ECE 7650 Scalable and Secure Internet Services and Architecture
---- A Systems Perspective

Part II: Software Infrastructure in Data Centers:

Distributed Execution Engines
MapReduce: Simplified Data Processing on Large Clusters

Motivation

Large-Scale Data Processing

• Want to use 1000s of CPUs
  – But don’t want hassle of managing things

MapReduce provides

• Automatic parallelization & distribution
• Fault tolerance
• I/O scheduling
• Monitoring & status updates

Jeffrey Dean, et al., MapReduce: Simplified Data Processing on Large Clusters in OSDI’04
Map/Reduce

- Map/Reduce
  - Programming model from Lisp
  - (and other functional languages)
- Many problems can be expressed this way
- Easy to distribute across nodes
- Nice retry/failure semantics
Map in Lisp (Scheme)

(map \textit{f} \textit{list} \ [\textit{list}_2 \ \textit{list}_3 \ \ldots])

(map \textit{square} \ '(1 2 3 4))
  - (1 4 9 16)

(reduce + \ '(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 1)))
  - 30
Map/Reduce ala Google

map(key, val) is run on each item in a set
• emits new-key / new-val pairs

reduce(key, vals) is run for each unique key emitted by map()
• emits final output
count words in docs

• Input consists of (url, contents) pairs

• map(key=url, val=contents):
  – For each word w in contents, emit (w, “1”)

• reduce(key=word, values=uniq_counts):
  – Sum all “1”s in values list
  – Emit result “(word, sum)”
Count, Illustrated

map(key=url, val=contents):
  For each word w in contents, emit (w, “1”)

reduce(key=word, values=uniq_counts):
  Sum all “1”s in values list
  Emit result “(word, sum)”

see bob throw
see spot run

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>see</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>bob</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>run</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>see</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>spot</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>throw</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>see</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>run</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>see</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>spot</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>throw</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Pseudo-code of WordCount

Map(String input_key, String input_value):
   //input_key: document name
   //input_value: document contents
   for each word w in input_values:
      EmitIntermediate(w, "1");

Reduce(String key, Iterator intermediate_values):
   //key: a word, same for input and output
   //intermediate_values: a list of counts
   int result = 0;
   for each v in intermediate_values:
      result += ParseInt(v);
   Emit(AsString(result))
Grep

- Input consists of (url+offset, single line)
- map(key=url+offset, val=line):
  - If contents matches regexp, emit (line, “1”)

- reduce(key=line, values=uniq_counts):
  - Don’t do anything; just emit line
Reverse Web-Link Graph

Map
- For each URL linking to target (the link) in a source page …
- Output <target, source> pairs

Reduce
- Concatenate list of all source URLs
- Outputs: <target, list (source)> pairs
More examples

Inverted index

Distributed sort
Implementation Overview

Typical cluster:

- 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
- Limited bisection bandwidth
- Storage is on local IDE disks
- GFS: distributed file system manages data (SOSP'03)
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

Implementation is a C++ library linked into user programs
Execution

How is this distributed?

1. Partition input key/value pairs into chunks, run map() tasks in parallel
2. After all map()s are complete, consolidate all emitted values for each unique emitted key
3. Now partition space of output map keys, and run reduce() in parallel

If map() or reduce() fails, reexecute!
MapReduce: A Bird’s-Eye View

In MapReduce, chunks are processed in isolation by tasks called *Mappers*

The outputs from the mappers are denoted as intermediate outputs (IOs) and are brought into a second set of tasks called *Reducers*

The process of bringing together IOs into a set of Reducers is known as *shuffling process*

The Reducers produce the final outputs (FOs)

Overall, MapReduce breaks the data flow into two phases, *map phase* and *reduce phase*
**Keys and Values**

- The programmer in MapReduce has to specify two functions, the map function and the reduce function that implement the Mapper and the Reducer in a MapReduce program.

- In MapReduce, data elements are always structured as key-value (i.e., (K, V)) pairs.

- The map and reduce functions receive and emit (K, V) pairs.
Partitions

- In MapReduce, intermediate output values are not usually reduced together.

- All values with the same key are presented to a single Reducer together.

- More specifically, a different subset of intermediate key space is assigned to each Reducer.

- These subsets are known as *partitions* using a partitioning function such as \( \text{hash(key)} \mod R \).

*Different colors represent different keys (potentially) from different Mappers.*

*Partitions are the input to Reducers.*
Task Scheduling in MapReduce

- MapReduce adopts a *master-slave architecture*

- The master node in MapReduce is referred to as *Job Tracker* (JT)

- Each slave node in MapReduce is referred to as *Task Tracker* (TT)

- MapReduce adopts a *pull scheduling* strategy rather than a *push one*
  
  - I.e., JT does not push map and reduce tasks to TTs but rather TTs pull them by making pertaining requests
Job Processing

1. Client submits “grep” job, indicating code and input files
2. JobTracker breaks input file into $k$ chunks, (in this case 6). Assigns work to ttrackers.
3. After map(), tasktrackers exchange map-output to build reduce() keyspace
4. JobTracker breaks reduce() keyspace into $m$ chunks (in this case 6). Assigns work.
5. reduce() output may go to GFS/HDFS
Execution

Input

Intermediate

Group by Key

Grouped

Output
Parallel Execution

Map Task 1

Partitioning Function

Sort and Group

Reduce Task 1

Map Task 2

Partitioning Function

Sort and Group

Reduce Task 2

Map Task 3

Partitioning Function
Task Granularity & Pipelining

Fine granularity tasks: map tasks >> machines

- Minimizes time for fault recovery
- Can pipeline shuffling with map execution
- Better dynamic load balancing

