Imperial College London
Department of Computing

AutoPig - Improving the Big Data user experience

Benjamin Jakobus

Submitted in partial fulfilment of the requirements for the MSc degree in Advanced Computing of

September 2013

Abstract

This project proposes solutions towards improving the "big data user experience". This means answering a range of questions such as how can deal with big data more effectively[1], identify the challenges in dealing with big data (both in terms of development and configuration) and improving this experience. How can we make the big data experience better both in terms of usability and performance?

[1]Within a Hadoop setting

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Peter McBrien, whose constant guidance and thoughts were crucial to the completion of this project. Dr. McBrien provided me with the input and support that brought reality and perspective to my thinking.

I would like to thank Yu Liu, PhD student at Imperial College London, who, over the course of the past year helped me with any technical problems I encountered. At any moment, Yu willingly gave his time to teaching and supporting me. His knowledge was invaluable to my understanding of the subject.

To my parents who provided me with a home, supported me throughout my studies and helped me in so many ways; thanks.

Apache Hive developers Edward Capriolo and Brock Noland: you endured and answered my many (and often times silly) questions and supervised my patch development. Thanks.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Apache Hadoop is an open-source distributed data processing framework derived from Google's BigTable. Its purpose is to facilitate the processing of large volumes of data over many machines and has been adopted by many large corporations[1] including Yahoo!, Google and Facebook.

Despite its popularity, the Hadoop user experience is still plagued by difficulties. The Pig and Hive codebases are in their infancy. Development can be cumbersome. No mature Pig/Hive development tools or IDEs exist. Users are often faced with the question whether to use Pig or Hive and no up-to-date scientific studies exist to help them answer this question. In addition, performance differences between Pig and Hive are not really well understood and not much literature in the field exists which examines these performance differences. This project proposes solutions towards improving this "big data user experience". This means answering a range of questions such as how one can deal with big data more effectively[2], identify the challenges in dealing with big data (both in terms of development and configuration) and improving this experience. How can we make the big data experience better both in terms of usability and performance?

1.1 Motivation and Objectives

The project's deliverables are subdivided into four steps:

[1] In fact, it possibly is the most widely used distributed data processing framework at the moment
[2] Within a Hadoop setting

1. Development Environment - To develop a toolset that allows for more effective development when using Pig and Hive. It should aid the benchmarking process by allowing for the generation of benchmark datasets and result analysis and automate mundane development tasks such as script deployment, file transfer to and from the Hadoop filesystem, script development, job termination notification, error detection, debugging and script scheduling.

2. Advanced benchmarking - Run benchmarks similar to those presented in the individual study option, however using more complex datasets, varying the number of map tasks and trying different schedulers. The explicit aim of this should be to determine a) the root cause of the performance differences between Pig and Hive and b) discover optimal cluster configuration in terms of the type of schedulers to use, the ratio of map and reduce tasks per job etc. That is: Which scheduler is best? What are the different schedulers good at? Given the cluster, what should the ratio of map and reduce tasks be? Currently it appears as though Hive is less efficient than Pig; is there a way of making Hive scripts more efficient? How and based on what should I choose a specific scheduler? Hadoop schedulers, such as the fair schedulers, seem to be designed for large clusters (that is, clusters containing hundreds of nodes)[6], therefore, strategies such as waiting (to be discussed in more detail in section **??** may not work well in small to medium sized clusters. For example, Zaharia et al show that in large clusters, jobs finish at such a high rate that resources can be reassigned to new jobs without having to kill old jobs (more on this later)[6]. Can the same be said for small to medium sized clusters? Why is it that previous benchmarks carried out by the author (see 2.4 showed that Pig outperformed Hive? What is it that the Pig compiler does differently to Hive? What results do we get if we vary other factors (such as io.sort.mb)?

3. Analyse the Pig and Hive codebase - Providing that the performance differences discovered as part of earlier research hold, how can they be explained? Can they be attributed to differences in the logical, physical or map-reduce plans? What about the map-reduce plans? Does either codebase contain performance bottlenecks or security issues? Can these be proven experimentally? What about overall code quality, design and structure? Which codebase is easier to maintain? Which is more prone to errors? How mature are the codebases really?

4. Knowledge integration - In essence, this answers the question as to how the big data experience can be improved. This involves developing a way to utilize the knowledge gathered in steps

2, 3 and 4, combine it with the developed IDE (named "AutoPig") and make recommendations as to how the Pig and Hive codebase may be improved upon. Are there any optimization recommendations that should be followed? Can these optimizations be demonstrated to be effective? Are there any specific design recommendations that should be followed? Can any of the optimizations be implemented?

1.2 Report structure

This document is divided into ten chapters:

Chapter 2

This chapter begins by introducing fundamental terminology to the reader, and then moves on to discussing existing literature and outlining the project motivation. The chapter concludes by summarizing the factors that could contribute towards improving big data development environments and cluster utilization.

Chapter 3

This chapter analyses the problems faced by the system's development and proposes suitable solutions. It focuses on the three distinct types of challenges that, if overcome, will improve the way we deal with big data.

Chapter 4

This chapter describes the experimental setup and discusses relevant problems and potential shortcomings. It presents the benchmark results and discusses the causes for any differences.

Chapter 5

This chapter compares and contrasts the logical, physical and map-reduce plans for Pig and Hive. It then moves on to analyse the codebases, identify short comings and illustrates concrete recommendations to fix these shortcomings.

Chapter 6

This chapter briefly presents the issues associated with developing an IDE.

Chapter 7

This chapter outlines the system's architecture and discusses the employed design strategies.

The chapter begins by giving an overview of the system's primary components and then elaborates on the project's internal packet structure. It then describes the development process' design strategies (such as the type of design patterns used, the prevision of error handling etc) and finishes by discussing the system's UI.

Chapter 8

This chapter describes, in detail, the implementation of the system's core components. It utilizes pseudocode, Java code snippets, flowchart diagrams and screenshots where applicable.

Chapter 9

This chapter analyses the system in terms of validity, correctness and usability. It outlines four levels of testing (Unit Testing, System Testing, Stress Testing and Usability Testing), describes the test designs for each level and concludes by presenting their results.

Chapter 10

This chapter provides a summary of the project and begins by outlining and reviewing the key project components. The chapter then moves on to re-iterating the project's outcome, discusses the project's future research potential and concludes by proposing a set of future project improvements.

