o.item = i.item

Order o

Node 1

Node 2

Node 3
Example Parallel Query Plan

Node 1

Join

Node 2

Join

Node 3

Join

Order(oid, item, date), Line(item, ...)

- Contains all orders and all lines where hash(item) = 1
- Contains all orders and all lines where hash(item) = 2
- Contains all orders and all lines where hash(item) = 3

Join

Join

Join

Join

Join

A Challenge

• Have P number of servers (say P=27 or P=1000)

• How do we compute this Datalog query in one step?

• $Q(x, y, z) :- R(x, y), S(y, z), T(z, x)$
A Challenge

- Have P number of servers (say P=27 or P=1000)
- How do we compute this Datalog query in one step? \( Q(x,y,z) = R(x,y), S(y,z), T(z,x) \)
- Organize the P servers into a cube with side \( P^{\frac{1}{3}} \)
  - Thus, each server is uniquely identified by \((i,j,k), i,j,k \leq P^{\frac{1}{3}}\)
HyperCube Join

• Have P number of servers (say P=27 or P=1000)
• How do we compute this Datalog query in one step?
  \[ Q(x,y,z) = R(x,y), S(y,z), T(z,x) \]
• Organize the P servers into a cube with side \( P^{1/3} \)
  – Thus, each server is uniquely identified by \((i,j,k)\), \(i,j,k \leq P^{1/3}\)
• Step 1:
  – Each server sends \( R(x,y) \) to all servers \((h(x), h(y), *)\)
  – Each server sends \( S(y,z) \) to all servers \((*, h(y), h(z))\)
  – Each server sends \( T(x,z) \) to all servers \((h(x), *, h(z))\)
HyperCube Join

• Have P number of servers (say P=27 or P=1000)
• How do we compute this Datalog query in one step?
  \[ Q(x,y,z) = R(x,y), S(y,z), T(z,x) \]
• Organize the P servers into a cube with side \( P^{\frac{1}{3}} \)
  – Thus, each server is uniquely identified by \( (i,j,k) \), \( i,j,k \leq P^{\frac{1}{3}} \)
• Step 1:
  – Each server sends \( R(x,y) \) to all servers \( \langle h(x), h(y), \ast \rangle \)
  – Each server sends \( S(y,z) \) to all servers \( \langle \ast, h(y), h(z) \rangle \)
  – Each server sends \( T(x,z) \) to all servers \( \langle h(x), \ast, h(z) \rangle \)
• Final output:
  – Each server \( (i,j,k) \) computes the query \( R(x,y), S(y,z), T(z,x) \) locally
HyperCube Join

• Have P number of servers (say P=27 or P=1000)
• How do we compute this Datalog query in one step? 
  \[ Q(x,y,z) = R(x,y), S(y,z), T(z,x) \]
• Organize the P servers into a cube with side \( P^{1/3} \)
  – Thus, each server is uniquely identified by \( (i,j,k) \), \( i,j,k \leq P^{1/3} \)
• Step 1:
  – Each server sends \( R(x,y) \) to all servers \( (h(x),h(y),*) \)
  – Each server sends \( S(y,z) \) to all servers \( (*,h(y),h(z)) \)
  – Each server sends \( T(x,z) \) to all servers \( (h(x),*,h(z)) \)
• Final output:
  – Each server \( (i,j,k) \) computes the query \( R(x,y), S(y,z), T(z,x) \) locally
• Analysis: each tuple \( R(x,y) \) is replicated at most \( P^{1/3} \) times
\[ Q(x, y, z) = R(x, y), S(y, z), T(z, x) \]
Q(x,y,z) = R(x,y),S(y,z),T(z,x)

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>S1</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R2</th>
<th>S2</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>5</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R3</th>
<th>S3</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>8</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>y</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

Hypercube join

Shuffle

What if
h(x): h(1) = h(3)
\[ Q(x,y,z) = R(x,y), S(y,z), T(z,x) \]

