Announcements

• HW 6 releases tonight
  – Due Nov. 20th
  – Waiting for AWS credit can take up to two days
  – Sign up early:
    – https://piazza.com/class/jmftm54e88t2kk?cid=452

• Extended office hours Friday to help with first parts of HW 6: 11:30 to 5:00pm in CSE 023
• Extra office hours 5:30pm today on 2nd Floor Breakout
Introduction to Database Systems
CSE 414

Lecture 19: Parallel DBMS
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
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

![Graph showing speedup vs. number of nodes (P)]

Ideal

Speedup

# nodes (P)

×1  ×5  ×10  ×15
Linear v.s. Non-linear Scaleup
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 Mapreduce

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:

<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

CSE 414 - Autumn 2018
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

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 \( i = \text{h}(t.A) \mod P + 1 \)
  - Recall: calling hash fn’s is free in this class compared to disk read time

- **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 $K$
    - Uniform
  - On the attribute $A$
    - May be skewed

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

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 $K$
Speedup and Scaleup

• Consider:
  – Query: \( γ_{A,\text{sum}(C)}(R) \)
  – 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 Data Processing in the 20th Century
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?
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
Sounds familiar after HW5?

A MapReduce program:
• Input: a bag of (inputkey, value) pairs
• Output: a bag of (outputkey, value) pairs
  – outputkey is optional
Step 1: the MAP Phase

User provides the MAP-function:
- Input: \text{(input key, value)}
- Output: bag of \text{(intermediate key, value)}

System applies the map function in parallel to all \text{(input key, value)} pairs in the input file.
Step 2: the REDUCE Phase

User provides the REDUCE function:
• Input: \((\text{intermediate key, bag of values})\)
• Output: bag of output \((\text{values})\)

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));
MAP

\[(\text{did1, v1}) \rightarrow \begin{array}{c}
(\text{w1, 1}) \\
(\text{w2, 1}) \\
(\text{w3, 1}) \\
\vdots \\
(\text{w1, 1}) \\
(\text{w2, 1}) \\
\vdots \\
\vdots \\
\vdots \\
\end{array} \rightarrow \begin{array}{c}
(\text{w1, (1, 1, 1, \ldots, 1)}) \\
(\text{w2, (1, 1, \ldots)}) \\
(\text{w3, (\ldots)}) \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\end{array} \rightarrow \begin{array}{c}
(\text{w1, 25}) \\
(\text{w2, 77}) \\
(\text{w3, 12}) \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\end{array}
\]

REDUCE

\[(\text{did2, v2}) \rightarrow \begin{array}{c}
(\text{w1, 1}) \\
(\text{w2, 1}) \\
(\text{w3, 1}) \\
\vdots \\
(\text{w1, 1}) \\
(\text{w2, 1}) \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\end{array} \rightarrow \begin{array}{c}
(\text{w1, (1, 1, 1, \ldots, 1)}) \\
(\text{w2, (1, 1, \ldots)}) \\
(\text{w3, (\ldots)}) \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\end{array} \rightarrow \begin{array}{c}
(\text{w1, 25}) \\
(\text{w2, 77}) \\
(\text{w3, 12}) \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\end{array}
\]

\[(\text{did3, v3}) \rightarrow \begin{array}{c}
(\text{w1, 1}) \\
(\text{w2, 1}) \\
(\text{w3, 1}) \\
\vdots \\
(\text{w1, 1}) \\
(\text{w2, 1}) \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\end{array} \rightarrow \begin{array}{c}
(\text{w1, (1, 1, 1, \ldots, 1)}) \\
(\text{w2, (1, 1, \ldots)}) \\
(\text{w3, (\ldots)}) \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\end{array} \rightarrow \begin{array}{c}
(\text{w1, 25}) \\
(\text{w2, 77}) \\
(\text{w3, 12}) \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\end{array}
\]

Shuffle

CSE 414 - Autumn 2018
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
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

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
Killed

Time
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);

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

Emit($v$);

<table>
<thead>
<tr>
<th></th>
<th>A</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>

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

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

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

reduce(String A, 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>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>10</td>
</tr>
<tr>
<td>123</td>
<td>21</td>
</tr>
<tr>
<td>123</td>
<td>4</td>
</tr>
<tr>
<td>42</td>
<td>6</td>
</tr>
</tbody>
</table>

reduce(String $A$, Iterator values):

$s = 0$

for each $v$ in values:
$s = s + v$

Emit($A$, $s$);

(23, [ $t_1$ ] ),
(42, [ $t_4$ ] ),
(123, [ $t_2$, $t_3$ ] )

(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

CSE 414 - Autumn 2018 52
Partitioned Hash-Join

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

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

Broadcast Join

Reshuffle $R$ on $R.B$

Broadcast $S$

$\begin{align*}
R_1 \rightarrow R'_1, S \\
R_2 \rightarrow R'_2, S \\
\ldots \\
R_p \rightarrow R'_p, S \\
\end{align*}$

$\begin{align*}
S \\
\end{align*}$
R(A,B) \bowtie_{B=C} S(C,D)

**Broadcast Join**

```java
map(String value):
    readFromNetwork(S); /* over the network */
    hashTable = new HashTable();
    for each w in S:
        hashTable.insert(w.C, w)

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

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

*map* should read several records of R: *value* = some group of tuples from R

Read entire table S, build a Hash Table
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?
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”

```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();
```
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

**lines, errors, sqlerrors have type JavaRDD<String>**
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”

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

RDD:

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

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

```python
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

CSE 414 - Autumn 2018
Example

```
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
Example

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

\[
\begin{align*}
R &= \text{s.read().textFile("R.csv").map(parseRecord).persist();} \\
S &= \text{s.read().textFile("S.csv").map(parseRecord).persist();} \\
RB &= \text{R.filter(t -> t.b > 200).persist();} \\
SC &= \text{S.filter(t -> t.c < 100).persist();} \\
J &= \text{RB.join(SC).persist();} \\
J\text{.count();}
\end{align*}
\]
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><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;</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 <code>RDD</code> 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?**
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