MapReduce Extension

Topics in Data Science

Cheng Ren, Lixing Lian
Outline

• Scientific Workloads
  - Hadoop’s Adolescence: An Analysis of Hadoop Usage in Scientific Workloads

• Iterative extension
  - Haloop: efficient iterative data processing on large clusters

• Adaptive Indexes
  - Only Aggressive Elephants are Fast Elephants (HAIL)
Hadoop's adolescence An analysis of Hadoop usage in scientific workload
Motivation

- How well Hadoop works for data scientists in a user-experience perspective
- User behavior Study of using Hadoop
Settings

- Three different clusters

  OpenCloud (computational astrophysics, computational biology ...)

  M45 (large-scale graph mining, text and web mining ...)

  Web Mining (web text mining)

- 20 months
- 113 data scientists from various disciplines
- 100,000 Hadoop Jobs
Hadoop Ecosystem

High-level Language:

- Hive
- Pig

High-level APIs: Cascading/Scoobi

Canned MapReduce Jobs APIs:

- Mahout
- LoadGen
Raw Hadoop applications VS High-level interfaces

![Graph showing the fraction of jobs per application type.](image)

Figure 1: Fraction of jobs per application type.
Raw Hadoop applications VS High-level interfaces

User still prefer to write their MapReduce jobs in plain Java

Other Finding:
- high-level tools that fail in the middle and difficult to resume
- existing tools cumbersome
- use Hadoop simply as a task scheduler.
## Variety of Job Structures

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Map Only</th>
<th>Single Job</th>
<th>Short Chain</th>
<th>Long Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCloud</td>
<td>21.00%</td>
<td>31.00%</td>
<td>8.00%</td>
<td>3.00%</td>
</tr>
<tr>
<td>M45</td>
<td>11.00%</td>
<td>35.00%</td>
<td>17.00%</td>
<td>9.00%</td>
</tr>
<tr>
<td>Web Mining</td>
<td>8.00%</td>
<td>70.00%</td>
<td>14.00%</td>
<td>8.00%</td>
</tr>
</tbody>
</table>

- **Short chain**: two or three jobs
- **Long chain**: more than three jobs

\[ \geq 50\% \text{ of applications can fit in the MapReduce paradigm} \]
Variety of Job Structures

Finding & Improvements

- Some jobs are hardly parallel.
- Should cater to extensive variety, rather than optimize for a single type of applications.
# Hadoop Optimization

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combiner</td>
<td>Used for efficiency reasons</td>
</tr>
<tr>
<td>Secondary Sort</td>
<td>An optimized join algorithm</td>
</tr>
<tr>
<td>Custom Partitioner</td>
<td>How to redistribute the map output to the reduce tasks</td>
</tr>
<tr>
<td>Custom Input and Output Format</td>
<td>Simplify handling of custom data formats and non-native storage systems</td>
</tr>
</tbody>
</table>
## Hadoop Optimization

**How many users perform customization at least once?**

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>OpenCloud</th>
<th>M45</th>
<th>Web Mining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combiner</td>
<td>62.00%</td>
<td>43.00%</td>
<td>80.00%</td>
</tr>
<tr>
<td>Secondary Sort</td>
<td>14.00%</td>
<td>0.00%</td>
<td>1.00%</td>
</tr>
<tr>
<td>Custom Partitioner</td>
<td>35.00%</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Custom Input and Output Format</td>
<td>27.00%</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
## Tunings

Number of users who changed tuning configuration

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>OpenCloud</th>
<th>M45</th>
<th>Web Mining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure Parameters</td>
<td>7</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>JVM</td>
<td>29</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Sort Parameters</td>
<td>N/A</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>HDFS Parameters</td>
<td>11</td>
<td>2</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Tunings-case study

Data Balancing

More than **20%** of jobs remain unbalanced in runtime.

- Optimizer: Input Splits

- less than **12%** of all jobs used this optimization in all datasets
Tuning

Other Finding & Improvements

- Tuned these options only when failures occur, few tune parameters related to performance.

