MapReduce Design Patterns

Kostas Solomos
Computer Science Department
University of Crete, Greece
Design Patterns

“Tools for solving problems in a reusable and general way so that the developer can spend less time figuring out how he’s going to overcome a hurdle and move onto the next one”

- Donald Miner and Adam Shook “MapReduce Design Patterns”
Patterns

Common:
- Counting
- Parsing
- Filtering (distributed “grep”)
- Binning
- Distributed tasks
- Chained jobs

Advanced:
- Grouping
- Distinct
- Secondary sorting
- Distributed global sorting
- Joins
Local Aggregation

- Moving data from Mappers to Reducers is costly!
  - data transfer
  - disk I/O

- Local aggregation reduces the amount of data needed to move

- Most popular technique: In-mapper combining
WordCount revisited (naïve)

- **map(key, value):**
  
  for each word:
  
  emit (word, 1);

- **reduce(word, counts[1,1,...,1])**
  
  sum = 0;
  
  for each c in counts:
    
    sum += c;
  
  emit(word, sum);
WordCount revisited (combiner)

- **map(key, value):**
  - for each word:
    - emit (word, 1);

- **reduce(word, counts[c1,c2,…]):**
  - sum = 0;
  - for each c in counts:
    - sum += c;
  - emit(word, sum);

- **combine(word, counts[c1,c2,…]):**
  - sum = 0;
  - for each c in counts:
    - sum += c;
  - emit(word, sum);

```java
job.setCombiner(WordCountReducer.class);
job.setReducer(WordCountReducer.class);
```
WordCount revisited
(in-mapper local aggregation)

- **map(key, value):**
  - for each word:
    - localCounts.put(word, localCounts.get(word) + 1); //a Map

- **cleanup():** // runs after map() is finished
  - for each word in localCounts:
    - emit(word, localCounts.get(word));

- **reduce(word, counts[1,1,...,1])**
  - sum = 0;
  - for each c in counts:
    - sum += c;
  - emit(word, sum);
Wrong use of a combiner: the mean

- map(object o, int v):
  emit (o, v);    //identity mapper*

- reduce(object o, values[v1,v2,...]):
  sum = 0;
  for each v in values:
    sum += v;
  emit(o, sum/|values|);
  *job.setMapper(Mapper.class);
  job.setCombiner(MeanReducer.class);
  Incorrect (mean != mean of means)
Wrong use of a combiner: the mean

- map(object o, int v):
  emit (o, v);

- combine(object o, values[\{v1,v2,\ldots\}]):
  sum = 0;
  for each v in values:
    sum += v;
  emit(o, pair(sum, |values|));

- reduce(object o, pairs[((v1,c1), (v2,c2), \ldots)]):
  sum = 0;  count = 0;
  for each pair (value,cnt) in pairs:
    sum += value;
    count += cnt;
  emit(o, sum/count);

Incorrect
Correct use of a combiner: the mean

- **map(object o, int v):**
  - emit (o, pair(v,1));

- **combine(object o, pairs[(v1,c1),(v2,c2),...]):**
  - sum = 0; count = 0;
  - for each pair (value,cnt) in pairs:
    - sum += value;
    - count += cnt;
  - emit(o, pair(sum, count));

- **reduce(object o, pairs[(v1,c1), (v2,c2), ...]):**
  - sum = 0; count = 0;
  - for each pair (value,cnt) in pairs:
    - sum += value;
    - count += cnt;
  - emit(o, sum/count);
Alternative Solution: In-mapper aggregation

- map(object o, int v):
  objects.put(o, objects.get(o) + v);
  counts.put(o, counts.get(o) + 1);

- cleanup():
  for each object o:
  emit(o, pair(objects.get(o), counts.get(o)));

- reduce(object o, pairs[(v1,c1), (v2,c2), ...]):
  sum = 0; count = 0;
  for each pair (value,cnt) in pairs:
    sum += value;
    count += cnt;
  emit(o, sum/count);
Local Aggregation vs Combiner

Pros:
- Controls when aggregation takes place
- More efficient (no disk spills and object creation&destruction overhead)

