Extending Starfish to Support the Growing Hadoop Ecosystem

by

Fei Dong

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Shivnath Babu, Supervisor

_____
Bruce Maggs

_____
Benjamin C. Lee

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in the Department of Computer Science
in the Graduate School of Duke University
2012

Abstract

Hadoop is a new open-source system designed to store and process large amounts of data (popularly called Big Data today). Hadoop has a large number of configuration options that have to be tuned for performance, availability, and scalability. The goal of the Starfish project at Duke is to enable automatic tuning of Hadoop through cost-based optimization techniques like those used in database systems. A major challenge faced by Starfish is the rapid evolution of Hadoop. A number of new software layers, storage systems, and programming paradigms have been added to extend Hadoop from a single system to an entire ecosystem for handling Big Data. Our project addresses the important research problem of adapting Starfish to deal with new extensions to the Hadoop ecosystem. Three types of extensions are considered in this project:

1. New query languages that generate multi-job workflows.
2. Iterative execution of jobs in a workflow.
3. Use of storage designed for random key-based lookups in addition to storage designed for scan-based access.

To address the first extension, we build on an existing cost-based optimization approach in Starfish. We develop a new cost-based optimization approach to address the second extension. The third extension is addressed through expert rules developed from an empirical study. We have evaluated the effectiveness of our techniques using a number of Hadoop workloads, including a workflow from a real-life application.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my special thanks to my advisor Shivnath Babu. I also want to thank my other two committee members Bruce Maggs and Benjamin C. Lee for their feedback and guidance.

I would like to acknowledge and thank Harold Lim, Balakrishnan Chandrasekaran, Herodotos Herodotou and Jie Li who have contributed to the growth of my work.

Lastly, I offer my regards and blessings to my family and friends who supported me during the completion of the project.

1

Introduction

1.1 Background

Big Data, voluminous and rapidly growing amounts of data that exceeds the processing capacity of conventional database systems, is now the main driving force for enterprises to innovate by gaining deeper insight about their customers, partners and overall business. EMC and IDC released their Digital Universe study in 2011 [1], estimating that the world will create 1.8 zettabytes of data this year and the growth of data is outpacing Moores Law. Now that we realize there is value in all that information, we are anxious to capture, analyze and use it, and that requires more and better Dig Data technology. As Figure 1.1 from Karmasphere [3] illustrates, Hadoop [11] is a very large part of the big data stack. In a nutshell, Hadoop is a set of open-source tools designed to enable the storage and processing of large amounts of unstructured data across a cluster of servers. Chief among those tools are Hadoop MapReduce and the Hadoop Distributed File System (HDFS), but there are numerous related ones, including Hive [18], Pig [23], HBase [12] and ZooKeeper [27].

1.1.1 Hadoop Ecosystem

Figure 1.2 shows some key components in the Hadoop ecosystem. Software components that are provided with the Hadoop stack are shown in the left panel. At the bottom, Hadoop is run on a cluster of machines. On top of that is a distributed file system, storing data in a replicated way and providing large read/write throughput. As a Hadoop core engine, the MapReduce [20] execution engine runs jobs in parallel by using a group of machines. Users can implement the MapReduce inter-

[1] http://karmasphere.com/In-The-News/gigaom-karmasphere-pushes-new-big-data-workflow.html

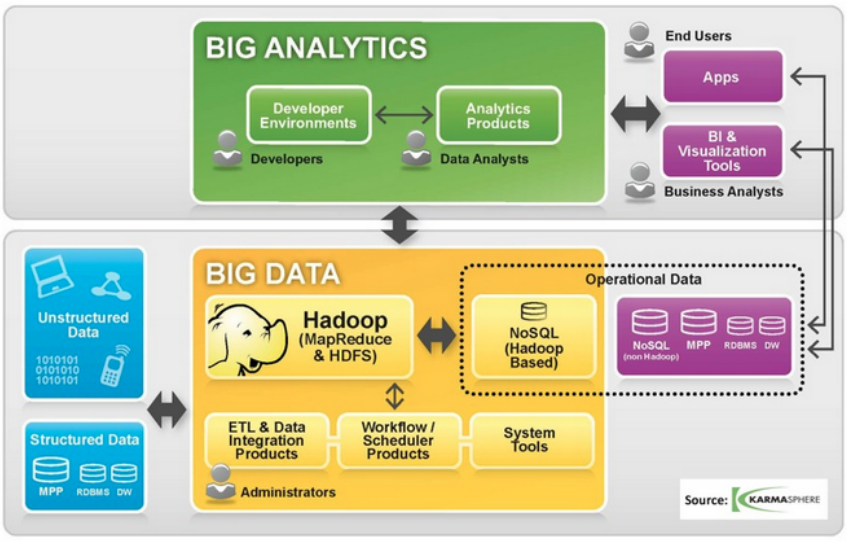


FIGURE 1.1: Big data overview

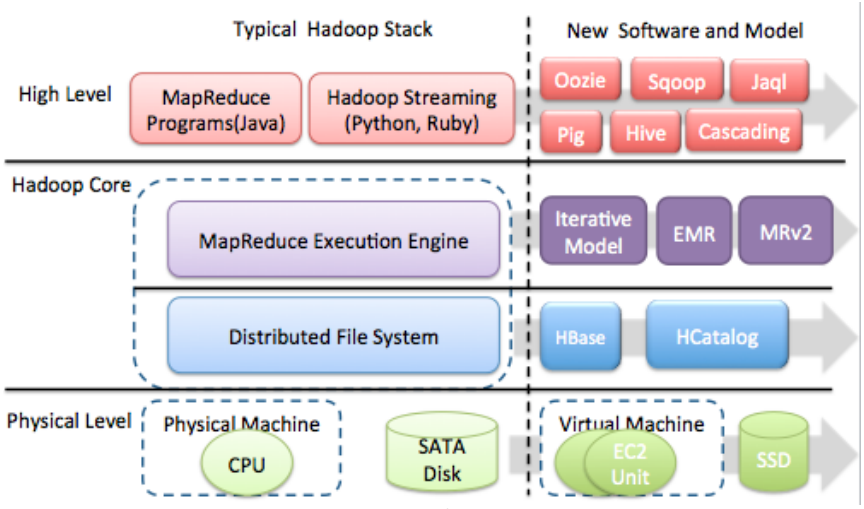


FIGURE 1.2: Hadoop ecosystem

face via Java API or Hadoop Streaming. The efficiency of Hadoop depends on the file size, number of files, the number of machines in a cluster, bandwidth, Hadoop configurations, data compression and so on.

On the right panel, we can see other new software on top of Hadoop that extend the spectrum of Hadoop. At the physical level, Amazon Web Service (AWS) [2] provides a virtual machine environment that leases resource through a computing unit. It attracts small to medium organizations that need to process large datasets on demand. In fact, all of the experiments in this thesis are run on Amazon Elastic Cloud Computing (EC2) platform. At the storage layer, HBase is introduced as a distributed big data storage on Hadoop. HCatalog [13] is a table and storage man-

agement service for data created using Hadoop. At the computation layer, Amazon provides Elastic MapReduce [1] that utilizes a hosted Hadoop framework running on the web-scale infrastructure of Amazon EC2 and Amazon Simple Storage Service (Amazon S3). Another inspiring news about the next generation of MapReduce [25] is that it will support multiple programming paradigms, such as MapReduce, MPI, Master-Worker, and iterative models. At the workflow layer, more declarative languages such as Pig Latin, Hive, Cascading, Jaql [19] are introduced to simplify the development process. Oozie [21] is a workflow service that manages data processing jobs for Hadoop. Actually, the Hadoop community leaves the platform decisions to end users, most of whom do not have a background in systems or the necessary lab environment to benchmark all possible design solutions.

On the other hand, Hadoop is a complex set of software with more than 200 tunable parameters. Each parameter affects others as tuning is completed for a Hadoop environment and will change over time as job structure changes, data layout evolves, and data volume grows. How to tune performance without knowing much is a hot research topic.

1.2 Starfish: Self-tuning Analytic System on Hadoop

Starfish [17, 14, 16] is motivated based on the background and developed as an intelligent performance tuning tool for Hadoop. It has the following features —-

1. It builds on Hadoop while adapting to user needs and system workloads to provide good performance automatically, without any need for users to understand and manipulate the many tuning knobs in Hadoop.

2. Without modifying custom MapReduce programs or Hadoop source code, it can probe your MapReduce behavior and has zero overhead to turn off Profiler.

Starfish consists of (i) a Profiler (collect job metrics); (ii) a What-if Engine (ask hypothetical questions and return virtual profilers); (iii) an Optimizer (find an optimal settings); (iv) a Visualizer (present visual effect for Starfish). These components are described next.

1.2.1 Profiler

The Profiler is responsible for collecting job profiles: statistical summaries of MapReduce job execution. A job profile is a vector of fields where each field captures some unique aspect of data-flow or cost estimates during job execution. Data-flow estimates represent information regarding the amount of bytes and key-value pairs processed during the jobs execution, as well as key-value distributions for input, intermediate, and output data. Cost estimates represent execution time and resource usage, including the usage trends of CPU, memory, I/O, and network resources during the jobs execution.

1.2.2 What-If Engine

The What-if Engine is given four inputs when asked to predict the performance of a MapReduce job j:

1. Job profile generated for j by the Profiler.
2. New configuration settings c to run j with.
3. Size, layout, and compression information of the input dataset d on which j will be run. Note that this input dataset can be different from the dataset used while generating the job profile.
4. Cluster setup and resource allocation r that will be used to run j. This information includes the number of nodes and network topology of the cluster, the number of map and reduce task slots per node, and the memory available for each task execution.

1.2.3 Optimizer

Given a MapReduce program p to be run on input data d and cluster resources r, the Optimizer must find the setting of configuration parameters $c_{opt} = \underset{c \in S}{\operatorname{argmin}} F(p, d, r, c)$ for the cost model F represented by the What-if Engine over the space S of configuration parameter settings. The Optimizer addresses this problem by making what-if calls with settings c of the configuration parameters selected through an enumeration and search over S. In order to provide both efficiency and effectiveness, the Optimizer must minimize the number of what-if calls while finding close-to-optimal configurations.

1.2.4 Visualizer

The Visualizer [15] is a new graphical tool that we have developed. Users can employ the Visualizer to (a) get a deep understanding of a MapReduce jobs behavior during execution, (b) ask hypothetical questions on how the job behavior will change when parameter settings, cluster resources, or input data properties change, and (c) ultimately optimize the job. Figure 1.3 presents a timeline of the map and reduce tasks of a MapReduce job. It is a very useful and agile tool to help us analysis performance in the project.
The released version of Starfish focuses on (i) profiling, (ii) predicting performance, and (iii) recommending configuration parameter settings for individual MapReduce jobs running on a particular cluster. Extending this capability to Pig, Hive, Cascading, and other system is under development.