- Demand for automatic tuning and interactive monitoring tool
Big Jobs or Small Jobs

Figure 10: Percentiles (min, 10%, median, 90%, max) of job durations in the OpenCloud cluster for each month.
Big Jobs or Small Jobs

Figure 11: Percentiles (min, 10%, median, 90%, max) of aggregate I/O activity (input+shuffle+output data size) of OpenCloud jobs each month.

50% are small jobs (touch less than 10 GB of data, run for less than 8 mins)
Hadoop is not the suboptimal solution
Resource Usage

(a) **OPENCLOUD**

(b) **M45**

Figure 13: Aggregated resource usage over users under three metrics. 20% of users consume about 90% of the resources.

90% of resources are consumed by 20% of the users.
User Education Missing

- Some users would rather break the large datasets into smaller ones
- Users primarily use Hadoop to process short jobs
- Large numbers of users do not leverage their Hadoop clusters much
Hadoop's adolescence

- Under-use of Hadoop features, extensions, and tools
- Optimization/Configuration are used in a narrow scope
- User education is missing to fully utilize the Hadoop resource
Future Work

1. Optimize for variety of jobs(long-chained jobs)

2. Advanced monitoring/tuning tools are needed
HaLoop: Efficient Iterative Data Processing On Large Clusters

Yingyi Bu, Bill Howe, Magda Balazinska, Michael D. Ernst
Outline

• Motivation

• Examples that cannot be executed perfectly

• Architecture

• Caching ideas
Motivation

• MapReduce can’t express recursion/iteration
• Lots of interesting programs need loops
  - graph algorithms
  - clustering
  - machine learning
  - recursive queries (CTEs, datalog, WITH clause)
• Dominant solution: Use a driver program outside of MapReduce
• Hypothesis: making MapReduce loop-aware affords optimization
  - lays a foundation for scalable implementations of recursive languages
Example 1: PageRank

<table>
<thead>
<tr>
<th>url</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.b.com">www.b.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.c.com">www.c.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.d.com">www.d.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.e.com">www.e.com</a></td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>url_src</th>
<th>url_dest</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td><a href="http://www.b.com">www.b.com</a></td>
</tr>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td><a href="http://www.c.com">www.c.com</a></td>
</tr>
<tr>
<td><a href="http://www.c.com">www.c.com</a></td>
<td><a href="http://www.a.com">www.a.com</a></td>
</tr>
<tr>
<td><a href="http://www.d.com">www.d.com</a></td>
<td><a href="http://www.b.com">www.b.com</a></td>
</tr>
<tr>
<td><a href="http://www.e.com">www.e.com</a></td>
<td><a href="http://www.c.com">www.c.com</a></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
R_{i+1} & = \pi(\text{url}_\text{dest}, \gamma_{\text{url}_\text{dest}} \sum \text{rank}) \\
R_i.\text{rank} & = R_i.\text{rank}/\gamma_{\text{url}} \sum \text{COUNT(url}_\text{dest}) \\
R_i.\text{url} & = L.\text{url}_\text{src}
\end{align*}
\]
PageRank Implementation on MapReduce

Join & compute rank

\( R_i \)

Count

L-split0

L-split1

\( i = i + 1 \)

Converged?

Client

done

Aggregate

Fixpoint evaluation
L and Count are loop invariants, but
1. They are loaded on each iteration
2. They are shuffled on each iteration
3. Also, fixpoint evaluated as a separate MapReduce job per iteration
Example 2: Transitive Closure

<table>
<thead>
<tr>
<th>Friend</th>
<th>name1</th>
<th>name2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tom</td>
<td>Bob</td>
</tr>
<tr>
<td></td>
<td>Tom</td>
<td>Alice</td>
</tr>
<tr>
<td>Elisa</td>
<td>Tom</td>
<td></td>
</tr>
<tr>
<td>Elisa</td>
<td>Harry</td>
<td></td>
</tr>
<tr>
<td>Sherry</td>
<td>Todd</td>
<td></td>
</tr>
<tr>
<td>Eric</td>
<td>Elisa</td>
<td></td>
</tr>
<tr>
<td>Todd</td>
<td>John</td>
<td></td>
</tr>
<tr>
<td>Robin</td>
<td>Edward</td>
<td></td>
</tr>
</tbody>
</table>