<table>
<thead>
<tr>
<th>R1</th>
<th>S1</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1'</th>
<th>S1'</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R2</th>
<th>S2</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R2'</th>
<th>S2'</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R3</th>
<th>S3</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R3'</th>
<th>S3'</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Diagram:
- **Partition**
  - P1: (1, 2, 7)
  - P2: (1, 2, 3)
  - P3: (3, 2, 3)

- **Shuffle**
  - What if \( h(x): h(1) = h(3) \)

- **Local Join**

**Hypercube join**
Introduction to Data Management
CSE 344

MapReduce
Announcements

• HW6 due tomorrow
• HW7 to be posted tomorrow; HW8 next week
• Makeup lectures:
  – Tuesday, Nov. 27, Room EEB 105
  – Tuesday, Dec. 4, Room EEB 105
• Canceled lectures:
  – Wednesday, Dec. 5
  – Friday, Dec. 7

Check the calendar!
Parallel Data Processing @ 2000
Optional Reading

• Original paper: https://www.usenix.org/legacy/events/osdi04/tech/dean.html

• Rebuttal to a comparison with parallel DBs: http://dl.acm.org/citation.cfm?doid=1629175.1629198

• Chapter 2 (Sections 1,2,3 only) of Mining of Massive Datasets, by Rajaraman and Ullman http://i.stanford.edu/~ullman/mmds.html
Motivation

• We learned how to parallelize relational database systems

• While useful, it might incur too much overhead if our query plans consist of simple operations

• MapReduce is a programming model for such computation

• First, let’s study how data is stored in such systems
Distributed File System (DFS)

- For very large files: TBs, PBs
- Each file is partitioned into *chunks*, typically 64MB
- Each chunk is replicated several times (≥3), on different racks, for fault tolerance
- Implementations:
  - Google’s DFS: **GFS**, proprietary
  - Hadoop’s DFS: **HDFS**, open source
MapReduce

• Google: paper published 2004
• Free variant: Hadoop
• MapReduce = high-level programming model and implementation for large-scale parallel data processing
Typical Problems Solved by MR

• Read a lot of data
• **Map**: extract something you care about from each record
• Shuffle and Sort
• **Reduce**: aggregate, summarize, filter, transform
• Write the results

Paradigm stays the same, change map and reduce functions for different problems
Data Model