1.3 Statement of Originality

In signing this declaration, you are confirming, in writing, that the submitted work is your own original work, unless clearly indicated otherwise, in which case, it has been fully and properly acknowledged.

I hereby declare that:

– this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;

– with respect to my own work: none of it has been submitted at any educational institution contributing in any way towards an educational award;

– with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Name: Benjamin Jakobus

Signed: ____________________

Date:

1.4 Publications

Publications here.

Chapter 2

Background Theory

2.1 Introduction

2.2 Literature Survey

2.2.1 Developmet tools, IDE plugins, text editors

There exist a wide variety of development tools, IDE plugins and text editor plugins for writing Pig and Hive scripts. However none provide the capabilities needed by the benchmarking application proposed in section **??**. In fact, a majority of the existing tools are not quite mature enough to allow for effective development. What follows is a list of text editor and IDE plugins.

- PigPen (Eclipse Plugin)
- TextMate Plugin
- Vim Plugin
- PigEditor (Eclipse Plugin)
- CodeMirror: Pig Latin mode (online Pig editor)

Initial websearches returned no useful Hive QL editors.

2.3 Schedulers

To date, several Hadoop schedulers have been developed. Although smaller, less well known schedulers may exist, the most notable schedulers are:

2.3.1 FIFO scheduler

This is the default, and most basic of schedulers. It basically consists of a FIFO queue of pending jobs. As a node completes a task, it notifies the scheduler that it has an empty task slot. The scheduler then assigns tasks in the following order: failed tasks are chosen first. If no failed tasks exist, non-running tasks are assigned. If neither failed nor non-running tasks exist, the scheduler uses "speculative execution" to choose a task to assign.[4] That is, it monitors the progress of individual tasks and assigns them a progress score. This score ranges between 0 and 1. The progress score for a map task is simply the fraction of input data read. For a reduce task, each of the following phases accounts for one third of the score:

1. The fraction of data processed during the copy phase
2. The fraction of data processed during the sort phase
3. The fraction of data processed during the reduce phase

As stated by [4]:

> Hadoop looks at the average progress score of each category of tasks (maps and reduces) to define a threshold for speculative execution: When a task's progress score is less than the average for its category minus 0.2, and the task has run for at least one minute, it is marked as a straggler. All tasks beyond the threshold are considered "equally slow," and ties between them are broken by data locality. The scheduler also ensures that at most one speculative copy of each task is running at a time.

Note that the FIFO principle still contributes to the selection process: that is, jobs submitted earlier do still take priority over jobs submitted later.

One problem with the aforementioned calculation of progress score is that it only works well within a homogeneous environment. That is, an environment in which all nodes use the same hardware and can process data at the same rate. To allow for effective scheduling within a heterogeneous environment, the **LATE**, algorithm developed in 2008 by Zaharia et al at the University of California, Berkeley, was introduced. LATE, short for (Longest Approximate Time to End), is used when running Map-Reduce jobs within a heterogeneous environment[4] since the original scheduler rested on the following assumptions[4]:

1. Nodes can perform work at roughly the same rate.
2. Tasks progress at a constant rate throughout time.
3. There is no cost to launching a speculative task on a node that would otherwise have an idle slot.
4. A task's progress score is representative of fraction of its total work that it has done. Specifically, in a reduce task, the copy, sort and reduce phases each take about 1/3 of the total time.
5. Tasks tend to finish in waves, so a task with a low progress score is likely a straggler.
6. Tasks in the same category (map or reduce) require roughly the same amount of work.

Given a heterogeneous environment, assumptions 1 and 2 may not hold (since different nodes may have different hardware) which resulted in Hadoop falsely identifying correct nodes as being faulty and hence not allocating them any work. LATE alleviates this problem by allowing for different pluggable time estimation methods, the default of which measures the progress rate of a given task using the simple formula:

$$\frac{ProgressScore}{T}$$

where T is the amount of time the task has been running for.

The *ProgressScore* is then used to predict the amount of time it takes for the task to complete:

$$\frac{1 - ProgressScore}{ProgressRate}$$

The tasks with the best score are launched. However to achieve best results, LATE defines a "slow node threshold" - tasks are only submitted to nodes that are above this threshold.

As noted by the authors:

> The LATE algorithm has several advantages. First, it is robust to node heterogeneity, because it will relaunch only the slowest tasks, and only a small number of tasks. LATE prioritizes among the slow tasks based on how much they hurt job response time. LATE also caps the number of speculative tasks to limit contention for shared resources. In contrast, Hadoop's native scheduler has a fixed threshold, beyond which all tasks that are "slow enough" have an equal chance of being launched. This fixed threshold can cause excessively many tasks to be speculated upon. Second, LATE takes into account node heterogeneity when deciding where to run speculative tasks.

2.3.2 Fair scheduler

As the name implies, this scheduler schedules jobs in such a way that each receive an equal share of the available resources. Developed through a collaboration between Facebook, Yahoo! and the University of California, the scheduler was introduced with Hadoop version 0.21[6]. In essence, the fair scheduler's primary design goal is to allow for cluster sharing, ensuring that smaller jobs make progress even in the presence of large jobs without actually starving the large job. As discussed above, this is an aspect that the FIFO scheduler does not necessarily allow for.[5].

Hadoop's Map-Reduce implementation was conceived for batch jobs [6] and as such sharing the cluster between different users, organizations or processes was not an inherent design consideration. However as Hadoop became more widely adopted, sharing became an ever more prevalent use-case which resulted in the inevitable conflict between data locality and fairness. That is, how can one ensure that all users get an equal (or allocated) share of the system whilst at the same time minimizing network overhead by ensuring that jobs are run on the nodes that contain their input data. Zaharia et al discuss present a solution to this problem in their 2010 paper "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling"[6], which resulted in the implementation of the "fair scheduler". Using Facebook's 600-node Hadoop cluster as a test bed, the authors begin by asking the question as to what should be done when submitting a new job to the scheduler if not enough resources exist to execute the job. Should running tasks be killed to allow the new job to run? Or should the new job wait until enough tasks finish execution? At first, both approaches seem undesirable[6]:

> Killing reallocates resources instantly and gives control over locality for the new jobs, but it has the serious disadvantage of wasting the work of killed tasks. Waiting, on the other hand, does not have this problem, but can negatively impact fairness, as a new job needs to wait for tasks to finish to achieve its share, and locality, as the new job may not have any input data on the nodes that free up.

