CS 347
Distributed Databases and Transaction Processing

Distributed Data Processing Using MapReduce

Hector Garcia-Molina
Zoltan Gyongyi
Motivation: Building a Text Index

1. Load data from disk.
2. Tokenize the data to extract words.
3. Sort the tokens.
4. Write the sorted data to disk.

Web page stream

LOADING

[rat: 1]
[dog: 1]
[cat: 2]

TOKENIZING

[dog: 2]
[cat: 2]
[rat: 3]

SORTING

[dog: 3]
[cat: 2]
[rat: 3]

FLUSHING

Intermediate runs

Disk
Motivation: Building a Text Index

Intermediate runs

[ant: 5]
[cat: 4]
[dog: 4]
[dog: 5]
[eel: 6]

MERGE

[ant: 5]
[cat: 2]
[cat: 4]
[dog: 1]
[dog: 2]
[dog: 3]
[dog: 4]
[dog: 5]
[eel: 6]
[rat: 1]
[rat: 3]

Final index

[ant: 5]
[cat: 2,4]
[dog: 1,2,3,4,5]
[eel: 6]
[rat: 1,3]
Generalization: MapReduce

Web page stream

LOADING

MAP

TOKENIZING

SORTING

Disk

FLUSHING

Intermediate runs
Generalization: MapReduce

Intermediate runs

MERGE

REDUCE

Final index
MapReduce

• Input
  – $R = \{ r_1, r_2, \ldots, r_n \}$
  – Functions $M, R$
    – $M(r_i) \rightarrow \{ [k_1: v_1], [k_2: v_2], \ldots \}$
    – $R(k_i, \text{value bag}) \rightarrow \text{“new” value for } k_i$

• Let
  – $S = \{ [k: v] \mid [k: v] \in M(r) \text{ for some } r \in R \}$
  – $K = \{ \{ k \mid [k: v] \in S, \text{ for any } v \} \}$
  – $V(k) = \{ v \mid [k: v] \in S \}$

• Output
  – $O = \{ \{ [k: t] \mid k \in K, t = R(k, V(k)) \} \}$

{\{ ... \} = \text{set} 
{\{ ... \} = \text{bag}
Example: Counting Word Occurrences

- Map(string key, string value):
  // key is the document ID
  // value is the document body
  for each word w in value:
    EmitIntermediate(w, '1')

- Example: Map('29875', 'cat dog cat bat dog') emits [cat: 1], [dog: 1], [cat: 1], [bat: 1], [dog: 1]

- Why does Map() have two parameters?
Example: Counting Word Occurrences

- Reduce(string key, string iterator values):
  // key is a word
  // values is a list of counts
  int result = 0
  for each value v in values:
    result += ParseInteger(v)
  EmitFinal(ToString(result))

- Example: Reduce('dog', {'1', '1', '1', '1'}) emits '4'
Google MapReduce Architecture

Input files → Map phase → Intermediate files (on local disks) → Reduce phase → Output files

- split 0
- split 1
- split 2
- split 3
- split 4

(1) fork
(2) assign map
(3) read
(4) local write
(5) remote read
(6) write

User Program → Master

output file 0 → output file 1
Implementation Issues

- File system
- Data partitioning
- Joined inputs
- Combine functions
- Result ordering
- Failure handling
- Backup tasks
File system

• All data transfer between workers occurs through distributed file system
  – Support for split files
  – Workers perform local writes
  – Each **map** worker performs *local or remote read of one or more* input splits
  – Each **reduce** worker performs *remote read of multiple* intermediate splits
  – Output is left in as many splits as reduce workers
Data partitioning

- Data partitioned (split) by hash on key
- Each worker responsible for certain hash bucket(s)
- How many workers/splits?
  - Best to have multiple splits per worker
    - Improves load balance
    - If worker fails, splits could be re-distributed across multiple other workers
  - Best to assign splits to “nearby” workers
  - Rules apply to both map and reduce workers
Joined inputs

- Two or more map inputs, partitioned (split) by same hash on key
- Map(string \textit{key}, string iterator \textit{values})
  - Each value from one of the inputs
  - For some inputs, value may be empty if key is not present in the input
  - SQL “equivalent”: full outer join on key

