UNIVERSITY OF CALIFORNIA,
IRVINE

HERDER: A Heterogeneous Engine for Running Data-Intensive Experiments & Reports

THESIS

submitted in partial satisfaction of the requirements
for the degree of

Master of Science

in Computer Science

by

Vandana Ayyalasomayajula

Thesis Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Professor Sharad Mehrotra

2011

© 2011Vandana

DEDICATION

This thesis is dedicated to my parents and my husband who have supported me all the way since the beginning of my studies and have been a great source of motivation and inspiration.

TABLE OF CONTENTS

List Of Figures vii

List of Tables ix

Acknowledgements x

Abstract of the Thesis xi

1 Introduction 1

1.1 HERDER 1

1.2 Motivation 2

1.3 Challenges 5

1.4 Thesis Organization 6

2 Related Benchmarks 7

2.1 Background 7

2.2 GridMix2 Benchmark 8

2.2.1 Data Generation 8

2.2.2 Synthetic Job Types 9

2.3 GridMix3 Benchmark 12

2.4 GridMix Summary 14

3 HERDER Overview 16

3.1 High Level Overview 16

3.2 Key Features of HERDER 20

4 HERDER User Model 22

4.1 HERDER Workload Specification 22

4.1.1 Workload 23

4.1.2 Stream 25

4.1.3 HERDER Job 29

4.2 HERDER Job Definition Files 30

4.3 Job Executors 32

4.4 Installing and Running HERDER 35

4.5 HERDER Execution Report 37

4.6 Advanced Topics 39

4.6.1 Determinism Using Seeds 39

4.6.2 Concurrent Execution of a HERDER Job 41

5 HERDER Architecture & Implementation 43

5.1 Workload Manager 43

5.2 Stream Manager 47

5.3 Worker 50

5.4 Job Statistics Aggregation 52

5.5 Failure Model 53

6 HERDER Workload Examples **55**

6.1 Hadoop Workload Example 55

6.2 Mixed Workload Example 61

7 Conclusion **67**

7.1 Future Work 68

References **70**

Appendix A **72**

LIST OF FIGURES

1.1 Mapping of real world scenario to a HERDER workload 4

2.1 Sample record from GridMix2 data 9

2.2 The chaining of jobs in Monster Query 11

3.1 Transformation of stream into workers at runtime in HERDER. . . . 17

3.2 Timelines of workers in HERDER 19

4.1 High level organization of workload definition file. 23

4.2 XML representation of a sample HERDER Workload 26

4.3 XML representation of a sample job descriptor set 26

4.4 XML representation of stream set 28

4.5 XML representation of HERDER job list 29

4.6 Example of Hadoop properties 31

4.7 Process of generating random number sequences using single seed . . 40

4.8 Process of generating deterministic order of execution of jobs in HERDER workload generator 40

5.1 High level architecture of HERDER workload generator. 44

5.2 Event reporting in HERDER 45

5.3 Failure model in HERDER 54

LIST OF TABLES

2.1 Data Set information in GridMix2 9

4.1 Probability distributions supported for job think times. 30

4.2 Basic configuration properties required for Hadoop Jobs 33

4.3 Job executors supported in HERDER 34

5.1 List of events reported to the workload manager and its corresponding response in HERDER. 45

5.2 List of events reported to the stream manager by its workers and corresponding responses. 48

ACKNOWLEDGEMENTS

I owe my deepest gratitude to Professor Michael J. Carey for providing me an opportunity to work with him. I am thankful for the encouragement and guidance provided by him all through my research work.

I would like to thank Raman Grover and Vinayak Borkar for regularly reviewing my work and providing me valuable feedback.

I am also grateful to Professor Chen Li and Professor Sharad Mehrotra for providing valuable comments about my work.

ABSTRACT OF THE THESIS

HERDER: A Heterogeneous Engine for Running Data-Intensive Experiments &

Reports

By

Vandana Ayyalasomayajula

Master of Science in Computer Science

University of California, 2011

Professor Michael J. Carey, Chair

There has been a tremendous increase in the amount of data collected everyday by Internet companies like Google, Yahoo!, Facebook and Twitter. These companies use large Hadoop clusters with thousands of machines to analyze the collected data. The usage model for data-intensive computing platforms like Hadoop is challenging, as many users can be submitting jobs to a cluster at the same time. Therefore, there is need to understand user behavior, cluster resource usage, and how data-intensive computing platforms react to multi-user workloads.

In this thesis we describe HERDER, a multi-user benchmarking tool to execute synthetic workloads on a cluster. HERDER provides a flexible user model to simulate

an actual environment with users working on a cluster and an extensible framework to execute jobs belonging to a variety of data-intensive computing platforms. The HERDER workload generator reports statistics for the steady-state and the overall execution time periods at the end of workload execution.

Chapter 1

Introduction

1.1 HERDER

The amount of data collected by Internet companies such as Google, Yahoo!, Facebook and Twitter is ever-increasing as more and more people are online daily. The collected data is used for business intelligence and other purposes to increase revenues of the companies. Traditional data warehousing tools have become very expensive at the scale at which these companies work. The MapReduce [1] programming model, introduced by Google, tries to solve this problem by supporting distributed computing on large data sets on clusters of commodity machines. Hadoop [3] is a very popular open source implementation of MapReduce. Today, many companies use large Hadoop clusters to analyze the data that they have collected. With so many users submitting jobs to these clusters, there is a need to understand user behavior, cluster resource usage and how data -intensive computing frameworks react to different usage

scenarios.

This thesis introduces HERDER which stands for **H**eterogeneous **E**ngine for **R**unning **D**ata-intensive **E**xperiments and **R**eports. HERDER is a benchmarking tool to execute synthetic workloads on a data-intensive computing cluster. It simulates multi-user workloads running on large shared resource environments. HERDER collects statistics during runs and produces a report about the workload at the end of its execution. The tool collects statistics on per job, per class basis during the steady-state period as well as during the overall execution period of the workload. HERDER users define jobs to be executed as part of the workload. Presently jobs are targeted to two data intensive computing platforms - Hadoop [3] and Hyracks [9]. Workloads belonging to both of these platforms can be executed using the HERDER workload generator.

1.2 Motivation

An early experiment benchmarking Hadoop [3] versus parallel databases was performed by Andrew Pavlo et. al [2] in 2009. In that experiment, a set of tasks representative of MapReduce [1] were taken and executed on Hadoop and two parallel databases to compare each task's running times. This experiment can be categorized as a single-user workload experiment, as the tasks were submitted to a cluster with no other users, one at a time, and their performance was measured. In real-world scenarios, the usage model of data-intensive platforms like Hadoop is much different. Web companies have large clusters of commodity machines with Hadoop installed

and multiple users submitting jobs to a cluster at the same time. Therefore, measuring and benchmarking the multi-user performance of a data-intensive computing platform is very useful and important.

There is an existing benchmark in the industry, known as GridMix [4], that executes synthetic workloads on a cluster. The GridMix benchmark is mainly used for measuring and comparing performance of various Hadoop [3] versions; the results obtained are used to identify performance bottlenecks in Hadoop. The latest version of GridMix is GridMix3, which takes a job trace from a live cluster as its input. A job trace contains JSON-encoded job descriptions derived from the job history logs on the cluster. Each job contains details like the original job submission time, the memory allocated to each task, and- for each task in that job- the byte and record counts read and written. The benchmark uses this information to construct synthetic jobs and submits them to the cluster at an interval matching the original submissions. The synthetic jobs produce a load on the I/O subsystems of the cluster that is comparable to the original multi-user workload. Some drawbacks of this model are:

1. The synthetic jobs submitted might not capture the complete details of the actual jobs.
2. It targets only specific data-intensive computing platform like Hadoop [3].
3. It does not provide a convenient way to generalize user behavior and model multi-user workload.
4. It tries to "replay" an actual scenario rather than modeling it.

HERDER is a benchmarking tool, that can be used to execute multi-user synthetic workloads. It aims to overcome the above mentioned drawbacks and offer the users a flexible framework to run custom jobs on custom frameworks. Figure 1.1 shows how actual real-world scenarios can be mapped into HERDER workload definitions and executed on a cluster.

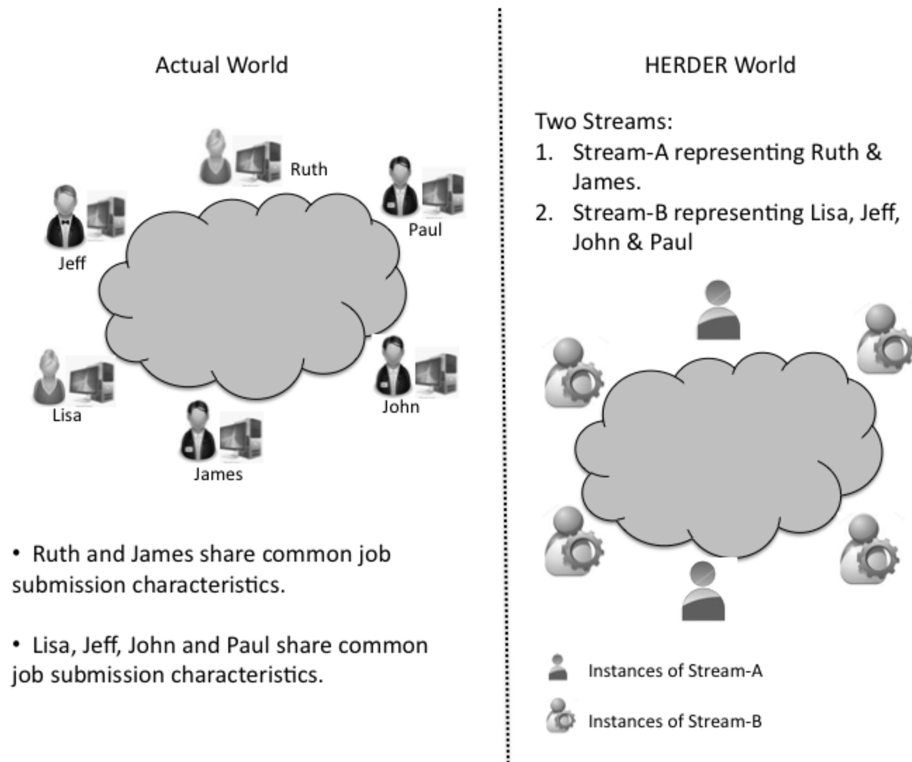

Figure 1.1: Mapping of real world scenario to a HERDER workload

In HERDER, "Streams" represent users with common job submission characteristics. Every stream requires the count of the number of users who are similar to each other. Therefore, a "population count" attribute of a stream indicates the number of actual "users", that constitute a stream. As all users belonging to a stream submit similar jobs, every stream contains a list of HERDER jobs. A HERDER job is the basic unit of execution. A HERDER job is very generic in nature and can be defined using a job

descriptor, the probability of this job being submitted among other jobs in the stream and a think time probability distribution. The job descriptor gives a HERDER job its generic nature. A job descriptor specifies actual job definition files plus an executor (e.g Hadoop or Hyracks). The framework then uses the executor to submit jobs to the cluster and uses the job definition file to determine the details of a job at runtime.