Often use 200,000 map & 5000 reduce tasks
Running on 2000 machines
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 00 min 18 sec
323 workers; 0 deaths

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13853</td>
<td>0</td>
<td>323</td>
<td>878934.6</td>
<td>1314.4</td>
<td>717.0</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>0</td>
<td>323</td>
<td>717.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Counters
<table>
<thead>
<tr>
<th>Variable</th>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped (MB/s)</td>
<td>72.5</td>
</tr>
<tr>
<td>Shuffle (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>Output (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>doc-index-hits</td>
<td>145825686</td>
</tr>
<tr>
<td>docs-indexed</td>
<td>506631</td>
</tr>
<tr>
<td>dups-in-index-merge</td>
<td>0</td>
</tr>
<tr>
<td>mr-operator-calls</td>
<td>508192</td>
</tr>
<tr>
<td>mr-operator-outputs</td>
<td>506631</td>
</tr>
</tbody>
</table>
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 05 min 07 sec
1707 workers, 1 deaths

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13853</td>
<td>1857</td>
<td>1707</td>
<td>878934.6</td>
<td>191995.8</td>
<td>113936.6</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>0</td>
<td>500</td>
<td>113936.6</td>
<td>57113.7</td>
<td>57113.7</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>57113.7</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Counters

<table>
<thead>
<tr>
<th>Variable</th>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped (MB/s)</td>
<td>699.1</td>
</tr>
<tr>
<td>Shuffle (MB/s)</td>
<td>349.5</td>
</tr>
<tr>
<td>Output (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>doc-index-hits</td>
<td>5004411944</td>
</tr>
<tr>
<td>docs-indexed</td>
<td>17290135</td>
</tr>
<tr>
<td>dups-in-index-merge</td>
<td>0</td>
</tr>
<tr>
<td>mr-operator-calls</td>
<td>17331371</td>
</tr>
<tr>
<td>mr-operator-outputs</td>
<td>17290135</td>
</tr>
</tbody>
</table>
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 10 min 18 sec

1707 workers; 1 deaths

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13853</td>
<td>5354</td>
<td>1707</td>
<td>878934.6</td>
<td>406020.1</td>
<td>241058.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>0</td>
<td>500</td>
<td>241058.2</td>
<td>196362.5</td>
<td>196362.5</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>196362.5</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

**Counters**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped (MB/s)</td>
<td>704.4</td>
</tr>
<tr>
<td>Shuffle (MB/s)</td>
<td>371.9</td>
</tr>
<tr>
<td>Output (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>doc-index-hits</td>
<td>5000364228</td>
</tr>
<tr>
<td>docs-indexed</td>
<td>17300709</td>
</tr>
<tr>
<td>dups-in-index-merge</td>
<td>0</td>
</tr>
<tr>
<td>mr-operator-calls</td>
<td>17342493</td>
</tr>
<tr>
<td>mr-operator-outputs</td>
<td>17300709</td>
</tr>
</tbody>
</table>
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 15 min 31 sec

1707 workers; 1 deaths

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13853</td>
<td>8841</td>
<td>1707</td>
<td>878934.6</td>
<td>621608.5</td>
<td>369459.8</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>0</td>
<td>500</td>
<td>369459.8</td>
<td>326986.8</td>
<td>326986.8</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>326986.8</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Counters

<table>
<thead>
<tr>
<th>Variable</th>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped (MB/s)</td>
<td>706.5</td>
</tr>
<tr>
<td>Shuffle (MB/s)</td>
<td>419.2</td>
</tr>
<tr>
<td>Output (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>doc-index-hits</td>
<td>4982870667</td>
</tr>
<tr>
<td>docs-indexed</td>
<td>17229926</td>
</tr>
<tr>
<td>dups-in-index-merge</td>
<td>0</td>
</tr>
<tr>
<td>mr-operator-calls</td>
<td>17272056</td>
</tr>
<tr>
<td>mr-operator-outputs</td>
<td>17229926</td>
</tr>
</tbody>
</table>
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 29 min 45 sec
1707 workers; 1 deaths

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13853</td>
<td>13853</td>
<td>0</td>
<td>878934.6</td>
<td>878934.6</td>
<td>523499.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>195</td>
<td>305</td>
<td>523499.2</td>
<td>523389.6</td>
<td>523389.6</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>0</td>
<td>195</td>
<td>523389.6</td>
<td>2685.2</td>
<td>2742.6</td>
</tr>
</tbody>
</table>

Counters

<table>
<thead>
<tr>
<th>Variable</th>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped (MB/s)</td>
<td>0.3</td>
</tr>
<tr>
<td>Shuffle (MB/s)</td>
<td>0.5</td>
</tr>
<tr>
<td>Output (MB/s)</td>
<td>45.7</td>
</tr>
<tr>
<td>docs-indexed</td>
<td>2313178</td>
</tr>
<tr>
<td>docs-indexed</td>
<td>7936</td>
</tr>
<tr>
<td>dups-in-index-merge</td>
<td>0</td>
</tr>
<tr>
<td>mrr-merge-calls</td>
<td>1954105</td>
</tr>
<tr>
<td>mrr-merge-outputs</td>
<td>1954105</td>
</tr>
</tbody>
</table>
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 31 min 34 sec
1707 workers, 1 deaths

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13853</td>
<td>13853</td>
<td>0</td>
<td>878934.6</td>
<td>878934.6</td>
<td>523499.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.2</td>
<td>523499.5</td>
<td>523499.5</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>0</td>
<td>500</td>
<td>523499.5</td>
<td>133837.8</td>
<td>136929.6</td>
</tr>
</tbody>
</table>