<table>
<thead>
<tr>
<th>Input 1</th>
<th>Input 2</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat: 1</td>
<td>ant: 6</td>
<td>Map('ant', { ⊕, '6'})</td>
</tr>
<tr>
<td>cow: 3</td>
<td>cow: 3</td>
<td>Map('cat', { '1', ⊕ })</td>
</tr>
<tr>
<td>eel: 4</td>
<td></td>
<td>Map('cow', { '3', '3'})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Map('eel', { '4', ⊕ })</td>
</tr>
</tbody>
</table>
Combine functions

Combine is like a local reduce applied (at map worker) before storing/distributing intermediate results:
Result ordering

- Results produced by workers are in key order

Input files | Map phase | Intermediate files (on local disks) | Reduce phase | Output files
---|---|---|---|---
split 0 | worker | worker | worker | output file 0
split 1 | | | | output file 1
split 2 | | | | 
split 3 | | | | 
split 4 | | | | 

User Program

Master

(1) fork

(2) assign map

(1) fork

(1) fork

(2) assign reduce

[cat: 2]
[cow: 1]
[dog: 3]

[ant: 2]
[bat: 1]
[cat: 5]
[cow: 7]
Result ordering

• Example: sorting records

Map: emit [k: v]
Reduce: emit v

Input not partitioned by key!

One or two records for 6?
Failure handling

- Worker failure
  - Detected by master through periodic pings
  - Handled via re-execution
    - Redo in-progress or completed map tasks
    - Redo in-progress reduce tasks
    - Map/reduce tasks committed through master

- Master failure
  - Not covered in original implementation
  - Could be detected by user program or monitor
  - Could recover persistent state from disk
Backup tasks

- **Straggler**: worker that takes unusually long to finish task
  - Possible causes include bad disks, network issues, overloaded machines

- Near the end of the map/reduce phase, master spawns backup copies of remaining tasks
  - Use workers that completed their task already
  - Whichever finishes first “wins”
Other Issues

• Handling bad records
  – Best is to debug and fix data/code
  – If master detects at least 2 task failures for a particular input record, skips record during 3rd attempt

• Debugging
  – Tricky in a distributed environment
  – Done through log messages and counters
MapReduce Advantages

• Easy to use
• General enough for expressing many practical problems
• Hides parallelization and fault recovery details
• Scales well, way beyond thousands of machines and terabytes of data
MapReduce Disadvantages

- One-input two-phase data flow rigid, hard to adapt
  - Does not allow for stateful multiple-step processing of records
- Procedural programming model requires (often repetitive) code for even the simplest operations (e.g., projection, filtering)
- Opaque nature of the map and reduce functions impedes optimization
Questions

• Could MapReduce be made more declarative?
• Could we perform (general) joins?
• Could we perform grouping?
  – As done through GROUP BY in SQL
Pig & Pig Latin

• Layer on top of MapReduce (Hadoop)
  – Hadoop is an open-source implementation of MapReduce
  – Pig is the system
  – Pig Latin is the language, a hybrid between
    ▪ A high-level declarative query language, such as SQL
    ▪ A low-level procedural language, such as C++/Java/Python typically used to define Map() and Reduce()
Example: Average score per category

- Input table: pages(url, category, score)
- Problem: find, for each sufficiently large category, the average score of high-score web pages in that category
- SQL solution:

  ```sql
  SELECT category, AVG(score)
  FROM pages
  WHERE score > 0.5
  GROUP BY category HAVING COUNT(*) > 1M
  ```
Example: Average score per category

• SQL solution:

```sql
SELECT category, AVG(score)
FROM pages
WHERE score > 0.5
GROUP BY category HAVING COUNT(*) > 1M
```

• Pig Latin solution:

```pig
topPages = FILTER pages BY score > 0.5;
groups = GROUP topPages BY category;
largeGroups = FILTER groups
    BY COUNT(topPages) > 1M;
output = FOREACH largeGroups
    GENERATE category, AVG(topPages.score);
```
### Example: Average score per category

```plaintext
topPages = FILTER pages BY score >= 0.5;
```