1.3 Challenges

The goal of this work has been the design and implementation of a tool for use within the ASTERIX group at UCI as well as for sharing with others (later) who are interested in the open-source software being created and shared by our group. Aside from the usual challenges related to creating software to share with others, some of the main challenges involved in developing the HERDER workload generator were as follows:

- The design of a workload specification that is rich enough to characterize a range of interesting multi-user workloads.
- The design of a framework capable of handling a heterogeneous mixture of data-intensive job types (e.g., MapReduce, Pig, and Hive).
- The design of an architecture capable of spanning platforms (e.g., Hyracks as well as Hadoop).
- The identification and handling of certain key "details" in order to support robust and reproducible performance experiments, e.g., support for various crite-

ria for workload termination and for executing multi-user job mixes in a pseudo-random yet deterministic manner.

- The provision of an extensible framework so that support for new platforms and languages (e.g., AQL from ASTERIX) will be easy to add in the future.

1.4 Thesis Organization

The remainder of the thesis report is structured as follows:

- Chapter 2 describes benchmarks related to HERDER.
- Chapter 3 gives a high level overview of HERDER and its key features.
- Chapter 4 describes the HERDER user model and provides details related to defining HERDER workloads.
- Chapter 5 describes the HERDER architecture and a real software implementation.
- Chapter 6 gives detailed examples of a HERDER workload to illustrate its use.
- Chapter 7 presents the conclusions from this work.

Chapter 2

Related Benchmarks

2.1 Background

GridMix [4] is an existing benchmark used to run synthetic workloads on Hadoop [3] clusters. This benchmark and HERDER are related, as both execute synthetic workloads on data-intensive computing platforms. However, the GridMix benchmark is tightly coupled to set of pre-defined jobs that are executed in the workload and to the Hadoop platform to which the jobs are submitted. In this chapter, we discuss several different versions of the Gridmix benchmark to understand the jobs executed by the benchmark, how each version of GridMix differed from its predecessors and how each one of them execute a given workload.

2.2 GridMix2 Benchmark

The GridMix2 benchmark models a Hadoop cluster workload. The benchmark defines six types of jobs which mimic the most common tasks the performed by Hadoop user(s). The input to the benchmark is a configuration file. The configuration file is an XML file that contains details about the jobs configured to run as part of the workload. At runtime, the configuration file is parsed and the jobs listed in the file are added to an instance of Hadoop JobControl [14]. JobControl is a Java class that encapsulates a set of MapReduce jobs and their dependencies, if any. It has a Java thread that submits jobs when they become ready, monitors the states of the running jobs, and updates the states of jobs based on the state changes of their depending jobs states. The GridMix2 benchmark, in this manner, submits multiple jobs to the cluster. However, it does not model the way in which actual users work on a cluster and their job inter-arrival times. The following subsections explain the GridMix2 data and the type of jobs used in the benchmark.

2.2.1 Data Generation

An independent program generates data for GridMix2. This program uses MapReduce to just run a distributed job where there is no interaction between the tasks, and each task writes a file containing a large unsorted random sequence of words. The data creation program takes words from a dictionary of 1000 words and generates three kinds of data for the benchmark: VARCOMPSEQ, FIXCOMPSEQ and

Data Category	Number of words per key	Number of words per value	Output Compressed ?	Output Format
VARCOMPSEQ	5 - 10	100-10000	Yes	SequenceFileOutputFormat
FIXCOMPSEQ	5	100	Yes	SequenceFileOutputFormat
VARINFLTEXT	1 - 10	0 - 200	No	TextOutputFormat

Table 2.1: Data Set information in GridMix2

VARINFLTEXT. Each of these data sets consist of records, and each record is a key-value pair delimited by tabs. Each key and value consist of one or more individual words. The name of each of the data sets indicates the kind of data that it contains: VAR/FIX indicates whether the key length is variable or fixed, COMP/INFL indicates whether the data is compressed or not and the last part SEQ/TEXT indicates the output format as being either SequenceFileOutputFormat or TextOutputFormat. Table 2.1 summarizes the data sets used in the GridMix2 benchmark.

Figure 2.1 shows a sample record from a data file generated for GridMix2 benchmark. The first four words form the key and the words after the tab form the associated value.

ultraobscure macropterous ethmopalatal commotion allotropic iniquitously triakistetrahedral aquiline glossing Bassaris sterilely tramplike Zuludom erythremia apopenptic oinomancy signifier choralcelo engrain

Figure 2.1: Sample record from GridMix2 data

2.2.2 Synthetic Job Types

There are six types of synthetic jobs are used in the GridMix2 benchmark. Each of the jobs listed below were chosen to represent common tasks execute by Hadoop users.

1. Stream Sort

This job uses the Hadoop streaming facility [5] that comes with the Hadoop Distribution. The streaming utility allows users to create and run Map-Reduce jobs with any executable or script as the mapper and/or as the reducer. In this job, both the mapper and the reducer are executables that read their input from standard input (line by line) and emit their output to standard output. The sorting of data happens during the reduce phase and the distribution phase that feeds it. The input data set type used for this job is VARINFLTEXT.

2. Java Sort

   This job is created using an "Identity" mapper and "Identity" reducer classes. These classes pass their input records to the output without any change. The sort happens during the reduce phase and the distribution phase that feeds it.

3. Web Data Sort

   This job uses "SampleMapper" and "SampleReducer" classes. In addition to the usual parameters like input/output formats, in/out keys, etc., this job takes "keepmap" and "keepreduce" percentages. SampleMapper and SampleReducer use these values to sample the data. For example, if the values of "keepmap" = 50 and "keepreduce" = 20, then 50% of the values coming to the mapper are passed to the output and 20% of the values coming to the reducer will appear as output. The remaining key-value pairs are ignored by the job. In summary, SampleMapper and SampleReducer do not modify their input pairs, but they pass only some percentage of total input to the output. The "keepmap" and "keepreduce" values used in this job are 100% and 100%. The input data set

type used for this job is VARCOMPSEQ. The sorting is performed during the reduce phase and the distribution phase that feeds it.

4. Web Data Scan

   This job tries to mimic a filter job. This job also uses SampleMapper and SampleReducer classes. The “keepmap” and “keepreduce” values used in this job are 0.2% and 5%. The input data set type used for this job is VARCOMPSEQ.

5. Monster Query

   This job simulates a three-stage pipelined Map-Reduce job. Each of the jobs use SampleMapper and SampleReducer in stages. The “keepmap” and “keepreduce” values for each of the jobs in the chain are 10% and 40%. The input data set type used for this job is FIXCOMPSEQ. Figure 2.2 shows the dependencies between the three Map-Reduce jobs.

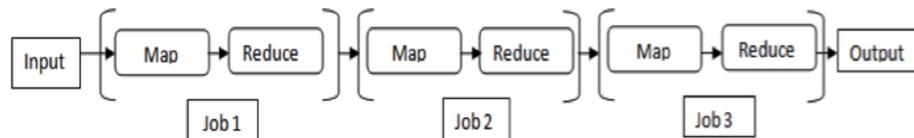

Figure 2.2: The chaining of jobs in Monster Query

6. Combiner

   This job simulates a word count job and uses a combiner in addition to a mapper and reducer. The combiner functions like a mini-reducer on the outputs of a mapper. The mapper of this job reads every word from the input file and emits $\langle word, 1\rangle$ as output. The reducer aggregates all the values (1’s in this case) associated with the key and emits $\langle word, aggregatedvalue\rangle$ as output.

2.3 GridMix3 Benchmark

GridMix3 submits a mix of synthetic jobs by modeling a job profile obtained from a production workload. This version of the GridMix benchmark attempts to model the resource profiles of production jobs to identify any bottlenecks. The input to this benchmark is a MapReduce job trace. Rumen [6] is the tool used for collecting job traces from a cluster. GridMix3 tries to overcome the drawbacks of GridMix2 and emulates actual workloads seen in a production cluster. GridMix3 replays the sequence of events that happened on a production cluster with the help of a job trace and a set of synthetic jobs. The job's inter-arrival times are obtained from the job trace. The input data required for the jobs that run in the benchmark can be generated by providing an additional argument while running the benchmarks.

In order to emulate the load of production jobs from a given cluster on the same or another cluster, the following steps need to be followed by a GridMix3 user:

- Locate the job history files on the production cluster. This location is specified by the "mapreduce.jobtracker.jobhistory.completed.location" configuration property of the cluster.

- Run Rumen to build a job trace in JSON format for all (or selected) jobs.

- Use Rumen to scale up or scale down this job trace to match a workload. This step is optional.

- Generate the input data required for the synthetic jobs.

- Use GridMix3 with the job trace on the benchmark cluster.

The input to the GridMix3 benchmark is a job trace, which is stream of JSON-encoded job descriptions. For each job description, the submitting client obtains the original job submission time, and for each task in that job, the byte and record counts read and written. Given this data, it constructs a synthetic job with the same byte and record patterns as those recorded in the trace. The workload then constructs two types of synthetic jobs:

1. LOADJOB

   This job emulates the workload found in the Rumen trace. The benchmark embeds detailed I/O information from every map and reduce task, such as the number of bytes and records read and written into each job's input splits. The map tasks further relay the I/O patterns of reduce tasks through the intermediate map output data.

2. SLEEPJOB

   The scalability of the job tracker in a Hadoop cluster is tested using this job. The scalability of the Hadoop job tracker is limited by the number of heartbeat messages it can handle from all the task trackers in the cluster. The number of heartbeat messages produced in a test cluster is always less than that in a production cluster. SLEEPJOB is used for the purpose of introducing artificial heart beat messages. This job does nothing except sleep for certain duration and send heartbeats to the job tracker.

The job submission client submits jobs depending on a job submission policy that the user specifies in the configuration. The three job submission policies that the benchmark supports are :

- STRESS

  This policy tells the job submission client to keep submitting jobs in order to keep the cluster stressed.

- REPLAY

  This policy tells the job submission client to submit the jobs in the same manner as the time-intervals present in the actual job trace.

- SERIAL

  This policy informs the job submission client to submit each job only after the previous job has finished its execution.

At runtime, a driver class in the benchmark deserializes the job trace. For each of the jobs in the trace, the job submission client submits a corresponding synthetic job to the cluster at a rate dictated by the job submission policy. In this manner, GridMix3 executes synthetic workloads on Hadoop cluster.

2.4 GridMix Summary

In this chapter, we have briefly reviewed various versions of the GridMix benchmark, which are the current state of the art for benchmarking Hadoop clusters. GridMix2 supports only a fixed set of pre-defined jobs and the executes the jobs using JobControl

object. GridMix3 executes a synthetic job trace derived from live cluster to model cluster I/O resource usage. Both GridMix2 and GridMix3 target jobs to be executed on Hadoop.

Chapter 3

HERDER Overview

3.1 High Level Overview

The goal of HERDER is to emulate an environment with multiple users working simultaneously on a cluster. To create such an environment, we need to understand users job submission characteristics and reproduce them. In reality, the users of a typical cluster submit a mix of jobs, and some jobs are submitted more often than others. Modeling such scenarios is possible by considering a set of jobs for a given type of user and then assigning probabilities to those jobs. Jobs that are submitted more often should be assigned higher probabilities than others. The time interval between two successive job submissions can be modeled by assigning a probabilistic think time distribution.