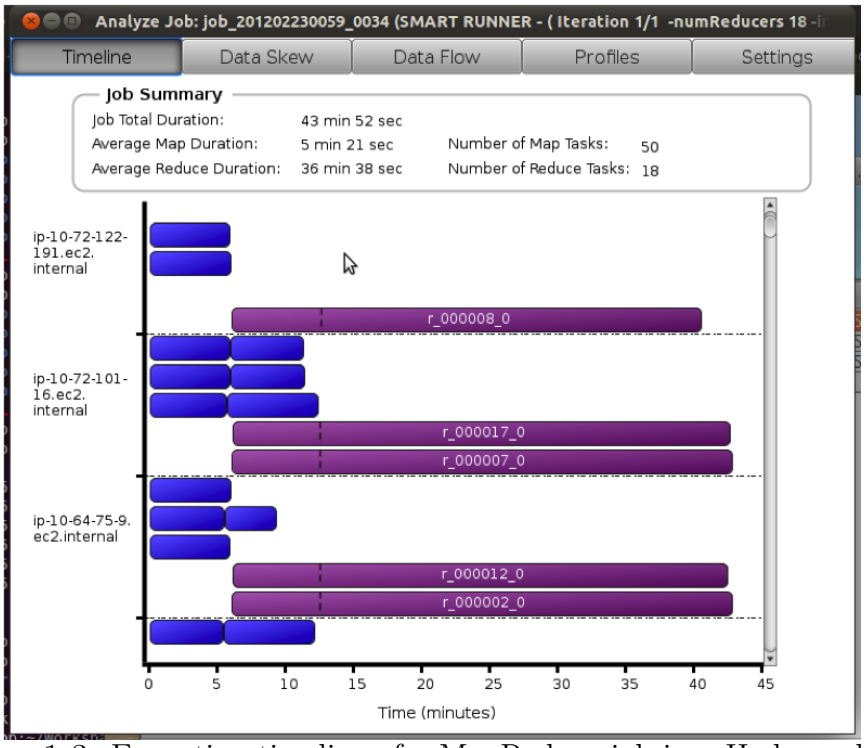


FIGURE 1.3: Execution timeline of a MapReduce job in a Hadoop cluster

1.3 Cloud Computing Environment

Infrastructure-as-a-service(IaaS) cloud platform has brought unprecedented changes in the cloud leasing market. Amazon EC2 [7] is a popular cloud provider to target the market, by providing the standard on-demand instances, reserved instances and Spot Instances(SI). EC2 is basically the equivalent of a virtual machine. When launching an instance, users are asked to choose an AMI (Amazon Machine Image), which is an image of an operating system. Next, users can choose the type of instance they like, there are quite a few options to choose from, depending on how much power they need and the kind of operations are running. Table 1.1 shows the features and renting costs of some representative EC2 node types.

In following Chapters, we prepare an AMI including all of software Hadoop needs and compare performance via choosing proper node type and size.

1.4 MapReduce Workflows on Hadoop

The Hadoop ecosystem evolves some high-level layers to support comprehensive workflows including Hive (with an SQL-like declarative interface), Pig (with an interface that mixes declarative and procedural elements), Cascading (with a Java interface for specifying workflows). We list some similarities they share and some different characteristics in Table 1.2.

EC2 Node Type	CPU (# EC2 Units)	Memory (GB)	Storage (GB)	I/O Performance	Cost (U.S. $ per hour)
m1.small	1	1.7	160	moderate	0.085
m1.large	4	7.5	850	high	0.34
m1.xlarge	8	15	1,690	high	0.68
m2.xlarge	6.5	17.1	1,690	high	0.54
c1.medium	5	1.7	350	moderate	0.17
c1.xlarge	20	7	1,690	high	0.68

Table 1.1: Six representative EC2 node types, along with resources and costs

Some similarities of Pig, Hive and Cascading.

- All translate their respective hig-level languages to MapReduce jobs.
- All provide interoperability with other languages.
- All provide points of extension to cover gaps in functionality.
- All offer significant reductions in program size over raw Java MapReduce program.

Characteristic	Pig	Hive	Cascading
Language name	Pig Latin	HiveQL	Java API
Type of Language	Data flow	Declarative (SQL dialect)	define operators in Java
Extension	User Defined Functions(UDF)	UDF	Java inherit
Shell	Yes	Yes	No
Server	No	Thrift (optional)	No
Hadoop API	New	Old	Old
Code Size	Medium	Large	Medium
Source code	60K lines	100K lines	40K lines

Table 1.2: Pig VS. Hive VS. Cascading

Starfish has already supported Pig. After considering the complexity and development timeline, we choose Cascading as a main workflow framework.

1.5 Hadoop Storage System

HDFS is the most used distributed filesystem in Hadoop. The primary reason HDFS is so popular is its built-in replication, fault tolerance, and scalability. However, it is not enough. Some application needs to efficiently store structured data while having

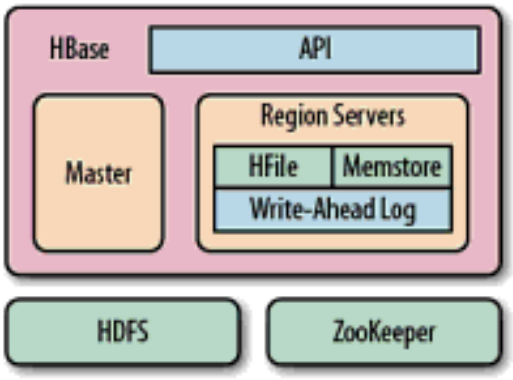


FIGURE 1.4: HBase components

a variable schema. HBase is useful in this scenario. Its goal is hosting of very large tables – billions of rows X millions of columns and provide high reliability.

Figure 1.4 shows how the various components of HBase are orchestrated to make use of existing system, like HDFS and Zookeeper, but also adding its own layers to form a complete platform.

Requirements	HDFS	HBase
Scalable Storage	✓	✓
System Fault Tolerance	✓	✓
Sequence Read/Write	✓	✓
Random Write	✗	✓
Client Fault Tolerance	✗	✓
Append & Flush	✗	✓

Table 1.3: HDFS vs. HBase

Table 1.3 discussed some tradeoffs of HBase. In general, HBase is a column-oriented store implemented as the open source version of Googles BigTable system. Each row in the sparse tables corresponds to a set of nested key-value pairs indexed by the same top level key (called "row key"). Scalability is achieved by transparently range-partitioning data based on row keys into partitions of equal total size following a shared-nothing architecture. As the size of data grows, more data partitions are created. Persistent distributed data storage systems are normally used to store all the data for fault tolerance purposes. In the thesis, we will analyze performance and propose some rules on optimization.

1.6 Cost Based Optimization vs. Rule Based Optimization

There are two kinds of optimization strategies used by Database engines for executing a plan.
Rule Based Optimization (RBO): RBO uses a set of rules to determine how to execute a plan. Cloudera [6] gives us some tips to improve MapReduce performance. For example: If a job has more than 1TB of input, consider increasing the block size of the input dataset to 256M or even 512M so that the number of tasks will be smaller. They recommend a number of reduce tasks equal to or a bit less than the number of reduce slots in the cluster.
Cost based optimization (CBO): The motivation behind CBO is to come up with the cheapest execution plan available for each query. The cheapest plan is the one that will use the least amount of resource (CPU, memory, I/O, etc.) to get the desired output. Query optimizers are responsible for finding a good execution plan p for a given query q, given an input set of tables with some data properties d, and some resources r allocated to run the plan. The kernel of CBO is a *costmodel*, which is used to estimate the performance cost y of a plan p. The query optimizer employs a search strategy to explore the space of all possible execution plans in search for the plan with the least estimated cost y.
$Starfish$ employs the cost-based optimization.

1.7 Contributions

As we described above, Hadoop is a powerful large-scale data processing platform and largely used in the industry. Within the self-tuning tool of Starfish, we can optimize jobs using profile-predict-optimize approach. However, since Hadoop is a constantly growing and complex software provides no guidance to the best platform for it to run on, Starfish should keep evolving and become agile. In this project, we focus on extending Starfish optimizer on several dimensions to support evolving Hadoop ecosystem. The specific contribution of this project can be briefly described as follows:

1. We extend Starfish to support a multi-job workflow — Cascading and evaluate some workflows which is described in Chapter 2.

2. We design and implement an iterative workflow optimizer in Chapter 3.

3. We analyze the performance of key-value store using rule-based technology in Chapter 4.

4. Based on some techniques discussed above, we optimize a real-life workflow Alidade in Chapter 5.

2

Optimization for Multi-Job Workflows

2.1 Cascading Overview

In Figure 2.1, we can clearly see the Cascading Structure. The top level is called Cascading which is composed of several flows. In each flow, it defines a source Tap, a sink Tap and Pipes. We also notice that one flow can have multiple pipes to do data operations like filter, grouping, aggregator.

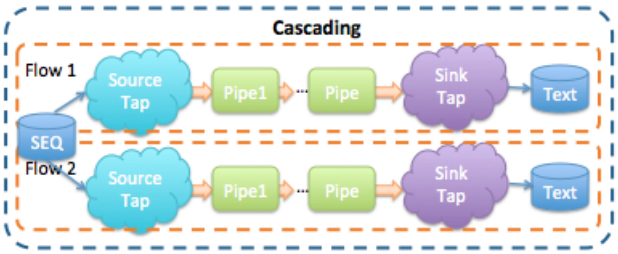


FIGURE 2.1: A typical cascading structure.

Internally, a Cascade worflow is constructed through the CascadeConnector class, by building an internal graph that makes each Flow a 'vertex', and each file an 'edge'. A topological traversal on this graph will touch each vertex in order of its dependencies. When a vertex has all it's incoming edges available, it will be scheduled on the cluster. Figure 2.2 gives us an example which goal is to statistic second and minute count from Apache logs. The dataflow is represented as a Graph. The first step is to import and parse source data. Next it generates two following steps to process "second" and "minutes" respectively.

The execution order for Log Analysis is:

1. calculate the dependency between flows, so we get $Flow1 \rightarrow Flow2$.

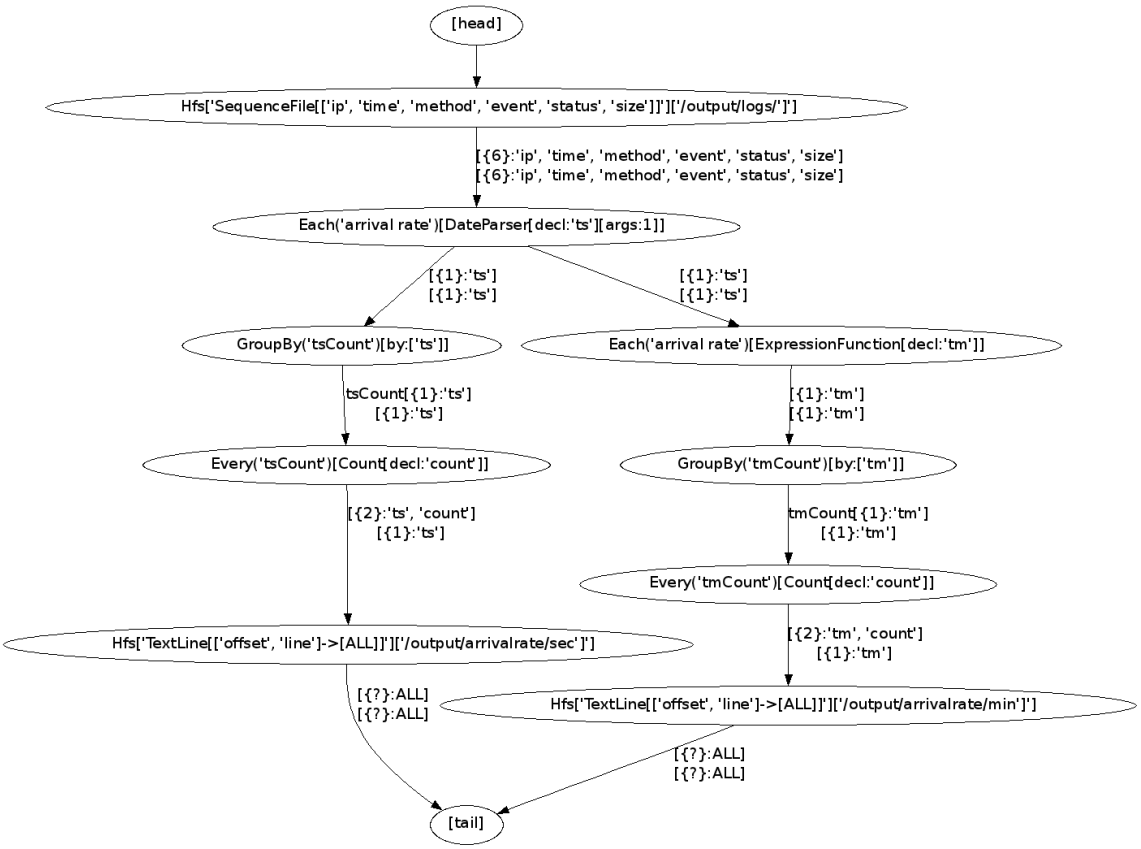


FIGURE 2.2: Cascading workflow example:log analysis.