Zaharia et al decide that waiting is the better approach after having shown that in large clusters jobs finish at such a high rate that resources can be reassigned to new jobs without having to kill old jobs. However pre-emption is included in the scheduler such that "if a pool's minimum share is not met for some period of time", the scheduler may "kill tasks from other pools to make room to run."[1].

Next, the authors address the problem of locality, since [6]

> [...] a strict implementation of fair sharing compromises locality, because the job to be scheduled next according to fairness might not have data on the nodes that are currently free.

An algorithm dubbed "delay scheduling" resolves this issue by, as is implied in its name, waiting for a fixed period of time until a job on the desired node completes execution (if not, then the job is allocated to a different node).

To summarize, fair scheduling is achieved by creating a set of pools (a pool represents a user or a user group). Pools are configured with the number of shares, constraints on number of jobs and guaranteed minimum shares and the scheduler then uses a combination of waiting and pre-emption to allocate the job to a node with emphasis on data locality.

2.3.3 Capacity scheduler

Basically a more fine-grained version of the fair scheduler designed to impose access restrictions and limit the waste of excess capacity . It differs to the aforementioned fair scheduler in that in that a) it is designed for large clusters that are shared by different organizations, b) developed by Yahoo!, c) instead of pools, it uses configurable queues. Jobs are submitted to a queue[3] and each queue can be configured to use a certain number of map and reduce slots, guaranteed capacity, prioritization etc. Queues are also monitored, so that when a queue isn't using its allocated capacity, the excess capacity is temporarily allocated to other queues.[2]

The capacity scheduler also supports pre-emption. Pre-emption with the capacity scheduler differs to fair scheduling pre-emption in that it uses priorities as opposed to time.

Furthermore, the capacity scheduler supports access controls[2]:

> Another difference is the presence of strict access controls on queues (given that queues are tied to a person or organization). These access controls are defined on a per-queue basis. They restrict the ability to submit jobs to queues and the ability to view and modify jobs in queues.

Capacity scheduler queues can be configured using the following properties:

- Capacity percentage
- Maximum capacity
- Priorities enabled / disabled

Queue properties can be changed at runtime.

2.3.4 Hadoop on Demand (HOD)

[2]

> The HOD approach uses the Torque resource manager for node allocation based on the needs of the virtual cluster. With allocated nodes, the HOD system automatically prepares configuration files, and then initializes the system based on the nodes within the virtual cluster. Once initialized, the HOD virtual cluster can be used in a relatively independent way. HOD is also adaptive in that it can shrink when the workload changes. HOD automatically de-allocates nodes from the virtual cluster after it detects no running jobs for a given time period. This behavior permits the most efficient use of the overall physical cluster assets.

The HOD scheduler is no longer actively supported has never achieved wide-spread use due to the fact that it violated data locality, making network bandwidth a serious bottleneck.[10]

As a side note, Seo et al addressed the general issue of data locality by implementing the High Performance MapReduce Engine (HPMR, available as part of Hadoop 0.18.3+) which pre-fetches and pre-shuffles data in an effort to reduce the execution time of a map-reduce job - an effort which was highly successful and reduces overall execution time by up to 73%[7].

The idea of pre-fetching is to reduce network traffic and minimize I/O latency, whilst pre-shuffling aims to reduce the overhead produced by the actual shuffling phase by analysing the input split and predicting the target reducer where the key-value pairs are partitioned[7].

2.3.5 Deadline Constraint Scheduler

The deadline constraint scheduler is based on the following problem statement[9]:

> Can a given query q that translates to a MapReduce job J and has to process data of size σ be completed within a deadline D, when run in a MapReduce cluster having N nodes with Nm map task slots, Nr reduce task slots and possibly k jobs executing at the time.

Rao and Reddy review the deadline constraint scheduler in [8]. The basic concept behind this scheduler is to increase system utilization whilst at the same time meeting given deadlines. The scheduler first constructs a job execution cost model using a variety of system properties such as the size of the input data, map-reduce runtimes and data distribution. Next, it acquires deadlines for a given job and then uses constraint programming to compute the best slot for the given job. One disadvantage of this scheduler is that it assumes a homogeneous system - that is, one in wich all nodes process data at an equal rate and that the data is processed in a uniform manner across all nodes.[9] Of course such an assumption conflicts with data locality and may significantly increase network overhead, possible to the point were network bandwidth becomes a serious bottleneck[1]

2.3.6 Priority parallel task scheduler

One shortcoming of the **fair scheduler** is that users cannot control and optimize their usage of the given cluster capacity nor can they respond to run-time problems such as node slowdowns or high network traffic. [10] Consequently, Sandholm and Lai[10] devised a scheduler that works on economic principles: map-reduce slots are (dynamically) assigned costs, based on various factors such as demand. Every user is assigned a "spending budget" which in essence is the equivalent to the fair scheduler's minimum and maximum capacity. If a user wants to execute a job, he will need to pay the corresponding "price" for that job. As demand fluctuates, so does pricing, and hence users can make more flexible and efficient decisions as to which job to run at what given time.

It should be noted that it appears as though the scheduler is still within its experimental stages - at least at the time of publishing.

2.3.7 Intelligent Schedulers

At the moment, Apache is also developing two "intelligent" schedulers, **MAPREDUCE-1349** and **MAPREDUCE-1380**. The former is a "learning scheduler" whose purpose is to maintain a given level of CPU utilization, network utilization and/or disk utilization under diverse and dynamic workloads.

The latter, MAPREDUCE-1380, is a so called "adaptive scheduler" whose aim is to dynamically adjust a job's resources (CPU, memory usage) based on some pre-defined criterion.

[1] This statement is a personal speculation by me and is not based on hard evidence.

It should be noted that, as discussed by Rao and Reddy[8], much research is currently done on making schedulers resource aware. That is, current schedulers all use static configuration. Instead, it may be beneficial to have ”intelligent” approaches, such as the aforementioned MAPREDUCE-1380 and MAPREDUCE-1349.

In their paper, Rao and Reddy discuss two possible mechanisms to make schedulers ”smarter”. The first is to have the task tracker dynamically compute slot configurations using some resource metric. The second is to borrow the concept of advertisements and ”markets” from multi-agent systems: nodes would ”advertise” their ”resource richness” and, instead of allocating jobs to the next available node, the job tracker would use these advertisements together with predicted runtimes to allocate the job to the most suitable node.[8]

2.4 Benchmark Overview

Note: The following is a summary of the author's ISO. Tables and figures stem from this ISO report.[13]

The essence of this dissertation builds on early work carried out by the author as part of an independent study option (ISO) titled ”Data Management in Big Data”. The aim of this project was to examine existing big data solutions and determine which performed best (if at all). Specifically, existing literature in the field was reviewed and the resulting map-reduce jobs produced by the big data languages Pig Latin and Hive QL were benchmarked and compared to each other as well as contrasted to PostgreSQL.