<table>
<thead>
<tr>
<th>pages:</th>
<th>url, category, score</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cnn.com, com, 0.9)</td>
<td></td>
</tr>
<tr>
<td>(yale.edu, edu, 0.5)</td>
<td></td>
</tr>
<tr>
<td>(berkeley.edu, edu, 0.1)</td>
<td></td>
</tr>
<tr>
<td>(nytimes.com, com, 0.8)</td>
<td></td>
</tr>
<tr>
<td>(stanford.edu, edu, 0.6)</td>
<td></td>
</tr>
<tr>
<td>(irs.gov, gov, 0.7)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>topPages:</th>
<th>url, category, score</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cnn.com, com, 0.9)</td>
<td></td>
</tr>
<tr>
<td>(yale.edu, edu, 0.5)</td>
<td></td>
</tr>
<tr>
<td>(nytimes.com, com, 0.8)</td>
<td></td>
</tr>
<tr>
<td>(stanford.edu, edu, 0.6)</td>
<td></td>
</tr>
<tr>
<td>(irs.gov, gov, 0.7)</td>
<td></td>
</tr>
</tbody>
</table>
Example: Average score per category

groups = GROUP topPages BY category;

topPages:
  url, category, score

(cnn.com, com, 0.9)
(yale.edu, edu, 0.5)
(nytimes.com, com, 0.8)
(stanford.edu, edu, 0.6)
(irs.gov, gov, 0.7)

groups:
  category, topPages

(com, {(cnn.com, com, 0.9), (nytimes.com, com, 0.8)})
(edu, {(yale.edu, edu, 0.5), (stanford.edu, edu, 0.6)})
(gov, {(irs.gov, gov, 0.7)})
Example: Average score per category

```plaintext
largeGroups = FILTER groups BY COUNT(topPages) > 1;
```

```plaintext
groups:
category, topPages
   (com,  {(cnn.com, com, 0.9), (nytimes.com, com, 0.8)})
   (edu,  {(yale.edu, edu, 0.5), (stanford.edu, edu, 0.6)})
   (gov,  {(irs.gov, gov, 0.7)})

largeGroups:
category, topPages
   (com,  {(cnn.com, com, 0.9), (nytimes.com, com, 0.8)})
   (edu,  {(yale.edu, edu, 0.5), (stanford.edu, edu, 0.6)})
```
Example: Average score per category

\[
\text{output} = \text{FOREACH largeGroups GENERATE category, AVG(topPages.score)};
\]

largeGroups:
\[
\begin{align*}
\text{category, topPages} \\
(\text{com, } \{(\text{cnn.com, com, 0.9}), (\text{nytimes.com, com, 0.8})\}) \\
(\text{edu, } \{(\text{yale.edu, edu, 0.5}), (\text{stanford.edu, edu, 0.6})\})
\end{align*}
\]

output:
\[
\begin{align*}
\text{category, topPages} \\
(\text{com, 0.85}) \\
(\text{edu, 0.55})
\end{align*}
\]
Pig (Latin) Features

- Similar to specifying a query execution plan (i.e., data flow graph)
  - Makes it easier for programmers to understand and control execution
- Flexible, fully nested data model
- Ability to operate over input files without schema information
- Debugging environment
Execution control: good or bad?

• Example:
  
  spamPages = FILTER pages BY isSpam(url);  
  culpritPages = FILTER spamPages BY score > 0.8;

• Should system reorder filters?  
  – Depends on selectivity
Data model

- **Atom**, e.g., ‘alice’
- **Tuple**, e.g., (‘alice’, ‘lakers’)
- **Bag**, e.g.,
  \[
  \{ (‘alice’, ‘lakers’), (‘alice’, (‘iPod’, ‘apple’)) \}
  \]
- **Map**, e.g.,
  \[
  [ ‘fan of’ \rightarrow \{ (‘lakers’), (‘iPod’) \},
    ‘age’ \rightarrow 20 ]
  \]
## Expressions

Let fields of tuple \( t \) be called \( f_1, f_2, f_3 \)