2. start to call $Flow1$.

   (a) initialize "import" flowStep and construct the Job1.

   (b) submit "import" Job1 to Hadoop.

3. start to call $Flow2$.

   (a) initialize "minute and second statistics" flowSteps and construct the Job2, Job3.

   (b) submit Job2, Job3 to Hadoop.

2.2 Cascading on Starfish

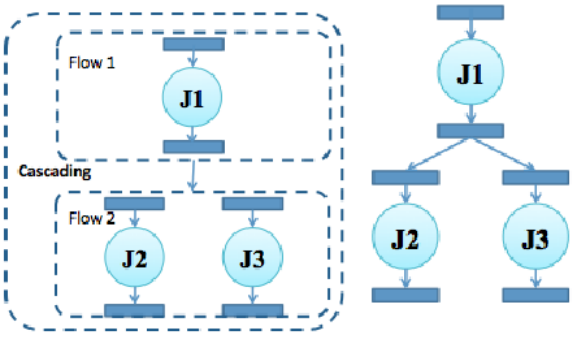
2.2.1 Change to new Hadoop API

The current version of Cascading uses the Hadoop Old-API. Since Starfish works with New-API (Old-API is supported recently), the first work is to connect those heterogeneous systems by replacing the Old-API of Hadoop. Although the Hadoop

community recommends using new API and provides some upgrade advice, it still takes us much energy on converting Cascading to use the new API. One reason is the system complexity (40K lines), we sacrifice some advanced features such as S3fs, TemplateTap, ZipSplit, Stats reports and Strategy to make the change work. Finally, we provide a revised version of Cascading that only uses Hadoop New-API.

2.2.2 Cascading Profiler

First, we need to decide when to capture the profilers. The position to enable the Profiler is the same as a single MapReduce job. We choose the return point of $blockTillCompleteOrStoped$ of $cascading.flow.FlowStepJob$ to collect job execution files when job completes. When all of the jobs are finished and the execution files are collected, we build a profile graph to represent dataflow dependencies among the jobs. In order to build the job directed acyclic graph(DAG), we decouple the hierarchy of Cascading and Flows. As we see before, Log Analysis workflow has two dependent Flows and finally will submit three MapReduce jobs to Hadoop. Figure 2.3 shows the original Workflow in Cascading and translating JobGraph in Starfish. We propose the following algorithm to build the Job DAG.



((a)) Cascading Represent ((b)) Dataflow Translation

FIGURE 2.3: Translate a Cascading workflow into job DAG.

2.2.3 Cascading What-if Engine and Optimizer

The What-if Engine predicts the behavior of a workflow W. To achieve that, the DAG Profilers ,Data Model, Cluster, DAG Configurations are given as parameters. Building the Conf Graph shares the same idea as building the Job Graph. We capture the returning point of $initializeNewJobMap$ in $cascading.cascade$ where we process what-if requests and exit the program afterwards.

For the Cascading optimizer, we make use of the data flow optimizer which is a part of Starfish and feed the related interface. When running the Optimizer, we keep the default Optimizer mode as $crossjob + dynamic$.

Algorithm 1 Build Job DAG Pseudo-Code

```
1: procedure BuildJobDAG(flowGraph)
2:     for flow ∈ flowGraph do                          ▷ Iterate over all flows
3:
4:         for flowStep ∈ flow.flowStepGraph do         ▷ Add the job vertices
5:             Create the jobVertex from the flowStep
6:         end for
7:
8:         for edge ∈ flow.flowStepGraph.edgeSet do     ▷ Add the job edges
   within a flow
9:             Create the corresponding edge in the jobGraph
10:        end for
11:    end for

12:    for flowEdge ∈ flowGraph.edgeSet do              ▷ Iterate over all flow edges
   (source → target)
13:        sourceFlowSteps ← flowEdge.sourceFlow.getLeafFlowSteps
14:        targetFlowSteps ← flowEdge.targetFlow.getRootFlowSteps
15:        for sourceFS ∈ sourceFlowSteps do
16:            for targetFS ∈ targetFlowSteps do
17:                Create the job edge from corresponding source to target
18:            end for
19:        end for
20:    end for
21: end procedure
```

2.2.4 Program Interface

The usage of Cascading on Starfish is simple and user-friendly. Users do not need to change the source code or import new package. We can list some cases as follows.

profile cascading jar loganalysis.jar

Profiler: collect task profiles when running a workflow and generate the profile files in $PROFILER\_OUTPUT\_DIR$.

execute cascading jar loganalysis.jar

Execute: only run program without collecting profiles.

*analyze cascading details workflow*_20120217205527

Analyze: list some basic or detailed statistical information regarding all jobs found in the $PROFILER\_OUTPUT\_DIR$

*whatif details workflow*_20120217205527 *cascading jar loganalysis.jar*

What-if Engine: ask a hypothetical question on a particular workflow and return

the predicted profiles.

optimize run workflow_20120217205527 cascading jar loganalysis.jar
Optimizer:Execute a MapReduce workflow using the configuration parameter settings automatically suggested by the Cost-based Optimizer.

2.3 Workflow Evaluation

We evaluate the end-to-end performance of optimizers on seven representative workflows used in different domains. **Term Frequency-Inverse Document Frequency(TF-IDF)**: TF-IDF calculates weights representing the importance of each word to a document in a collection. The workflow contains three jobs: 1) the total terms in each document. 2) calculate the number of documents containing each term. 3) calculate tf * idf. Job 2 depends on Job 1 and Job 3 depends on Job 2. **Top 20 Coauthor Pairs:** Suppose you have a large datasets of papers and authors. You want to know who and if there is any correlation between being collaborative and being a prolific author. It can take three jobs: 1) Group authors by paper. 2) Generate co-authorship pairs (map) and count (reduce). 3) Sort by count. Job 2 depends on Job 1 and Job 3 depends on Job 2. **Log Analysis:** Given an Apache log, parse it with specified format, statistic the minute count and second count separately and dump each results. There are three jobs: 1) Import and parse raw log. 2) group by minutes and statistic counts. 3) Group by seconds and statistic counts. Job 2 and Job 3 depends on Job 1. **PageRank:** The goal is to find the ranking of web pages. The algorithm can be implemented as an iterative workflow containing two jobs:1) Join on the pageId of two datasets.2) Calculate the new rankings of each webpage. Job 2 depends on Job 1. **TPC-H:** TPC-H benchmark as a representative example of a complex SQL query. Query 3 is implemented in four-job workflow. 1) Join the order and customer table, with filter conditions on each table. 2) Join lineitem and result table in job one. 3) Calculate the volume by discount. 4) Get the sum after grouping by some keys. Job 2 depends on Job 1 and Job 3 depends on Job 2 and Job 4 depends on Job 3. **HTML Parser and WordCount:** The workflow processes a collection of web source pages. It has three jobs: 1) Parse the raw data with HTML SAX Parser. 2) Statistic the number of words with the same urls. 3) Aggregate the total word count. Job 2 depends on Job 1 and Job 3 depends on Job 2. **User-defined Partition:** It spill the dataset into three parts by the range of key. Some statistics are collected on each spilled part. In general, it is run in three jobs and each job is responsible for one part of dataset. There is no dependency between those three job, which means three jobs can be run in parallel.

2.3.1 Speedup with Starfish Optimizer

Figure 2.4 shows the timeline for TPC-H Query 3 workflow. When using the profiler, it spends 20% more time. The final cross-job optimizer causes 1.3x speedup. Figure 2.5 analysis the speedup for six workflows respectively. The optimizer is effective for most of workflows with only exception of the user-defined partition. One possible reason is that workflows generates three jobs in parallel which compete the limited cluster resource (30 available map slots and 20 available reduce slots) from each other.

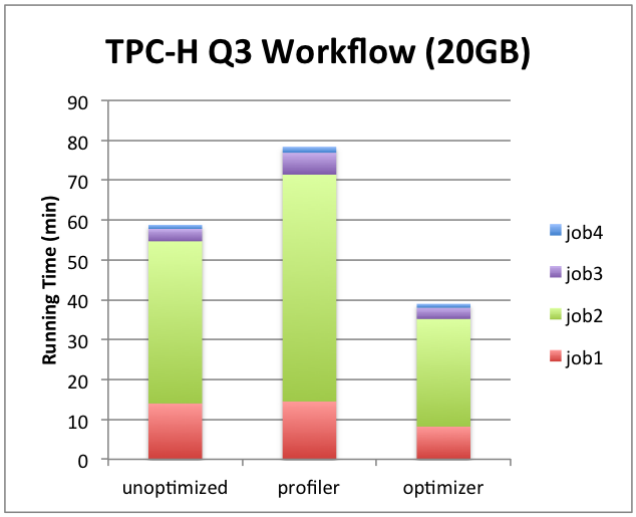


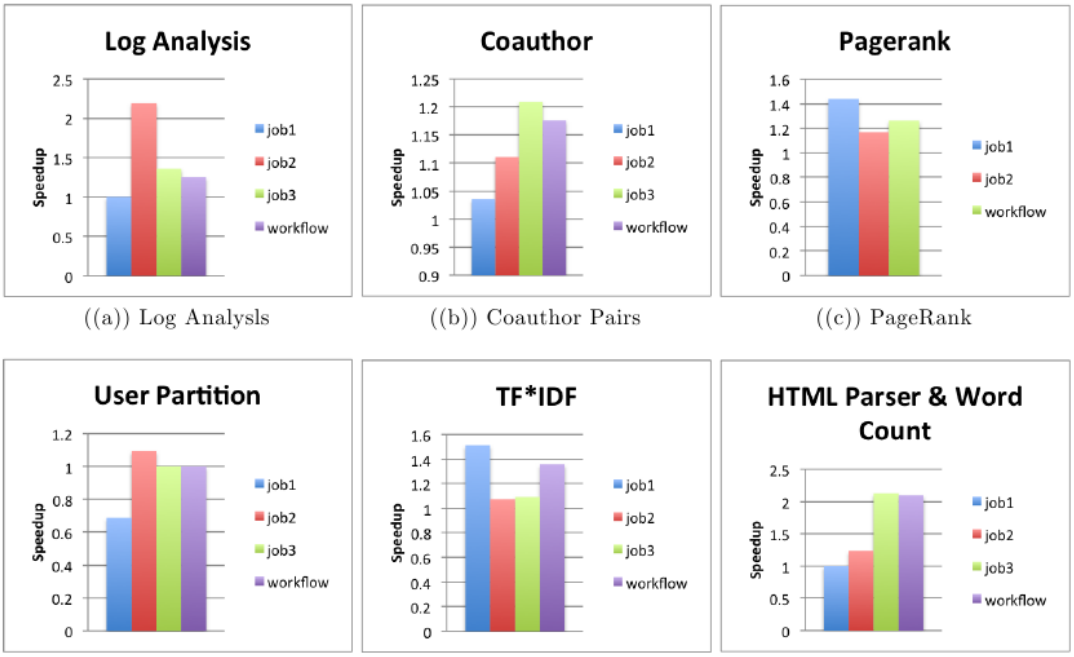
FIGURE 2.4: Run TPC-H query3 with no optimizer, profiler and optimizer.