Cons:
- May run out of memory
- Algorithmic behaviour may depend on the order of input keys/values
Pairs And Stripes
(Word co-occurrence problem)
doc1.txt: “I have a dog”
doc2.txt: “I have a cat”
doc3.txt: “I have no pets”

Co-occurrence matrix:

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>have</th>
<th>a</th>
<th>dog</th>
<th>cat</th>
<th>no</th>
<th>pets</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>-</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>have</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dog</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td>-</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>-</td>
<td></td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pets</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pairs

- map(docid d, text s):
  - for each word x in s:
    - for each word y in s:
      - emit(pair(x,y), 1);
  - reduce(pair p, counts[1,1,...,1]):
    - sum = 0;
    - for each c in counts:
      - sum += counts;
    - emit (p, sum);  //one cell in the co-occurrence matrix

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>have</th>
<th>a</th>
<th>dog</th>
<th>cat</th>
<th>no</th>
<th>pets</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>-</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>have</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>-</td>
<td></td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pets</td>
<td>-</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stripes

- **map**(docid d, text s):
  for each word x in s:
    for each word y in s:
      stripe\{x\}.put(y, stripe\{x\}.get(y) + 1);
      emit(x, stripe\{x\});

- **reduce**(word w, stripes[s1{w}, s2{w}, ...]):
  stripe\{w\} ← zeros
  for each s{w} in stripes:
    stripe\{w\} ← stripe\{w\} + s{w};
  emit (w, stripe\{w\});  //one row in the co-occurrence matrix

<table>
<thead>
<tr>
<th>i</th>
<th>have</th>
<th>a</th>
<th>dog</th>
<th>cat</th>
<th>no</th>
<th>pets</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>-</td>
<td>3</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>have</td>
<td>-</td>
<td>2</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>-</td>
<td>1</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dog</td>
<td>-</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cat</td>
<td>-</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>-</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>
Pairs vs Stripes

- Pairs generates too much intermediate data
- Stripes generates fewer and shorter intermediate keys
  - less sorting needed
- Stripes generated more complex values
  - more (de-)serialization overhead
- Both can benefit from a Combiner
  - Stripes are more likely to benefit more

Main-memory limitation:
- each stripe is small enough to fit in main-memory
- the pairs approach does not suffer from this limitation
Pairs vs Stripes

from Jimmy Lin’s and Chris Dyer’s “Data-Intensive Text Processing with MapReduce” book

1 hour
2.6 billion intermediate key/value pairs;
after combiner run: 1.1B
final key/value pairs: 142M

11 minutes
653M intermediate key/value pairs;
after combiner run: 29M
final: 1.69M rows
Pairs vs Stripes

from Jimmy Lin’s and Chris Dyer’s “Data-Intensive Text Processing with MapReduce” book
Using the Counters

- Use the Counters when:
  - you need to count or summarize large data sets
  - the number of counters you will need is small (<< 100)

- Design Pattern:
  - A single map job (no combiner/partitioner/reducer)*
  - The mapper increments counter(s) by one or more, based on certain criteria
  - No actual output, just the aggregated value of the counter(s)

- Applications (usually as a preprocessing job):
  - Count records
  - Count a specific subset of the input
  - Summations

*job.setNumReduceTasks(0)
Using the Counters - Examples

- map(document d, text t):
  for each word w of t:
    dCounter.increment(1);
  //keeps a Counter for each distinct input document and counts the #words per doc

- map(document d, text t):
  for each word w of t:
    if (w in stopwords):
      stopWordsCounter.increment(1);
  //counts how many times a stopword appears in the corpus
Filtering Patterns

- Identity a specific subset of the input
  - e.g. top K, deduplication, searching, sampling

- Filtering
  - decide if an input record should be returned or not

- Top 10
  - keep the top 10 of the input records

- Distinct
  - keep all the distinct records
Filtering – Distributed Grep

- setup(Context context):
  String regex = context.getConfiguration().get("regex");

- map(docId d, text t):
  if (t.matches(regex))
    emit (d, t);
Filtering - Threshold

- **setup(Context context):**
  
  ```java
  Double t = context.getConfiguration().get("thresh");
  ```

- **map(object o, double value):**
  
  ```java
  if (value > t)
      emit (o, value);
  ```

  E.g. 
  