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Example</th>
<th>Value for ( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>‘bob’</td>
<td>Independent of ( t )</td>
</tr>
<tr>
<td>Field by position</td>
<td>$0$</td>
<td>‘alice’</td>
</tr>
<tr>
<td>Field by name</td>
<td>( f_3 )</td>
<td>( \text{[‘age’} \rightarrow 20 ) )</td>
</tr>
</tbody>
</table>
| Projection              | \( f_2.$0 \)  | \( \{ \text{‘lakers’} \}
|                         |               | \( \text{‘iPod’} \)        | |
| Map Lookup              | \( f_3\#‘age’\) | 20                           |
| Function Evaluation     | \( \text{SUM}(f_2.$1) \) | 1 + 2 = 3                   |
| Conditional Expression  | \( f_3\#‘age’>18? \) \( \text{‘adult’} : \text{‘minor’} \) | ‘adult’                     |
| Flattening              | \( \text{FLATTEN}(f_2) \) | ‘lakers’, 1
|                         |               | ‘iPod’, 2                    |

\[ t = (‘alice’, \{ (‘lakers’, 1) (‘iPod’, 2) \}, [‘age’ \rightarrow 20]) \]
Reading input

queries = LOAD 'query_log.txt'
USING myLoad()
AS (userId, queryString, timestamp);

input file

custom deserializer

schema

handle
For each

expandedQueries = FOREACH queries GENERATE userId, expandQuery(queryString);

• Each tuple is processed independently ↵
  good for parallelism

• Can flatten output to remove one level of nesting:

  expandedQueries = FOREACH queries GENERATE userId, FLATTEN(expandQuery(queryString));
For each

queries:
(userId, queryString, timestamp)

(alice, lakers, 1)
(bob, iPod, 3)

FOREACH queries GENERATE
expandQuery(queryString)
(without flattening)

(a) 
(b)

FOREACH queries GENERATE

(a)
(b)

with flattening

(a)
(b)

(a)
(b)
Flattening example

\[
\begin{array}{ccc}
  x & a & b & c \\
  (a1, \{(b1, b2), (b3, b4), (b5)\}, \{(c1), (c2)\}) \\
  (a2, \{(b6, (b7, b8))\}, \{(c3), (c4)\}) \\
\end{array}
\]

\[
y = \text{FOREACH } x \ \text{GENERATE } a, \ \text{FLATTEN}(b), \ c;
\]
Flattening example

\[
x \quad a \quad b \quad c
\]
\[
(a_1, \{(b_1, b_2), (b_3, b_4), (b_5)\}, \{(c_1), (c_2)\})
\]
\[
(a_2, \{(b_6, (b_7, b_8))\}, \{(c_3), (c_4)\})
\]

\[y = \text{FOREACH } x \ \text{GENERATE } a, \text{ FLATTEN}(b), c;\]

\[
(a_1, b_1, b_2, \{(c_1), (c_2)\})
\]
\[
(a_1, b_3, b_4, \{(c_1), (c_2)\})
\]
\[
(a_1, b_5, ? \{(c_1), (c_2)\})
\]
\[
(a_2, b_6, (b_7, b_8), \{(c_3), (c_4)\})
\]
Flattening example

• Also flattening $c$ (in addition to $b$) yields:
  
  $(a_1, b_1, b_2, c_1)$
  $(a_1, b_1, b_2, c_2)$
  $(a_1, b_3, b_4, c_1)$
  $(a_1, b_3, b_4, c_2)$
  ...

Filter

realQueries = FILTER queries BY userId NEQ 'bot';

realQueries = FILTER queries
              BY NOT isBot(userId);
Co-group

• Two input tables:
  – results(queryString, url, position)
  – revenue(queryString, adSlot, amount)

resultsWithRevenue =
  COGROUP results BY queryString,
  revenue BY queryString;

revenues = FOREACH resultsWithRevenue GENERATE
  FLATTEN(distributeRevenue(results, revenue));

• More flexible than SQL joins
Co-group

results: (queryString, url, rank)
(lakers, nba.com, 1)
(lakers, espn.com, 2)
(kings, nhl.com, 1)
(kings, nba.com, 2)

resultsWithRevenue: (queryString, results, revenue)
(lakers, {lakers, nba.com, 1}, {lakers, top, 50}, {lakers, side, 20})
(lakers, {lakers, espn.com, 2}, {lakers, top, 50}, {lakers, side, 20})
(kings, {kings, nhl.com, 1}, {kings, top, 30}, {kings, side, 10})
(kings, {kings, nba.com, 2}, {kings, top, 30}, {kings, side, 10})

revenue: (queryString, adSlot, amount)
(lakers, top, 50)
(lakers, side, 20)
(kings, top, 30)
(kings, side, 10)

JOIN

distributeRevenue

(lakers, nba.com, 1, top, 50)
(lakers, nba.com, 1, side, 20)
(lakers, espn.com, 2, top, 50)
(lakers, espn.com, 2, side, 20)

...
Group

• Simplified co-group (single input)

  groupedRevenue = GROUP revenue BY queryString;
  queryRevenues = FOREACH groupedRevenue
                  GENERATE queryString,
                  SUM(revenue.amount) AS total;
Co-group example 1

<table>
<thead>
<tr>
<th align="right">x</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td align="right">(1, 1,</td>
<td>c1</td>
<td></td>
<td></td>
</tr>
<tr>
<td align="right">(1, 1,</td>
<td>c2</td>
<td></td>
<td></td>
</tr>
<tr>
<td align="right">(2, 2,</td>
<td>c3</td>
<td></td>
<td></td>
</tr>
<tr>
<td align="right">(2, 2,</td>
<td>c4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th align="right">y</th>
<th>a</th>
<th>b</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td align="right">(1, 1,</td>
<td>d1</td>
<td></td>
<td></td>
</tr>
<tr>
<td align="right">(1, 2,</td>
<td>d2</td>
<td></td>
<td></td>
</tr>
<tr>
<td align="right">(2, 1,</td>
<td>d3</td>
<td></td>
<td></td>
</tr>
<tr>
<td align="right">(2, 2,</td>
<td>d4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ s = \text{GROUP x BY a;} \]
Co-group example 1

\[ x \quad a \quad b \quad c \quad y \quad a \quad b \quad d \]

\[
(1, 1, c1) \\
(1, 1, c2) \\
(2, 2, c3) \\
(2, 2, c4) \\
\]

\[
(1, 1, d1) \\
(1, 2, d2) \\
(2, 1, d3) \\
(2, 2, d4) \\
\]

\[ s = \text{GROUP } x \text{ BY } a; \]

\[ s \quad a \quad x \]

\[
(1, \{(1, 1, c1), (1, 1, c2)\}) \\
(2, \{(2, 2, c3), (2, 2, c4)\}) \\
\]
Group and flatten

s = GROUP x BY a;

\[ s \quad a \quad x \]
\[
(1, \{(1, 1, c1), (1, 1, c2)\}) \\
(2, \{(2, 2, c3), (2, 2, c4)\})
\]

z = FOREACH s GENERATE FLATTEN(x);

\[ z \quad a \quad b \quad c \]
\[
(1, 1, c1) \\
(1, 1, c2) \\
(2, 2, c3) \\
(2, 2, c4)
\]
Co-group example 2

\[
\begin{array}{cccc}
  x & a & b & c \\
  (1, 1, c1) & (1, 1, c2) & (2, 2, c3) & (2, 2, c4) \\
  y & a & b & d \\
  (1, 1, d1) & (1, 2, d2) & (2, 1, d3) & (2, 2, d4)
\end{array}
\]

t = \text{GROUP } x \text{ BY } (a, b);
Co-group example 2

<table>
<thead>
<tr>
<th>x</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>y</th>
<th>a</th>
<th>b</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 1, c1)</td>
<td></td>
<td></td>
<td></td>
<td>(1, 1, d1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1, 1, c2)</td>
<td></td>
<td></td>
<td></td>
<td>(1, 2, d2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2, 2, c3)</td>
<td></td>
<td></td>
<td></td>
<td>(2, 1, d3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2, 2, c4)</td>
<td></td>
<td></td>
<td></td>
<td>(2, 2, d4)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ t = \text{GROUP } x \text{ BY } (a, b); \]

\[ t \quad a/b? \quad x \]

\[ ((1, 1), \{(1, 1, c1), (1, 1, c2)\}) \]
\[ ((2, 2), \{(2, 2, c3), (2, 2, c4)\}) \]
### Co-group example 3

\[
\begin{array}{cccc}
  x & a & b & c \\
  (1, 1, c1) & & & \\
  (1, 1, c2) & & & \\
  (2, 2, c3) & & & \\
  (2, 2, c4) & & & \\
\end{array}
\quad
\begin{array}{cccc}
  y & a & b & d \\
  (1, 1, d1) & & & \\
  (1, 2, d2) & & & \\
  (2, 1, d3) & & & \\
  (2, 2, d4) & & & \\
\end{array}
\]