Of specific interest was the finding that Pig consistently outperformed Hive (with the exception of grouping data - see tables 5.1, 2.2, 2.3 and 2.4). Specifically[13]

- For arithmetic operations, Pig is 46% faster (on average) than Hive
- For filtering 10% of the data, Pig is 49% faster (on average) than Hive
- For filtering 90% of the data, Pig is 18% faster (on average) than Hive
- For joining datasets, Pig is 36% faster (on average) than Hive

This conflicted with existing literature that found Hive to outperform Pig: In 2009, Apache's own performance benchmarks[14] found that Pig was significantly slower than Hive. These findings were validated in 2011 by Stewart and Trinder et al[17][13] who also found that Hive map-reduce jobs outperformed those produced by the Pig compiler and that Hive was in fact only fractionally slower than map-reduce jobs written using Java.

Dataset size	**% Pig being faster**
1	0.061%
2	3%
3	32%
4	72%
5	83%
6	85%
Avg.:	46%

Table 2.1: The percentage (in terms of real time) that Pig is faster than Hive when performing **arithmetic** operations

Dataset size	**% Pig being faster**
1	-1.8%
2	36%
3	25%
4	68%
5	82%
6	86%
Avg.:	49%

Table 2.2: The percentage (in terms of real time) that Pig is faster than Hive when **filtering 10% of the data**

Dataset size	**% Pig being faster**
1	-9%
2	0.4%
3	3%
4	25%
5	41%
6	50%
Avg.:	18.4%

Table 2.3: The percentage (in terms of real time) that Pig is faster than Hive when **filtering 90% of the data**

Dataset size	**% Pig being faster**
1	-3%
2	12%
3	25%
4	71%
5	76%
6	-
Avg.:	36%

Table 2.4: The percentage (in terms of real time) that Pig is faster than Hive when **joining two datasets**

Furthermore, the ISO confirmed the expectation that relational database management systems are always a better choice (in terms of runtime), providing that the data fits[13]. The benchmarks proving this argument were supported by earlier experiments carried out at the University of Tunis in which researchers applied TPC-H benchmarks to compare Oracle SQL Engine to Apache's Pig (2012)[15]. These findings came of no surprise as a study conducted in 2009 by Loebman et al[16] using large astrophysical datasets already produced the same conclusion.

As part of the ISO's conclusion, it was hypothesized that this initial performance difference between Pig and Hive were due to bugs in the Pig compiler as well as issues with the compiler's logical plan[13]. Upon examining Apache's Pig repository, two releases stood out:

29 July, 2011: release 0.9.0

This release introduces control structures, changes query parser, and performs semantic cleanup.

24 April, 2011: release 0.8.1

This is a maintenance release of Pig 0.8, contains several critical bug fixes.

Closer inspection found that the following tickets appeared to account for the aforementioned problems:

PIG-1775: Removal of old logical plan

PIG-1787: Error in logical plan generated

PIG-1618: Switch to new parser generator technology.

PIG-1868: New logical plan fails when I have complex data types

PIG-2159: New logical plan uses incorrect class for SUM causing

As will be discussed in chapter 5, this hypothesis was largely correct.

The ISO also found that[13]

> [...] the number of map tasks cannot be set manually using either Hive or Pig. Instead, the JobTracker calculates the number of map tasks based on the input, the number of input splits and the number of slots per node (this is configured using Hadoop's MapRed-site.xml configuration file or by setting mapred.min.split.size amd mapred.max.split.size inside your script. i.e. **when changing these configuration options and keeping them relative to the input, one can force both Hive and Pig to use a certain number of map tasks**) and the number of jobs already running on the cluster. Therefore one can vary the number map tasks by manipulating the split sizes. The number of reduce tasks are set in Hive this is using the `set mapred.reduce.tasks=num tasks;` statement; in Pig one uses the `PARALLEL` keyword).
>
> As seen in section **??**, Hive allocates only 4 map tasks for the given dataset when using the default configuration, whilst Pig's translation into Map-Reduce results in 11 map tasks. This appears to be the primary reason as to why Pig requires only 29% of the time that it takes Hive to perform the JOIN on the two datasets (Pig has an average real time runtime of 168.44 seconds; Hive on the other hand takes 581.25 seconds) as fewer map tasks results in less parallelism.

Furthermore:

> The map-reduce job produced by Hive also appears to be slightly less efficient than Pig's: Hive requires an average total CPU time of 824.23 seconds whilst Pig requires 796.3 seconds (i.e. Pig is 3% faster than Hive in terms of CPU time).
>
> Of interest is also the fact that Hive is 28% faster at mapping (241.51 seconds (CPU time) on average as opposed to Pig's 337.12 seconds (CPU time)), yet 26% slower at reducing (622.06 seconds (CPU time) on average) than Pig (458.77 seconds (CPU time) on average).

When forced to equal terms (that is, when forcing Hive to use the same number of mappers as Pig), Hive remains 67% slower than Pig when comparing real time runtime (i.e. it takes Pig roughly 1/3 of the time to compute the JOIN (as seen in table **??**)). That is, increasing the number of map tasks in Hive from 4 to 11 only resulted in a 13% speed-up. [...]

It should also be noted that the performance difference between Pig and Hive does not scale linearly. That is, initially there is little difference in performance (this is due to the large start-up costs). However as the datasets increase in size, Hive becomes consistently slower (to the point of crashing when attempting to join large datasets).

Chapter 3

Problem Analysis and Discussion

Only over the past 3 years or so have big data technologies caught on with main stream tech companies. Before then, the likes of Hadoop (conceived in 2005) and Cassandra (2009) were used primarily by just a small handful of large corporations, such as Google and Yahoo, to solve very domain specific problems. Fast forwarding to 2013, this means that the big data environment is still living its teenage years and consequently exhibits many immature behaviour patterns: erratic performance difference, frequent and drastic changes to codebases, incomplete development tools, few scientific studies examining performance differences and poorly understood codebases. This begs the question as to how the situation can be improved upon.