Counters

<table>
<thead>
<tr>
<th>Variable</th>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>Shuffle (MB/s)</td>
<td>0.1</td>
</tr>
<tr>
<td>Output (MB/s)</td>
<td>1238.8</td>
</tr>
<tr>
<td>doc-index-hits</td>
<td>0.10</td>
</tr>
<tr>
<td>docs-indexed</td>
<td>0</td>
</tr>
<tr>
<td>dups-in-index-merge</td>
<td>0</td>
</tr>
<tr>
<td>mr-merge-calls</td>
<td>51738599</td>
</tr>
<tr>
<td>mr-merge-outputs</td>
<td>51738599</td>
</tr>
</tbody>
</table>
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 33 min 22 sec
1707 workers; 1 deaths

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13853</td>
<td>13853</td>
<td>0</td>
<td>878934.6</td>
<td>878934.6</td>
<td>523499.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.2</td>
<td>523499.5</td>
<td>523499.5</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>0</td>
<td>500</td>
<td>523499.5</td>
<td>263283.3</td>
<td>269351.2</td>
</tr>
</tbody>
</table>

Counters

<table>
<thead>
<tr>
<th>Variable</th>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>Shuffle (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>Output (MB/s)</td>
<td>1225.1</td>
</tr>
<tr>
<td>doc-index-hits</td>
<td>0</td>
</tr>
<tr>
<td>docs-indexed</td>
<td>0</td>
</tr>
</tbody>
</table>
| dups-in-index-
merge | 0 |
| mr-merge-calls     | 51842100 |
| mr-merge-
outputs | 51842100 |
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 35 min 08 sec
1707 workers; 1 deaths

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13853</td>
<td>13853</td>
<td>0</td>
<td>878934.6</td>
<td>878934.6</td>
<td>523499.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.2</td>
<td>523499.5</td>
<td>523499.5</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>0</td>
<td>500</td>
<td>523499.5</td>
<td>390447.6</td>
<td>399457.2</td>
</tr>
</tbody>
</table>

Counters

<table>
<thead>
<tr>
<th>Variable</th>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>Shuffle (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>Output (MB/s)</td>
<td>1222.0</td>
</tr>
<tr>
<td>doc-index-hits</td>
<td>0.10</td>
</tr>
<tr>
<td>docs-indexed</td>
<td>0.0</td>
</tr>
<tr>
<td>dups-in-indexed</td>
<td>0.0</td>
</tr>
<tr>
<td>mr-merge-calls</td>
<td>51640600</td>
</tr>
<tr>
<td>mr-merge-outputs</td>
<td>51640600</td>
</tr>
</tbody>
</table>
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

- Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 37 min 01 sec
- 1707 workers; 1 deaths

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done (MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13853</td>
<td>13853</td>
<td>0</td>
<td>878934.6</td>
<td>878934.6</td>
<td>523499.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.2</td>
<td>520468.6</td>
<td>520468.6</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>406</td>
<td>94</td>
<td>520468.6</td>
<td>512265.2</td>
<td>514373.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Counter</th>
<th>Variable</th>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped (MB/s)</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>Shuffle (MB/s)</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>Output (MB/s)</td>
<td></td>
<td>849.5</td>
</tr>
<tr>
<td>docs-indexed</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>docs-indexed</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>dups-in-index-merge</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>mr-merge-calls</td>
<td></td>
<td>35083350</td>
</tr>
<tr>
<td>mr-merge-outputs</td>
<td></td>
<td>35083350</td>
</tr>
</tbody>
</table>
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 38 min 56 sec
1707 workers; 1 deaths

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13853</td>
<td>13853</td>
<td>0</td>
<td>878934.6</td>
<td>878934.6</td>
<td>523499.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.2</td>
<td>519781.8</td>
<td>519781.8</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>498</td>
<td>2</td>
<td>519781.8</td>
<td>519394.7</td>
<td>519440.7</td>
</tr>
</tbody>
</table>

Counters

<table>
<thead>
<tr>
<th>Variable</th>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>Shuffle (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>Output (MB/s)</td>
<td>9.4</td>
</tr>
<tr>
<td>doc-index-hits</td>
<td>0</td>
</tr>
<tr>
<td>docs-indexed</td>
<td>0</td>
</tr>
<tr>
<td>dups-in-index-merge</td>
<td>0</td>
</tr>
<tr>
<td>mr-merge-calls</td>
<td>394792</td>
</tr>
<tr>
<td>mr-merge-outputs</td>
<td>394792</td>
</tr>
</tbody>
</table>
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 40 min 43 sec
1707 workers; 1 deaths

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13853</td>
<td>13853</td>
<td>0</td>
<td>878934.6</td>
<td>878934.6</td>
<td>523499.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.2</td>
<td>519774.3</td>
<td>519774.3</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>499</td>
<td>1</td>
<td>519774.3</td>
<td>519735.2</td>
<td>519764.0</td>
</tr>
</tbody>
</table>

Counters

<table>
<thead>
<tr>
<th>Variable</th>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>Shuffle (MB/s)</td>
<td>0.0</td>
</tr>
<tr>
<td>Output (MB/s)</td>
<td>1.9</td>
</tr>
<tr>
<td>doc-index-hits</td>
<td>0 1050</td>
</tr>
<tr>
<td>docs-indexed</td>
<td>0</td>
</tr>
<tr>
<td>dups-in-index-merge</td>
<td>0</td>
</tr>
<tr>
<td>mr-merge-calls</td>
<td>73442</td>
</tr>
<tr>
<td>mr-merge-outputs</td>
<td>73442</td>
</tr>
</tbody>
</table>
Fault Tolerance / Workers

Handled via re-execution

- Detect failure via periodic heartbeats
- Re-execute completed + in-progress *map* tasks
  - The map outputs are on the local disks.
- Re-execute in progress *reduce* tasks
- Task completion committed through master

Robust: lost 1600/1800 machines once \(\rightarrow\) finished ok
Master Failure

Could handle, … ?
But don't yet
  • (master failure unlikely)
Refinement: Redundant Execution

Slow workers significantly delay completion time
  • Other jobs consuming resources on machine
  • Bad disks w/ soft errors transfer data slowly
  • Weird things: processor caches disabled (!!)

