Anti-Combining for MapReduce

Alper Okcan    Mirek Riedewald
Northeastern University, Boston, USA
SIGMOD 2014
MapReduce Overview

Shuffle

Input HDFS

- Split 0
- Split 1
- Split 2

Map

Sort

Copy

Merge

Reduce

Output HDFS

- Part 0
- Part 1

HDFS Replication
Shuffle is always the bottleneck of a MR job execution

- large amounts of data are grouped, sorted, and moved across the network
- data transfer is inherent to enable parallel execution
- Network links and switches are in fact limited

Reducing network load is essential for increasing throughput in highly utilized environments
## The methods of reducing network load

<table>
<thead>
<tr>
<th>Methods</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combiner</td>
<td>User-defined</td>
<td>Limitation to applications; Efficiency;</td>
</tr>
<tr>
<td>Compression</td>
<td>The reduction of data</td>
<td>Overheads for compression and decompression;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Same pattern;</td>
</tr>
<tr>
<td>Active Storage</td>
<td>Make good use of devices’</td>
<td>complex</td>
</tr>
<tr>
<td></td>
<td>computation</td>
<td></td>
</tr>
</tbody>
</table>
Motivation

• Two design goals
  – Simple encoding/decoding functions
    • CPU cost
    • buffer space
  – Fine-grained adaptive optimization
    • the choice of encode
      – driven by the data
The Query-Suggestion problem

We are given a log of search queries. For any string \( P \) that occurred as a prefix of some query in the log, pre-compute the five most frequent queries in the log starting with prefix \( P \).
The Problem of current Query Suggestion execution

• For a search query of length \( n \)
  – the Map function will generate \( n \) output records
  – each Map function call's output is quadratic in its input size
    • each output record contains the query itself

• Combiner
  – The large number of distinct query strings
  – a Combiner might not be applicable at all
Eager Sharing Strategy

- **EagerSH Map Phase**

  First executes the original Map on the given input record.

  Then groups the original Map's output by value and partition number.

  For each group, a single record is emitted.

---

**Algorithm 1: EagerSH’s Map Function**

**Input:** input tuple $I$

1. $\text{MapOutput} = O\text{-map}(I)$ /* Original map */
2. $\text{result} = \text{SELECT MIN}(O.\text{key}) \text{ AS key,}
   (\text{setOfOtherKeysInGroup}(), O.\text{value}) \text{ AS value FROM MapOutput} O$
   GROUP BY getPartition($O.\text{key}$), $O.\text{value}$
3. for all $r \in \text{result}$ do
4. $\text{Emit}(r.\text{key}, r.\text{value})$
Eager Sharing Strategy

- **EagerSH Reduce Phase**

  EagerSH's Reduce only receives three encoded records, in this case all those with key "m", EagerSH's Reduce scans through all records with that key and inserts into Shared the corresponding key/value combinations for all keys encoded in the value component.

  reduce task 1 receives all the records with the keys assigned to it by the Partitioner, in key order. It then calls Reduce three times, first for key "m", followed by "man", and finally "mango" in the example.
Algorithm 2: EagerSH’s Reduce Function

Input: \(<key, KVAL = \text{listOf}(key \text{ set } K, \text{ value})>\>

1: repeat
2: \hspace{1em} altKey = Shared.\text{peekMinKey}()
3: \hspace{1em} if altKey < key then
4: \hspace{2em} O-reduce(altKey, Shared.\text{popMinKeyValues}()) /* Original reduce on values with smaller key altKey */
5: \hspace{1em} until altKey ≥ key
6: for all (K, value) in KVAL do
7: \hspace{1em} for all key in K do
8: \hspace{2em} Shared.\text{add}(key, value) /* Store for later reduce calls */
9: Values = KVAL.\text{getValues}()
10: if altKey = key then
11: \hspace{1em} Values = Values \cup Shared.\text{popMinKeyValues}() /* Append values with same key in Shared */
12: O-reduce(key, Values) /* Original reduce */
Anti-Combining Design — Lazy Sharing Strategy

- To decrease mapper output size
  - a Combiner
    - requires records with the same key
  - EagerSH
    - requires records with the same value
  - traditional compression techniques
    - require some form of redundancy among the keys and values

When all keys and values in the output of a Map call are unique, how to decrease mapper output size
Anti-Combining Design — Lazy Sharing Strategy