Naturally the scope of this dissertation is limited, and therefore so are the number of issues that can be addressed. Careful consideration suggests that there are three distinct types of challenges that, if overcome, will improve the way we deal with big data:

- **Language** - More specifically, language choice when it comes to writing Hadoop jobs. Which is better: Pig or Hive? And why? How can either be improved upon?

- **Tools** - The need for good development tools is crucial. How can development be made more efficient?

- **Configuration** - How should Hadoop be configured? What types of schedulers are best? What is the ratio for map and reduce tasks? So on, so forth.

3.1 Unanswered questions - How should we configure Hadoop?

The independent study option presented in section 2.4 produced interesting results which resulted in further questions that need to be answered before the project's core question of how the overall Hadoop experience can be improved, can be answered. Specifically, the following issues need to be addressed:

- What initially caused Hive to outperform Pig[17][15][14]?
- Given previous benchmark results[13], how do the logical and physical plans generated by Pig and Hive differ? What makes Pig outperform Hive?
- How do Pig and Hive perform as other Hadoop properties are varied (e.g. number of map tasks)?
- Do more complex datasets and queries (e.g. TPC-H benchmarks) yield the same results than the ISO?
- How does real time runtime scale with regards to CPU runtime?
- What should the ratio of map and reduce tasks be?

Having answered these questions, the question as to which scheduler is best needs to be answered. To this end, TPC-H benchmarks would be run using different schedulers under different conditions. What are the different schedulers good at? How and based on what should I choose a specific scheduler?

If the benchmark results show significant performance differences (as is to be expected given the benchmarking outcome observed as part of the ISO), how can these differences be rectified? Is there a way of making the cluster more efficient? What causes the differences in runtime (this intersects with section 3.2?

One significant obstacle in answering these questions is time: running the TPC-H benchmarks on a relatively small dataset (300GB) takes approximately 3 days. Consequently mistakes in analysis, lost results or incorrect setup may waste large amounts of time.

3.2 How do Pig and Hive compare? How can the two projects be improved upon?

As discussed in section 2.4 there exist significant performance difference between Pig and Hive. To come to the bottom of this, a systematic dissection of both projects needs to be undertaken. First we must confirm initial results presented in [13] by running more advanced benchmarks. If results coincide, as is to be expected, we must take a top-to-bottom approach by comparing the logical, physical and map reduce plans generated by both compilers. How do they differ semantically (if at all)? Are there any logical differences in how operations are interpreted? Do the order or types of map-reduce operations differ? And so on. Next (and this is by far the largest task), the codebase of Apache Pig and Apache Hive needs to be scrutinized. Naturally, the manner in which this is done depends on the findings of the logical, physical and map-reduce plan analysis (for example, if significant differences in the logical plans exist it follows that emphasis is placed on examining the logical plan generator) however the overall approach should consider flawed code / bugs, general design issues, areas in which the code could be made more efficient, complexity, codesize etc. Objective metrics for code quality should be used, such as n-path complexity, cyclomatic complexity, number of issues per lines of code (e.g. number of issues per 100 lines of code) contrasted against well-known, mature software projects. The primary challenges in answering these questions are that a) the Pig and Hive codebases are very large and complex and b) poorly documented. Together they consist of over 342,000 lines of code. Analysing and examining such large codebases will require a considerable amount time.

3.3 How can we improve the overall development experience?

Last but not least: what were some of the common problems and challenges when developing and running the benchmarks? Did suitable tools for overcoming these challenges exist? If not, why? How can Hadoop (i.e. Pig and Hive) development be streamlined and made more efficient? The development of an IDE is quite complex - can it be accomplished within a reasonable amount of time? The fact that many mature and usable IDEs for dozens of different languages exist should be used as an advantage as one clearly knows what features are desirable, what type of UI is most effective and what functionality is useful but missing in existing IDEs. Nevertheless, developing an IDE from scratch will be challenging.

Chapter 4

Advanced Benchmarks

As previously noted, the TPC-H benchmark was used to confirm the existence of a performance difference between Pig and Hive. TPC-H is a decision support benchmark published by the Transaction Processing Performance Council [12] (Transaction Processing Performance Council (TPC) is an organization founded for the purpose to define global database benchmarks). As stated in the official TPC-H specification[18]

> [TPC-H] consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance while maintaining a sufficient degree of ease of implementation. This benchmark illustrates decision support systems that
>
> - Examine large volumes of data;
> - Execute queries with a high degree of complexity;
> - Give answers to critical business questions.

The performance metrics used for these benchmarks are the same than those used as part of the ISO [13] benchmarks:

- Real time runtime (using the Unix `time` command)
- Cumulative CPU time
- Map CPU time
- Reduce CPU time

In addition, 4 new metrics were added:

- Number of map tasks launched
- Number of reduce tasks launched
- HDFS reads
- HDFS writes

The TPC-H benchmarks differ to the ISO benchmarks and that a) they consist of more queries and b) the queries are more complex and intended to simulate a realistic business environment.

4.1 Benchmark design

4.1.1 Test Data

As stated in the ISO report[13], the original benchmarks attempted to replicate the Apache Pig benchmark published by the Apache Foundation on 11/07/07[?] which served as a baseline to compare major Pig Latin releases. Consequently, the data was generated using the `generate_data.pl` perl script available for download on the Apache website.[?] which produced tab delimited text files with the following schema[13]

name - string age - integer gpa - float

Six separate datasets were generated[1] in an order to measure the performance of, arithmetic, group, join and filter operations. The datasets scaled scaled linearly, whereby the size equates to 3000 * 10^n: dataset size 1 consisted of 30,000 records (772KB), dataset size 2 consisted of 300,000 records (6.4MB), dataset size 3 consisted of 3,000,000 records (63MB), dataset size 4 consisted of 30 million records (628MB), dataset size 5 consisted of 300 million records (6.2GB) and dataset size 6 consisted of 3 billion records (62GB).

One obvious downside to the above datasets is their simplicity: in reality, databases tend to be much more complex and most certainly consist of tables containing more than just three columns. Furthermore, databases usually don't just consist of one or two tables (the queries executed as part of the original benchmarks[13] involved 2 tables at most. In fact all queries, except the join,

[1] These datasets were joined against seventh dataset consisting of 1,000 records (23KB)

involved only 1 table).