Solution: Near end of phase, spawn backup tasks
  • Whichever one finishes first "wins"

Dramatically shortens job completion time
Refinement: Locality Optimization

Master scheduling policy:
  • Asks GFS for locations of replicas of input file blocks
  • Map tasks typically split into 64MB (GFS block size)
  • Map tasks scheduled so GFS input block replica are on same machine or same rack

Effect
  • Thousands of machines read input at local disk speed
    – Without this, rack switches limit read rate
Refinement
Skipping Bad Records

Map/Reduce functions sometimes fail for particular inputs

• Best solution is to debug & fix
  – Not always possible ~ third-party source libraries

• On segmentation fault:
  – Send UDP packet to master from signal handler
  – Include sequence number of record being processed

• If master sees two failures for same record:
  – Next worker is told to skip the record
Other Refinements

- **Sorting guarantees**
  - within each reduce partition
- **Combiner**
  - Useful for saving network bandwidth
- **Local execution for debugging/testing**
- **User-defined counters**
What Makes MapReduce Unique?

MapReduce is characterized by:

1. Its simplified programming model which allows the user to quickly write and test distributed systems

2. Its efficient and automatic distribution of data and workload across machines

3. Its flat scalability curve. Specifically, after a Mapreduce program is written and functioning on 10 nodes, very little-if any- work is required for making that same program run on 1000 nodes
## Comparison With Traditional Models

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Shared Memory</th>
<th>Message Passing</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>Implicit (via loads/stores)</td>
<td>Explicit Messages</td>
<td>Limited and Implicit</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Explicit</td>
<td>Implicit (via messages)</td>
<td>Immutable (K, V) Pairs</td>
</tr>
<tr>
<td>Hardware Support</td>
<td>Typically Required</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Development Effort</td>
<td>Lower</td>
<td>Higher</td>
<td>Lowest</td>
</tr>
<tr>
<td>Tuning Effort</td>
<td>Higher</td>
<td>Lower</td>
<td>Lowest</td>
</tr>
</tbody>
</table>
Summary of MapReduce

MapReduce proven to be useful abstraction

Greatly simplifies large-scale computations

Fun to use:

• focus on problem,
• let library deal w/ messy details
Google was interested in applications that could perform internet-related graph algorithms, such as PageRank, so they designed Pregel to perform these tasks efficiently.

It is a scalable, general-purpose system for implementing graph algorithms in a distributed environment.

Focus on “Thinking Like a Vertex” and parallelism

Grzegorz Greg Malewicz, et al., Pregel: A System for Large-Scale Graph Processing, in SIGMOD’10
What is a Graph?

A graph is a representation of a set of objects (vertices) where some pairs of objects are connected by links (edges).
Introduction

Many practical applications concern large graphs

**Large graph data**
- Web graph
- Transportation routes
- Citation relationships
- Social networks

**Graph Applications**
- PageRank
- Shortest path
- Connected components
- Clustering techniques
PageRank

What's the rank of this user?

Depends on rank of who follows her

Depends on rank of who follows them...

Loops in graph ➔ Iterations!
Introduction

Graph Processing

- Computation is vertex-centric
- Many iterations

Real world graphs are really large!

- the World Wide Web has billions of pages with billions of links
- Facebook’s social graph had more than 700 million users and more than 68 billion friendships in 2011
- Twitter’s social graph has billions of follower relationships
Why not MapReduce?

- Map Reduce is ill-suited for graph processing
  - Program Model is not intuitive: hard to implement
  - Not design for iterations
  - Unnecessarily slow: Each iteration is scheduled as separate MapReduce job with lots of overhead
Example: SSSP – Parallel BFS in MapReduce

Adjacency matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adjacency List

A: (B, 10), (D, 5)
B: (C, 1), (D, 2)
C: (E, 4)
D: (B, 3), (C, 9), (E, 2)
E: (A, 7), (C, 6)
Example: SSSP – Parallel BFS in MapReduce

Map input: <node ID, <dist, adj list>>

- <A, <0, <(B, 10), (D, 5)>>>
- <B, <inf, <(C, 1), (D, 2)>>>
- <C, <inf, <(E, 4)>>>
- <D, <inf, <(B, 3), (C, 9), (E, 2)>>>
- <E, <inf, <(A, 7), (C, 6)>>>

Map output: <dest node ID, dist>

- <B, 10>
- <D, 5>
- <C, inf>
- <D, inf>
- <E, inf>

Flushed to local disk!!
Example: SSSP – Parallel BFS in MapReduce

Reduce input: \( <\text{node ID, dist}> \)

\(<A, <0, (<B, 10), (D, 5)>>>\)
\(<A, \text{inf}>>\)

\(<B, <\text{inf}, (<C, 1), (D, 2)>>>\)
\(<B, 10> <B, \text{inf}>>\)

\(<C, <\text{inf}, (<E, 4)>>>>\)
\(<C, \text{inf}>> <C, \text{inf}>> <C, \text{inf}>>\)