2.3.2 Overhead on Profiler

Figure 2.6 shows the profiling overhead by comparing againest the same job run with profiling turned off. In average, profiling consumes 20% of the running time.

2.4 Summary

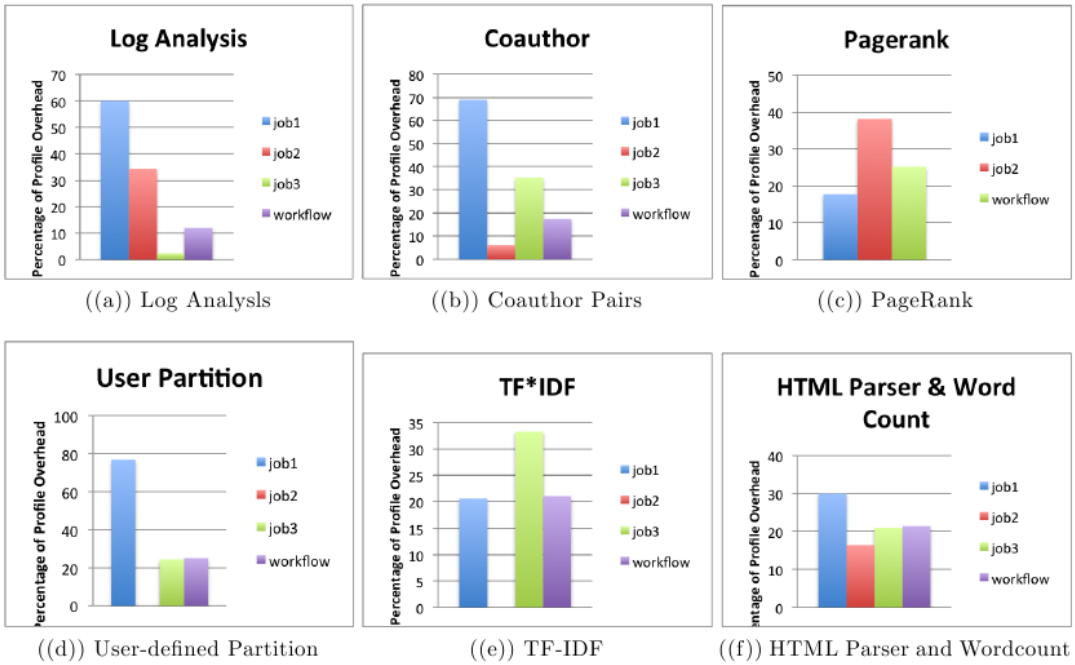
Cascading aims to help developers build pwwerful MapReduce workflow through a well-reasoned API. WIth Starfish Optimizer, we can boost the original Cascading program performance by 20% to 200% without modifying source code. It appears similar syntax as Pig in representation. Considering the coding scale, learning cost and performance, we recommend users to build out applications in Cascading as prototype. If part of the problem is best represented in Pig, we refactor Pig with Cascading. If seeing a solvable bottleneck, we replace that piece of the process with a custom MapReduce job.



((a)) Log Analysls ((b)) Coauthor Pairs ((c)) PageRank

((d)) User-defined Partition ((e)) TF-IDF ((f)) HTML Parser and Wordcount

FIGURE 2.5: Speedup with Starfish optimizer.



((a)) Log Analysls ((b)) Coauthor Pairs ((c)) PageRank

((d)) User-defined Partition ((e)) TF-IDF ((f)) HTML Parser and Wordcount

FIGURE 2.6: Overhead to measure profiler.

3

Optimization for Iterative Workflows

3.1 Iterative Workflows on Hadoop

Many data analysis techniques require *iterative* computations, including PageRank [22], clustering, social network analysis. The techniques has a common trait: data is processed iteratively until the computation satisfies a convergence or stopping condition. Figure 3.1 shows an iterative workflow. It illustrates some data(input/output) is reused during the iterations. The MapReduce framework does not directly support iterations. Instead, developers must implement manually. There are some work on the MapReduce systems that optimize the framework for iterative applications. HaLoop [4] is a modified version of Hadoop that supports iterative large scale applications. Twister [8] is a stream-based MapReduce implementation that support iterative applications. iMapReduce [26] also provides a framework that can model iterative algorithms. Unlike those systems that modify Hadoop or MapReduce model, we propose a simple glue iterative workflow layer on top of the Starfish core that uses cost-based optimization and has zero overhead when instrumentation is turned off.

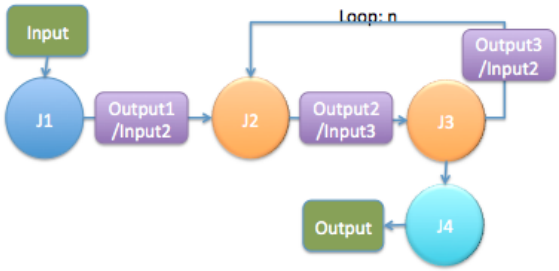


FIGURE 3.1: An iteration workflow

3.2 Design Principles

In Starfish, we have supported a workflow framework that bundles a Job Graph, Configuration Graph. However, here we discuss about a loop instead of fixed jobs in one workflow, which differs at

a) Changing the number of iterations for the optimized job will lead to a different new graph; which in turn means that the optimizer will not work.

b) Each iteration has the same input and/or output paths. In this case, creating a graph based on paths will not work.

There are different scenarios to consider to implement an iterative workflow optimizer:

1. The driver will launch a fixed number of MapReduce jobs in order. The profiler will collect all profiles. Now, we need to give the optimizer all profiles. One solution would be to extend the interface to accept multiple profiles as input. Another would be for the profiler to generate a graph at the end and then use the dataflow optimizer.

2. The driver will launch multiple jobs from a loop. If all jobs have similar execution behavior, then using one profile for optimization would work. Starfish might already support this.

3. The driver is using JobControl to build a graph (not necessarily a chain) of jobs to submit. Here, we need to build the profile graph and follow the dataflow route. It would be similar to the work done for Pig and Cascading.

Here we pick the second one as an implementation. A more general solution would be to introduce the notion of an "iterative job". An iterative job could have a single profile or a linear set of rotating profiles. Using the above example above, the iterative job would have profiles $P_2 -> P_3$. If the optimized job has two iterations, P_2 will be used first, then P_3. If the optimized job has four iterations, P_2 is used first, then P_3, then P_3, and finally P_3.

For optimizing new iterations of the job, we can follow one of the following approaches:

A) Use P_2 and P_3 to create an average profile P'. Use P' to optimize each new iteration of the job.

B) Use P_2 to optimize iteration 1 of the job. Use P_3 to optimize iteration 2 of the job. Use P_2 to optimize iteration 3 of the job. Use P_3 to optimize iteration 4 of the job. And so on.

There are pros and cons for using either approach. We pick approach B as it offers more flexibility and can potentially capture more subtleties across iterations.

3.2.1 Schema Design

The design is inspired by Oozie, which allows one to combine multiple Map/Reduce jobs into a logical unit of work, accomplishing the larger task. Oozie workflow is a collection of actions (i.e. Hadoop Map/Reduce jobs, Pig jobs) arranged in a control dependency DAG (Direct Acyclic Graph), specifying a sequence of actions execution. While it does not support iterative jobs. We extend the workflow description in XML to support this type. This is a template for creating such workflow.

```
<?xml version="1.0"?>
<workflow name="${workflowName}">
<global>
    <job-tracker>${jobTracker}</job-tracker>
    <name-node>${nameNode}</name-node>
</global>

<mapreduce>
    <mapper>${map.class}</mapper>
    <reducer>${reduce.class}</reducer>
    <configuration>
        <property>
            <name>${key}</name>
            <value>${value}</value>
        </property>
    </configuration>
</mapreduce>

<iteration loop="${iteration_times}">
    <mapreduce>....</mapreduce>
    <mapreduce>...</mapreduce>
</iteration>
</workflow>
```

The Alidade workflow topology is shown as following:

```
<?xml version="1.0"?>
<workflow name="Alidade">
<global>
    <job-tracker>${jobTracker}</job-tracker>
    <name-node>${nameNode}</name-node>
</global>

<mapreduce>
    <mapper>RawInputMapper</mapper>
```

```
    <reducer>PreprocessReducer</reducer>
    <configuration>
        <property>
            <name>mapred.input.dir</name>
            <value>${hdfs_input_dir}</value>
        </property>
    </configuration>
</mapreduce>

<iteration loop="10">
    <mapreduce>
        <mapper>ObservationRecordMapper</mapper>
        <reducer>DirectReducer</reducer>
        <configuration>
            <property>
                <name>mapred.input.dir</name>
                <value>${hdfs_input_dir}</value>
            </property>
        </configuration>
    </mapreduce>
</iteration>
</workflow>
```

3.3 Workflow Optimizer based on Starfish

3.3.1 Logic Flow

Figure 3.2 and Figure 3.3 presents logic flow of our system.

3.3.2 Usage

The usage of iterative workflow optimizer is simple and user-friendly. We list some operators as follows:

1. Profiler:

```
profile iterative_dataflow topology.xml jar $ALDADE_HOME/AlidaeVersion2.jar \
pMapReduce.SmartRunner $CONFIG_OPTS $COMMAND_OPTS

analyze iterative_dataflow list_all
analyze iterative_dataflow details workflow_20120330055215

optimize iterative_dataflow topology.xml jar $ALDADE_HOME/AlidaeVersion2.jar
pMapReduce.SmartRunner $CONFIG_OPTS $COMMAND_OPTS
```

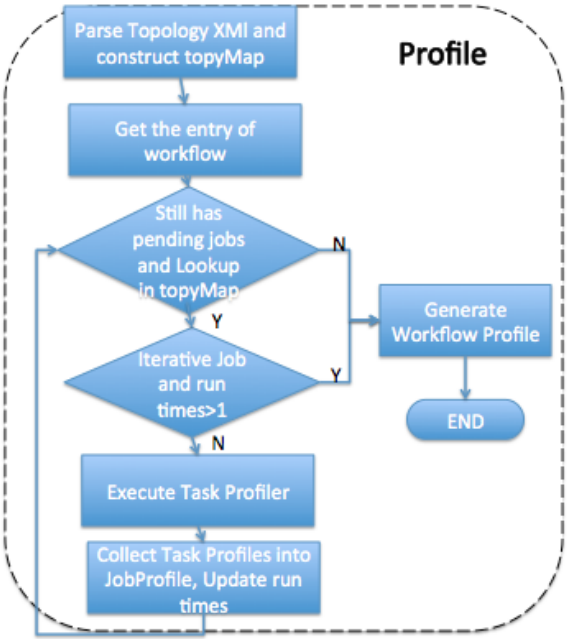


FIGURE 3.2: Profiler logic flow

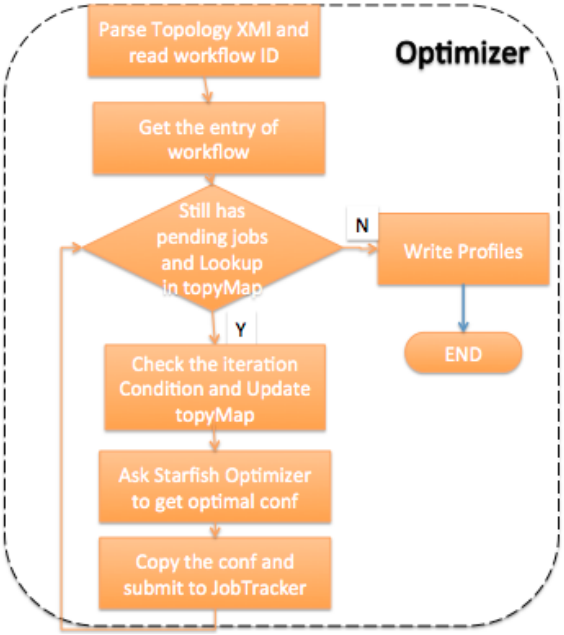


FIGURE 3.3: Optimizer logic flow

3.4 Evaluation

3.4.1 PageRank Introduction

PageRank is the link-based analysis algorithm used by Google to rank webpages was at first present as citation ranking [22]. In a nutshell, a link from a page to another is understood as a recommendation and status of recommender is also very important. Page et al. ranking formula started with a simple summation equation as in Equation (3.1). The PageRank of node u is denoted as $PR(u)$, is the sum of the PageRanks of all nodes pointing into node u where B_u is the set of pages pointing into u. Finally $N(v)$ is the number of outlinks from node v.