  thresh = mean of all values found in a previous job
Top 10 - Workflow

*job.setNumReduceTasks(1);
Top 10 - Mapper

- **setup:**
  
  ```
  localTop10 ← initialize as sorted list
  ```

- **map(object o, value v):**
  
  ```
  for each value v:
  localTop10.add(v);
  if (localTop10.size() > 10): //always keep up to 10 elements
    localTop10.removeLast(); //removes the lowest element(11th)
  ```

- **cleanup:**
  
  ```
  for each value v in localTop10:
    emit(null, v); //emit the local top 10 (after map)
  ```
Top 10 - Reducer

- **setup():**
  
  ```java
  globalTop10 ← initialize as sorted list
  ```

- **reduce():**  //use a single reducer
  
  for each value v:
  ```java
  globalTop10.add(v);
  if (globalTop10.size() > 10):  //always keep up to 10 elements
      globalTop10.removeLast();  //removes the lowest element (11th)
  ```

- **cleanup:**
  
  for each value v in globalTop10:
  ```java
  emit(null, v);  //emit the global top 10 values
  ```
Distinct

Given a dataset of records, output the set of unique records/unique values of the records for a specific criterion

Similar to the SQL query:
SELECT DISTINCT personId FROM persons;
Distinct

- map(object o, value v):
  id = v.extractPersonId();  //or any other field
  emit (id, null);

- reduce(id, [null...null]):
  emit (id, null);
Data Organization Patterns

- Structured to hierarchical
  - creates new records from data of a very different structure

- Partitioning
  - moves the records into categories (i.e., partitions) not really caring about the order of records
  - using a Partitioner

- Binning
  - same as partitioning, but without using a Reducer & Partitioner

- Shuffling
  - randomizes the order of the input data
Structured to hierarchical

Example:
- **Input1**: a set of posts
- **Input2**: a set of comments, each related to a post id
- **Output**: an XML document with the hierarchy of posts and comments.

Pattern (join):
- Use `MultipleInputs`, assigning a different mapper to each input type
- emit as intermediate output key the root id (here: post id)
<?xml version="1.0" encoding="UTF-8"?>
<posts>

<post>
</post>

<post>
</post>

<post>
</post>

<XML (part)>
</XML (part)>

<XML (part)>
</XML (part)>

<XML (part)>
</XML (part)>

</posts>
Structural to XML

- **mapPost**(int postId, text post):
  
etit (postId, “P”+post);

- **reduce**(int postId, values[v1,v2,...]):
  
  ```java
  String post;
  List<String> comments;
  for each v in values:
      if v.startsWith(“P”):
          post = v.substring(1);
      else
          comments.add(v.substring(1));
  String XMLdoc = nest(post, comments); //creates hierarchy
  emit(XMLdoc, null);
  ```

- **mapComment**(int postId, text post):
  
etit (postId, “C”+post);
Partitioning

- Divide input records into partitions
- The number of required partitions is known
  - otherwise, run a pre-processing job to count the desired partitions

Example:
Partition a given set of NBA players, based on their position (1-5)
Partitioning

NBA Players -> Mapper -> Partitioner -> Reducer

NBA Players -> Mapper -> Partitioner -> Reducer

NBA Players -> Mapper -> Partitioner -> Reducer

Reducer -> Point Guards

Reducer -> Shooting Guards

Reducer -> Small Forwards

Reducer -> Power Forwards

Reducer -> Centers
Partitioning

- run():
  ```java
  job.setPartitioner(MyPartitioner.class);
  job.setNumReduceTasks(5); //one for each position
  ```

- map(String playerName, int positionId):
  ```java
  emit (positionId-1, playerName); //counting starts from 0...
  ```

- getPartition(int key, String playerName, int numPartitions):
  ```java
  return key;
  ```

- reduce(positionId, playerName):
  ```java
  emit (playerName, null);
  ```
Binning

- **Same task with Partitioning, but:**
  - data grouping takes place within each mapper
    - no partitioner
    - no reducer

- **Uses MultipleOutputs**
  - one output for each category

*job.setNumReduceTasks(0)*
additional merging is required (e.g., concat PG-0, PG-1, PG-2)
Binning