In the HERDER workload generator, a "stream" defines a group of users with similar job submission characteristics. A stream contains the following primary information:

1. Population Count

   The population count represents the number of similar users.

2. HERDER Job List

   The list of HERDER jobs represent the representative set of jobs that the users submit to the cluster.

At runtime, HERDER creates as many clones of a stream as indicated by the population count. Each of the clones is called a "Worker", which is a Java thread responsible for submitting jobs to the cluster. Figure 3.1 shows how a stream gets transformed into workers at runtime. The basic unit of execution in HERDER is a "HERDER

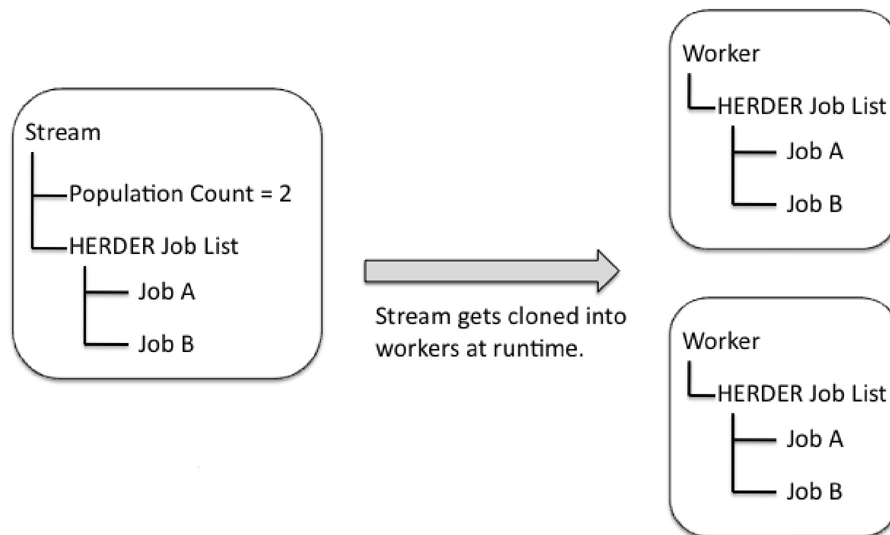

Figure 3.1: Transformation of stream into workers at runtime in HERDER.

job".

A HERDER job description contains the following information:

1. Job Descriptor

   The Job Descriptor contains the actual job definition file and specifies the ex-

ecutor associated with the job.

2. Probability

   The probability that this HERDER job would be chosen among the list of HERDER jobs in the stream.

3. Think Time

   The probability distribution that dictates the pre-job submission interval for this job.

HERDER uses a random number generator to choose the next job to execute for a given worker. To carefully control the sequence of job submission and think times, the users of HERDER can specify a "seed" for each stream in the workload definition file. A seed is an integer which initializes the random number generator to a random starting point; a random number generator gives the same sequence of random numbers each time it is initialized with a given seed. Section 4.6 of Chapter 4 of this report explains in detail how HERDER uses seeds to produce job submissions and think times which are deterministic (reproducible) in nature.

At runtime, when a worker chooses a HERDER job to submit, the HERDER job thinks for some amount of time and then proceeds to execute. This thinking time is dependent on the think-time probability distribution specified in the job. In this manner, the time interval between the jobs need not be constant. The total number of workers actively submitting jobs will be the sum of the population counts of all streams in the workload definition file. All of the workers become initially active at the same time with a job at hand for execution. However, every HERDER job thinks

for some time and then gets ready for job submission. As a result, the time at which different jobs start execution is different for each of the workers. The time period in which all workers are submitting and executing jobs is called the steady-state period for the workload. This time period is the duration in which the cluster experiences the full stress from the workload. After a criterion for stopping the workload is met, all the workers stop submitting new jobs. Figure 3.2 shows example timelines of three workers as part of a mixed workload execution.

A HERDER workload is terminated using one or more criteria provided in the work-

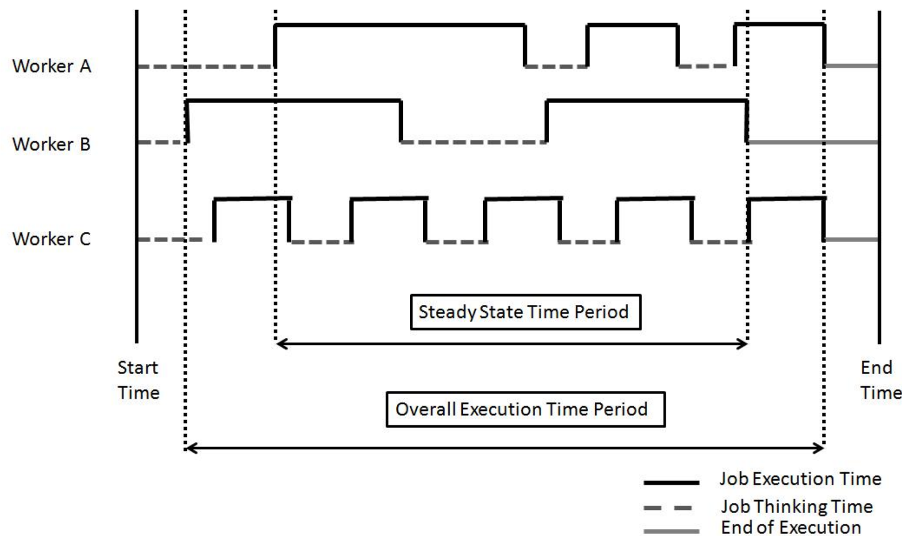

Figure 3.2: Timelines of workers in HERDER

load definition file. There are three criteria that a HERDER workload designer can use to terminate a workload's execution. They are :

- The total execution time of the workload.
- The total number of jobs executed in the workload by all streams together.
- The minimum number of jobs that each of the workers in the workload must

complete.

A user of HERDER is required to specify the first criterion in the workload definition file, and the remaining two criteria are optional.

HERDER does not generate any input data for the jobs running as part of the workload. Instead, HERDER users are responsible for the generating the data required by the HERDER jobs and for ensuring the correctness of the job definition. In addition, the jobs that run as part of the workload should be independent of one another. This means that the execution of one job should not affect the running of another job (i.e., it is required to ensure that two concurrently executing jobs will not compete for any shared resources like directory files during their execution).

3.2 Key Features of HERDER

The key features of the HERDER workload generator are the following:

1. The ability to execute custom jobs belonging to different data-intensive computing platforms like Hadoop [3] and Hyracks [9].

2. The provision of an extensible framework to run jobs belonging to custom data-intensive computing platforms.

3. The ability to run scripts containing commands from high level languages like HiveQL [7] and PigLatin [8].

4. The ability to model the execution of a set of jobs in a flexible manner by assigning probabilities.

5. The ability to model inter-arrival times of jobs using a think-time probability distribution.

The remaining chapters cover these features in more detail.

Chapter 4

HERDER User Model

4.1 HERDER Workload Specification

The first step to run a workload using the HERDER workload generator is to define the workload. In HERDER, the workload is specified using an XML file that serves as an input to the workload generator. The XML tags used for workload definitions are summarized in Appendix A. The top level element in a workload definition file is "workload". All of the other elements are contained in this element. The workload definition file must be well-formed in order to be run by HERDER. Figure 4.1 shows the high-level organization of the workload definition file and the following sub-sections explain how to create a HERDER workload definition file in more detail.

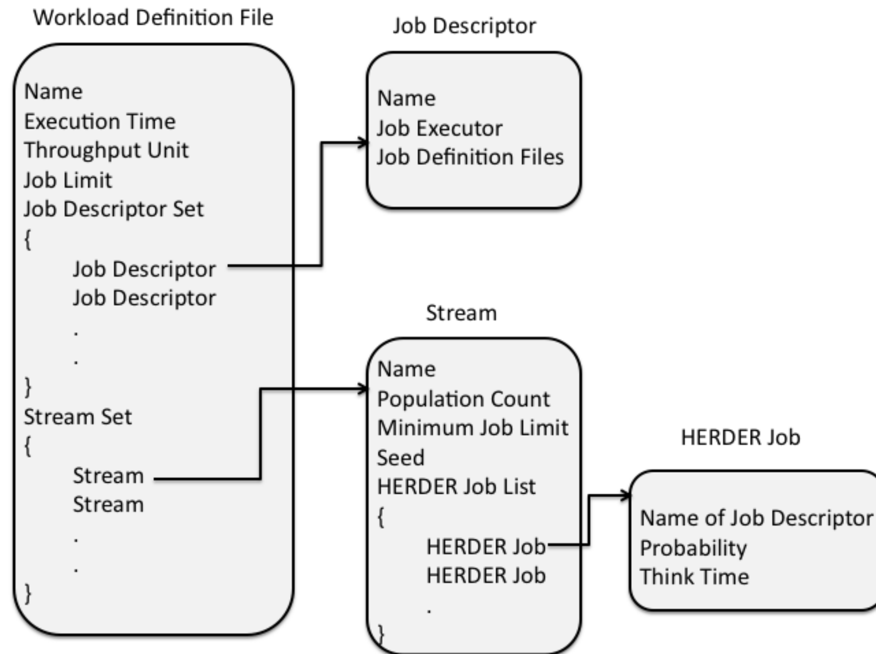

Figure 4.1: High level organization of workload definition file.

4.1.1 Workload

The HERDER workload specification is identified by the "workload" tag, the root element in a HERDER workload definition file. Figure 4.2 shows an outline of a sample workload definition file. The list of elements directly under the workload element are:

- Name

  This element identifies a unique name for the workload. It is identified by the tag "name" in the workload definition file and its value is a string. This string will be used to name the log file and the statistics report produced at the end of workload execution.

- Execution Time

This element specifies the total desired (maximum) execution time for the workload. The execution time should be specified in seconds. It is identified by the tag “executiontime” and takes an integer value.

- Throughput Unit

  This element specifies the units desired for reporting the throughput measurement. It is identified by the tag “throughputunit”, and can take any of the three values “secs”, “mins” or “hrs”. If the user specifies the value to be “secs”, then the units for throughput reporting will be “jobs per second”; if “mins is specified then “jobs per minute”; and, if “hrs” is specified then “jobs per hour”. The default value for this element is “secs”.

- Job Limit

  This element specifies the number of jobs that the workload should complete before execution is terminated by the HERDER workload generator. This property is optional. Recall that the end-user can choose among the three criteria for termination mentioned in Section 3.1 of Chapter 3. The tag used for this property is “joblimit”. It takes an integer value.

- Job Descriptor Set

  This element specifies the set of job descriptors. The job descriptor is the core part of a HERDER job definition. The job descriptors for all the HERDER jobs used in the workload are required to be defined as children of the XML tag “jobdescriptorset”. Each job descriptor has a unique name, parameter string, a job executor, and a list of job definition files. A job executor is a Java class that

is responsible for the execution of a HERDER job. The parameter string is an optional element which can be used to provide additional parameters to the job. A job definition file contains all the information required for running a given job. Note that support for a list of job definition files is necessary to support scenarios in which a single job may consist of multiple small jobs connected as a linear chain. For example, a sort job using the Hadoop [3] framework can be accomplished using a single Hadoop job configuration file, whereas a job that performs a multi-attribute join on two datasets requires two Map-Reduce jobs, with the latter case resulting in two job definition files. The order of execution of chained jobs will be the order in which their job definition files are listed. Figure 4.3 shows the XML representation of a sample job descriptor set.

- Stream Set

  This element specifies the set of streams for the workload. The tag used for this component is “streamset”. All the streams which belong to this workload are required to defined as the children of this element. The tag used for each of the children is “stream”.

4.1.2 Stream

A Stream in a workload identifies a group or class of users who are similar in their job submission characteristics. It is identified by the tag “stream” in the workload definition file. Figure 4.4 shows the XML representation of a sample stream.

```
<workload>
    <name>Sample-Workload</name>
    <executiontime> 1800 </executiontime>
    <throughputunit>mins</throughputunit>
    <joblimit> 40 </joblimit>
    <jobdescriptorset>
        <jobdescriptor>
            <name>word-sort </name>
            <executor> executors.HadoopJobExecutor</executor>
            <jobdefinitionfiles>
                <file>/home/documents/wordsort.xml </file>
            </jobdefinitionfiles>
        </jobdescriptor>
    </jobdescriptorset>
    <streamset>
      ⋮
    </streamset>
</workload>
```

Figure 4.2: XML representation of a sample HERDER Workload

```
<jobdescriptorset>
    <jobdescriptor>
        <name>Multi-Attribute-Group</name>
        <executor> executors.HadoopJobExecutor</executor>
        <parameters> -libjars mylib.jar </parameters>
        <jobdefinitionfiles>
            <file>/home/documents/group1.xml</file>
            <file>/home/documents/group2.xml</file>
            <file>/home/documents/group3.xml</file>
        </jobdefinitionfiles>
    </jobdescriptor>
</jobdescriptorset>
```

Figure 4.3: XML representation of a sample job descriptor set

The list of elements under the stream element are :

- Name

  This element identifies a unique name for the stream. It is identified by the tag "name" and takes a string as its value.

- Population Count

  This element identifies the number of users who belong to the stream. It is identified by the tag "populationcount" and takes an integer as its value.

- Minimum Job Limit

  This element is optional and serves as another termination criterion for the workload. It indicates the number of successful jobs that each of the workers belonging to a stream should complete before the workload can be terminated. Note that, it is also possible that the workload will be terminated by one of the other criterion before this stream criterion is met. In such scenarios, the early workload termination is not an intended behavior because the streams are not "done" yet and a warning appears in the log file. The warning serves as an indication that something is wrong with one or more of the following:

  - The other termination criteria provided by the user in the workload definition file.
  - The behavior of the data-intensive computing framework for this workload.
  - The behavior of the cluster to which the workload's jobs are submitted.

The tag for this element is “minjoblimit” and it takes an integer as its value.

- Seed

  This element specifies a seed for initializing random number generators. A seed sets the starting point of the stream’s random number generator to a specified “random” starting point. A unique seed returns a unique sequence of random numbers. The tag for this property is “seed” and it takes an integer as its value.

- HERDER Job List

  This element specifies the list of HERDER jobs that will run as part of the given stream in the workload. It is identified by the tag “herderjoblist” and takes a list of HERDER jobs as children with the tag name “herderjob”.

```
<streamset>
   <stream>
      <name>Stream-A </name>
      <populationcount> 3 </populationcount>
      <minjoblimit>10</minjoblimit>
      <seed>9999</seed>
      <herderjoblist>
       </herderjoblist>
   </stream>
</streamset>
```

Figure 4.4: XML representation of stream set

4.1.3 HERDER Job

The HERDER job is the core execution component in the workload. The jobs are listed as part of the stream definition. Figure 4.5 shows the XML representation of a sample list of HERDER jobs. The list of elements under a HERDER job are:

```
<herderjoblist>
    <herderjob>
      <name>word-sort</name>
      <probability>0.8</probability>
       <thinktime>
          <name>uniform</name>
          <min>5</min>
          <max>10</max>
       </thinktime>
    </herderjob>
    <herderjob>
       <name>word-count</name>
       <probability>0.2</probability>
       <thinktime>
          <name>exponential</name>
          <mean>10</mean>
       </thinktime>
    </herderjob>
</herderjoblist>
```

Figure 4.5: XML representation of HERDER job list

- Name

  This element denotes the unique name of the job descriptor to be used to run this job. This name serves as a pointer to a job descriptor in the workload definition. The tag used for this property is “name” and it takes a string value.

- Probability

  This element denotes the probability of this job being chosen to execute in a

stream from among its list of jobs. The tag used for this property is “probability” and it takes a float value. The sum of the probabilities of all the jobs in a stream must be 1.

- Think-Time

  This element describes the think time probability distribution of the job. It is denoted by the XML tag “thinktime”. The user is required to specify the name of the probability distribution and all of the required parameters. Table 4.1 shows the various probability distributions supported for think times and the respective parameters required. The think time for a given job is drawn randomly from this distribution at runtime and occurs immediately *before* the job executes.

4.2 HERDER Job Definition Files

An important step in constructing a HERDER job is creating its job definition file(s). After a job definition file is created, it should be listed in the “jobdefinitionfiles” section of the workload definition file. Currently, HERDER can run jobs belonging to two data-intensive computing platforms - Hadoop [3] and Hyracks [9] - and two

Probability distribution	**Parameters**	**XML tags**
Uniform	Minimum, Maximum	“min”, “max”
Exponential	Mean	“mean”
Normal	Mean, Standard deviation	“mean”, “stddeviation”
Poisson	Mean	“mean”

Table 4.1: Probability distributions supported for job think times.

high-level languages - PigLatin [8] and HiveQL [7]. Jobs described in the PigLatin and HiveQL languages are finally translated into Map-Reduce jobs by the Pig [10] and Hive [11] frameworks, respectively, and then submitted to Hadoop. The process to create a job definition file for each of the supported data-intensive platforms/high-level languages is as follows:

1. Hadoop

   Hadoop [3] (from its 0.20.2 version) uses configuration [12] to describe a Map-Reduce job. The configuration contains a set of properties that describe a job. Each property contains a "name-value" pair. Figure 4.6 shows examples of two such properties.

   The user is required to create a XML file with "configuration" element as the

```
<property>
  <name>mapred.job.name</name>
  <value>word-sort</value>
</property>
<property>
  <name>mapreduce.map.class</name>
  <value>org.apache.hadoop.mapred.lib.IdentityMapper</value>
</property>
```

Figure 4.6: Example of Hadoop properties

   root element and the "property" elements as its children. This file serves as the job definition file for a Hadoop job. All of the properties required for successful completion of the job should be captured in this file. Some of the basic properties required to run a Hadoop Map-Reduce job are listed in Table 4.2. In the case of chained jobs, the value indicated by the "mapred.output.dir" property should be the same as the value of the "mapred.input.dir" of the next

job in the chain. Figure 2.2 showed an example of chained jobs.

2. Hyracks

   A Hyracks [9] job is a dataflow DAG composed of operators and connectors. The Hyracks framework uses a "job specification" to capture the dataflow DAG of a Hyracks job. Users are required to serialize a Hyracks job specification into a file with ".hyracks" extension. This file will serve as the job definition file. The job executor associated with Hyracks is required to deserialize the job definition file and execute it.

3. HiveQL

   For Hive [11], users are required to create a file with a ".hive" file extension. All the desired HiveQL [7] commands can be written in the file, with each command written on a new line. This file is then ready to be used as a job definition file.

4. PigLatin

   For Pig [10], users are required to create a file with a ".pig" file extension. All the desired PigLatin [8] commands can be written in the file, with each command written on a new line. This file can then be used as a job definition file.

4.3 Job Executors

In the HERDER runtime architecture, every HERDER job is associated with a job executor. The job executor is responsible for executing a HERDER job on the cluster.

Property	Explanation
mapred.job.name	The name of the Hadoop job
mapred.input.dir	The directory path of the job input files
mapred.output.dir	The directory path of the job output files
mapreduce.map.class	The qualified name of the Java class that will server as mapper for the job
mapreduce.reduce.class	The qualified name of the Java class that will server as reducer for the job
mapred.output.key.class	The qualified name of the key class for the final output data
mapred.output.value.class	The qualified name of the value class for the final output data
mapred.mapoutput.key.class	The qualified name of the key class for the map output data
mapred.mapoutput.value.class	The qualified name of the value class for the map output data
mapred.input.format.class	The qualified name of the Java class which implements the input format for the Hadoop job
mapred.output.format.class	The qualified name of the Java class which implements the output format for the Hadoop job
mapred.output.compress	A boolean value indicating whether the final outputs should be compressed
mapred.reduce.tasks	The number of reducers required for the job to complete
mapred.compress.map.output	A boolean value indicating whether the outputs of the mapper should be compressed

Table 4.2: Basic configuration properties required for Hadoop Jobs

There are four in-built job executors in HERDER. Table 4.3 shows the executors and the kind of jobs they execute. All the job executors are present in the "executors" package in the HERDER source code. The job descriptor of a HERDER job contains information about its job executor. In the job descriptor, the fully qualified name for the job executor is required to be provided. For example, if there is a Hadoop job in the workload, then its job descriptor would look like that line in Figure 4.3.

The HERDER workload generator can be extended by adding new job executors. I.e., HERDER users can create their own job executors and use them to launch their own custom jobs on the cluster. A Java interface "IJobExecutor" is provided to enable users to create custom job executors. This interface provides two methods:

- execute

  Description: Submits a HERDER job to the cluster for execution.

  Inputs: Instance of "components.HerderJob" class

  Instance of "java.util.logger" class

- clearArtifacts

  Description: Clears all outputs produced by a HERDER job during its execution.

Type of Job	**Job Executor**
Hadoop	HadoopJobExecutor
Hyracks	HyracksJobExecutor
HiveQL	HiveJobExecutor
PigLatin	PigJobExecutor

Table 4.3: Job executors supported in HERDER

Inputs: None

A custom job executor is required to implement the above two methods. At runtime, HERDER creates an instance of the job executor and hands a HERDER job to it through the "execute" method. It is the responsibility of the job executor to submit the job to the cluster and to ensure its successful completion. After successful job completion, the"clearArtifacts" method is used to delete all of the outputs produced by the job during its execution. This is necessary to make sure there is enough disk space for other jobs to run during the workload's execution.

4.4 Installing and Running HERDER

The prerequisites for running the HERDER workload generator on a cluster are:

- Java version 1.6 or above.
- Linux operating system.
- Frameworks like Hadoop [3], Hyracks [9], Pig [10] and/or Hive [11] that are required for the desired workload's job execution installed on the cluster.

After these prerequisites are met, users are required to download the HERDER workload generator and save it into a folder on the local machine. The folder contains a "herder.jar" jar file and a script file "runherder.sh" to execute HERDER. The steps required for running a workload are:

1. Workload Definition File

A workload definition file must be created using the specification provided in Section 4.1.

2. Data Generation

   The data required for HERDER job execution should be generated before running the workload. The HERDER workload generator does not generate the data required for the jobs. The job definition files associated with each job should therefore contain all of the required information about the input data for the job and the directory location for writing any output data.

