Unit 5: Parallel Data Processing

Parallel RDBMS
MapReduce
Spark

(4 lectures)
Introduction to Data Management
CSE 414

Spark
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

Ideal

Speedup

×1

×5

×10

×15

# nodes (=P)
Linear v.s. Non-linear Scaleup

Batch Scaleup

# nodes (=P) AND data size

Ideal

×1  ×5  ×10  ×15
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
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”

```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

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”

```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

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

is the same as:

```
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”

```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

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:

<table>
<thead>
<tr>
<th>Error</th>
<th>Warning</th>
<th>Warning</th>
<th>Error</th>
<th>Aborted</th>
<th>Aborted</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();
Example

The RDDs:

```
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"));
sqllerrors = errors.filter(l->l.contains("sqlite"));
sqllerrors.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"));
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.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();
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect();
```
Persistence

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

Spark can recompute the result from errors
Example

\[
\text{SELECT count(*) FROM R, S WHERE R.B > 200 and S.C < 100 and R.A = S.A}
\]

\[
R = \text{strm.read().textFile("R.csv").map(parseRecord).persist();}
\]

\[
S = \text{strm.read().textFile("S.csv").map(parseRecord).persist();}
\]

Parses each line into an object

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

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

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>Return Type</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>Return Type</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>
Spark 2.0

The DataFrame and Dataset Interfaces
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, 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 414

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:** $R \bowtie_{R.B=S.B} S$
  
  – 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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></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>

**Data:**

**Servers:**

1

2

...
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**:  
  - 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 $v_{i-1} < t.A < v_i$
Data: $R(K,A,B,C)$

- Informally: we say that the data is skewed if one server holds much more data than 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.
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

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

Assuming 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

May be skewed

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 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 $K$

Reshuffle $R$ on attribute $A$

Run grouping on reshuffled partitions
Speedup and Scaleup

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

- 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!
Parallel Execution of RA Operators: Partitioned Hash-Join

- **Data:** $R(K_1, A, B), S(K_2, B, C)$
- **Query:** $R(K_1, A, B) \bowtie_{R.B=S.B} S(K_2, B, C)$
  - Initially, $R$ and $S$ are partitioned on $K_1$ and $K_2$

Each server computes the join locally.
Data: $R(K_1, A, B)$, $S(K_2, B, C)$
Query: $R(K_1, A, B) \bowtie S(K_2, B, C)$

Parallel Join Illustration

Partition

Shuffle on $B$

Local Join
Data: $R(A, B), S(C, D)$
Query: $R(A, B) \bowtie_{B=C} S(C, D)$
Data: $R(A, B), S(C, D)$
Query: $R(A, B) \bowtie_{B=C} S(C, D)$

Broadcast Join

Broadcast S

Keep R in place

$R_1, S$  $R_2, S$  . . .  $R_P, S$
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()

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

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

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

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

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
Example Parallel Query Plan

Node 1

\[ \text{o.item} = \text{i.item} \]

contains all orders and all lines where hash(item) = 1

Node 2

\[ \text{o.item} = \text{i.item} \]

contains all orders and all lines where hash(item) = 2

Node 3

\[ \text{o.item} = \text{i.item} \]

contains all orders and all lines where hash(item) = 3

Order(oid, item, date), Line(item, …)
Summary

- Parallel query evaluation is based on data partitioning
- Main challenge: skew
- When the data is skewed (has “heavy hitter” values) then hash partitioning will lead to uneven load, and poor performance
- Skewed data values must be broadcast, e.g. Broadcast join