- `map(String playerName, int positionId):
  switch(positionId):
    case 1:  mos.write("PG", playerName, null);
    case 2:  mos.write("SG", playerName, null);
    case 3:  mos.write("SF", playerName, null);
    case 4:  mos.write("PF", playerName, null);
    case 5:  mos.write("C", playerName, null);

(mos is an instance of MultipleOutputs)
Binning vs Partitioning

- **Binning:**
  - eliminates the need for shuffling and sorting (more efficient)
  - does not require a custom Partitioner (easier to implement)
  - each mapper creates a file for each bin
    - if 1K bins are needed and 1K mappers are run → 1M files are created
    - bad for namenode

- **Partitioning:**
  - requires shuffling and sorting
  - requires to implement a custom Partitioner
  - one output file per category
Shuffling

- Return the input data in a random order

Application examples:
- Data anonymization (to protect privacy)
- Random sampling (to use a subset of the input data for experiments)

Use case:
  Given a log of emergency events in a hospital, anonymize each record, removing the patient’s name and randomly shuffling the records of the log
Shuffling

- map(int recordId, String record):
  record = record.remove(patientName);
  emit (randomInt, record);

- reduce(int Id, String record):
  emit (record, null);
Graph algorithms

● A graph can be represented as:
  ◦ an adjacency matrix (N x N)
    ● each cell $e_{ij}$ denoting
      ● the existence of an edge between nodes $n_i$ and $n_j$ (with values 0 or 1), or
      ● the weight of the edge connecting nodes $n_i$ and $n_j$
  ◦ adjacency lists
    ● each node is associated with neighbors, reachable through (outgoing) edges

● An adjacency matrix representation is not efficient
  ◦ sparsity (most cells are zeros) $\rightarrow$ too much space needed
Breadth-first search
Single-source shortest path (SSSP) (Dijkstra’s algorithm)

image from https://www.cs.indiana.edu/~achauhan/Teaching/B403/LectureNotes/10-graphalgo.html
Parallel SSSP

Each node is represented as:

```
nodeId  adjacency_list|distance_from_the_source|color|parentNode
```

Assume all edges have the same weight (1)

Sample Input:

1 2,3|0|GRAY|source
2 1,3,4,5|∞|WHITE|null
3 1,4,2|∞|WHITE|null
4 2,3|∞|WHITE|null
5 2|∞|WHITE|null
Parallel SSSP (cont’d)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2,3</td>
<td>0</td>
<td>BLACK</td>
<td>source</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1,3,4,5</td>
<td>1</td>
<td>GRAY</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1,4,2</td>
<td>1</td>
<td>GRAY</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2,3</td>
<td>∞</td>
<td>WHITE</td>
<td>null</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>∞</td>
<td>WHITE</td>
<td>null</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 2,3|0|BLACK|source
2 1,3,4,5|1|BLACK|1
3 1,4,2|1|BLACK|1
4 2,3|2|GRAY|2
5 2|2|GRAY|2

1 2,3|0|BLACK|source
2 1,3,4,5|1|BLACK|1
3 1,4,2|1|BLACK|1
4 2,3|2|BLACK|2
5 2|2|BLACK|2
Parallel SSSP - Mapper

- map(int nId, Node n):
  
  if n.color == GRAY:
    for each node m in n.adjacency_list:
      m.distance = n.distance + 1;
      m.color = GRAY;
      m.parent = nId;
      emit(m.id, m); //update all neighbors
  
  n.color = BLACK;
  emit(nId, n); //GRAY→ BLACK, otherwise no change
Parallel SSSP - Reducer

- **reduce(int nId, neighbors [m1, m2, ...]):**
  
  Node n = new Node(nId);
  for each m in neighbors:
    if (m.adjacency_list != empty):
      n.adjacency_list.add(m.adjacency_list);
      if (m.distance < n.distance):
        n.distance = m.distance;
        n.parent = m.parent;
    if (m.color > n.color):
      n.color = m.color;
    emit(nId, n);
Resources

- “MapReduce Design Patters” book, by Donald Miner and Adam Shook
- “Data-Intensive Text Processing with MapReduce” book, by Jimmy Lin and Chris Dyer
- http://hadooptutorial.wikispaces.com/Iterative+MapReduce+and+Counters
- On-line code examples: