Unit 5: Parallel Data Processing

Parallel RDBMS
MapReduce
Spark

(3-4 lectures)
Introduction to Data Management

CSE 344

Parallel DBMS
Announcement

• HW6 is posted

• We use Amazon Web Services (AWS)

• Urgent: please sign up for AWS credits (see instructions on the homework)
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 and Hadoop
• Unit 6: DBMS usability, conceptual design
• Unit 7: Transactions
• Unit 8: Advanced topics (time permitting)
Why compute in parallel?

• 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
  – Recall HW3 and HW6
Performance Metrics for Parallel DBMSs

Nodes = processors, computers

- **Speedup:**
  - More nodes, same data $\rightarrow$ higher speed

- **Scaleup:**
  - More nodes, more data $\rightarrow$ same speed
Linear v.s. Non-linear Speedup

Speedup

×1

×5

×10

×15

# nodes (=P)

Ideal
Linear v.s. Non-linear Scaleup

Batch Scaleup

# nodes (=P) AND data size

Ideal

×1

×5

×10

×15
Why Sub-linear Speedup and Scaleup?

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

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

• **Skew**
  – Slowest node becomes the bottleneck
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
  - last remaining cash cows in the hardware industry
Shared Disk

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

Example: Oracle

- No need to worry about shared memory
- 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: Google

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 many servers

• Operators may require data reshuffling

• First let’s discuss how to distribute data across multiple nodes / servers
Horizontal Data Partitioning

Data:

Servers:

<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>
<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>K</td>
<td>A</td>
<td>B</td>
</tr>
<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>K</td>
<td>A</td>
<td>B</td>
</tr>
<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>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. $\text{size}(R_1) \approx \cdots \approx \text{size}(R_p)$

• **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 $-\infty = v_0 < v_1 < \cdots < v_P = \infty$
  - 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
  • Hash-partition
    – On the key K
    – On the attribute A

  Uniform

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: $\gamma_{A,sum(C)}(R)$
• $R$ is block-partitioned or hash-partitioned on $K$

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

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

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

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 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
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 K1 and K2
Parallel Join Illustration

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, ...)

join

o.item = i.item

select
date = today()
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

join
o.item = i.item
date = today()

Order o
Example Parallel Query Plan

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

Node 1: Contains all orders and all lines where hash(item) = 1

Node 2: Contains all orders and all lines where hash(item) = 2

Node 3: Contains all orders and all lines where hash(item) = 3

join: o.item = i.item
A Challenge

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

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

• \( Q(x,y,z) :\equiv R(x,y), S(y,z), T(z,x) \)
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 \( P^{\frac{1}{3}} \)
  - 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 \((h(x), h(y), *)\)
  - 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 \( P^{1/3} \)
  – 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
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≤P^{\frac{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^{\frac{1}{3}} times
\( Q(x,y,z) = R(x,y), S(y,z), T(z,x) \)

### Partition

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

\[ \text{P1: } (1, 2, 7) \]

<table>
<thead>
<tr>
<th></th>
<th>R2</th>
<th>S2</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

\[ \text{P2: } (1, 2, 3) \]

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

\[ \text{P3: } (3, 2, 3) \]

### Shuffle

### Local Join

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

\[ \text{P1: } (1, 2, 7) \]

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

\[ \text{P2: } (1, 2, 3) \]

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

\[ \text{P3: } (3, 2, 3) \]

### Hypercube join
\( Q(x, y, z) = R(x, y), S(y, z), T(z, x) \)

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

P1

<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>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

P2

<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>

P3

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></th>
<th>( R(x,y) )</th>
<th>( S(y,z) )</th>
<th>( T(z,x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>( \begin{array}{cc} x &amp; y \ 1 &amp; 2 \ 3 &amp; 2 \end{array} )</td>
<td>( \begin{array}{cc} y &amp; z \ 4 &amp; 9 \ 7 &amp; 3 \end{array} )</td>
<td>( \begin{array}{cc} z &amp; x \ 1 &amp; 1 \ 3 &amp; 3 \end{array} )</td>
</tr>
<tr>
<td>P2</td>
<td>( \begin{array}{cc} x &amp; y \ 5 &amp; 4 \ 7 &amp; 6 \end{array} )</td>
<td>( \begin{array}{cc} y &amp; z \ 2 &amp; 3 \ 2 &amp; 9 \end{array} )</td>
<td>( \begin{array}{cc} z &amp; x \ 9 &amp; 5 \ 3 &amp; 1 \end{array} )</td>
</tr>
<tr>
<td>P3</td>
<td>( \begin{array}{cc} x &amp; y \ 8 &amp; 6 \ 9 &amp; 3 \end{array} )</td>
<td>( \begin{array}{cc} y &amp; z \ 6 &amp; 7 \ 6 &amp; 9 \end{array} )</td>
<td>( \begin{array}{cc} z &amp; x \ 7 &amp; 1 \ 3 &amp; 1 \end{array} )</td>
</tr>
</tbody>
</table>