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

System applies the map function in parallel to all \((\text{input key}, \text{value})\) pairs in the 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));
```
Comparison to Spark

MapReduce in Spark:

- `col.flatMap(f)` applies in parallel the predicate `f` to all elements `x` of the partitioned collection; for each `x`, `f(x)` is a list; then `flatMap` returns their concatenation.

- `col.reduceByKey(g)` applies in parallel the function `g` to all elements with a common key.
Jobs v.s. Tasks

• A MapReduce Job
  – One single “query”, e.g. count the words in all docs
  – More complex queries may consists of multiple jobs

• A Map Task, or a Reduce Task
  – A group of instantiations of the map-, or reduce-function, which are 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

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 (M)

REDUCE Tasks (R)

Shuffle

CSE 344 - 2018au

87
MapReduce Execution Details

Reduce
(Shuffle)
Map

Task
Intermediate data goes to local disk: $M \times R$ files (why?)
Data not necessarily local
File system: GFS or HDFS

Output to disk, replicated in cluster

CSE 344 - 2018au
MapReduce Phases

Map Task

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

Reduce Task

{P 3} Copy → Sort → Reduce

Local storage

HDFS

file
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. E.g.:
  – 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*
Straggler Example

Worker 1

Worker 2

Worker 3

Backup execution

Straggler

Killed
Using MapReduce in Practice:
Implementing RA Operators in MR
Relational Operators in MapReduce

Given relations $R(A, B)$ and $S(B, C)$ compute:

- **Selection:** $\sigma_{A=123}(R)$
- **Group-by:** $\gamma_{A,\text{sum}(B)}(R)$
- **Join:** $R \bowtie S$
Selection $\sigma_{A=123}(R)$

map(Tuple $t$):
  if $t.A = 123$:
    EmitIntermediate($t.A$, $t$);

<table>
<thead>
<tr>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>23</td>
</tr>
<tr>
<td>$t_2$</td>
<td>123</td>
</tr>
<tr>
<td>$t_3$</td>
<td>123</td>
</tr>
<tr>
<td>$t_4$</td>
<td>42</td>
</tr>
</tbody>
</table>

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

(123, [ $t_2$, $t_3$ ])

( $t_2$, $t_3$ )
Selection $\sigma_{A=123}(R)$

(map(String value):
    if value.A = 123:
        EmitIntermediate(value.key, value);
)

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

No need for reduce.
But need system hacking in Hadoop
  to remove reduce from MapReduce
Group By $\gamma_A,\text{sum}(B)(R)$

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>23</td>
<td>10</td>
</tr>
<tr>
<td>t2</td>
<td>123</td>
<td>21</td>
</tr>
<tr>
<td>t3</td>
<td>123</td>
<td>4</td>
</tr>
<tr>
<td>t4</td>
<td>42</td>
<td>6</td>
</tr>
</tbody>
</table>

reduce(String A, Iterator values):

\[
\begin{align*}
    s &= 0 \\
    \text{for each } v \text{ in values:} & \\
    s &= s + v \\
    \text{ Emit}(A, s); & \\
\end{align*}
\]

(23, [ t1 ]), (42, [ t4 ]), (123, [ t2, t3 ])

(23, 10), (42, 6), (123, 25)
Join

Two simple parallel join algorithms:

• Partitioned hash-join (we saw it, will recap)

• Broadcast join
R(A,B) \bowtie_{B=C} S(C,D)

Partitioned Hash-Join

Initially, both R and S are horizontally partitioned

Reshuffle R on R.B and S on S.B

Each server computes the join locally
Partitioned Hash-Join

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

map(Tuple \( t \)):
- case \( t.\text{relationName} \) of
  - ‘R’: EmitIntermediate(t.B, t);
  - ‘S’: EmitIntermediate(t.C, t);

reduce(String \( k \), Iterator \( values \)):
- \( R = \text{empty} \);
- \( S = \text{empty} \);
- for each \( v \) in \( values \):
  - case \( v.\text{relationName} \) of:
    - ‘R’: \( R.\text{insert}(v) \)
    - ‘S’: \( S.\text{insert}(v) \);
  - for \( v_1 \) in \( R \), for \( v_2 \) in \( S \)
    - Emit(\( v_1, v_2 \));

actual tuple:
- \( (5, [ t_1, t_3, t_4 ] ) \)
- \( (6, [ t_2, t_5 ] ) \)

\begin{array}{|c|c|}
\hline
A & B \\
\hline
t_1 & 10 & 5 \\
t_2 & 20 & 6 \\
t_3 & 20 & 5 \\
\hline
\end{array}

\begin{array}{|c|c|}
\hline
C & D \\
\hline
t_4 & 5 & 2 \\
t_5 & 6 & 10 \\
\hline
\end{array}
Broadcast Join

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

Reshuffle R on R.B

Broadcast S

R_1 \rightarrow R_1', S
R_2 \rightarrow R_2', S
\ldots
R_p \rightarrow R_p', S

S
Broadcast Join

map(String value):
  open(S); /* over the network */
  hashTbl = new()
  for each w in S:
    hashTbl.insert(w.C, w)
  close(S);

  for each v in value:
    for each w in hashTbl.find(v.B)
      Emit(v,w);

reduce(…):
  /* empty: map-side only */

R(A,B) ⋈_{B=C} S(C,D)
Summary

• MapReduce: simple abstraction, distribution, fault tolerance
• Load balancing (needed for speedup): by dynamic task allocation; skew still possible
• Writing intermediate results to disk is necessary for fault tolerance, but very slow.
• Spark replaces this with “Resilient Distributed Datasets” = main memory + lineage