3. Configuring Environment Variables

   The file "runherder.sh" should be updated to set the environment and classpath variables required for the successful execution of jobs. The directory path of the workload definition file and the output directory for the workload generator are also required to be set in this file.

4. Running HERDER

   The HERDER workload generator can be started by typing the command "./runherder.sh" in the command line. The requirement is that the script file and the HERDER jar file are required to be in the folder where this command is run. At the end of execution, a log file and a report file are produced in the output folder specified in the script file.

4.5 HERDER Execution Report

The HERDER workload generator produces a report at the end of workload execution. This report can be found in the output directory that was specified by the user in the script file (runherder.sh). The report contains statistics about the workload's execution. The statistics are measured for two time intervals, namely the steady-state period and the overall execution period. The steady-state period is the time interval during which the cluster experiences the full load from its users. The overall execution period is the time interval during which atleast one job is executing on the cluster. Figure 3.2 showed an example of the steady state time period and the overall execution time period for a sample workload with three workers.

The reported statistics are calculated on per stream basis. For example, consider a workload defined to have two streams, Stream-A and Stream-B, and where each of the streams has a population count of two. At runtime, two workers of Stream-A and two workers of Stream-B will all be actively submitting jobs to the cluster. At the end of execution, all of the jobs completed by each of the workers of Stream-A will be used for calculating the statistics for Stream-A and written to the report file; similarly, separate statistics will be calculated and reported for Stream-B.

The reported statistics for each of the streams in the workload are:

- The individual execution times of all the jobs in the stream (in seconds).
- The minimum execution time of a job in a stream (in seconds).
- The maximum execution time of a job in a stream (in seconds).

- The average execution time of all jobs belonging to a stream (in seconds).
- The standard deviation of the jobs in a stream.
- The throughput of each stream in the units requested by the user in the workload definition file.

The report gives information to users of HERDER in the following order:

1. General Information

   This section of the report contains information about the start time, end time, duration of the steady-state period, and the overall execution period for the workload.

2. Steady-State Period Statistics

   This section contains the statistics for each of the streams in the workload. A job is considered eligible to be a part of the steady-state statistics calculation if its start time and end time both fall within the steady state period. In addition to the per stream statistics, the overall throughput of the system for the steady-state period is reported.

3. Overall Execution Period Statistics

   This section contains statistics for each of the streams in the workload. All the jobs belonging to a stream are considered while calculating statistics. In addition to the per stream statistics, the overall throughput of the system for the entire execution period is reported.

4.6 Advanced Topics

4.6.1 Determinism Using Seeds

The HERDER workload generator uses random number generators to choose a job for execution and to decide the think time of a job. As a result, each time a workload is run, a new set of jobs are executed with different think times. If this were truly random, the user would have no way to have an experiment behave in the same way in terms of having the exact same workload presented to the system as was generated the previous time. This kind of behavior is not acceptable, as most likely a single workload should be run more than once, e.g. against different schedulers during an experimentation process. In addition, benchmarking two systems in such a scenario would be difficult. Random yet deterministic behavior is therefore very important for workload generators.

A seed is an integer that sets the starting point of a random number generator. The random number generator then gives a series of "random" numbers determined by the starting point, i.e, by the seed. A unique seed will result in a unique sequence of pseudo-random numbers. A given seed can be in turn used to generate other seeds, and these can be used to initialize other random number generators. In this manner, we can generate multiple sequences random numbers using a single seed value in a deterministic manner. Figure 4.7 shows the process of generating multiple sequences of psuedo-random numbers using a single seed.

HERDER achieves repeatable workload generation behavior by using a "seed" for

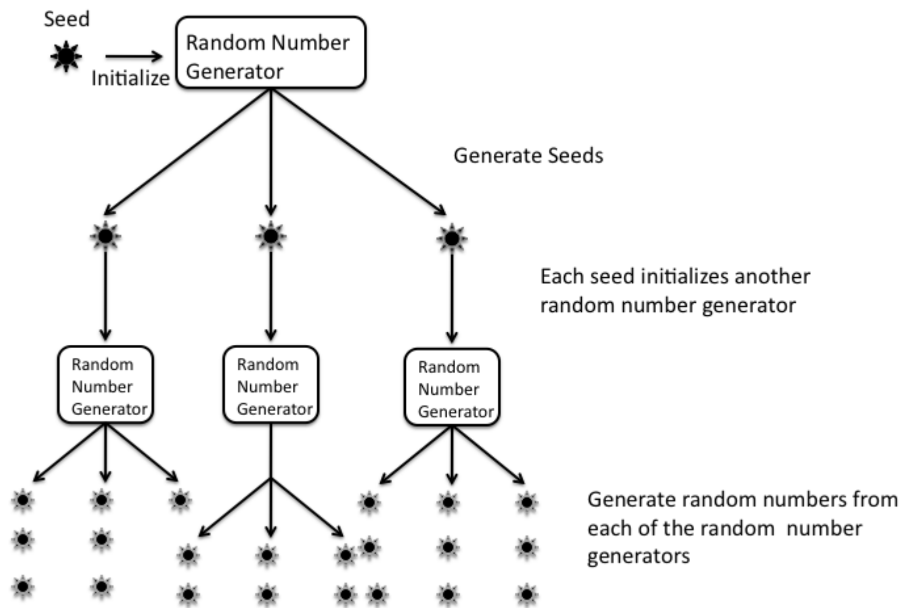

Figure 4.7: Process of generating random number sequences using single seed

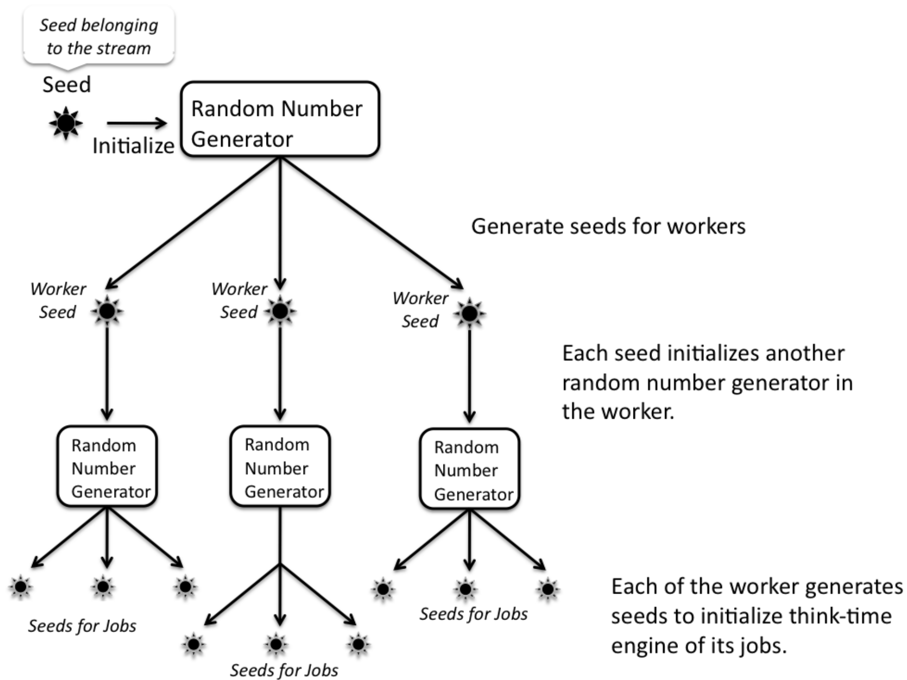

Figure 4.8: Process of generating deterministic order of execution of jobs in HERDER workload generator

each of its streams and applying the process depicted in Figure 4.7. The user provides a seed for each of the streams in the workload definition file. At runtime, every stream has a random number generator which is initialized from its seed. Depending on the population count of the stream, several random numbers are generated. Each random number generated assumes the role of seed for each of the worker's threads. Each worker initializes its random number generator with its seed and again generates random numbers; these numbers serve as seeds for the think time generators of the jobs belonging to the worker. In this manner, the single seed provided for a stream is used to generate a pseudo-random yet repeatable sequence of jobs with think times. Figure 4.8 illustrates the process of generating a deterministic order of job execution in the HERDER workload generator.

4.6.2 Concurrent Execution of a HERDER Job

In the HERDER workload generator, multiple instances of a stream, known as workers, are instantiated at runtime. Each of the workers belonging to the stream executes the same set of HERDER jobs in a pseudo-random manner. It is thus possible that a scenario might arise where two workers of a stream are executing the exact same HERDER job at the same time. In such a situation, interference might occur between the two executing jobs. The most common case occurs when one of the two jobs holds a lock on a resource and the other job fails as a consequence. Therefore, there is a need for users to ensure that concurrent instances of a HERDER job successfully complete their execution.

The HERDER job executors can be used to ensure concurrent execution of a HERDER job. At runtime, each worker delegates the execution of a HERDER job to a separate instance of the job executor. Hence, the job creators can create separate execution environments for each of the parallel instances of the HERDER job. For example, in the current implementation of the job executor associated with Hadoop, the "mapred.output.dir" property of a job is not used as an output directory during execution. Instead, the job executor uses the output directory as a *parent* directory and creates temporary directory inside it, and using the newly created directory as the output for each "Hadoop" job'ss execution. In this manner, two parallel instances of the same Hadoop job can be run at the same time.

Chapter 5

HERDER Architecture & Implementation

The HERDER workload generator runs the workload defined by the user. As described in Chapter 4, the workload definition file (an XML file) contains details about the streams, the jobs to be run as part of a stream, and the criteria for the workload's execution termination.

Figure 5.1 shows the high-level architecture of HERDER.