Hypercube join

Shuffle

What if \( h(x): h(1) = h(3) \)

Local Join

P1: (1, 2, 7)
P2: (1, 2, 3)
P3: (3, 2, 3)
Introduction to Data Management
CSE 344

MapReduce
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 \((\text{key}, \text{value})\) pairs

A MapReduce program:
- Input: a bag of \((\text{inputkey}, \text{value})\) pairs
- Output: a bag of \((\text{outputkey}, \text{value})\) 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 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**)

```
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));
```
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

• 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
MAP Tasks (M)

REDUCE Tasks (R)

Shuffle
MapReduce Execution Details

Map (Shuffle)
Reduce

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 - 2017au
MapReduce Phases

Map Task

{P 1} Split → {P 2} Record Reader → Map → Combine → {P 3} Copy → {P 4} Sort → {P 5} Reduce

Reduce Task

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 $\rightarrow$ 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

Killed

CSE 344 - 2017au
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(String value):
  if value.A = 123:
    EmitIntermediate(value.key, value);

reduce(String k, Iterator values):
  for each v in values:
    Emit(v);
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(String value):
    EmitIntermediate(value.A, value.B);

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

Two simple parallel join algorithms:

- Partitioned hash-join (we saw it, will recap)
- Broadcast join
Partitioned Hash-Join

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

Initially, both R and S are horizontally partitioned

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

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

Partitioned Hash-Join

map(String value):
  case value.relationName of
    'R': EmitIntermediate(value.B, ('R', value));
    'S': EmitIntermediate(value.C, ('S', value));

reduce(String k, Iterator values):
  R = empty;  S = empty;
  for each v in values:
    case v.type of:
      'R': R.insert(v)
      'S': S.insert(v);
  for v1 in R, for v2 in S
    Emit(v1,v2);
Broadcast Join

\[ R(A,B) \times_{B=C} S(C,D) \]
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 */
HW6

• HW6 will ask you to write SQL queries and MapReduce tasks using Spark

• You will get to “implement” SQL using MapReduce tasks
  – Can you beat Spark’s implementation?
Conclusions

• MapReduce offers a simple abstraction, and handles distribution + fault tolerance
• Speedup/scaleup achieved by allocating dynamically map tasks and reduce tasks to available server. However, skew is possible (e.g., one huge reduce task)
• Writing intermediate results to disk is necessary for fault tolerance, but very slow.
• Spark replaces this with “Resilient Distributed Datasets” = main memory + lineage
Introduction to Data Management
CSE 344

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

• Difficult to write more complex queries

• Need multiple MapReduce jobs: dramatically slows down because it writes all results to disk
Spark

• Open source system from UC Berkeley
• Distributed processing over HDFS
• Differences from MapReduce:
  – 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
Resilient Distributed Datasets

- **RDD = Resilient Distributed Datasets**
  - A distributed, immutable relation, together with 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
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

What are the benefits of lazy execution?
The RDD Interface
Collections in Spark

• **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
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();
```
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();
```

Transformation: Not executed yet...

Action: triggers execution of entire program

Lines, errors, sqlerrors have type JavaRDD<String>
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 {
    boolean apply (String l)
    { 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”

```python
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
MapReduce Again…

Steps in Spark resemble MapReduce:

- `col.filter(p)` applies in parallel the predicate `p` to all elements `x` of the partitioned collection, and returns collection with those `x` where `p(x) = true`

- `col.map(f)` applies in parallel the function `f` to all elements `x` of the partitioned collection, and returns a new partitioned collection
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();

Persistence

If any server fails before the end, then Spark must restart
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"));
sqllerrors = errors.filter(l->l.contains("sqlite"));
sqllerrors.collect();
```

New RDD

```java
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();
```

Spark can recompute the result from errors
Persistence

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

Spark can recompute the result from errors

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

```
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();
```

RDD:

```
hdfs://logfile.log
filter(...startsWith("ERROR"))
filter(...contains("sqlite"))
result
```

```
hdfs://logfile.log
filter(...startsWith("ERROR"))
errors
filter(...contains("sqlite"))
result
```