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{N(v)} \tag{3.1}$$

The problem with Equation (3.1) is that the ranking value $PR(v)$ is not known. Page et al. use iterative procedure to overcome this problem. All node is assign with initial equal PageRank like 1. Then rule is successively applied, subtituting the values of the previous iterate into $PR(u)$. This process will be repeated until the PageRank scores will eventually converge to some final stable values. The formula representing the iterative procedure is as in Equation (3.2). Each iteration is denoted by value of k. $k+1$ means next iteration and k is previous iteration.

$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{N(v)} \tag{3.2}$$

3.4.2 PageRank Evaluation

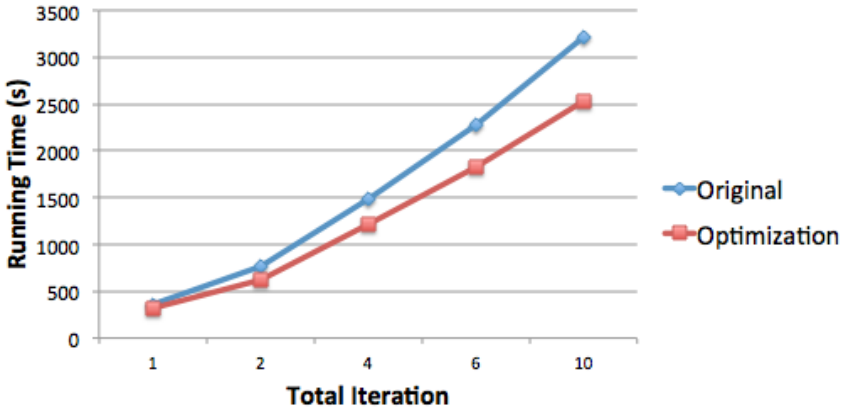


FIGURE 3.4: Iterative workflow evaluation

For time cost evaluation, Figure 3.4 shows that iterative workflow optimizor performs better (more than 10% speedup) than default default settings.

4

Optimization on Key-value Stores

4.1 HBase Overview

The HBase is a faithful, open source implementation of Googleś Bigtable [5]. Basically it is a distributed column-oriented database built on top of HDFS. HBase is the Hadoop application to use when you require real-time read/write random-access to very large datasets and it provides a scalable, distributed database. See Figure 4.1 to present the HBase cluster architecture. In HBase, there are some concepts.

Tables: HBase tables are like those in an RDBMS, only cells are versioned, rows are sorted, and columns can be added on the fly by the client as long as the column family they belong to prefix exists. Tables are automatically partitioned horizontally by HBase into regions.

Rows: Row keys are uninterpreted bytes. Rows are lexicographically sorted with the lowest order appearing rst in a table. The empty byte array is used to denote both the start and end of a tables' namespace.

Column Family: Columns in HBase are grouped into column families. All Column family members have a common prefix. Physically, all column family members are stored together on the file system. Because tunings and storage specifications are done at the column family level, it is advised that all column family members have the same general access pattern and size characteristics.

Cells: A row, column, version tuple exactly species a cell in HBase. Cell content is uninterpreted bytes.

Regions: The Tables are automatically partitioned horizontally by HBase into regions and each of them comprises a subset of a tables rows. HBase characterized with an HBase master node orchestrating a cluster of one or more regionserver slaves. The HBase Master is responsible for boot strapping, for assigning regions to registered regionservers, and for recovering regionserver failure. HBase keeps special catalog tables named -ROOT- and .META, which maintains the current list,

state, recent history, and location of all regions afloat on the cluster. The -ROOT- table holds the list of .META table regions. The .META table holds the list of all user-space regions. Entries in these tables are keyed using the regions start row.

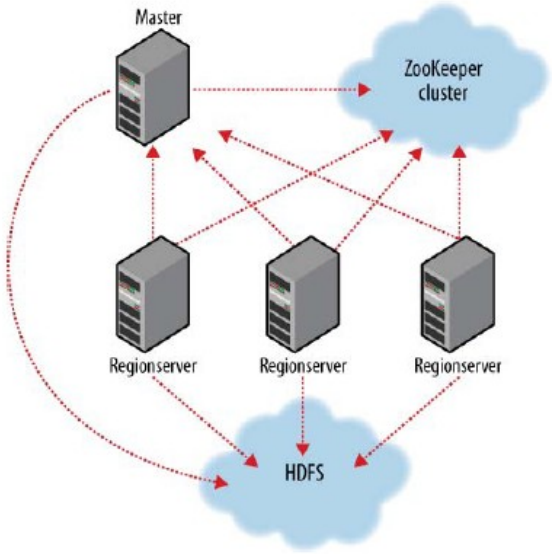


FIGURE 4.1: HBase cluster architecture

4.2 HBase Profile

Alidade relies on HDFS and HBase for data storage. HBase records important calculation results such as intersection area, target geolocation. We also want to measure the cost for HBase usage. Hence, we develop the HBase Profiler.

The principle is to hook HBase client API such as put(), get() when HBase is enabled. At the entry of methods, we will capture the input size and starting time. When the function returns, we update the counter and finally statistics below metrics after MapReduce jobs finish.

The task profiler populates the metrics:

```
HBASE PUT_DURATION 36002716
HBASE PUT_COUNT 3435
HBASE PUT_BYTE_COUNT 5789092
HBASE GET_DURATION 162122328
HBASE GET_COUNT 23232
HBASE GET_RESULT_COUNT 27692
HBASE GET_RESULT_BYTE_COUNT  3889092
```

When the job finished running, the job profile will gather the task profile together and export as a XML report.

```
<counter key="HBASE_PUT_SIZE" value="3435"/>
<counter key="HBASE_PUT_BYTES" value="4780"/>
<counter key="HBASE_GET_SIZE" value="23232"/>
<counter key="HBASE_GET_RESULT_SIZE" value="27692"/>
<counter key="HBASE_GET_RESULT_BYTES" value="3889092"/>

<cost_factor key="HBASE_PUT_COST" value="106064.0"/>
<cost_factor key="HBASE_GET_COST" value="212321.0"/>

<timing key="HBASE_PUT" value="36.633596"/>
<timing key="HBASE_GET" value="74.985654"/>
```

How to integrate the HBase profile with Starfishś cost-based model is still a pending problem.

4.3 Memory Allocation

Process	Heap	Description
Namenode	2G	about 1G for each 10TB data
SecondaryNameNode	2G	Applies the edits in memory, needs about the same amount as the NameNode
JobTracker	2G	Moderate requirement
HBase Master	4G	Usually light loaded
DataNode	1G	Moderate requirement
TaskTracker	1G	Moderate requirement
Task Attempts	1G	Multiply by the maximum number alllowed
HBase Region Server	8G	Majority of available memory
Zookeeper	1G	Moderate requirement

Table 4.1: Memory allocation per Java process for a cluster

Suggested by the HBase book [10], Table 4.1 shows a basic distribution of memory to specific processes. The setup is as such: for the master machine, running the NameNode, Secondary NameNode, JobTracker and HBase Master, 17 GB of memory; and for the slaves, running the DataNode, TaskTrackers, and HBase RegionServers.

We note that setting the heap of region servers to larger than 16GB is considered dangerous. Once a stop-the-world garbage collection is required, it simply takes too long to rewrite the fragmented heap.

4.4 HBase Performance Tuning

4.4.1 Optimization Space

1. Operation System: ulimit, nproc, JVM Settings

2. Hadoop Configuration: xciever, handlers

3. HBase Configuration: garbage collections, heap size, block cache.

4. HBase Schema: row key, compression, block size, max filesize, bloomfilter

5. HBase processes: split, compacting

4.4.2 JVM Setting

Hadoop JVM instances was set to run with "-server" option with a HEAP size of 4GB. HBase nodes ran with 4GB heap with JVM settings.

"-server -XX:+UseParallelGC -XX:ParallelGCThraed=8 -XX:+AggressivHeap -XX:+HeapDumpOnOutOfMemoryError".

The parallel GC leverages multiple CPUs.

4.4.3 Configurations Settings

Some of the configurations carried out in Table 4.2

Configuration File	Property	Description	Value
hdfs-site.xml	dfs.block.size	Lower value offers more parallelism	33554432
	dfs.datanode.max.xcievers	upper bound on the number of files that it will serve at any one time	4096
core-site.xml	io.file.buffer.size	Read and write buffer size. By setting limit to 16KB it allows continous streaming	16384
hbase-site.xml	hbase.regionserver.handler.count	RPC Server instances suun up on HBase regionservers	100
	hfile.min.blocksize.size	small size increases the index but reduces the less fetch on a random access	65536
	hbase.hregion.max.filesize	Maximum HStoreFile size. If any HStoreFiles has grown to exceed this value, the hosting HRegion is split in two.	512M
	zookeeper.session.timeout	ZooKeeper session timeout.	60000
	hfile.block.cache.size	Percentage of maximum heap to allocate to block cache used by HFile/StoreFile.	0.39

Table 4.2: HBase configuration tuning

4.4.4 HBase Operation Performance

Figure 4.3 describes the basic operation performance under the workload of 10M records on 10 m2.xlarge nodes.