**EagerSH**

\[
(k_{in}, v_{in}) \xrightarrow{\text{map}} (k_1, v_1)^{r_1} (k_2, v_1)^{r_2} (k_3, v_2)^{r_2} (k_4, v_2)^{r_2} (k_5, v_2)^{r_2}
\]

\[
\xrightarrow{\text{encode}} (k_1, (\{\}, v_1))^{r_1} (k_2, (\{\}, v_1))^{r_2} (k_3, (\{k_4, k_5\}, v_2))^{r_2}
\]

---

**LazySH**

\[
(k_{in}, v_{in}) \xrightarrow{\text{map}} (k_1, v_1)^{r_1} (k_2, v_2)^{r_1} (k_3, v_3)^{r_2} (k_4, v_4)^{r_2} (k_5, v_5)^{r_2}
\]

\[
\xrightarrow{\text{encode}} (k_1, (k_{in}, v_{in}))^{r_1} (k_3, (k_{in}, v_{in}))^{r_2}
\]

---

**LazySH:** transfers the Map input record to all reduce tasks
Lazy Sharing Strategy

- **LazySH Map Phase**

  First computes the output of the original Map call for input record
  then finds the minimal key for each reduce task, i.e., partition
  record I is emitted for each of these minimal keys

### Algorithm 3: LazySH’s Map Function

<table>
<thead>
<tr>
<th>Input: input tuple I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: MapOutput = O-map(I) /* Original map */</td>
</tr>
<tr>
<td>2: result = SELECT MIN(O.key) AS key FROM MapOutput O GROUP BY getPartition(O.key)</td>
</tr>
<tr>
<td>3: for all r ∈ result do</td>
</tr>
<tr>
<td>4: Emit(r.key, I)</td>
</tr>
</tbody>
</table>
Analysis of an example

- A real query, and its length is n
  - If the Partitioner assigns all its prefixes to the same reduce task
    - Original program
      - \(1+n + 2+n + \ldots + n+n = n(1+n)/2 + n^2\)
    - EagerSH
      - \(1 + (2 + 3 + \ldots + n) + n = n(1+n)/2 + n\)
    - LazySH
      - \(1+n\)
Lazy Sharing Strategy

- **LazySH Reduce Phase**

  The reduce tasks of LazySH receive Map input, not output, therefore decoding in the reducer requires re-execution of the original Map function.
Anti-Combining Design — ENABLING ADAPTIVE RUNTIME OPTIMIZATION

// Original Mapper class
Class Mapper {
    map(K, V, context) {...
    setup(...) {...
    cleanup(...) {...
}

// Adaptive Mapper for Anti-Combining
Class AntiMapper {
    Mapper o_mapper, AntiContext a_context

    setup(...) { call o_mapper.setup() }
    cleanup(...) { call o_mapper.cleanup() }

    map(K key, V val, context) {
        o_mapper.map(key, val, a_context) //Call original map, measure cost
        mapOutput = a_context.getOutput()
        mapOutput.partition(Partitioner)  //Call Partitioner, measure cost
        if ((cost of map + cost of partition call) * number of partitions > T)
            for all partitions P in mapOutput do context.write(EagerSH–encoded P)
        else
            for all partitions P in mapOutput
                if (size of EagerSH–encoded P < size of input record (key, val))
                    use EagerSH to encode P
                else
                    use LazySH to encode P
                context.write(encoded partition P)
    }

// Extended Context class
Class AntiContext {
    mapOutput

    write(K key, V val) {
        mapOutput.insert(key, val)
    }

    getOutput() {
        return mapOutput
    }

    then partitions the output

    first executes the original program's Map function on input record (key, val) (through the o_mapper object)

    compute the total cost of re-executing o_mapper.map and getPartition to determine EagerSH or LazySH
Anti-Combining Design — ENABLING ADAPTIVE RUNTIME OPTIMIZATION

// Original Reducer class
Class Reducer {
    reduce(K, iter<V>, context) {...}
    setup(...) {...}
    cleanup(...) {...}
}