The benchmarks produced within this report address these shortcomings by employing the much richer TPC-H datasets generated using the TPC `dbgen` utility. This utility produces 8 individual tables (customer.tbl consisting of 15,000,000 records (2.3GB), lineitem.tbl consisting of 600,037,902 records (75GB), nation.tbl consisting of 25 records (4KB, orders.tbl consisting of 150,000,000 records (17GB), partsupp.tbl consisting of 80,000,000 records (12GB), part.tbl consisting of 20,000,000 records (2.3GB), region.tbl consisting of 5 records (4KB), supplier.tbl consisting of 1,000,000 records (137MB)) as illustrated in figure 4.1.

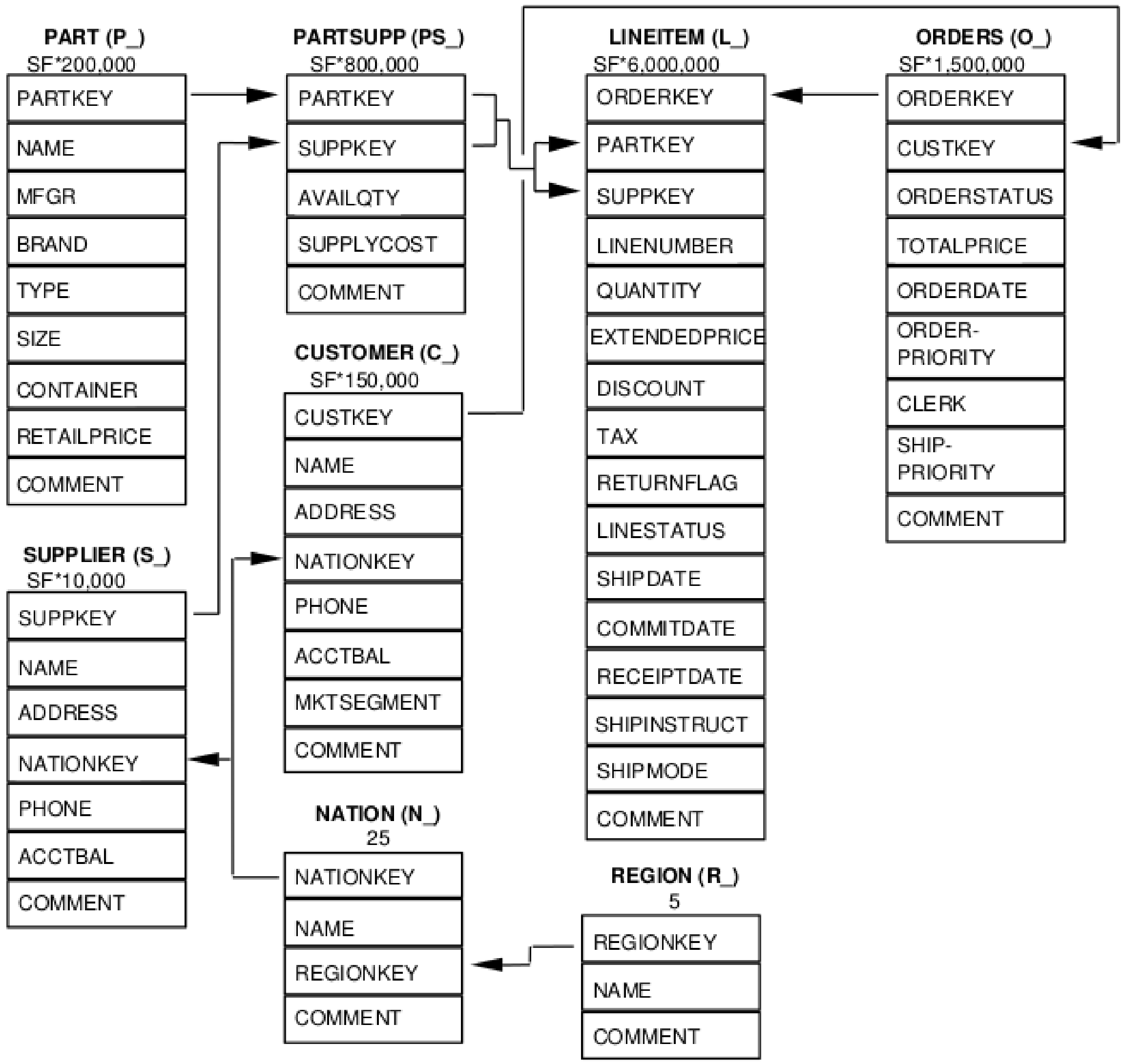


Figure 4.1: The TPC-H schema as per the TPC-H specification 2.15.0

As per the TPC-H specification, the dataset schema is as follows[18]:

Column Name	**Datatype Requirements**
P_PARTKEY	identifier
P_NAME	variable text, size 55
P_MFGR	fixed text, size 25
P_BRAND	fixed text, size 10
P_TYPE	variable text, size 25
P_SIZE	integer
P_CONTAINER	fixed text, size 10
P_RETAILPRICE	decimal
P_COMMENT	variable text, size 23
Primary Key	P_PARTKEY

Table 4.1: TPC-H benchmark schema for the **part** table as per the TPC-H specification [18].

Column Name	**Datatype Requirements**
S_SUPPKEY	identifier
S_NAME	fixed text, size 25
S_ADDRESS	variable text, size 40
S_NATIONKEY	Identifier
S_PHONE	fixed text, size 15
S_ACCTBAL	decimal
S_COMMENT	variable text, size 101
Primary Key	S_SUPPKEY
Foreign Key	S_NATIONKEY to N_NATIONKEY

Table 4.2: TPC-H benchmark schema for the **supplier** table as per the TPC-H specification [18].

Column Name	**Datatype Requirements**
PS_PARTKEY	Identifier
PS_SUPPKEY	Identifier
PS_AVAILQTY	integer
PS_SUPPLYCOST	Decimal
PS_COMMENT	variable text, size 199
Primary Key PS_PARTKEY, PS_SUPPKEY	
Foreign Key	PS_PARTKEY to P_PARTKEY
Foreign Key	S_SUPPKEY

Table 4.3: TPC-H benchmark schema for the **partsupp** table as per the TPC-H specification [18].

4.1.2 Test Cases

The TPC-H test cases consist of 22 distinct queries, each of which were designed to exhibit a high degree of complexity, consist of varying query parameters and various types of access. They are explicitly designed such that each query examines a large percentage of each table/dataset[18].