Operation	Throughput	Latency(ms)
random read	2539	3.921
random insert	46268	0.2
random update	12361	0.797
read-modify-write	4245	2.214
scan	592	16.59

Table 4.3: Performance test in HBase

Writing

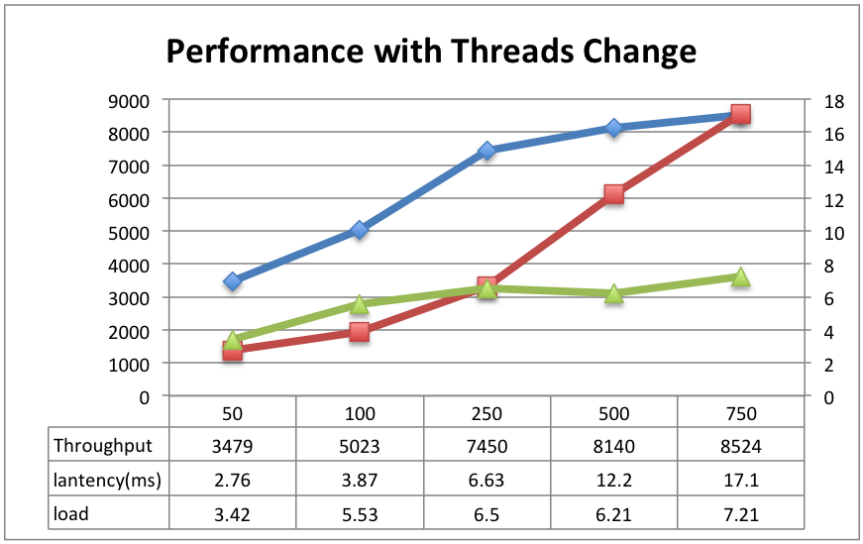


FIGURE 4.2: HBase write performance change with threads

From Figure 4.2, we can see that when client threads reach 250, the response time is around 6ms and throughput is 7000. While we find some fluctuations during the running. In order to tune HBase, we choose six different scenarios to test the write speed. The following two graphs show testing results with client or server-side settings that may affect speed and fluctuation.

The first test case is using the default settings. From the first graph, it shakes from 5ms to 25ms, and lasts several seconds on waiting. For the online business, it is not acceptable. We also meet a common problem of HBase, that is when writing an empty table, the stress would aggregate into one region server for a long period of time, that waste cluster cability. It can be solved by pre-splitting regions.

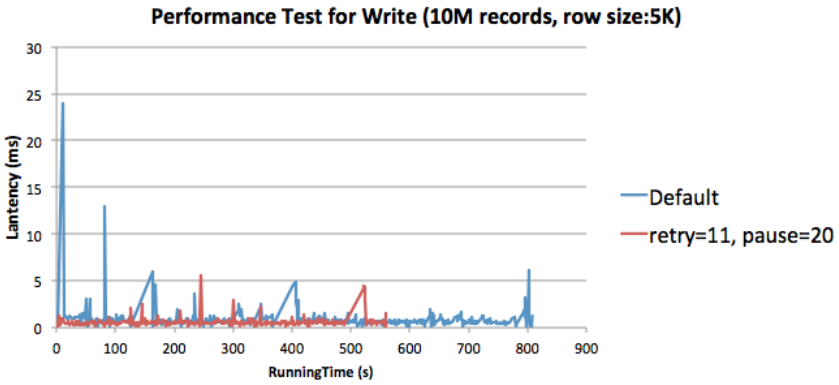


FIGURE 4.3: HBase performance test with different client parameters

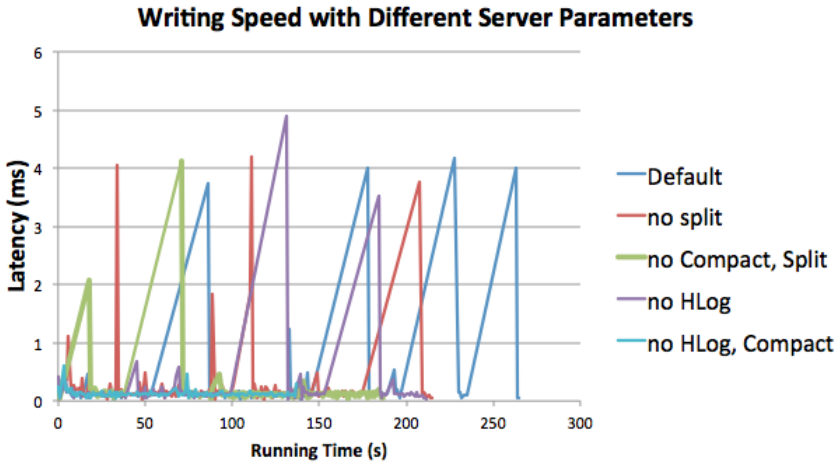


FIGURE 4.4: HBase performance test with different server parameters

After reading the source code in HBase client, it will retry several times when writing. The strategy is to increase the step exponentially to retry. For example: it starts from 1 second and retry 10 times. Hence, the interval sequence is [1,1,1,2,2,4,4,8,16,32]. If server occurs some exceptions, clients will wait for long time. Therefore one solution is to limit the pause time to 20 ms (hbase.client.pause), retry times is set 11 times. Within those, it can guarantee clients retry in 2s. and set $hbase.ipc.client.tcpnodelay = true$, $ipc.ping.interval = 3s$ that can avoid long time waiting. After applying those solutions, we can see that although writing still sometimes fluctuates, the peak is under 5 ms which is much better than before.

After tuning the client, we should also tune the server. By mining the server logs, we find one key factor is split. If disable split, fluctuation frequency is less than baseline, but it still happens at some time.

Another problem is with more data writing, server will compact larger files, then the disk IO is bottleneck during compacting phase. Then we experiment to disable

compact. In the first test of closing compact, the fluctuation range becomes larger and logs show "Blocking updates for" and "delay flush up to". Actually when the memstore flush as a region, it will first check the number of store files. If there are too many files, it will first execute memstore and defer flushing. And in the process of deferring, it may lead to generate more data in memstore. If the memstore size is larger than 2X space, it will block wirte/delete request until the memestore size becomes normal. We modify the related parameter:

$$base.hstore.blockingStoreFiles = Integer.MAX\_VALUE$$

$$hbase.hregion.memstore.block.multipiler = 8$$

When testing another region server with many regions, we find that though disable compact/split and some configuration changes, it still has large fluctuation. One reason is it maintains too many regions, which can be solved by decreasing region numbers or memstore size; Another reason is HLog. HLog will asks memstore to flush regions regardless of the size of the memstore. The final graph shows closing HLog can reduce fluctuation even more.

Some other parameters that affects writing:

1. Batch Operations
2. Write Buffer Size
3. WAL (HLog)
4. AutoFlush

   WAL=false, autoFlush=false, buffer=25165824 insert costs: 0.058ms

   WAL=true, autoFlush=false, buffer=25165824, insert costs: 0.081ms

   WAL=false, autoFlush=true, insert costs: 0.75ms

   WAL=true, autoFlush=true, insert costs: 1.00ms

Our experiment shows that without autoFlush, the insert can be 10X faster.

Reading

We continue to test read performance. Figure 4.5 show read throughput in default is 2539 while 7385 with Bloomfilter. We can conclude that using bloom filters with the matching update or read patterns can save much IO.

4.4.5 Compression

HBase comes with support for a number of compression algorithms that can be enabled at the column family level. We run 10Million records to insert with different compression algorithms.

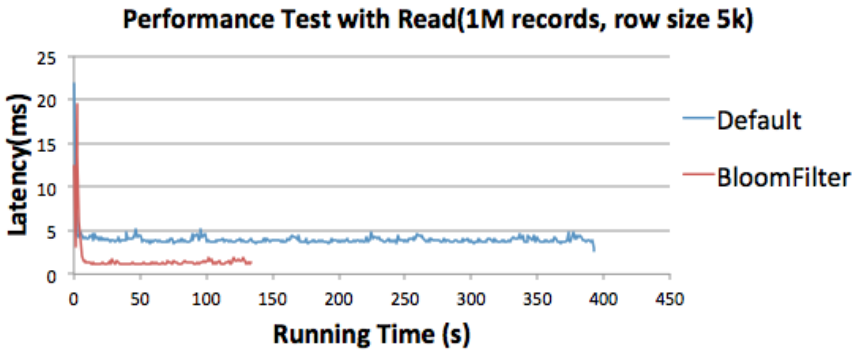


FIGURE 4.5: HBase read performance

Algorithm	Insert Time (ms)
NONE	0.076
GZIP	0.150
LZO	0.12
SNAPPY	0.086

Table 4.4: Compression performance in HBase

4.4.6 Costs of Running Benchmark

Benchmarking can take a lot of time and money; there are many permutations of factors to test so the cost of each test in terms of setup time(launch services and load data) and compute resources used can be a real limitation. The table 4.6 shows the test duration and AWS cost at the normal list price. This cost could be reduced by using spot pricing, or by sharing unused reservations with the production site. Figure 4.6 shows the monetary cost when we run those experiments.

	10 m1.large	20 m1.large	10 m2.xlarge
Write Capacity	43593/s	87012/s	58273/s
Storage Capacity	8.5T	17T	17T
Nodes cost per hour	$3.4	$6.8	$5.4
Traffic Cost	$4	$8	$4
Setup Duration	2hr	2hr	2hr
AWS Billed Duration	19hr	11hr	12hr
Total Cost	$68.6	$82.8	$68.8

FIGURE 4.6: EC2 cost for running experiments

4.5 Rules and Guidelines

Based on the above tests and my understanding of Alidade model, we can extract some best practice:

- Recommend c1.xlarge, m2.xlarge or even better node type ro run HBase. Isolate HBase cluster to avoid memory competition with other services.
- Some important factors affect writing speed. Writing HLog > Split > Compact.
- When HBase writes, it is hard to get rid of "shaking", the factors are ranked as Split > Writing hlog > Compact
- If applications do not need "delete" operation, we can shutdown compact/split which can save some cost.
- In reducer phase, Alidade writes each result into HDFS and HBase. If we use batch operation or set autoFlush=false and proper buffersize, it speeds up to 10X of the default condition.
- For high-frequency write applications, one region server should not have too many regions. It depends on row size and machine memory.
- If applications do not require strict data security, closing HLog can get 2X speedup.
- hbase.regionserver.handler.count (default is 10). Around 100 threads are blocked when load increases. If we set it to 100, it is fair and then throughput becomes the bottleneck.
- If not considering prespliting region, we can manually move and design proper row key to distribute evenly onto region servers.
- Split takes about seconds. If visiting region when in splitting phase, it may throw NotServingRegion exception.
- Compression can save space on storage. Snappy provide high speeds and reasonable compression.
- In a read-busy system, using bloom filters with the matching update or read patterns can save a huge amount of IO.

5

A Real-life Application on Hadoop

5.1 Alidade Overview

Alidade is a constraint based geolocation system. In common with other geolocation systems like Octant [24], Alidades goal is to accurately predict the geographical location of any given Internet IP address. For example, given a query for the IP address 152.3.215.24, Alidade should present an answer such as "Durham, NC, or perhaps an even more specific answer, "Duke West Campus". Alidade is designed to operate on data at a very large scale. In order to achieve that, it employs a Hadoop cluster of servers, allowing it to process much larger data sets. Furthermore, Alidade has no notion of a single target address. Instead, based on the entirety of its input data, Alidade can geolocate every IP address in a trace no matter it has some measurements or not.

5.1.1 Terminology

- Target: Name/IP which Alidade attempts to geolocate.
- RTT/Latency: round-trip time, the time required for a packet to travel from the source to the destination and back.
- Landmark: a.k.a: beacon. IPs with "ground-truth" information on their location.
- IntersectionWritable: structure defines data transferred from Reduce Stage to final MapReduce output. In particular, each instance contains information pertaining to a target's localization results.

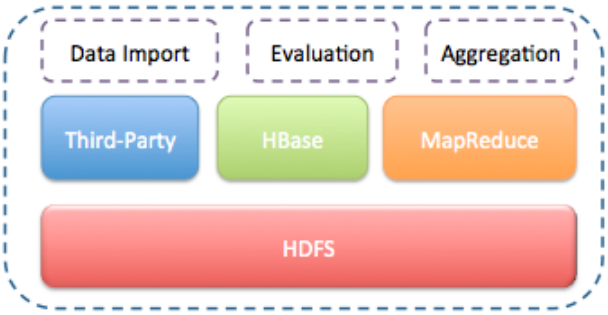


FIGURE 5.1: Alidade architecture

5.1.2 Architecture

Figure 5.1 illustrates the architecture of Alidade, a component of the open source MapReduce implementation Hadoop. In general, Alidade has two phases.