\(<D, <\text{inf}, (<B, 3), (C, 9), (E, 2)>>>>\)
\(<D, 5> <D, \text{inf}>>\)

\(<E, <\text{inf}, (<A, 7), (C, 6)>>>>\)
Example: SSSP – Parallel BFS in MapReduce

Reduce input: \(<\text{node ID}, \text{dist}>\)

- \(<A, 0; \langle B, 10 \rangle, \langle D, 5 \rangle>\)
- \(<B, \infty; \langle C, 1 \rangle, \langle D, 2 \rangle>\)
- \(<B, 10; \infty, \infty>\)
- \(<C, \infty; \langle E, 4 \rangle>\)
- \(<D, \infty; \langle B, 3 \rangle, \langle C, 9 \rangle, \langle E, 2 \rangle>\)
- \(<D, 5; \infty>\)
Example: SSSP – Parallel BFS in MapReduce

Flushed to DFS!!

Flushed to local disk!!
Example: SSSP – Parallel BFS in MapReduce

The diagram illustrates a graph with vertices A, B, C, D, and E. The distances between the vertices are as follows:

- From A to B: 10
- From A to D: 5
- From B to C: 1
- From D to E: 2
- From C to E: 6
- From A to D: 3
- From D to E: 7
- From C to D: 9

The Reduce input includes:

- <node ID, dist>
- <A, 0, <(B, 10), (D, 5)>>
- <A, inf>
- <B, 10, <(C, 1), (D, 2)>>
- <B, 8>
- <C, inf, <(E, 4)>>
- <C, 11, <(E, 7)>>
- <D, inf, <(B, 3), (C, 9), (E, 2)>>
- <D, 5>
- <E, inf, <(A, 7), (C, 6)>>
- <E, 7>
Example: SSSP – Parallel BFS in MapReduce

Reduce input: <node ID, dist>

- \( <A, 0, <(B, 10), (D, 5)> > \)
- \( <B, <2>, <(C, 1), (D, 2)> > \)
- \( <B, 10, <B, 8> > \)
- \( <C, <inf>, <(E, 4)> > \)
- \( <C, 11, <inf> > \)
- \( <D, <5>, <(B, 3), (C, 9), (E, 2)> > \)
- \( <E, <inf>, <(A, 7), (C, 6)> > \)
Example: SSSP – Parallel BFS in MapReduce

Reduce output: <node ID, <dist, adj list>>
  = Map input for next iteration

<A, <0, <(B, 10), (D, 5)>>>
Flushed to DFS!!

<B, <8, <(C, 1), (D, 2)>>>

<C, <11, <(E, 4)>>>

<D, <5, <(B, 3), (C, 9), (E, 2)>>>

<E, <7, <(A, 7), (C, 6)>>>

... the rest omitted ...
Computation Model (1/3)

Input

Supersteps
(a sequence of iterations)

Output
“Think like a vertex”

Inspired by Valiant’s Bulk Synchronous Parallel model (1990)

Source: http://en.wikipedia.org/wiki/Bulk_synchronous_parallel
Computation Model (3/3)

Superstep: the vertices compute in parallel

- Each vertex
  - Receives messages sent in the previous superstep
  - Executes the same user-defined function
  - Modifies its value or its outgoing edges
  - Sends messages to other vertices (to be received in the next superstep)
  - Mutates the topology of the graph
  - Votes to halt if it has no further work to do

- Termination condition
  - All vertices are simultaneously inactive
  - There are no messages in transit
Example: SSSP – Parallel BFS in Pregel
Example: SSSP – Parallel BFS in Pregel
Example: SSSP – Parallel BFS in Pregel
Example: SSSP – Parallel BFS in Pregel
Example: SSSP – Parallel BFS in Pregel

![Graph]

Nodes: 0, 5, 7, 8, 11

Edge Weights:
- 0 to 8: 10
- 0 to 5: 2
- 0 to 7: 5
- 5 to 8: 2
- 5 to 11: 4
- 5 to 7: 2
- 8 to 11: 1
- 7 to 11: 6
- 7 to 5: 2

Graph represents the shortest path problem solution.
Example: SSSP – Parallel BFS in Pregel
Example: SSSP – Parallel BFS in Pregel
Example: SSSP – Parallel BFS in Pregel
Example: SSSP – Parallel BFS in Pregel
Differences from MapReduce

Graph algorithms can be written as a series of chained MapReduce invocation

Pregel
  • Keeps vertices & edges on the machine that performs computation
  • Uses network transfers only for messages

MapReduce
  • Passes the entire state of the graph from one stage to the next
  • Needs to coordinate the steps of a chained MapReduce
Writing a Pregel program

- Subclassing the predefined **Vertex** class

```cpp
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
  public:
    virtual void Compute(MessageIterator* msgs) = 0;

    const string& vertex_id() const;
    int64 superstep() const;

    const VertexValue& GetValue();
    VertexValue* MutableValue();
    OutEdgeIterator GetOutEdgeIterator();

    void SendMessageTo(const string& dest_vertex,
                        const MessageValue& message);
    void VoteToHalt();
};
```

*Override this!*
Example: Vertex Class for SSSP

class ShortestPathVertex
    : public Vertex<int, int, int> {
    void Compute(MessageIterator* msgs) {
        int mindist = IsSource(vertex_id()) ? 0 : INF;
        for (; !msgs->Done(); msgs->Next())
            mindist = min(mindist, msgs->Value());
        if (mindist < GetValue()) {
            *MutableValue() = mindist;
            OutEdgeIterator iter = GetOutEdgeIterator();
            for (; !iter.Done(); iter.Next())
                SendMessageTo(iter.Target(),
                    mindist + iter.GetValue());
        }
    VoteToHalt();
    }
};
System Architecture