5.1 Workload Manager

The workload manager orchestrates the process of executing a given workload. A driver class parses the input workload specification file using an XML parser and gives it to the workload manager for execution. Based on the number of streams defined by the user, the workload manager creates a stream manager for each of

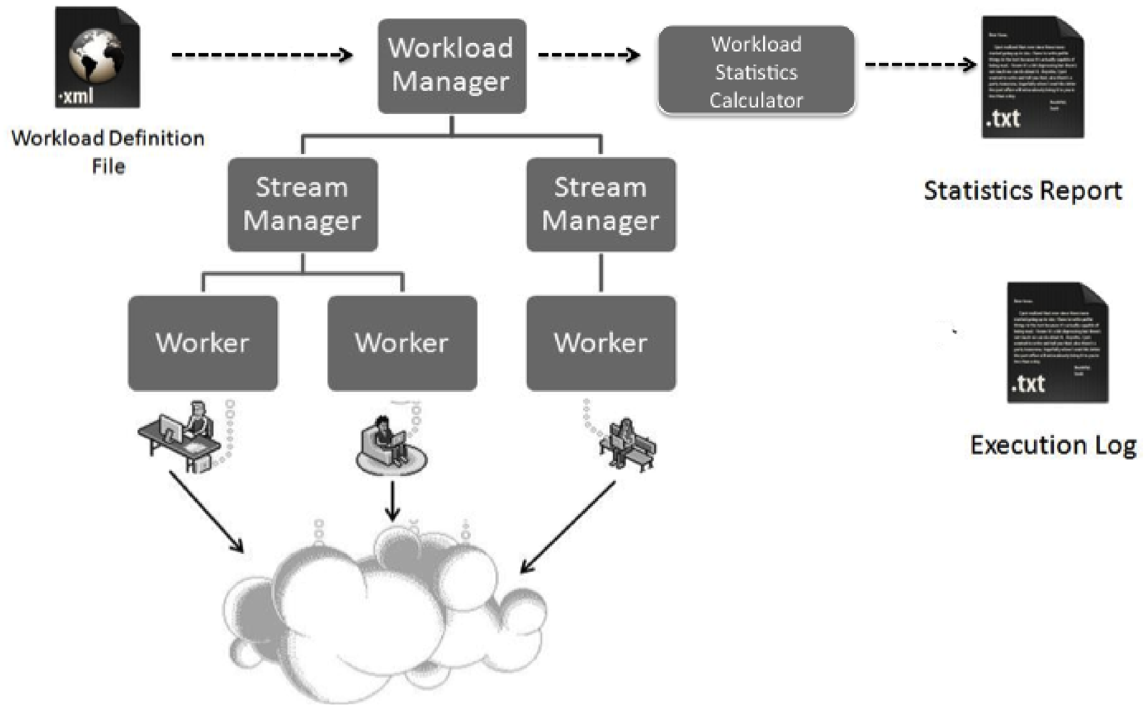

Figure 5.1: High level architecture of HERDER workload generator.

the streams. Each stream is then handled by its own stream manager. The workload manager controls all of the stream managers and, after the execution of the workload, collects statistics about the jobs executed in the workload. It also keeps track of the number of jobs completed in the workload and the status of each job's execution. All of the stream managers report the following events to the workload manager:

1. The successful completion of a job.

2. The failure of a job.

3. The event that each of the workers belonging to a stream manager successfully completed the number of jobs specified by the minimum job limit property of the stream.

Figure 5.2 shows the event reporting mechanism in HERDER. The workload manager

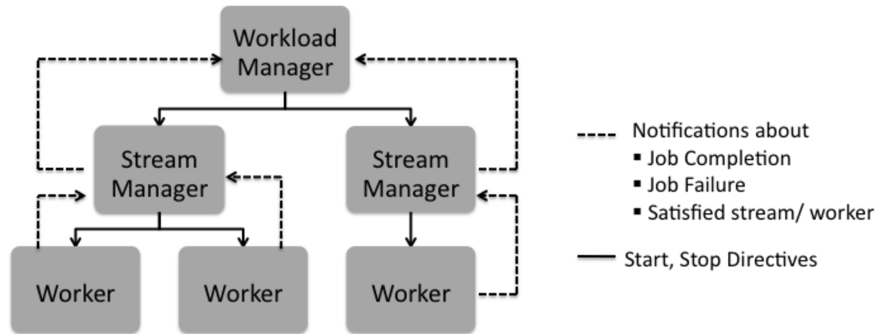

Figure 5.2: Event reporting in HERDER

maintains counters for the total number of jobs completed in the workload as well as counting the number of stream managers that have had all of their workers successfully complete the number of jobs specified by the minimum job limit property of the stream. Upon receiving each of the above events, the workload manager updates its job and stream counters and/or sends appropriate messages back to the stream managers. Table 5.1 shows the action taken by the workload manager for each these events.

At the end of workload execution, the workload manager is responsible for collecting statistics about each of the jobs executed. The workload manager collects all of the job execution statistics from each of the stream managers and hands them to

Event Reported	**Response**
Job Success	The workload manager updates the job completion counters.
Job Failure	The workload manager asks all of its stream managers to abort their workers. The workload's execution is terminated.
Stream Minimum Job Limit Criterion Satisfied	The workload manager updates the satisfied stream counters.

Table 5.1: List of events reported to the workload manager and its corresponding response in HERDER.

the driver class. The driver class passes the collected job execution statistics to the workload statistics calculator, which is responsible for calculating the statistics for the completed execution of the workload.

The main Java methods present in the workload manager class are:

- runWorkload

  Description: This method executes a workload.

  Input: An instance of workload class.

  Output: None

- notifyCompletionOfJob

  Description: This method notifies the workload manager about the completion of job.

  Input: None.

  Output: None

- notifySatisfiedStream

  Description: This method notifies the workload manager about a stream manager where each of the workers have completed the number of jobs indicated by the minimum job limit property of the stream.

  Input: None.

  Output: None

- notifyJobFailure

  Description: This method notifies the workload manager about the failure of a job's execution.

Input: None.

Output: None

- getExecutionStatus

  Description: This method returns the final status of workload execution.

  Input: None

  Output: A boolean value (true or false) indicating the success or failure of a workload execution.

- getJobStatsByStream

  Description: This method returns the statistics related to each of the jobs executed in the workload, grouped on a stream basis.

  Input: None

  Output: A hash map of stream name and the list of execution statistics for each of the jobs run by the workers of the stream's manager.

5.2 Stream Manager

The stream manager handles a single stream's execution. It is responsible for spawning workers based on the population count of a stream. The stream manager contains a random number generator which is initialized using the seed specified for a stream in the workload definition file. The random number generator is used to generate random numbers which are used as seeds for each of the spawned workers. All of the workers report to the stream manager about the job execution status, and the

stream manager in turn reports the stream's status to the workload manager. All of the workers report the following events to the stream manager:

1. The successful completion of a job.
2. The failure of a job.
3. The event that a worker has successfully completed the number of jobs specified by the minimum job limit property of the stream.

The stream manager maintains a counter for the number of workers that have completed the minimum number of jobs indicated by the minimum job limit property of their stream. After all of its workers have satisfied the minimum job limit property of the stream, it reports to the workload manager that it has satisfied the minimum job limit criterion. Table 5.2 shows the list of events reported by the workers to the stream managers and the response of the stream managers. The stream manager also receives directives from the workload manager. The directives issued by the workload manager are "Start" and "Stop". Upon receiving the "Start" directive, the stream manager spawns the worker threads and waits for them to complete execution. When

Event Reported	**Response**
Job Success	The stream manager reports a job completion event to the workload manager.
Job Failure	The stream manager reports job failure event to the workload manager.
Minimum Job Limit Criterion Satisfied	The stream manager updates its internal counter to track the number of workers who have satisfied the criterion.

Table 5.2: List of events reported to the stream manager by its workers and corresponding responses.

the “Stop” directive is received, the stream manager stops all of its workers. After all the workers have finished their execution, the stream manager calculates the steady state period and overall execution period for the stream.

The main Java methods present in the stream manager Java class are:

- startStreamManager

  Description: This method starts a stream manager.

  Input: None

  Output: None

- stopStreamManager

  Description: This method stops a stream manager.

  Input: None

  Output: None

- waitForCompletion

  Description: This method waits for all the workers to complete their execution.

  Input: None

  Output: None

- notifyJobCompletion

  Description: This method notifies the stream manager about the completion of a job.

  Input: None

  Output: None

- notifyJobFailure

  Description: This method notifies the stream manager about the failure of a job.

  Input: None

  Output: None

- notifyMinJobCompletion

  Description: This method notifies the stream manager about workers who have completed the number of jobs indicated by the minimum job limit property of the stream.

  Input: None

  Output: None

- getWorkerJobStatistics

  Description: This method returns the execution statistics related to each of the jobs executed by all of the workers belonging to a stream manager.

  Input: None

  Output: A list of execution statistics of all the jobs.

5.3 Worker

The worker is a Java thread that is responsible for choosing a HERDER job to execute and collecting its job execution statistics. The worker contains a list of HERDER jobs to be executed. At runtime, the random number generator in the worker is used

to generate seeds for each of its HERDER jobs. Each HERDER job uses the seed to initialize its think time generator. The worker runs for the duration specified in the workload definition file, unless any of the termination criteria becomes satisfied and cause the workload manager to send a "Stop" directive to its stream managers. The worker randomly chooses a job from among the list of jobs to execute and the executor associated with the job is used for executing the job. After the job completes successfully, the worker stores statistics such as the start time, end time and status of the job. If the job fails for some reason, the worker notifies its stream manager about the job failure. The worker notifies the stream manager about the following events:

1. The successful completion of job.
2. The failure of a job.
3. A worker completing the minimum number of jobs specified in the stream's definition.

The main Java methods present in the worker Java class are:

- startWorker

  Description: This method starts a worker.

  Input: None

  Output: None

- stopWorker

  Description: This method stops a worker.

Input: None

Output: None

- getAllJobExecutionStats

  Description: This method returns a list of execution statistics related to all jobs submitted by a worker.

  Input: None

  Output: A list of job execution statistics.

- getExecutionTimeInterval

  Description: This method returns the time interval during which the worker was actively executing jobs.

  Input: None

  Output: A time interval indicating start time and end time of a worker.

5.4 Job Statistics Aggregation

HERDER obtains the execution statistics related to each of the jobs executed in a workload from the workload manager and hands them to the workload statistics calculator. As described in section 4.5 of chapter 4, the workload statistics calculator calculates statistics for both the steady-state time period and the overall execution time period. After the execution of each of the workload's HERDER jobs, the following per job information is collected:

- The start time of the job.

- The end time of the job.
- The status of job execution.

The steps followed by the workload statistics calculator to calculate the statistics for a given time period are:

1. It collects all the jobs statistics related to a stream.
2. It checks if the start time and end time of each of the job both lie in the time period for which statistics are being calculated to create a qualified set of job statistics.
3. Then, it iterates over the qualified set of job statistics to calculate the minimum running time of a job in the stream, the maximum running time of a job in the stream, the average running time of a job in the stream, the standard deviation of the job running times, and the throughput of the stream.

After all statistics are calculated for the steady-state time period and the overall execution time period, the workload calculator writes the resulting information into a file in the output directory specified by the user in the script file.

5.5 Failure Model

When any job fails during workload execution, the in-progress workload execution is halted and termed as a failure. A job failure is first detected by a worker. The worker notifies its stream manager about the failure, which in turn notifies the workload

manager. The workload manager then sends “Stop” directives to all of its stream managers. Each of the stream managers in turn issues “Stop” directives to all of its workers. In this way, workload execution comes to a halt in case of any failure. After a worker is asked to stop by the stream manager, it stops submitting new jobs to the cluster. Figure 5.3 illustrates how job failures are reported to the workload manager and how workload execution is terminated.

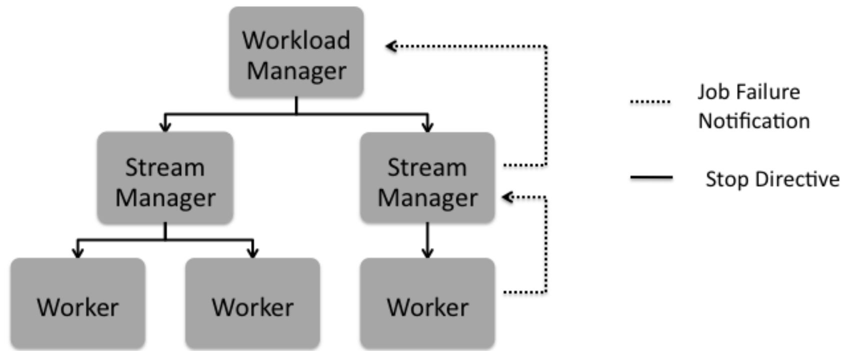

Figure 5.3: Failure model in HERDER

Chapter 6

HERDER Workload Examples

This chapter provides examples of HERDER workloads. The goal of this chapter is to familiarize users with HERDER workload specification files and the execution reports produced at the end of each workload's execution.