- Preprocessing data
- Iterative geolocation

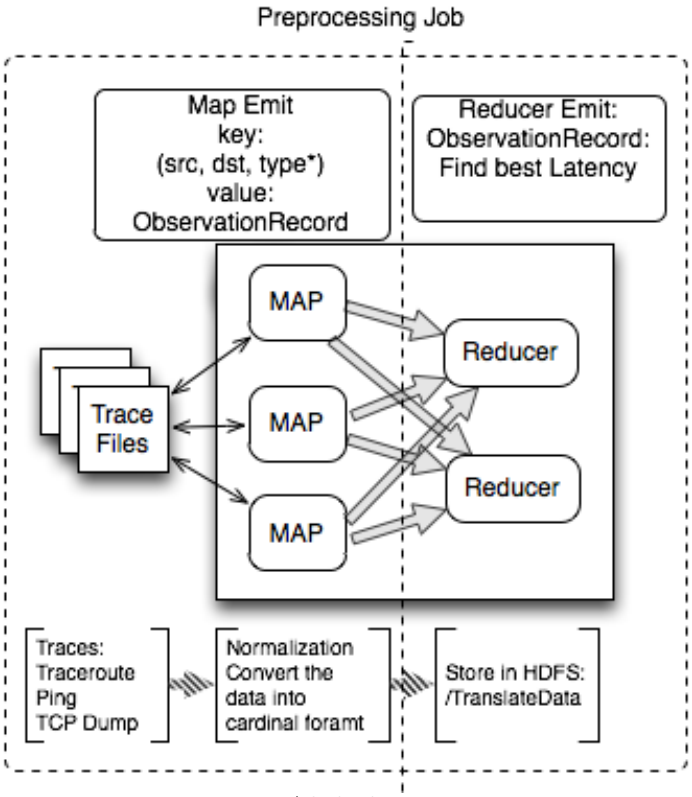


FIGURE 5.2: Alidade preprocessing

Figure 5.2 shows the Preprocssing Job. It reads from diverse data sets viz. *traceroutes*, *ping*, and *TCPstats*. The import is followed with a transformation to an internal canonical format. The transformation process finds the "best latency" for any given landmark-target pair. The strategy to choose the "best RTT" is to apply statistical method to reasonable low RTT. It does not guarantee to pick the minimum value because network measurement are buggy.

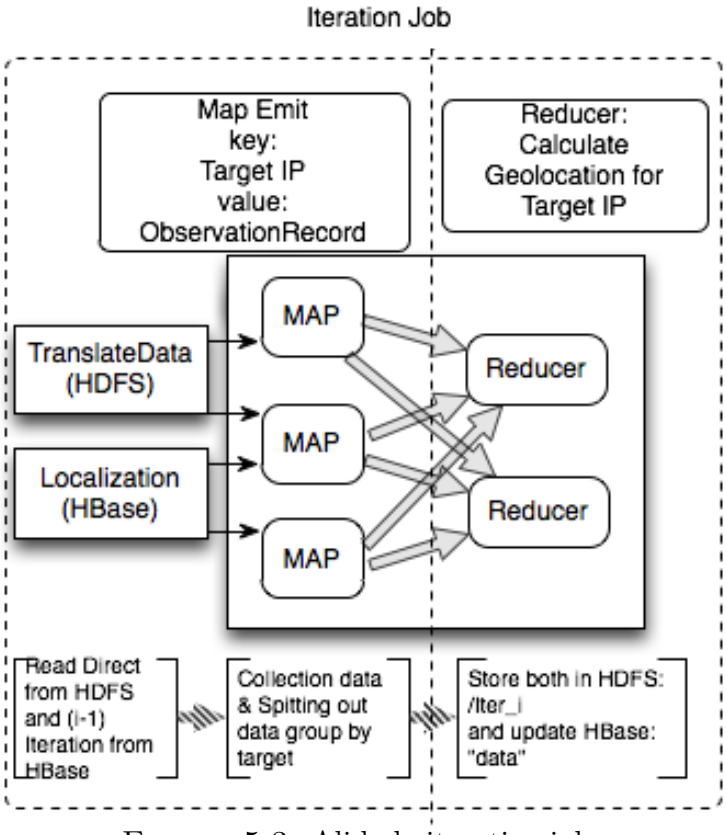


FIGURE 5.3: Alidade iterative jobs

Figure 5.3 shows the iterative Jobs. The iterative program that Alidade develop can be distilled into the following core construct:

$$I_{i+1} = I_0 \cup (I_i \bowtie R)$$

where I_0 is an initial result and R is an invariant relation. It terminates when a fixpoint is reached. Here we manually set $i = 10$.

5.1.3 Geolocation Calculation

In order to calculate the distance between two targets, we can keep track of each RTT. There are two kinds of RTTs, one is called Direct RTT that represents records

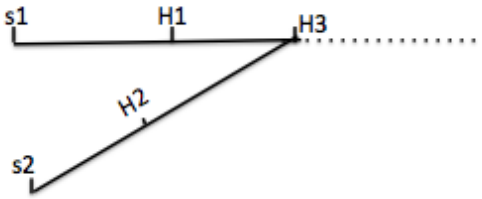


FIGURE 5.4: Alidade path example

that can be measured from a landmark. The other is Indirect RTT that can not be measured from a landmark, while it can be inferred from a Direct RTT. Take Figure 5.4 for an example:

- Directed Records:
  - S1 - H1
  - S1 - H3
  - S2 - H2
  - S2 - H3
- Indirected Records:
  - H1 - H3
  - H2 - H3

Initially, we already know the Landmark location. We can propose the following equation to calculate directed targets.

$$Distance = RTT/2 \times Speed$$

Where *Speed* is the packet transferred speed in Internet that at most 2/3 of the Speed of light. We take the maximum speed because we can cover all possible area around this point.

For the indirect records, we can infer latency from subtraction. i.e.: $latency_{13} = latency_3 - latency_1$.

In the first iteration, we have collected all direct latency and calculate related distance from source points. Since source points location are accurate, we can locate the target region with some geometric knowledge. Figure 5.5 shows that target get the distance of $d1$ from $P1$ and distance of $d2$ from $P2$. The shadow should be the target location area. In fact, Alidade describes the intersection through a set of sample points. When it goes to Iteration 2, $P3$ could appear after the first iteration. Based on Indirect RTT, we can extend the boundary and update the intersection. Compared to Iteration 1, the intersection area is shrinked in Figure 5.6.

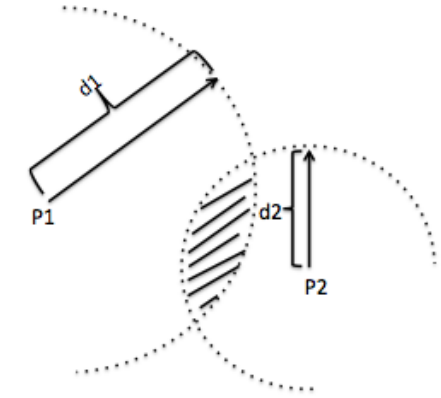


FIGURE 5.5: Alidade intersection example, Iteration 1

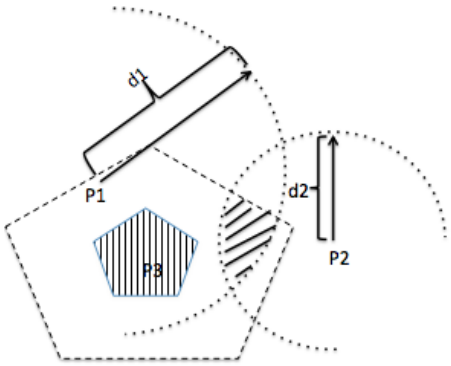


FIGURE 5.6: Alidade intersection example, Iteration 2

After more iterations, the final result will present a acceptablely small region. Alidade supports some evaluation tools to measure the presicion and efficiency of the iterative algorithm.

5.1.4 Cluster Specification

Specification for the Alidade clusters:

- 30 1x2 servers
- 10 1x8 servers

The 1x2 servers consist of

- 1 quadcore AMD 1389 processor
- 8 Gb RAM

– 2x 250Gb HDD

The 1x8 servers consist of

– 1 quadcore Intel Xeon x3440 processor with hyperthreading

– 16Gb RAM

– 8x600Gb HDD

HBase Configuration:

`hbase.hregion.memstore.block.multiplier`	`4`
`hbase.hstore.blockingStoreFiles`	`20`
`hbase.regionserver.handler.count`	`40`
`hfile.block.cache.size`	`0.2 (default)`
`zookeeper.session.timeout`	`1800000 (in ms)`
`hbase.hregion.max.filesize`	`1073741824 (1 GB)`
`hbase.regionserver.codecs`	`snappy`
`hbase.regionserver.maxlogs`	`16`

5.2 Challenges

Alidade meets three big challenges when running.

1. Heavy Computation

Alidade runs iterative jobs to get the appropriate area for each target. In each iteration, operations like calculating intersections or converting a set of longitude, latitude to an area need much computation based on linear algebra and geometry. Unlike efficient library in Matlab, Alidade is implemented in pure java.

2. Large Scale

Alidade data sets are huge and informative. The number of records varies from 160,000,000 to 700,000,000. It takes days to run iterative jobs.

3. Limited Resource Allocation

Cluster resource is limited. It depends on many services such as HDFS, JobTrackers, TaskTrackers, HBase, Customed Jobs. Some jobs or services are blocked or terminated by other services.

5.3 Experiment Environment

– Cluster Type1: m1.large 10-20 nodes. 7.5 GB memory, 2 virtual cores, 850 GB storage, set 3 map tasks and 2 reduce tasks concurrently.

– Cluster Type2: m2.xlarge 10 nodes: 17.1 GB of memory, 6.5 EC2 Compute Units (2 virtual cores with 3.25 EC2 Compute Units each)

– Hadoop Version1: Apache 0.20.205.

– Hadoop Version2: CDH3U3 (Cloudera Update Version) based on 0.20.2.

– HBase Version1: Apache 0.90.4.

– HBase Version2: CDH3U3 (support LZO, SNAPPY compression library).

– Data Set: logs for 30 days, each day has 8G raw data.