*(semi-naïve evaluation)*

Find all transitive friends of Eric

\[ R_0 \quad \{\text{Eric, Eric}\} \]

\[ R_1 \quad \{\text{Eric, Elisa}\} \]

\[ R_2 \quad \{\text{Eric, Tom, Eric, Harry}\} \]

\[ R_3 \quad \{\} \]
Transitive Closure on MapReduce

\[
\begin{align*}
S_i & \rightarrow M \\
\text{Friend0} & \rightarrow M \\
\text{Friend1} & \rightarrow M \\
\text{Join} & \rightarrow r \\
\text{Dup-elim} & \rightarrow r \\
\text{Anything new?} & \rightarrow i = i + 1 \\
\text{Client} & \rightarrow \text{done}
\end{align*}
\]
What’s the problem?

Friend is loop invariant, but
1. Friend is loaded on each iteration
2. Friend is shuffled on each iteration
Push loops into MapReduce!

• Architecture

• Cache loop-invariant data

• Programming Model
HaLoop Architecture

The diagram illustrates the architecture of HaLoop, a distributed computing framework. It consists of a Master node and multiple Slave nodes. The Master node manages tasks and distributes them to the Slave nodes. Each Slave node runs tasks from the Master and communicates back to the Master.

Key components include:
- **Application**: Forms jobs that are managed by the Master.
- **Framework**: Contains a Task Scheduler, Loop Control, Task Queue, and Task Tracker.
- **File System**: Includes a Distributed File System and a Local File System.

The diagram also highlights the communication channels between nodes and the types of tasks and features unique to HaLoop compared to Hadoop.
Inter-iteration caching

Mapper output cache (MO)

Reducer input cache (RI)

Reducer output cache (RO)

Mapper input cache (MI)
RI: Reducer Input Cache

- **Provides:**
  - Access to loop invariant data without map/shuffle
- **Data:**
  - Reducer function
- **Assumes:**
  1. Static partitioning (implies: no new nodes)
  2. Deterministic mapper implementation

- **PageRank**
  - Avoid loading and shuffling the web graph at every iteration
- **Transitive Closure**
  - Avoid loading and shuffling the friends graph at every iteration
RO: Reducer Output Cache

- **Provides:**
  - Distributed access to output of previous iterations

- **Used by:**
  - Fixpoint evaluation

- **Assumes:**
  1. Partitioning constant across iterations
  2. Reducer output key functionally determines Reducer input key

- **PageRank**
  - Allows distributed fixpoint evaluation
  - Obviates extra MapReduce job

- **Transitive Closure**
  - No help
MI: Mapper Input Cache

- **Provides:**
  - Access to non-local mapper input on later iterations
- **Data for:**
  - Map function
- **Assumes:**
  - Mapper input does not change
  - Avoids non-local data reads on iterations > 0
Programming Model

- Mapper/reducer stay the same!
- Touch points
  - Input/Output: for each <iteration, step>
  - Cache filter: which tuple to cache?
  - Distance function: optional
- Nested job containing child jobs as loop body
- Minimize extra programming efforts
Conclusions

• Relatively simple changes to MapReduce/Hadoop can
  - support iterative/recursive programs
  - TaskTracker (Cache management)
  - Scheduler (Cache awareness)
  - Programming model (multi-step loop bodies, cache control)

• Optimizations
  - Caching reducer input realizes the largest gain
  - Good to eliminate extra MapReduce step for termination checks
  - Mapper input cache benefit inconclusive; need a busier cluster
Only Aggressive Elephants are fast Elephants

Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, Jörg Schad
Outline

• Motivation
• Comparison between Hadoop and HAIL
• Upload pipeline
• Query pipeline
• Analyze a large web log by filtering conditions. (source IP, web address)

• He uses a sequence of different filter conditions, each one triggering a new MapReduce job.