6.1 Hadoop Workload Example

This section provides an example of a HERDER workload consisting just of Hadoop jobs. The jobs used in this example workload are same as the jobs used in the GridMix2 benchmark, except for the "Stream sort" job that uses the Hadoop streaming utility for its execution (HERDER does not support this Hadoop feature yet). Recall that the GridMix2 benchmark has six types of jobs:

1. Stream Sort
2. Java Sort

3. Web Data Sort

4. Web Data Scan

5. Combiner

6. Monster Query

Section 2.1 of Chapter 2 provides detailed information about each of the above listed jobs. The example workload here has two streams: Stream-A and Stream-B. All of the jobs in this example workload use the *HadoopJobExecutor* for their execution. "Stream-A" has a population count of 2 and consists of the following three sort/count jobs:

- Combiner

  This job counts the number of occurrences of a word in a text file. The probability of this job in the stream is 0.5.

- Web-Data-Sort

  This job performs a sort operation on the input data. The probability of this job in the stream is 0.3.

- Java-Sort

  This job performs a sort operation on the input data. The probability of this job in the stream is 0.2.

"Stream-B" has a population count of 1 and consists of the following filter jobs:

- Monster-Query

  This job consists of three chained jobs, with each of the job in the chain performing a filter operation. The probability of this job in the stream is 0.5.

- Web-Data-Scan

  This job performs a scan operation on the input data. The probability of this job in the stream is 0.5.

The example workload is configured to execute a total of 10 HERDER jobs, and the total execution time of the workload is set to be a maximum of 30 minutes. This implies that whenever 10 jobs complete their execution or the running time of the workload reaches 30 minutes, whichever comes first, the execution of the workload will be terminated.

The workload definition file for the above configuration is as follows:

```
    <workload>
 <name>Hadoop-Workload</name>
 <executiontime>1800</executiontime>
 <throughputunit>hrs</throughputunit>
 <joblimit>10</joblimit>
 <jobdescriptorset>
  <jobdescriptor>
           <name>Monster-Query</name>
           <executor>executors.HadoopJobExecutor</executor>
           <jobdefinitionfiles>
            <file>/home/vandana/Demo/hadoop-jobs/MonsterQuery/MQ1.xml</file>
      <file>/home/vandana/Demo/hadoop-jobs/MonsterQuery/MQ2.xml</file>
      <file>/home/vandana/Demo/hadoop-jobs/MonsterQuery/MQ3.xml</file>
           </jobdefinitionfiles>
        </jobdescriptor>
        <jobdescriptor>
           <name>Java-Sort</name>
           <executor>executors.HadoopJobExecutor</executor>
           <jobdefinitionfiles>
            <file>/home/vandana/Demo/hadoop-jobs/javasort.xml</file>
           </jobdefinitionfiles>
        </jobdescriptor>
  <jobdescriptor>
           <name>Web-Data-Sort</name>
           <executor>executors.HadoopJobExecutor</executor>
           <jobdefinitionfiles>
            <file>/home/vandana/Demo/hadoop-jobs/websort.xml</file>
           </jobdefinitionfiles>
        </jobdescriptor>
        <jobdescriptor>
```

```
                <name>Combiner</name>
                <executor>executors.HadoopJobExecutor</executor>
                <jobdefinitionfiles>
                 <file>/home/vandana/Demo/hadoop-jobs/wordcount.xml</file>
                </jobdefinitionfiles>
            </jobdescriptor>
  <jobdescriptor>
                <name>Web-Data-Scan</name>
                <executor>executors.HadoopJobExecutor</executor>
                <jobdefinitionfiles>
                 <file>/home/vandana/Demo/hadoop-jobs/webscan.xml</file>
                </jobdefinitionfiles>
            </jobdescriptor>
</jobdescriptorset>
<streamset>
<stream>
      <name>Stream-A</name>
       <populationcount>2</populationcount>
       <seed>1999</seed>
       <herderjoblist>
                <herderjob>
  <name>Combiner</name>
  <probability>0.5</probability>
                  <thinktime>
                    <distribution>uniform</distribution>
                    <min>3</min>
            <max>10</max>
                  </thinktime>
               </herderjob>
       <herderjob>
  <name>Web-Data-Sort</name>
  <probability>0.3</probability>
                  <thinktime>
                    <distribution>exponential</distribution>
                    <mean>20</mean>
                  </thinktime>
               </herderjob>
       <herderjob>
  <name>Java-Sort</name>
  <probability>0.2</probability>
                  <thinktime>
                    <distribution>uniform</distribution>
                    <min>10</min>
            <max>40</max>
                  </thinktime>
               </herderjob>

       </herderjoblist>

</stream>
<stream>
      <name>Stream-B</name>
       <populationcount>1</populationcount>
       <seed>1011</seed>
       <herderjoblist>
 <herderjob>
   <name>Monster-Query</name>
   <probability>0.5</probability>
          <thinktime>
                    <distribution>uniform</distribution>
                    <min>5</min>
            <max>10</max>
          </thinktime>
        </herderjob>
 <herderjob>
   <name>Web-Data-Scan</name>
   <probability>0.5</probability>
          <thinktime>
```

```
                <distribution>exponential</distribution>
                <mean>10</mean>
            </thinktime>
          </herderjob>
        </herderjoblist>
</stream>
</streamset>
</workload>
```

At runtime, given this workload specification, the HERDER workload generator will create two workers belonging to “Stream-A” stream and one worker belonging to “Stream-B”. All of the workers will continue to submit jobs belonging to their respective streams until one of the termination criteria becomes satisfied. The content of the statistics report produced at the end of execution for this workload looks as follows:

```
************************Workload Statistics****************************

Steady state start time : 03/03/2011 09:22:55.524
Steady state end time : 03/03/2011 09:37:32.286
Steady state duration : 876 seconds.
Overall execution start time : 03/03/2011 09:22:55.520
Overall execution end time : 03/03/2011 09:41:13.786
Actual duration : 1098 seconds.
=====================================================================
Steady State Statistics
=====================================================================
Class : Stream-B
Web-Data-Scan - runningtime : 289.0 seconds.
Web-Data-Scan - runningtime : 226.0 seconds.
Web-Data-Scan - runningtime : 246.0 seconds.
Total number of jobs = 3
Average running time = 253.67 seconds.
Minimum running time = 226.0 seconds.
Maximum running time = 289.0 seconds.
Throughput of stream = 12.33 jobs/hour.
Standard deviation of job running times = 26.28.
=====================================================================
```

Class : Stream-A
Web-Data-Sort - runningtime : 374.0 seconds.
Combiner - runningtime : 245.0 seconds.
Combiner - runningtime : 241.0 seconds.
Web-Data-Sort - runningtime : 174.0 seconds.
Combiner - runningtime : 200.0 seconds.
Web-Data-Sort - runningtime : 241.0 seconds.
Web-Data-Sort - runningtime : 251.0 seconds.
Total number of jobs = 7
Average running time = 246.57 seconds.
Minimum running time = 174.0 seconds.
Maximum running time = 374.0 seconds.
Throughput of stream = 28.77 jobs/hour .
Standard deviation of job running times = 58.26.

==

Total number of jobs = 10
Steady State System Throughput = 41.10 jobs/hour .

==

Overall Running Time Statistics

==

Class : Stream-B
Web-Data-Scan - runningtime : 289.0 seconds.
Web-Data-Scan - runningtime : 226.0 seconds.
Web-Data-Scan - runningtime : 246.0 seconds.
Monster-Query - runningtime : 330.0 seconds.
Total number of jobs = 4
Average running time = 272.75 seconds.
Minimum running time = 226.0 seconds.
Maximum running time = 330.0 seconds.
Throughput of stream = 13.11 jobs/hour .
Standard deviation of job running times = 40.13.

==

Class : Stream-A
Web-Data-Sort - runningtime : 374.0 seconds.
Combiner - runningtime : 245.0 seconds.
Combiner - runningtime : 241.0 seconds.
Web-Data-Sort - runningtime : 174.0 seconds.
Combiner - runningtime : 200.0 seconds.
Web-Data-Sort - runningtime : 241.0 seconds.
Web-Data-Sort - runningtime : 251.0 seconds.
Combiner - runningtime : 120.0 seconds.
Total number of jobs = 8
Average running time = 230.75 seconds.
Minimum running time = 120.0 seconds.
Maximum running time = 374.0 seconds.

Throughput of stream = 26.23 jobs/hour .
Standard deviation of job running times = 68.72.
==
Total number of jobs = 12
Overall System Throughput = 39.34 jobs/hour .

6.2 Mixed Workload Example

This section provides an example of a heterogeneous HERDER workload. The workload consists of two streams: Little-Queries and Big-Query. The "Little-Queries" stream has a population count of 2 and consists of four small HERDER jobs listed below:

- Word-Count

  This job counts the number of occurrences of a word in a text file. This job is defined using a PigLatin script file and uses the *PigJobExecutor* for its execution. The probability of this job in the stream is 0.5.

- Web-Data-Sort

  This Hadoop job performs a sort operation on the input data. This job uses the *HadoopJobExecutor* for its execution. The probability of this job in the stream is 0.4.

- File-Scan

  This Hyracks job does a file scan. This job uses the *HyracksJobExecutor* for its execution. The probability of this job in the stream is 0.1.

The “Big-Query” stream has a population count of 1 and consists of two larger HERDER jobs:

- Monster-Query

  This Hadoop job consists of three chained jobs. Each job in the chain performs a filter operation. This job mimics the "MonsterQuery" job present in the GridMix2 benchmark. This job uses the *HadoopJobExecutor* for its execution. The probability of this job in the stream is 0.6.

- Aggregate-Query

  This job is defined in HiveQL and performs a group operation followed by an aggregation. The jobs uses the *HiveJobExecutor* for its execution. The probability of this job in the stream is 0.4.

This example workload is configured to execute a total of 8 HERDER jobs, and the total execution time of the workload is set to be a maximum of 30 minutes. This implies that whenever 8 jobs complete their execution or the running time of the workload reaches 30 minutes, whichever event happens first, the execution of the workload will be terminated.