4.1.3 Test Setup

The ISO experiments whose results are quoted throughout this document were run on a cluster consisting of 6 nodes (1 dedicated to Name Node and Job Tracker and 5 compute nodes). Each node was equippted with a 2 dual-core Intel(R) Xeon(R) CPU @2.13GHz and 4 GB of memory. Furthermore, the cluster had Hadoop 0.14.1 installed, configured to 1024MB memory and 2 map + 2 reduce jobs per node. Our experiment was run on a 32-node cluster (totalling 500 GB of memory), with each node being equipped with an 8-core 2.70GHz Intel(R) Xeon(R))[13]. Several modifications have been made to the cluster since then, which now consists of 9 hosts:

- **chewbacca.doc.ic.ac.uk** - 3.00GHz Intel(R) Core(TM)2 Duo CPU, 3822MiB system memory. Running Ubuntu 12.04.2 LTS, Precise Pangolin.
- **queen.doc.ic.ac.uk** - 3.20GHz Intel(R) Core(TM) i5 CPU, 7847MiB system memory. Running Ubuntu 12.04.2 LTS, Precise Pangolin.
- **awake.doc.ic.ac.uk** - 3.20GHz Intel(R) Core(TM) i5 CPU, 7847MiB system memory. Running Ubuntu 12.04.2 LTS, Precise Pangolin.
- **mavolio.doc.ic.ac.uk** - 3.00GHz Intel(R) Core(TM)2 Duo CPU, 5712MiB system memory. Running Ubuntu 12.04.2 LTS, Precise Pangolin.
- **zim.doc.ic.ac.uk** - 3.00GHz Intel(R) Core(TM)2 Duo CPU, 3824MiB system memory. Running Ubuntu 12.04.2 LTS, Precise Pangolin.
- **zorin.doc.ic.ac.uk** - 2.66GHz Intel(R) Core(TM)2 Duo CPU, 3872MiB system memory. Running Ubuntu 12.04.2 LTS, Precise Pangolin.
- **tiffanycase.doc.ic.ac.uk** - 2.66GHz Intel(R) Core(TM)2 Duo CPU, 3872MiB system memory. Running Ubuntu 12.04.2 LTS, Precise Pangolin.
- **zosimus.doc.ic.ac.uk** - 3.00GHz Intel(R) Core(TM)2 Duo CPU, 3825MiB system memory. Running Ubuntu 12.04.2 LTS, Precise Pangolin.
- **artemis.doc.ic.ac.uk** - 3.20GHz Intel(R) Core(TM) i5 CPU, 7847MiB system memory. Running Ubuntu 12.04.2 LTS, Precise Pangolin.

Both the Hive and Pig TPC-H scripts are available for download from the Apache website.

As noted in sections 4.4 and 5.2, additional benchmarks were run to test Hive's join operations using two transitive self-join datasets consisting of 1,000 and 10,000,000 records (the scripts and dataset generator used for this benchmark were provided by Yu Liu).

Section 4.6 presents additions to the ISO[13] benchmarks - the datasets and scripts used are identical to those presented in the ISO report.

Note: The Linux `time` utility was used to measure the average wall-cock time of each operation. For other metrics (CPU time, heap usage, etc) the Hadoop logs were used.

4.2 Implementation

Bash scripts were written to automate the benchmarking and to re-direct the output to files (see Appendix). At later stages, the IDE developed by the author (and discussed in chapters 6, 7 and 8) was used.

To reduce the size of this report, the TPC-H scripts are not included in the Appendix. The scripts are available for download from: https://issues.apache.org/jira/browse/PIG-2397 (Pig Latin) and https://issues.apache.org/jira/browse/PIG-2397 (Hive QL).

Additions to the ISO benchmarks were performed using the original ISO benchmark scripts and datasets - see [13] for implementation details.

4.3 Results

This section presents the results for both Pig and Hive.

4.3.1 Hive (TPC-H)

Running the TPC-H benchmarks for Hive produced the following results:

Script	**Avg. runtime**	**Std. dev.**	**Avg. cumulative CPU time**	**Avg. map tasks**	**Avg. reduce tasks**
q1	623.64	26.2	4393.5	309	81
q2	516.36	3.94	2015	82	21
q3	1063.73	8.22	10144	402.5	102
q4	344.88	70.74	0	0	0
q5	1472.62	28.67	0	0	0
q6	502.27	7.66	2325.5	300	1
q7	2303.63	101.51	6	5	2
q8	1494.1	0.06	13235	428	111
q9	3921.66	239.36	48817	747	192
q10	1155.33	44.71	7427	416	103
q11	434.28	1.26	1446.5	59.5	15
q12	763.14	11.4	4911.5	380	82
q13	409.16	11.31	3157	93.5	24
q14	515.39	9.82	3231.5	322	80
q15	687.81	14.62	3168.5	300	80
q16	698.14	69.94	14	3	0
q17	1890.16	36.81	10643	300	80
q18	2147	38.57	5591	300	69
q19	1234.5	13.15	17168.5	322	80
q20	1228.72	36.92	91	13	3
q21	3327.84	16.3	10588.5	300	80
q22	580.18	26.86	158	14	0

Table 4.4: TPC-H benchmark results for Hive using 6 trials (time is in seconds, unless indicated otherwise).

Script	**Avg. map heap usage**	**Avg. reduce heap usage**	**Avg.total heap usage**	**Avg. map CPU time**	**Avg. reduce CPU time**	**Avg. total CPU time**
q1	1428	57	1486	6225	2890	9115
q2	790	162	953	18485	13425	31910
q3	1743	241	1985	54985	22820	77805
q4	0	0	0	0	0	0
q5	0	0	0	0	0	0
q6	0	0	0	0	0	0
q7	561	174	737	3275	4285	7560
q8	1620	469	2092	31625	23975	55600
q9	1882	199	2082	18055	12585	30640
q10	3960	367	4328	268270	233640	501910
q11	1468	254	1722	60365	33730	94095
q12	1588	145	1733	5665	4565	10230
q13	1663	349	2013	134420	42070	176490
q14	1421	57	1478	5525	2180	7705
q15	0	0	0	0	0	0
q16	216	0	216	14435	0	14435
q17	0	0	0	0	0	0
q18	0	0	0	0	0	0
q19	1421	71	1493	5250	2395	7645
q20	0	0	0	0	0	0
q21	0	0	0	0	0	0
q22	1202	0	1202	159390	0	159390

Table 4.5: TPC-H benchmark results for Hive using 6 trials.