• He is not exactly sure what he is looking for.

• “Let’s see what I am going to encounter on the way.”
This kind of use-case illustrates an exploratory usage of Hadoop MapReduce.

It is a major use-case of Hadoop MapReduce.
- One major problem: slow query runtimes.
- Time dominated by the I/O for reading all input data.
Hadoop Aggressive Indexing Library
HDFS + MapReduce VS. HAIL + MapReduce
HDFS + MapReduce
HDFS

horizontal partitions

Datanodes

HDFS blocks 64MB (default)
HDFS
HDFS
HDFS
HDFS

Allows two Failovers
MapReduce

map(row) -> set of (ikey, value)
MapReduce

map(row) -> set of (ikey, value)
MapReduce

map(row) -> set of (ikey, value)
MapReduce

map(docID, document) \rightarrow set of (term, docID)
HAIL + MapReduce
HAIL

horizontal partitions

HDFS blocks 64MB (default)
HAIL

<table>
<thead>
<tr>
<th>DN1</th>
<th>DN2</th>
<th>DN3</th>
<th>DN4</th>
<th>DN5</th>
<th>DN6</th>
<th>DNn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 5 6

7 8 9
HAIL
HAIL
HAIL
HAIL

1. Convert the input file into binary PAX

2. Create a series of different sort orders

3. Create multiple clustered indexes.
   - If indexes cannot help, fall back to standard Hadoop scanning.
HAIL changes the upload pipeline of HDFS in order to create different clustered indexes on each data block replica.
HAIL Upload Pipeline

Why Clustered Indexes?
- Unclustered indexes are only competitive for very selective queries as they may trigger considerable random I/O for non-selective index traversals.
- Clustered index do not have that problem. Whatever the selectivity, we will read the clustered index and scan the qualifying blocks.
HAIL Upload Pipeline

Semantics of an ACK for a packet of a block are changed
From “packet received, validated, and flushed”
To “packet received and validated”.

In parallel to forwarding and reassembling packets, each
datanode sorts the data, creates indexes, and forms a HAIL
block.
HAIL Upload Pipeline

Enrich the HDFS namenode to schedule map tasks close to replicas having a suitable index

- Dir_rep mapping \((\text{blockID}, \text{datanode})\) → HAILBlockReplicaInfo
- HAILBlockReplicaInfo contains detailed information about the types of available indexes
HAIL Query Pipeline

1. Write Job
2. Run Job

MapReduce Job
- Main Class
- map(...)
- reduce(...)

HAIL Annotation

3. Split Phase
   for each block\textsubscript{i} in input {
     locations = block\textsubscript{i}.getHostWithIndex(\#3);
     splitBuilder.add(locations, block\textsubscript{i});
   }
   splits() = splitBuilder.result;

4. Scheduler
   for each split\textsubscript{i} in splits {
     allocate split\textsubscript{i} to closest datanode storing block\textsubscript{i}
   }

5. Map Phase
   allocate Map Task
   Task Tracker
   Record Reader:
   - Perform Index Scan
   - Perform Post-Filtering
   - for each record invoke map(HailRecord)

6. HDFS
   DN1, DN2, ...
Upload Time

![Graph showing the upload time for Hadoop and HAIL with different numbers of created replicas.](image)

- **Hadoop**
  - 3 replicas: 717 sec
  - 5 replicas: 956 sec
  - 6 replicas: 1089 sec
  - 7 replicas: 1254 sec
  - 10 replicas: 1700 sec

- **HAIL**
  - 3 replicas: 1132 sec
  - 5 replicas: 1773 sec
  - 6 replicas: 2256 sec
  - 7 replicas: 2712 sec
  - 10 replicas: 3710 sec

- Dashed line represents Hadoop upload time with 3 replicas.
Query Times

Individual Jobs: Weblog
Fast Indexing and Fast Querying
Questions? Ask Bob.