The workload definition file for the above configuration is as follows:

```
   <workload>
<name>Mixed-Workload</name>
<executiontime>1800</executiontime>
<throughputunit>hrs</throughputunit>
<joblimit>8</joblimit>
<jobdescriptorset>
 <jobdescriptor>
          <name>Monster-Query</name>
          <executor>executors.HadoopJobExecutor</executor>
          <jobdefinitionfiles>
           <file>/home/vandana/Demo/hadoop-jobs/MonsterQuery/MQ1.xml</file>
    <file>/home/vandana/Demo/hadoop-jobs/MonsterQuery/MQ2.xml</file>
    <file>/home/vandana/Demo/hadoop-jobs/MonsterQuery/MQ3.xml</file>
          </jobdefinitionfiles>
```

```
        </jobdescriptor>
        <jobdescriptor>
            <name>File-Scan</name>
            <parameters>-ccipaddr 127.0.0.1 -ccport 3080</parameters>
            <executor>executors.HyracksJobExecutor</executor>
            <jobdefinitionfiles>
             <file>/home/vandana/Demo/hyracks-jobs/filescan.hyracks</file>
            </jobdefinitionfiles>
        </jobdescriptor>
  <jobdescriptor>
            <name>Web-Data-Sort</name>
            <executor>executors.HadoopJobExecutor</executor>
            <jobdefinitionfiles>
             <file>/home/vandana/Demo/hadoop-jobs/websort.xml</file>
            </jobdefinitionfiles>
        </jobdescriptor>
        <jobdescriptor>
            <name>Word-Count</name>
            <executor>executors.PigJobExecutor</executor>
            <jobdefinitionfiles>
             <file>/home/vandana/Demo/pig-script/wordcount.pig</file>
            </jobdefinitionfiles>
        </jobdescriptor>
        <jobdescriptor>
            <name>Aggregate-Query</name>
            <executor>executors.HiveJobExecutor</executor>
            <jobdefinitionfiles>
             <file>/home/vandana/Demo/hive-script/aggrquery.hive</file>
            </jobdefinitionfiles>
        </jobdescriptor>
</jobdescriptorset>
<streamset>
<stream>
     <name>Little-Queries</name>
      <populationcount>2</populationcount>
      <seed>1999</seed>
      <herderjoblist>
              <herderjob>
  <name>Word-Count</name>
  <probability>0.5</probability>
                <thinktime>
                  <distribution>uniform</distribution>
                  <min>3</min>
            <max>10</max>
                 </thinktime>
             </herderjob>
      <herderjob>
  <name>Web-Data-Sort</name>
  <probability>0.4</probability>
                <thinktime>
                  <distribution>exponential</distribution>
                  <mean>20</mean>
                 </thinktime>
             </herderjob>
      <herderjob>
  <name>File-Scan</name>
  <probability>0.1</probability>
                <thinktime>
                  <distribution>uniform</distribution>
                  <min>10</min>
            <max>40</max>
                 </thinktime>
             </herderjob>

      </herderjoblist>

</stream>
<stream>
```

```
      <name>Big−Query</name>
       <populationcount>1</populationcount>
       <seed>1011</seed>
       <herderjoblist>
  <herderjob>
    <name>Monster−Query</name>
    <probability>0.6</probability>
            <thinktime>
                    <distribution>uniform</distribution>
                    <min>5</min>
             <max>10</max>
            </thinktime>
          </herderjob>
  <herderjob>
    <name>Aggregate−Query</name>
    <probability>0.4</probability>
            <thinktime>
                    <distribution>poisson</distribution>
                    <mean>8</mean>
            </thinktime>
          </herderjob>
       </herderjoblist>
 </stream>
 </streamset>
 </workload>
```

At runtime, the HERDER workload generator will create two workers belonging to the “Little-Queries” stream and one worker belonging to “Big-Query”. All of the workers, continue to submit jobs belonging to their streams until one of the termination criteria will becomes satisfied. The content of the statistics report produced at the end of the execution of this workload looks as follows:

************************Workload Statistics*****************************

Steady state start time : 03/03/2011 12:06:31.482
Steady state end time : 03/03/2011 12:18:55.539
Steady state duration : 744 seconds.
Overall execution start time : 03/03/2011 12:06:26.485
Overall execution end time : 03/03/2011 12:21:23.701
Actual duration : 897 seconds.

==

Steady State Statistics

==

Class : Little-Queries
Word-Count - runningtime : 291.0 seconds.

Word-Count - runningtime : 329.0 seconds.
Web-Data-Sort - runningtime : 236.0 seconds.
Total number of jobs = 3
Average running time = 285.34 seconds.
Minimum running time = 236.0 seconds.
Maximum running time = 329.0 seconds.
Throughput of stream = 14.52 jobs/hour .
Standard deviation of job running times = 38.18.

==

Class : Big-Query
Aggregate-Query - runningtime : 271.0 seconds.
Aggregate-Query - runningtime : 157.0 seconds.
Aggregate-Query - runningtime : 107.0 seconds.
Total number of jobs = 3
Average running time = 178.34 seconds.
Minimum running time = 107.0 seconds.
Maximum running time = 271.0 seconds.
Throughput of stream = 14.52 jobs/hour .
Standard deviation of job running times = 68.63.

==

Total number of jobs = 6
Steady State System Throughput = 29.03 jobs/hour .

==

Overall Running Time Statistics

==

Class : Little-Queries
Web-Data-Sort - runningtime : 283.0 seconds.
Word-Count - runningtime : 291.0 seconds.
Web-Data-Sort - runningtime : 178.0 seconds.
Word-Count - runningtime : 329.0 seconds.
Web-Data-Sort - runningtime : 236.0 seconds.
Total number of jobs = 5
Average running time = 263.4 seconds.
Minimum running time = 178.0 seconds.
Maximum running time = 329.0 seconds.
Throughput of stream = 20.07 jobs/hour .
Standard deviation of job running times = 51.95.

==

Class : Big-Query
Aggregate-Query - runningtime : 271.0 seconds.
Aggregate-Query - runningtime : 157.0 seconds.
Aggregate-Query - runningtime : 107.0 seconds.
Monster-Query - runningtime : 328.0 seconds.
Total number of jobs = 4
Average running time = 215.75 seconds.

Minimum running time = 107.0 seconds.
Maximum running time = 328.0 seconds.
Throughput of stream = 16.05 jobs/hour .
Standard deviation of job running times = 87.94.
==
Total number of jobs = 9
Overall System Throughput = 36.12 jobs/hour .

Chapter 7

Conclusion

In this thesis, we set out to build a workload generator that models actual sets of users working on large-shared resource environments. The goal was to create a generic tool that can execute synthetic workloads on a cluster with any data-intensive computing platform, offer a convenient user model to express multi-user workloads, and provide statistics about workload execution. We have built and described the HERDER workload generator for use as a benchmarking tool for multi-user data-intensive computing platforms.

The HERDER workload generator provides an extensible framework to execute synthetic workloads consisting of jobs belonging to different data-intensive platforms. In the current implementation, jobs belonging to data-intensive platforms like Hadoop [3], Hyracks [9] and high-level languages like HiveQL [7] and Pig [8] can be executed. The HERDER user model offers an intuitive way to describe actual users working on a cluster using "streams". Each stream is defined via a population count, a seed and list of HERDER jobs . The seed is used achieve pseudo-random yet deterministic

behavior for the HERDER workload generator. A HERDER job is described using a job descriptor, the probability of this job being selected from among other jobs in the stream and a think time probability distribution. The think time probability distribution is used to randomly draw inter-job submission times. The users of HERDER can terminate a workload's execution by specifying a desired maximum execution time, the maximum number of jobs to be executed in a workload, or the minimum number of jobs that all instances of a stream should execute. At runtime, HERDER creates stream instances based on the population count of a stream and an instance is known as a "worker". Each worker mimics an actual user working on a cluster.

In this manner, we have been successful in building a workload generator that can be used to execute jobs belonging to several data-intensive computing platforms and also offer a flexible user model to model sets of concurrent users. HERDER has been used at UCI to study "sampling" policies for Hive queries and their multi-user performance implications [15].

7.1 Future Work

The current implementation of HERDER is limited by the number of workers (Java threads) that can be spawned on a single machine. Therefore, at least for small jobs, HERDER may not scale to model very large multi-user workloads. The present architecture can be extended in the future to create a master-slave architecture, where a single machine would act as a master and the remaining machines would act as slaves. At runtime, the "Master-HERDER" would spawn "Slave-HERDER"s on a

set of slave machines and these slaves can create the workers that submit the jobs to the cluster. Upon completion of workload execution, the slaves would report their job execution statistics to the master. The master could then aggregate the global job statistics and produce the overall workload execution report.

REFERENCES

[1] J. Dean and S. Ghemawat. MapReduce: Simplied Data Processing on Large Clusters. In OSDI '04, pages137-150, December 2004.

[2] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in SIGMOD 09: Proceedings of the 35th SIGMOD international conference on Management of data, New York, NY, USA, 2009, pp. 165-178.

[3] Apache Hadoop. http://hadoop.apache.org/.

[4] GridMix. http://hadoop.apache.org/mapreduce/docs/current/gridmix.html.

[5] Hadoop Streaming Utility, http://hadoop.apache.org/common/docs/r0.15.2/streaming.html.

[6] Rumen. git://git.apache.org/hadoop-mapreduce.git/src/tools/org/apache/hadoop/tools/rumen/

[7] HiveQL. http://wiki.apache.org/hadoop/Hive/HiveQL.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig Latin: A Not-So-Foreign Language for Data Processing in SIGMOD 08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. New York,NY, USA: ACM, 2008, pp. 10991110.

[9] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, “Hyracks: A exible and extensible foundation for data-intensive computing , in ICDE ’11. Available from `http://asterix.ics.uci.edu/papers/ICDE11_conf_full_690.pdf`

[10] Apache Pig. `http://pig.apache.org/`.

[11] A. Thusoo, J. Sen Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony,H. Liu, P.Wyckoff, and R. Murthy, “Hive - a Warehousing Solution Over a Mapreduce Framework, in VLDB 09: Proceedings of the 35th International Conference on Very Large Data Bases. New York, NY,USA: ACM, 2009.

[12] Hadoop Configuration. `http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/conf/Configuration.html`.

[13] Hadoop JobClient. `http://hadoop.apache.org/common/docs/r0.20.0/api/org/apache/hadoop/mapred/JobClient.html` .

[14] Hadoop JobControl. `http://hadoop.apache.org/common/docs/r0.20.1/api/org/apache/hadoop/mapred/jobcontrol/JobControl.html`

[15] R. Grover, ”Sampling-Based Task Scheduling in Hadoop: Design and Performance Evaluation”, in preparation, March 2011.

Appendix A

XML tags for HERDER Workload Definition

This section contains the list of XML tags required for specifying the HERDER workload definition file. All of the tags are case-sensitive.

XML Tag	Description
workload	The root element in the workload definition file.
name	A unique name for the workload.
executiontime	The total desired (maximum) execution time for workload.
throughputunit	The units desired for throughput measurement.
joblimit	The number of jobs that the workload should complete before execution is terminated.
jobdescriptorset	The set of job descriptors.
jobdescriptor	A job descriptor.
executor	A Java class responsible for the execution of a HERDER job.
parameters	Any additional parameters required by the HERDER job.
jobdefinitionfiles	The list of job definition files.
file	The directory path of a job definition file.
streamset	The set of streams.
stream	A stream.
populationcount	The number of users belonging to a stream.
seed	A seed for the stream's random number generator.

minjoblimit	The number of successful jobs that each of the workers belonging to the stream should complete before the workload can be terminated.
herderjoblist	The list of HERDER jobs belonging to a stream.
herderjob	A HERDER job.
probability	The probability of a job being chosen to execute in a stream from among its list of jobs.
thinktime	The think time probability distribution for this job in the stream.
distribution	The name of the think time probability distribution.
min	The minimum value for the Uniform probability distribution.
max	The maximum value for the Uniform probability distribution.
mean	The mean value for the Exponential, Normal and Poisson probability distribution.
stddeviation	The standard deviation for the Normal probability distribution.