\[
u = \text{COGROUP } x \text{ BY } a, \ y \text{ BY } a;
\]
Co-group example 3

\[
\begin{array}{cccc}
   x & a & b & c \\
   (1, 1, c1) & (1, 1, d1) \\
   (1, 1, c2) & (1, 2, d2) \\
   (2, 2, c3) & (2, 1, d3) \\
   (2, 2, c4) & (2, 2, d4) \\
\end{array}
\]

\[
u = \text{COGROUP} \ x \ \text{BY} \ a, \ y \ \text{BY} \ a;
\]

\[
u
\begin{array}{cccc}
   x & a & b & c \\
   (1, \{(1, 1, c1), (1, 1, c2)\}, \{(1, 1, d1), (1, 2, d2)\}) \\
   (2, \{(2, 2, c3), (2, 2, c4)\}, \{(2, 1, d3), (2, 2, d4)\}) \\
\end{array}
\]
Co-group example 4

\[
\begin{array}{cccc}
  x & a & b & c \\
  (1, 1, c1) & (1, 1, c2) & (2, 2, c3) & (2, 2, c4) \\
  y & a & b & d \\
  (1, 1, d1) & (1, 2, d2) & (2, 1, d3) & (2, 2, d4) \\
\end{array}
\]

\[v = \text{COGROUP} \ x \ \text{BY} \ a, \ y \ \text{BY} \ b;\]
Co-group example 4

\[
\begin{array}{cccc}
  x & a & b & c \\
  (1, 1, c1) & & & \\
  (1, 1, c2) & & & \\
  (2, 2, c3) & & & \\
  (2, 2, c4) & & & \\
\end{array}
\quad
\begin{array}{cccc}
  y & a & b & d \\
  (1, 1, d1) & & & \\
  (1, 2, d2) & & & \\
  (2, 1, d3) & & & \\
  (2, 2, d4) & & & \\
\end{array}
\]

\[
v = \text{COGROUP } x \text{ BY } a, y \text{ BY } b;
\]

\[
v
\begin{array}{cc}
  a/b? & \text{x} \\
  (1, \{(1, 1, c1), (1, 1, c2)\}, \{(1, 1, d1), (2, 1, d3)\}) & \text{y} \\
  (2, \{(2, 2, c3), (2, 2, c4)\}, \{(1, 2, d2), (2, 2, d4)\})
\end{array}
\]
Join

• Syntax:

\[
\text{joinedResults} = \text{JOIN results BY queryString, revenue BY queryString;}
\]

• Shorthand for:

\[
\text{temp} = \text{COGROUP results BY queryString, revenue BY queryString;}
\]

\[
\text{joinedResults} = \text{FOREACH temp GENERATE FLATTEN(results), FLATTEN(revenue);} 
\]
MapReduce in Pig Latin

mapResult = FOREACH input GENERATE FLATTEN(map(*));
keyGroups = GROUP mapResult BY $0;
output = FOREACH keyGroups GENERATE reduce(*);
Storing output

STORE queryRevenues INTO ‘output.txt’
USING myStore();

custom serializer
Pig on Top of MapReduce

- Pig Latin program can be “compiled” into a sequence of mapreductions
- Load, for each, filter: can be implemented as map functions
- Group, store: can be implemented as reduce functions (given proper intermediate data)
- Cogroup and join: special map functions that handle multiple inputs split using the same hash function
- Depending on sequence of operations, include identity mapper and reducer phases as needed
Hive & HiveQL

- Data warehouse on top of Hadoop
  - Hive is the system
  - HiveQL is the language
    - Fully declarative, SQL-like
    - Most SQL features present
    - Supports custom mapreduce scripts
  - MetaStore system catalog
    - Table schemas and statistics
    - Also for keeping track of underlying distributed file structure
HiveQL Features

- Data model: relations with cells containing
  - Atoms
  - Lists, maps, and structs that may be nested
- Table creation

```sql
CREATE TABLE t(
  x string,
  y list<map<struct<a: string, b:int>>>)
```
- Default serializers and deserializes
- Incorporate “external” data using `SerDe` Java interface
HiveQL Features

• Table updates
  – UPDATE and DELETE not supported
  – INSERT overwrites entire table
    ```java
    INSERT OVERWRITE t SELECT * FROM s
    ```