// Adaptive Reducer for Anti-Combining
Class AntiReducer {
    Shared
    Reducer o_reducer; Mapper o_mapper
    setup(...) { initialize Shared; call o_reducer.setup() }
    cleanup(...) { cleanup Shared; call o_reducer.cleanup() }

    reduce(K key, iter<adaptiveV> values, context) {
        repeat {
            altKey = Shared.peekMinKey()
            if (altKey < key)
                o_reducer.reduce(altKey, Shared.popMinKeyValues(), context)
        } until altKey >= key
        for all val in values {
            determine encoding used for val
            decode val and insert key-value pairs into Shared
            // Decoding for LazySH calls o_mapper.map and the
            // original Partitioner, which is available through the context
        }
        o_reducer.reduce(key, Shared.popMinKeyValues(), context) }
    }

Class Shared<K, V> {
    minHeap<K>
    hashMap<K, V>
    keyComparator
    groupingComparator
    add(K, V) {...}
    popMinKeyValues() {...}
}
Evaluation

• **Platform**
  - a 12-machine cluster running Hadoop 1.0.3
  - Each machine
    • a single quad-core Xeon 2.4GHz processor
    • 8MB cache
    • 8GB RAM
    • two 250 GB 7.2K RPM SATA hard disks

• **Data sets**

<table>
<thead>
<tr>
<th>Data set</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>QLog</td>
<td>140 million real queries</td>
<td>4.3GB</td>
</tr>
<tr>
<td>ClueWeb09</td>
<td>a real data set containing the first English segment of a web crawl</td>
<td>7GB</td>
</tr>
<tr>
<td>Cloud</td>
<td>a real data set containing extended cloud reports from ships and land stations</td>
<td>28.8GB</td>
</tr>
<tr>
<td>RandomText</td>
<td>a synthetic data set containing randomly generated text records</td>
<td>360GB</td>
</tr>
</tbody>
</table>
Query Suggestion I

- the Query-Suggestion problem on QLog and explore the effect of the choice of Partitioner by comparing three alternatives

![Bar chart](image)

Figure 9: Total Map Output Size for Query-Suggestion
Figure 10: Total Map Output Size for Query-Suggestion using Combiner and Compression

Table 1: Total Cost Breakdown for Prefix-5, using different Compression Techniques

<table>
<thead>
<tr>
<th></th>
<th>Deflate</th>
<th>Gzip</th>
<th>Bzip2</th>
<th>Snappy</th>
<th>AdaptiveSH with Gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Disk Read(GB)</td>
<td>65</td>
<td>65</td>
<td>56</td>
<td>105</td>
<td>15</td>
</tr>
<tr>
<td>Total Disk Write(GB)</td>
<td>82</td>
<td>82</td>
<td>70</td>
<td>133</td>
<td>21</td>
</tr>
<tr>
<td>Total Map Output Size(GB)</td>
<td>18</td>
<td>18</td>
<td>15</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>Total CPU Time(1000 sec)</td>
<td>126.9</td>
<td>125.2</td>
<td>332.4</td>
<td>77.4</td>
<td>27.9</td>
</tr>
</tbody>
</table>
Effect on Disk I/O and CPU

- The cost breakdown for total disk read/write and total CPU time of the Query Suggestion
  - "-CB" and "-CP" means Combiner and compression, respectively.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Total CPU Time (1000 sec)</th>
<th>Disk Read (GB)</th>
<th>Disk Write (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>168.8</td>
<td>566.1</td>
<td>741.5</td>
</tr>
<tr>
<td>Original-CB</td>
<td>172.9</td>
<td>510.4</td>
<td>664.6</td>
</tr>
<tr>
<td>Original-CP</td>
<td>125.2</td>
<td>64.5</td>
<td>82.3</td>
</tr>
<tr>
<td>AdaptiveSH</td>
<td>30.8</td>
<td>150.8</td>
<td>179.9</td>
</tr>
<tr>
<td>AdaptiveSH-CB</td>
<td>20.8</td>
<td>61.9</td>
<td>84.9</td>
</tr>
<tr>
<td>AdaptiveSH-CP</td>
<td>27.9</td>
<td>15</td>
<td>20.6</td>
</tr>
</tbody>
</table>
CPU Intensive Workloads

- the performance of Anti-Combining for CPU intensive workloads by adding extra CPU intensive calls to the Map function

![Graph showing total CPU time using runtime cost-based optimization.](image)
Conclusions

• **Anti-Combining**
  – A novel approach for reducing the amount of data transferred between mappers and reducers
  – Shifts mapper-side processing to the reducers
  – Adaptive runtime optimization

• **Applicable的问题**