Pregel system also uses the master/worker model

- Master
  - Maintains worker
  - Recovers faults of workers
  - Provides Web-UI monitoring tool of job progress
- Worker
  - Processes its task
  - Communicates with the other workers

Persistent data is stored as files on a distributed storage system (such as GFS or BigTable)
Temporary data is stored on local disk
Execution of a Pregel Program

1. Many copies of the program begin executing on a cluster of machines

2. The master assigns a partition of the input to each worker
   • Each worker loads the vertices and marks them as active

3. The master instructs each worker to perform a superstep
   • Each worker loops through its active vertices & computes for each vertex
   • Messages are sent asynchronously, but are delivered before the end of the superstep
   • This step is repeated as long as any vertices are active, or any messages are in transit

4. After the computation halts, the master may instruct each worker to save its portion of the graph
Fault Tolerance

- Checkpointing
  - The master periodically instructs the workers to save the state of their partitions to persistent storage
    - e.g., Vertex values, edge values, incoming messages

- Failure detection
  - Using regular “ping” messages

- Recovery
  - The master reassigns graph partitions to the currently available workers
  - The workers all reload their partition state from most recent available checkpoint
Spark: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

The current frameworks are not efficient in two types of applications:

• Iterative algorithms
  (e.g. iterative machine learning & graph processing)

• More **interactive** ad-hoc queries
  (ad-hoc query: a user customize defined, specific query)
Motivation Discussion

Why?
The important character of iterative algorithm and interactive query application:

Temporal locality and intermediate data reuse.

Interactive query
User always run multiple ad-hoc queries on the same subset of data
Keep data in distributed memory of the cluster is effective and necessary for this character!

iterative algorithms
The same dataset reuse and iterative processing
Current framework weakness:

MapReduce --- Lack efficient primitives for data sharing (slow): Between two Map-Reduce jobs, all the intermediate data would be written into the external stable storage (after reduce phase) such as GFS/HDFS or local disk (after map phase). (each iteration or interactive query is one MapReduce job).

Pregel --- Specific and limited models, not flexible, fault tolerance not efficient: Although it keeps intermediate data in memory, it could only provide the strict BSP computation model, it can’t provide more general data reuse abstractions such as let user load several datasets into memory and run arbitrary / ad-hoc queries.

The efficient fault-tolerance problem would be discussed in Piccolo part. Different tasks has dependency.

Dryad --- Acyclic data flow: More flexible than MapReduce (Because of acyclic model), but do not support iterative computation because of its acyclic work flow and dynamic task creation. And similar to MapReduce, not utilize distributed memory.
Current framework weakness:

CIEL --- Only support MapReduce, : CIEL is designed for iterative algorithms and implemented by a acyclic model, it pushes the iteration scheduler job to the master which would exacerbate the single node failure problem and make master become the bottleneck. And it also doesn’t use distributed memory caching.

Piccolo --- Lack efficient fault-tolerance:

>>For in-memory storage, the fault tolerance problem is more sensitive than the disk-based approach because if the memory fails, then all the data loss.

>>Although Piccolo is a distributed shared memory(DSM) abstraction which maintains in-memory storage, its fault tolerance approach is using expensive checking point which interactive with disk to log each updates of the operation across the cluster because it provides fine-grained updates to mutable state (e.g. cells in a table). When it wants to recover, it needs to rollback and re-execute all the tasks from previous snapshot using checkpointing of the cluster which is not efficient as MapReduce only to restart independent task.

>> Could not perform straggler mitigation like MapReduce.

>>Pregel has the similar not efficient fault tolerance problem because its in-memory storage.
Examples

Slow due to replication and disk I/O, but necessary for fault tolerance
Goal: In-Memory Data Sharing

10-100× faster than network/disk, but how to get FT?
Challenge

How to design a distributed memory abstraction that is both fault-tolerant and efficient?
Solution: Resilient Distributed Datasets (RDDs) Overview

- RDD is an *in-memory abstraction* for cluster computing with efficient fault tolerance. It is not a new framework or data workflow.

- RDD is designed for batch analytics applications rather than asynchronous fine-grained updates shared state applications such as web application storage system.

- For these systems, traditional logging and data checking point such as databases, RAMCloud, Piccolo is more suitable.

- Simple programming interface: Most transformations in Spark less than 20 lines of code.
Solution: Resilient Distributed Datasets (RDDs) Overview

- Restricted form of distributed shared memory for efficient fault tolerance. (borrow from MapReduce, Pregel restrict the programming model for better fault tolerance)
  - Immutable, partitioned collections of records
  - Can only be built through coarse-grained deterministic transformations (map, filter, join, …) to apply the same operation to multiple data items. (like MapReduce --- the same map and reduce function to all of the data sets to achieve good independent task fault tolerance)

- Efficient fault recovery using lineage
  - Log one operation to apply to many data elements. (lineage)
  - Recompute specific lost partitions on failure of its derived RDDs: do not need to re-execute all the tasks. (like MapReduce fault tolerance)
  - No cost if nothing fails (no needs to replicate data)
  - MapReduce and Dryad’s dependency among DAG are lost after each job finishes. (master maintains, and would burden the master’s responsibility)
RDD Recovery

Input

iter. 1

X

iter. 2

X

...
Another significant advantage: Generality of RDDs

- The main purpose of RDD is to utilize a new in-memory storage abstraction to re-express all the existing frameworks such as MapReduce, Pregel, Dryad --- The generality character is very important for RDD.

- Although the coarse-grained transformation seemed restricted, RDD could express many parallel algorithms flexibly because most of the parallel programming model apply the same operation to multiple data items.