The cluster topology is shown as 5.7

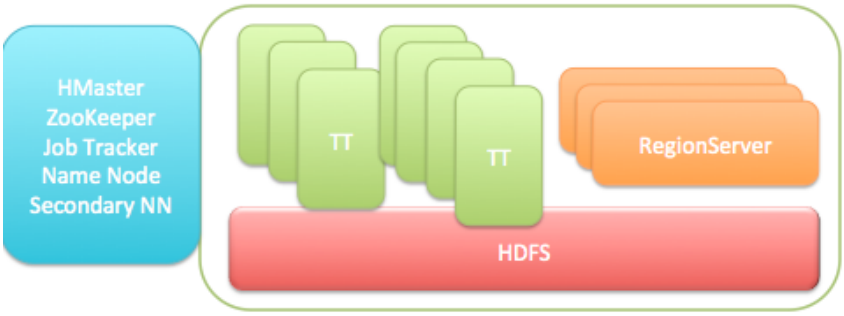


FIGURE 5.7: Alidade cluster topology

5.4 Alidade Evaluation

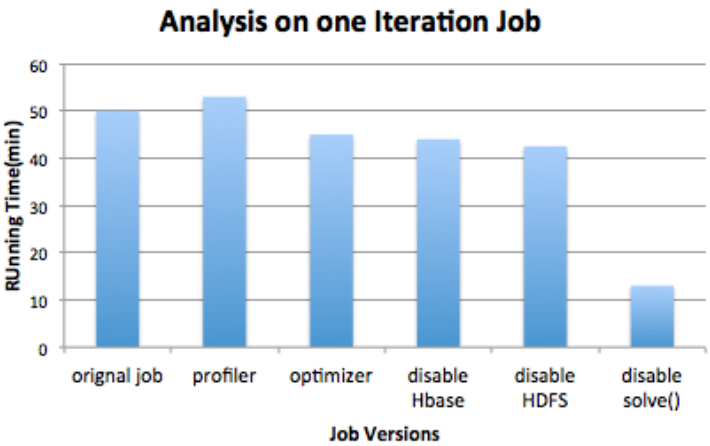


FIGURE 5.8: Performance analysis for one Iteration Job

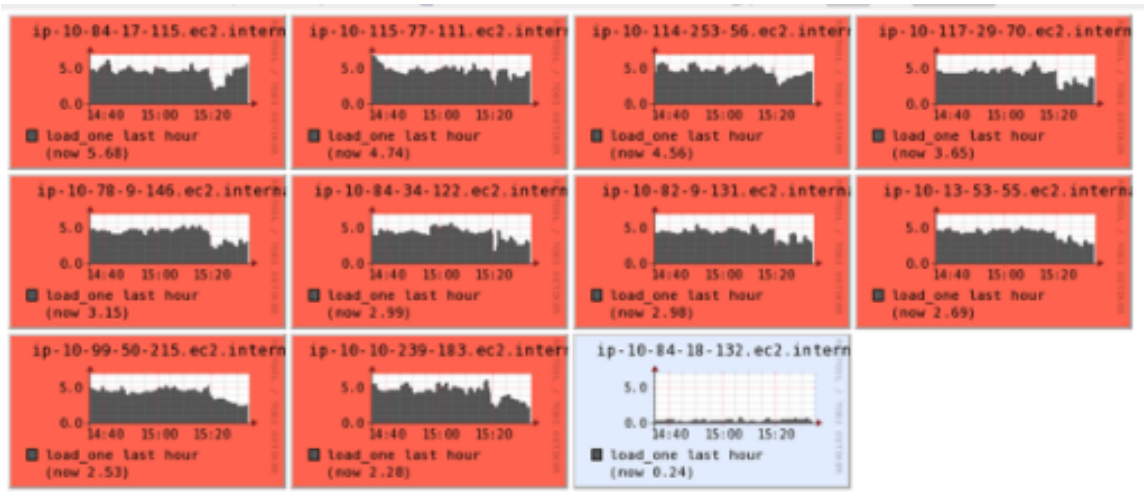


FIGURE 5.9: Cluster load monitor using Gangia

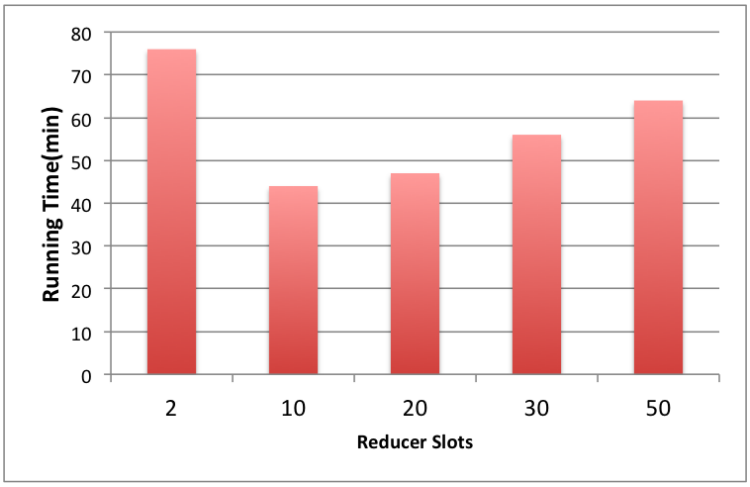


FIGURE 5.10: Number of reduce slots effect Alidade performance

After the cost-based optimization, Alidade can improve 10-30% performance shown in Figure 5.11. In Figure 5.8. Optimizer contributes much (cut off 30% time) on the first job which function is to translate data into proper type. An interesting observation is when we disable the solve function which is the core logic in Alidade, it can immediately save about 70% of total time. When we monitor the cluster via Ganglia 5.9, it shows the average load for host is 3.7, which is dangerous for machine liveness. We notice that the reduce phase is compute-intensive. In order to evaluate the reduce factor, we also modify the number of reducer slots in conf/mapred-site.xml, the results Figure 5.10 show that it is not always good to increase machine reduce slots because it depends on machine CPU number and other resources. Compared with some typical workflows we present in Chapter 2, we can conclude that currently Starfish Optimizer is better fitted for I/O intensive jobs

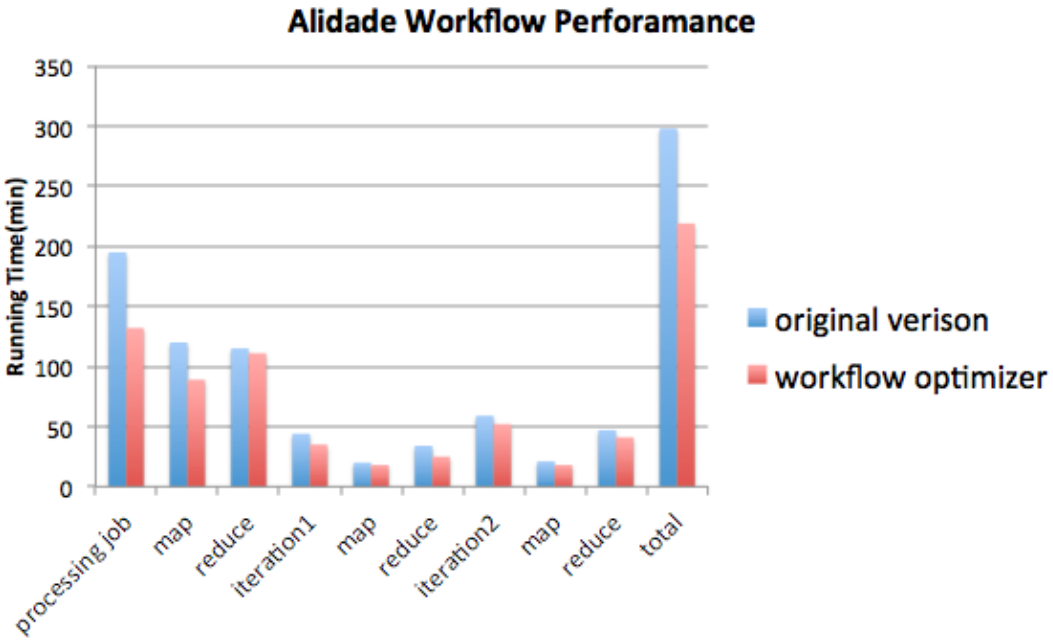


FIGURE 5.11: Alidade performance comparison using workflow optimizer

instead of CPU-intensive jobs.

Another issue Alidade workflow facing is the reliability. When running Alidade in the EC2 environment, it throws some exceptions, such as the following:

```
Error: java.lang.OutOfMemoryError: Java heap space

org.apache.hadoop.hbase.client.RetriesExhaustedException: \
 Trying to contact region server ip-10-99-50-215.ec2.internal:60020 \
  for region exp_0322, but failed after 10 attempts.

Exceptions:
java.net.ConnectException: Connection refused
```

To solve the problem, we isolate HBase cluster to avoid memory competition with other services and tune some Hadoop and HBase configurations as shown in chapter 4. After those optimization, the errors go away.

6

Conclusion

In this thesis, we have described our approach to optimize MapReduce workflows in an automatic way using cost-based model and rule-based technology.

The first contribution is automatic tuning of workflow. We choose Cascading which is a Java API for building a DAG of MapReduce jobs. Using an extended Starfish Optimizer, we can boost the original Cascading program by 20% to 200% without modifying any source code.

The second contribution is to optimize a real-life workflow that has heavy computational jobs and relies on HBase. We develop an iterative workflow optimizer and successfully apply it in the Alidade system.

The third contribution is we leverage the cost-based optimizer and rule-based optimizer to get better performance for workflows. After finding the existing bottleneck in Alidade workflow, we pick appropriate combination of clusters, in-memory settings, HBase configurations to achieve a reliable state. Finally, we summarize some optimization rules that can be applied in practice.

For future work, we can continue to involve HBase, Hive or other sub systems into Starfish roadmap. We will develop related profiler to exactly capture the running behavior. Furthermore, diagnosing the cause of performance problems is a challenging exercise in workflow. If we have a diagnosis tool that makes use of different log sources and profiles and present in a nice visual way, it will help developers to improve their productivity to find the root of problem.

Through these research efforts, we obtained fruitful results in developing novel techniques to extend and enhance the large-scale data processing capability of Hadoop. As cloud computing becomes more and more popular in the academia and industry, we believe that there are promising opportunities in further extending the data processing capability of Hadoop on clouds for a wide spectrum of usage scenarios.

Bibliography

[1] *Amazon Elastic MapReduce*. http://aws.amazon.com/elasticmapreduce.

[2] *Amazon Web Service*, 2012. http://aws.amazon.com/.

[3] *Karmasphere pushes new big data workflow*, 2011. http://karmasphere.com/In-The-News/gigaom-karmasphere-pushes-new-big-data-workflow.html.

[4] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1), 2010.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[6] *Cloudera: 7 tips for Improving MapReduce Performance*, 2009. http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/.

[7] *Amazon Elastic Cloud Computing*, 2012. http://aws.amazon.com/ec2/.

[8] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[9] *2011 digital universe study: extracting value from chaos*, 2011. http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.

[10] L. George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[11] *Apache Hadoop*, 2012. http://hadoop.apache.org/.

[12] *Apache HBase*, 2012. http://hbase.apache.org/.

[13] *Apache HCatalog*, 2012. http://incubator.apache.org/hcatalog/.

[14] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. In *VLDB*, 2011. To Appear In.

[15] H. Herodotou, F. Dong, and S. Babu. Mapreduce programming and cost-based optimization? crossing this chasm with starfish. *PVLDB*, 4(12):1446–1449, 2011.

[16] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[17] H. Herodotou et al. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, 2011.

[18] *Hive*, 2012. `http://hadoop.apache.org/hive/`.

[19] *Jaql, Query Language for JSON*, 2012. `http://code.google.com/p/jaql/`.

[20] *Hadoop MapReduce Tutorial*. `http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html`.

[21] *Apache Oozie Workflow Scheduler for Hadoop*, 2012. `http://incubator.apache.org/oozie/`.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[23] *Pig*, 2012. `http://hadoop.apache.org/pig/`.

[24] B. Wong, I. Stoyanov, and E. G. Sirer. Octant: a comprehensive framework for the geolocalization of internet hosts. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 23–23, Berkeley, CA, USA, 2007. USENIX Association.

[25] *NextGen MapReduce*, 2011. `http://hadoop.apache.org/common/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[26] Y. Zhang, Q. Gao, L. Gao, and C. Wang. imapreduce: A distributed computing framework for iterative computation. In *IPDPS Workshops*, pages 1112–1121, 2011.

[27] *Apache ZooKeeper*, 2012. `http://zookeeper.apache.org/`.