New RDD
Example

```scala
R = s.read().textFile("R.csv").map(parseRecord).persist();
S = s.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

Example

R = s.read().textFile("R.csv").map(parseRecord).persist();
S = s.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>Method</th>
<th>Input Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T -&gt; U)</code></td>
<td>RDD&lt;T&gt;</td>
<td>RDD&lt;U&gt;</td>
</tr>
<tr>
<td><code>flatMap(f : T -&gt; Seq(U))</code></td>
<td>RDD&lt;T&gt;</td>
<td>RDD&lt;U&gt;</td>
</tr>
<tr>
<td><code>filter(f : T -&gt; Bool)</code></td>
<td>RDD&lt;T&gt;</td>
<td>RDD&lt;T&gt;</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>RDD&lt;(K,V)&gt;</td>
<td>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;</td>
<td>RDD&lt;(K,V)&gt;</td>
</tr>
<tr>
<td><code>union()</code></td>
<td>(RDD&lt;T&gt;, RDD&lt;T&gt;)</td>
<td>RDD&lt;T&gt;</td>
</tr>
<tr>
<td><code>join()</code></td>
<td>(RDD&lt;(K,V)&gt;, RDD&lt;(K,W)&gt;)</td>
<td>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;)</td>
<td>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;)</td>
<td>RDD&lt;(T,U)&gt;</td>
</tr>
</tbody>
</table>

### Actions:

<table>
<thead>
<tr>
<th>Method</th>
<th>Input Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td>RDD&lt;T&gt;</td>
<td>Long</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>RDD&lt;T&gt;</td>
<td>Seq&lt;T&gt;</td>
</tr>
<tr>
<td><code>reduce(f : (T,T) -&gt; T)</code></td>
<td>RDD&lt;T&gt;</td>
<td>T</td>
</tr>
<tr>
<td><code>save(path : String)</code></td>
<td></td>
<td>Outputs RDD to a storage system e.g., HDFS</td>
</tr>
</tbody>
</table>
### 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&lt;T&gt; -&gt; RDD&lt;U&gt;</code></td>
</tr>
<tr>
<td><code>flatMap(f: T -&gt; Seq(U))</code></td>
<td><code>RDD&lt;T&gt; -&gt; RDD&lt;U&gt;</code></td>
</tr>
<tr>
<td><code>filter(f:T-&gt;Bool)</code></td>
<td><code>RDD&lt;T&gt; -&gt; RDD&lt;T&gt;</code></td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td><code>RDD&lt;(K,V)&gt; -&gt; RDD&lt;(K,Seq[V])&gt;</code></td>
</tr>
<tr>
<td><code>reduceByKey(F:(V,V)-&gt;V)</code></td>
<td><code>RDD&lt;(K,V)&gt; -&gt; RDD&lt;(K,V)&gt;</code></td>
</tr>
<tr>
<td><code>union()</code></td>
<td><code>(RDD&lt;T&gt;,RDD&lt;T&gt;) -&gt; RDD&lt;T&gt;</code></td>
</tr>
<tr>
<td><code>join()</code></td>
<td><code>(RDD&lt;(K,V)&gt;,RDD&lt;(K,W)&gt;) -&gt; RDD&lt;(K,(V,W))&gt;</code></td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td><code>(RDD&lt;(K,V)&gt;,RDD&lt;(K,W)&gt;)-&gt; RDD&lt;(K,(Seq&lt;V&gt;,Seq&lt;W&gt;))&gt;</code></td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td><code>(RDD&lt;T&gt;,RDD&lt;U&gt;) -&gt; RDD&lt;(T,U)&gt;</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&lt;T&gt; -&gt; Long</code></td>
</tr>
<tr>
<td><code>collect()</code></td>
<td><code>RDD&lt;T&gt; -&gt; Seq&lt;T&gt;</code></td>
</tr>
<tr>
<td><code>reduce(f:(T,T)-&gt;T)</code></td>
<td><code>RDD&lt;T&gt; -&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>
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?
An Example Application
PageRank

- Page Rank is an algorithm that assigns to each page a score such that pages have higher scores if more pages with high scores link to them.

- Page Rank was introduced by Google, and, essentially, defined Google.
PageRank toy example

Superstep 0

Superstep 1

Superstep 2

Input graph

http://www.slideshare.net/sscdotopen/large-scale/20
PageRank

for i = 1 to n:
  r[i] = 1/n

repeat
  for j = 1 to n: contribs[j] = 0
  for i = 1 to n:
    k = links[i].length()
    for j in links[i]:
      contribs[j] += r[i] / k
  for i = 1 to n: r[i] = contribs[i]
until convergence
/* usually 10-20 iterations */

Random walk interpretation:

Start at a random node i
At each step, randomly choose an outgoing link and follow it.

Repeat for a very long time

r[i] = prob. that we are at node i
PageRank

for i = 1 to n:
    r[i] = 1/n

repeat
    for j = 1 to n: contrbs[j] = 0
    for i = 1 to n:
        k = links[i].length()
        for j in links[i]:
            contrbs[j] += r[i] / k
    for i = 1 to n: r[i] = contrbs[i]
until convergence
/* usually 10-20 iterations */

Random walk interpretation:

Start at a random node i
At each step, randomly choose an outgoing link and follow it.

Improvement: with small prob. a restart at a random node.

\[ r[i] = a/N + (1-a) \cdot \text{contrbs}[i] \]

where a \( \in (0,1) \) is the restart probability
for i = 1 to n:
    r[i] = 1/n

repeat
    for j = 1 to n: contribs[j] = 0
    for i = 1 to n:
        k = links[i].length()
        for j in links[i]:
            contribs[j] += r[i] / k
    for i = 1 to n: r[i] = a/N + (1-a)*contribs[i]
until convergence
/* usually 10-20 iterations */

// spark

links = spark.read().textFile( .. ).map(...);

ranks = // RDD of (URL, 1/n) pairs

for (k = 1 to ITERATIONS) {

    // Build RDD of (targetURL, float) pairs
    // with contributions sent by each page
    contribs = links.join(ranks).flatMap {
        (url, lr) -> // lr: a (link, rank) pair
            links.map(dest ->
                (dest, lr._2/outlinks.size()))
    }

    // Sum contributions by URL and get new ranks
    ranks = contribs.reduceByKey((x,y) -> x+y)
        .mapValues(sum -> a/n + (1-a)*sum)
}

links: RDD<url:string, outlinks:SEQ<string>>
ranks: RDD<url:string, rank:float>
for $i = 1$ to $n$:
    $r[i] = \frac{1}{n}$

repeat
    for $j = 1$ to $n$: $\text{contribs}[j] = 0$
    for $i = 1$ to $n$:
        $k = \text{links}[i].\text{length()}$
        for $j$ in $\text{links}[i]$:
            $\text{contribs}[j] += \frac{r[i]}{k}$
    for $i = 1$ to $n$: $r[i] = \alpha/N + (1-\alpha)*\text{contribs}[i]$
until convergence
/* usually 10-20 iterations */

// spark

links = spark.read().textFile(..).map(...);
ranks = // RDD of (URL, 1/n) pairs

for ($k = 1$ to ITERATIONS) {

    // Build RDD of (targetURL, float) pairs with contributions sent by each page
    \text{contribs} = \text{links}.join(ranks).flatMap {
        (url, lr) -> // lr: a (link, rank) pair
            \text{links.map}(dest -> (dest, lr._2/outlinks.size()))
    }

    // Sum contributions by URL and get new ranks
    \text{ranks} = \text{contribs.reduceByKey}((x,y) -> x+y)
        .mapValues(sum -> \frac{\alpha}{N} + (1-\alpha)*sum)
}

\text{links}: \text{RDD<url:string, outlinks:SEQ<string>>>}
\text{ranks}: \text{RDD<url:string, rank:float>}

PageRank
for i = 1 to n:
    r[i] = 1/n

repeat
    for j = 1 to n: contribs[j] = 0
    for i = 1 to n:
        k = links[i].length()
        for j in links[i]:
            contribs[j] += r[i] / k
    for i = 1 to n: r[i] = a/N + (1-a)*contribs[i]

until convergence /* usually 10-20 iterations */

// spark

links = spark.read().textFile(..).map(...);
ranks = // RDD of (URL, 1/n) pairs

for (k = 1 to ITERATIONS) {
    // Build RDD of (targetURL, float) pairs
    // with contributions sent by each page
    contribs = links.join(ranks).flatMap {
        (url, lr) -> // lr: a (link, rank) pair
            links.map(dest ->
                (dest, lr._2/outlinks.size()))
    }

    // Sum contributions by URL and get new ranks
    ranks = contribs.reduceByKey(((x,y) -> x+y))
        .mapValues(sum -> a/N + (1-a)*sum)
}

Key: url₁,  
Value: rank₁/outlink₁.size)
Conclusions

- **Parallel databases**
  - Predefined relational operators
  - Optimization
  - Transactions

- **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