- Unify many current programming models
  - *Data flow models*: MapReduce, Dryad, SQL, …
  - *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, …

- Support *new apps* that these models don’t include.

  e.g. User could run a MapReduce operation to build a graph and then run Pregel on it by sharing data through RDD.
Tradeoff Space

Granularity of Updates

Write Throughput

Fine

Coarse

Low

High

Network bandwidth

Memory bandwidth

K-V stores, databases, RAMCloud

K-V stores, databases, RAMCloud

Best for transactional workloads

Best for batch workloads

HDFS

HDFS

RDDs

RDDs

86
RDD implementation --- Spark

RDD abstraction
Spark programming interface
Fault Tolerance
Implementation
RDD Abstraction

1. Immutable --- Read only, owns multiple partitioned collection of records.
   >> RDD could only be created through deterministic operations(transformations) on either data in stable storage or other RDDs.
   >> Purposes : a. consistency for Lineage
       b. straggler mitigation

2. Lineage --- Each RDD has enough information about how it is derived from other data sets to compute its partitions from data in stable storage.

3. Persistence and partitioning --- User could indicate which RDDs they would reuse and store in-memory or disk(not enough memory) in future operation; User could indicate an RDD’s element be partitioned across machines based on a key on record (like MapReduce Reduce phase hash function partition).

4. Two types Operations on RDD : Transformation and Action ---
   >>a. Transformation are a set of specific lazy transformations to create new RDDs.
   >>b. Action launch a computation on the existing RDDs to return a value result to the program or write to external storage.
**RDD Abstraction**

Each RDD through a common interface includes the five pieces of information:

1. A set of partitions, which are atomic pieces of dataset.
2. A set of dependencies on parent RDDs.
3. Metadata about its partitioning scheme and data placement.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>partitions()</td>
<td>Return a list of Partition objects</td>
</tr>
<tr>
<td>preferredLocations(p)</td>
<td>List nodes where partition ( p ) can be accessed faster due to data locality</td>
</tr>
<tr>
<td>dependencies()</td>
<td>Return a list of dependencies</td>
</tr>
<tr>
<td>iterator( (p, parentIters) )</td>
<td>Compute the elements of partition ( p ) given iterators for its parent partitions</td>
</tr>
<tr>
<td>partitioner()</td>
<td>Return metadata specifying whether the RDD is hash/range partitioned</td>
</tr>
</tbody>
</table>

Table 3: Interface used to represent RDDs in Spark.
RDD Abstraction

Narrow dependency: one partition to one partition.
Wide dependency: multiple child partitions depend on one parent partition.

Narrow dependency advantage:
1. pipelined execution. (not like MapReduce wide dependency, there is a barrier between map phase and reduce phase)
2. More efficient fault recovery. (each failed partition only needs one partition re-compute)
Spark Programming Interface

Provides:

- Resilient distributed datasets (RDDs)
- Operations on RDDs: *transformations* (build new RDDs), *actions* (compute and output results)
- Control of each RDD’s *partitioning* (layout across nodes) and *persistence* (storage in RAM, on disk, etc)
**Spark Operations**

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>Map: one-to-one mapping. filter sample groupByKey reduceByKey sortByKey</th>
<th>flatMap: each input value to one or more outputs (like map in MapReduce) union join cogroup cross mapValues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actions (return a result to driver program)</td>
<td>collect reduce count save lookupKey</td>
<td></td>
</tr>
</tbody>
</table>
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split(\t')(2))
messages.persist()

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count

Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)
```
1. Start each page with a rank of 1
2. On each iteration, update each page’s rank to:
\[
\sum_{i \in \text{neighbors}} \frac{\text{rank}_i}{|\text{neighbors}_i|}
\]

```scala
Tlinks = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
    ranks = Tlinks.map((url, links) => (dest, rank))
        .flatMap{
            (dest, rank) => links.map(dest => (dest, rank/links.size))
        }
        .reduceByKey(_ + _)
}
```

Example: PageRank
Optimizing Placement

links & ranks repeatedly joined

Can co-partition them (e.g. hash both on URL) to avoid shuffles. (Make sure rank partition and link partition at the same machine) --- Partition based on data locality

\[
\text{Links} = \text{Links}.\text{partitionBy(new URLPartitioner())}
\]

Only needs to replicate some versions of ranks for fault tolerance rather than replicate large data set

Links in HDFS like MapReduce
PageRank Performance

Time per iteration (s)

- Hadoop: 171
- Basic Spark: 72
- Spark + Controlled Partitioning: 23
How to fault tolerance?

- Efficient Parallel re-compute the lost partition of the RDD. The master node knows the lineage graph and would assign parallel re-computation tasks to other available worker machine.

- Because for each error data set, we only perform the coarse-fined specific transformation filter and map, then if one partition of the message RDD is lost, spark would rebuild it by applying the map transformation on the corresponding partition of lost.

- There are only limited number of transformation operation, so spark only needs to log the name of the operation to each RDD, and then could use this information attached with lineage.

- The immutable character of RDD makes the fault tolerance efficient. Because each RDD is created by coarse-grained transformation, it is not the detail specific operation (Pregel), otherwise we need checking point to record every detail operation for fault tolerance rather than re-computing using lineage.
How to fault tolerance?

- The immutable character also makes RDDs could utilize backup tasks to solve the slow node straggler problem like MapReduce.
  >> DSM like Piccolo could not use backup tasks because the two copies of a task would access the same memory locations and interfere each other’s update.

- RDD could efficiently remember each transformation as one step in a lineage graph and recover lost partitions without having to log large amounts of data. (where the data come from and what coarse-fined operation to these data)
RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data.

E.g.:

```scala
messages = textFile(...).filter(_.contains("error"))
           .map(_.split(\t')(2))
```

**Fault Recovery**

HadoopRDD

FilteredRDD

MappedRDD
Implementation

1. Developer write a driver program that connects to a cluster of the workers.
2. The driver defines one or more RDDs and invokes actions on them.
3. The spark code on the driver also tracks the RDDs’ lineage.
4. The workers are long-lived processes that can store RDD partitions in RAM across operations.
5. The worker which runs the driver program is the master of each job, so each job has its own master to avoid MapReduce single master’s bottleneck.
6. Data partition for each transformation is defined by a Partitioner class in a hash or range partitioned RDD. (during runtime, input data could also based on data locality)

Figure 2: Spark runtime. The user’s driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.
Implementation

7. For each worker in same iteration, they would perform the same operation for different data partition (like Map-Reduce).

One of RDD design purpose is to perform the same operation for different data partition to solve iteration job.

Task Scheduler based on data locality:
1. If a task needs to process a partition that is available in memory on a node, we send it to that node.
2. Otherwise, if a task processes a partition for which the containing RDD provides preferred locations, we send it to those.
Memory Management

1. RDD Data format:
   >> In-memory storage as deserialized Java objects --- Fastest performance. (JVM directly access each RDD element natively)
   >> In-memory storage as serialized data --- Memory-efficient at the cost of lower performance.
   >> On-disk RDD storage ---- When too large keep in memory. The poorest performance.

2. RDD Data reclaim:
   >> When user doesn’t claim, using LRU eviction if the memory do not have enough space.
   >> User could claim the “persistence priority” to each RDD they want to reuse in memory.

3. RDD Data reclaim: Every Spark job has separate memory space.
Memory Management

4. Task resource management within one node:

a. Each task of a job is a thread in one process rather than a process like MapReduce task. (all the tasks of a job would share one executor process)

>> Multi-threads within one Java VM could fast launch task rather than MapReduce’s weakness: slow task launch(launch a new process: 1 second).

>> Multi-threads would shared the memory more sufficiently---important for Spark’s in-memory computation whose memory space is critical for the total performance. (multi-process could not share each processor’s memory)

>> One JVM executor would launch once and used by all the subsequent tasks, not like MapReduce each task would separately apply new resource(process) and release the resource after task finishes : it would reduce the resource request overhead which would significantly reduce the total completion time for the job with a high number of tasks.

RDD In-memory computation is mainly designed for lightweight fast data processing, so the completion time is more critical for Spark rather than MapReduce.
Memory Management

4. Task resource management within one node:

b. Disadvantage of Multi-threads in one JVM:
>> Multi-threads within one JVM would compete for the resource, so it is hard to scheduler fine-grained resource allocation like MapReduce.

MapReduce is designed for batch processing large job, so the resource isolation (multi- process) is critical for large job smooth execution.
Support for Check-pointing

1. Only be used for long-lineage chains and wide dependencies:
   Because one partition lost may cause the full re-computation across the whole cluster: not efficiency.

2. Narrow dependency RDD never needs check-pointing:
   They could be re-computing parallel to recover very efficiently.

3. Spark provides the check-pointing service by letting user define which RDD to check-pointing:
   In the paper, they mention that they are investing how to perform automatic check-pointing because the scheduler knows the size of each dataset as well as the time it took to first computing it, it should be able to select an optimal set of RDDs to checkpoint.
Conclusion

RDDs offer a simple and efficient programming model for a broad range of applications.

RDD provides in-memory computing for data reuse application such as iteration algorithm or interactive ad-hoc query.

Leverage the coarse-grained nature of many parallel algorithms for low-overhead recovery.
Task Scheduler

Dryad-like DAGs
Pipelines functions within a stage
Locality & data reuse aware
Partitioning-aware to avoid shuffles
Breaking Down the Speedup

- Text Input
- Binary Input

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Iteration Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-mem HDFS</td>
<td>15.4</td>
</tr>
<tr>
<td>In-mem local file</td>
<td>8.4</td>
</tr>
<tr>
<td>Spark RDD</td>
<td>6.9</td>
</tr>
<tr>
<td>Spark RDD</td>
<td>2.9</td>
</tr>
</tbody>
</table>

**Notes:**
-iteration times are in seconds.
Scalability

Logistic Regression

<table>
<thead>
<tr>
<th>Number of machines</th>
<th>Hadoop</th>
<th>HadoopBinMem</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>184</td>
<td>116</td>
<td>111</td>
</tr>
<tr>
<td>50</td>
<td>15</td>
<td>80</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>1.3</td>
<td>76</td>
<td>62</td>
</tr>
</tbody>
</table>

K-Means

<table>
<thead>
<tr>
<th>Number of machines</th>
<th>Hadoop</th>
<th>HadoopBinMem</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>274</td>
<td>197</td>
<td>143</td>
</tr>
<tr>
<td>50</td>
<td>157</td>
<td>121</td>
<td>61</td>
</tr>
<tr>
<td>100</td>
<td>87</td>
<td>33</td>
<td>87</td>
</tr>
</tbody>
</table>
Fault Recovery Results

Iteratrion time (s)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>119</td>
</tr>
<tr>
<td>2</td>
<td>57</td>
</tr>
<tr>
<td>3</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
</tr>
<tr>
<td>6</td>
<td>81</td>
</tr>
<tr>
<td>7</td>
<td>57</td>
</tr>
<tr>
<td>8</td>
<td>59</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
</tr>
<tr>
<td>10</td>
<td>59</td>
</tr>
</tbody>
</table>

Failure happens