• Joins
  – Only equality-based SQL joins
    ▪ Equijoin, Cartesian product, left/right/full outer join
  – Explicit syntax
    ```sql
    SELECT t.a AS x, s.b AS y
    FROM t JOIN s ON (t.a = s.b)
    ```
HiveQL Features

• MapReduce scripts
  – Word count example:
    
    FROM (  
      FROM docs  
      MAP text USING `python wc_map.py` AS (word, count)  
      CLUSTER BY word  
    ) temp  
    REDUCE word, count USING `python wc_reduce.py`  

  – May have SELECT instead of MAP or REDUCE
    ▪ Order of FROM and SELECT/MAP/REDUCE keywords is interchangeable
Hive Query Processing

1. Parsing: query $\rightarrow$ abstract syntax tree (AST)
2. Type checking and semantic analysis: AST $\rightarrow$ query block tree (QBT)
3. Optimization: QBT $\rightarrow$ optimized operator DAG
   - Map operators: TableScan, Filter, ReduceSink
   - Reduce operators: GroupBy, Join, Select, FileSink
   - Extensible optimization logic (set of heuristics) through \textit{Transform} Java interface
   - No explicit notion of cost or search of plan space
4. Generation of physical plan: operator DAG $\rightarrow$ Hadoop mapreduce sequence
Optimization heuristics

- **Column pruning**: add projections to remove unneeded columns
- **Partition pruning**: eliminate unneeded file partitions (splits)
- **Predicate pushdown**: early filtering of input records
- **Map-side joins**: hash-join in mapper if one table is small
  - Triggered explicitly by HiveQL hint

```sql
SELECT /*+ MAPJOIN(t) */ t.a, s.b ...
```
**Sawzall**

- Procedural scripting language and interpreter on top of MapReduce
- Simplifies the formulation of one mapreduction
  - Record-oriented processing
- Word count example:

```plaintext
wc_table: table sum[word: string] of count: int;
most_freq: table top(100) of word: string;

words: array of string = tokenize(input);
for (i: int = 0; i < len(words); i++) {
    emit wc_table[words[i]] ← 1;
    emit most_freq ← words[i];
}
```
Sawzall

• Mapper executes script body for each input record

• Reducer generates table outputs by aggregating data emitted by mapper
  – Table (aggregation) types
    ▪ sum, maximum
    ▪ collection: bag of values
    ▪ unique(n): set of values (of max size n for efficiency)
    ▪ sample(n): random sample of size n
    ▪ top(n): n most frequent values
    ▪ quantile(n): quantile thresholds (e.g., percentile for n=100)
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Sawzall</th>
<th>Pig</th>
<th>Hive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
<td>Procedural</td>
<td>Semi-declarative</td>
<td>Declarative</td>
</tr>
<tr>
<td><strong>Schemas</strong></td>
<td>Yes*</td>
<td>Yes (implicit)</td>
<td>Yes (explicit)</td>
</tr>
<tr>
<td><strong>Nesting</strong></td>
<td>Containers</td>
<td>Containers</td>
<td>Full</td>
</tr>
<tr>
<td><strong>User-defined functions</strong></td>
<td>No</td>
<td>Yes (Java)</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Custom serialization/deserialization</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Joins</strong></td>
<td>No</td>
<td>Yes+</td>
<td>Yes (equality)</td>
</tr>
<tr>
<td><strong>MapReduce steps</strong></td>
<td>Single</td>
<td>Multiple</td>
<td>Multiple</td>
</tr>
</tbody>
</table>

* Through *protocol buffers*, i.e., complex data type declaration
References

• MapReduce: Simplified Data Processing on Large Clusters (Dean and Ghemawat)
  http://labs.google.com/papers/mapreduce.html

• Pig Latin: A Not-so-foreign Language for Data Processing (Olston et al.)
  http://wiki.apache.org/pig/

• Hive: A Petabyte Scale Data Warehouse Using Hadoop (Thusoo et al.)

• Interpreting the Data: Parallel Analysis with Sawzall (Pike et al.)
  http://labs.google.com/papers/sawzall.html
Summary

• MapReduce
  – Two phases: map and reduce
  – Transparent distribution, fault tolerance, and scaling

• Sawzall, Pig, and Hive
  – Various layers on top of MapReduce
  – Procedural, semi-declarative, and declarative “query” languages