Note: script names were abbreviated. See appendix A for a mapping from abbreviation to actual names.

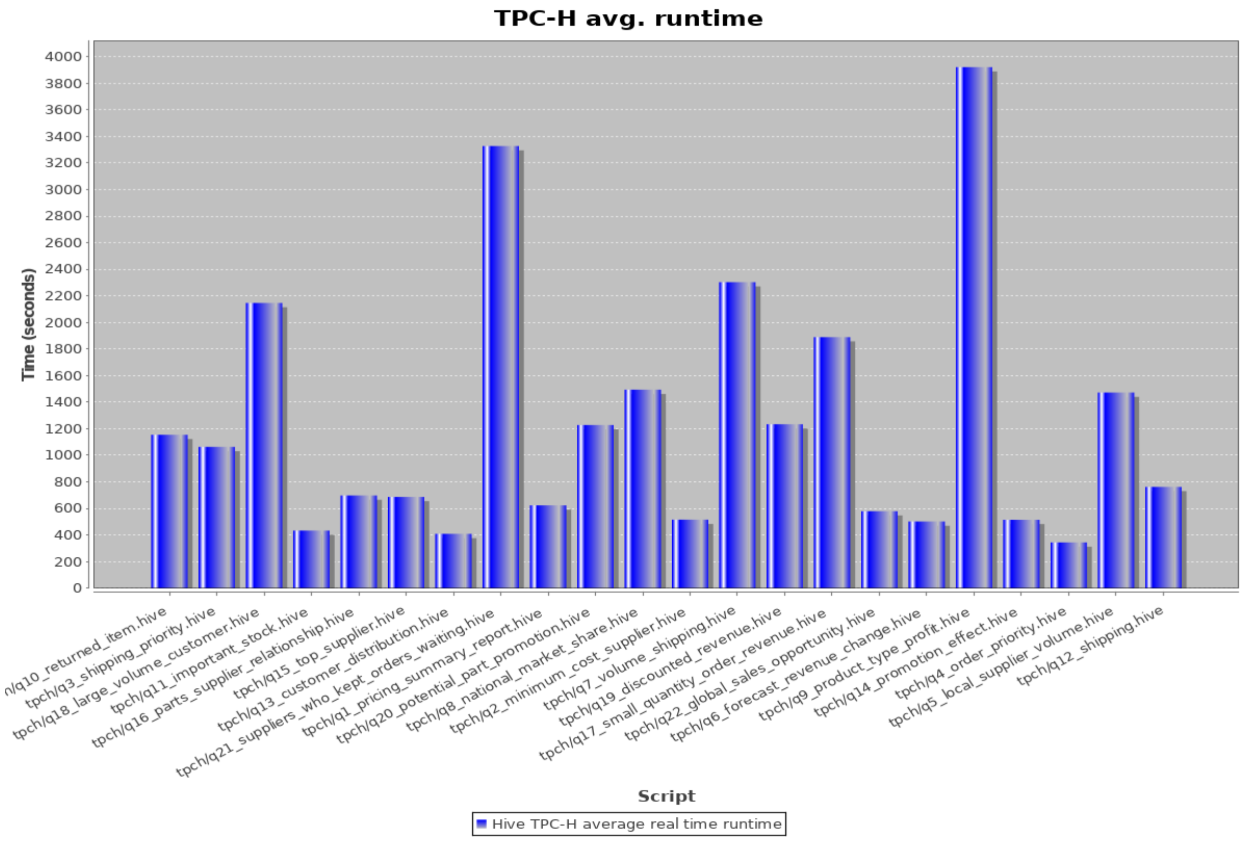


Figure 4.2: Real time runtimes of all 22 TPC-H benchmark scripts for Hive.

4.3.2 Pig (TPC-H)

Running the TPC-H benchmarks for Hive produced the following results:

Script	Avg. runtime	Std. dev.	Avg. cumulative CPU time	Avg. map tasks	Avg. reduce tasks
q1	2192.34	5.88	0	0	0
q2	2264.28	48.35	0	0	0
q3	2365.21	355.49	0	0	0
q4	1947.12	262.88	0	0	0
q5	5998.67	250.99	0	0	0
q6	589.74	2.65	0	0	0
q7	1813.7	148.62	0	0	0
q8	5405.69	811.68	0	0	0
q9	7999.28	640.31	0	0	0
q10	1871.74	93.54	0	0	0
q11	824.42	103.37	0	0	0
q12	1401.48	120.69	0	0	0
q13	818.79	104.89	0	0	0
q14	913.31	3.79	0	0	0
q15	878.67	0.98	0	0	0
q16	925.32	133.34	0	0	0
q17	2935.41	178.31	0	0	0
q18	4909.62	67.7	0	0	0
q19	8375.02	438.12	0	0	0
q20	2669.12	299.79	0	0	0
q21	9065.29	543.42	0	0	0
q22	818.79	14.74	0	0	0

Table 4.6: TPC-H benchmark results for Pig using 6 trials (time is in seconds, unless indicated otherwise).

Script	Avg. map heap usage	Avg. reduce heap usage	Avg.total heap usage	Avg. map CPU time	Avg. reduce CPU time	Avg. total CPU time
q1	3023	400	3426	187750	12980	200730
q2	3221	861	4087	69670	43780	113450
q3	1582	631	2218	110120	46090	156210
q4	633	363	999	1340	9190	10530
q5	3129	878	4010	56020	28550	84570
q6	0	0	0	0	0	0
q7	1336	372	1713	20410	12870	33280
q8	5978	882	6865	306900	162330	469230
q9	874	348	1222	2670	8620	11290
q10	0	0	0	0	0	0
q11	2875	807	3686	172670	46840	219510
q12	1016	751	1771	33880	49830	83710
q13	600	346	950	1740	9860	11600
q14	115	0	115	710	0	710
q15	2561	559	3123	67130	24960	92090
q16	3538	576	4116	606220	70250	676470
q17	375	174	551	5470	3740	9210
q18	965	385	1353	9090	13810	22900
q19	328	175	504	4700	5390	10090
q20	3232	858	4094	69820	48000	117820
q21	1668	486	2160	34440	16800	51240
q22	989	417	1409	20260	13080	33340

Table 4.7: TPC-H benchmark results for Pig using 6 trials.

Note: script names were abbreviated. See appendix A for a mapping from abbreviation to actual names.

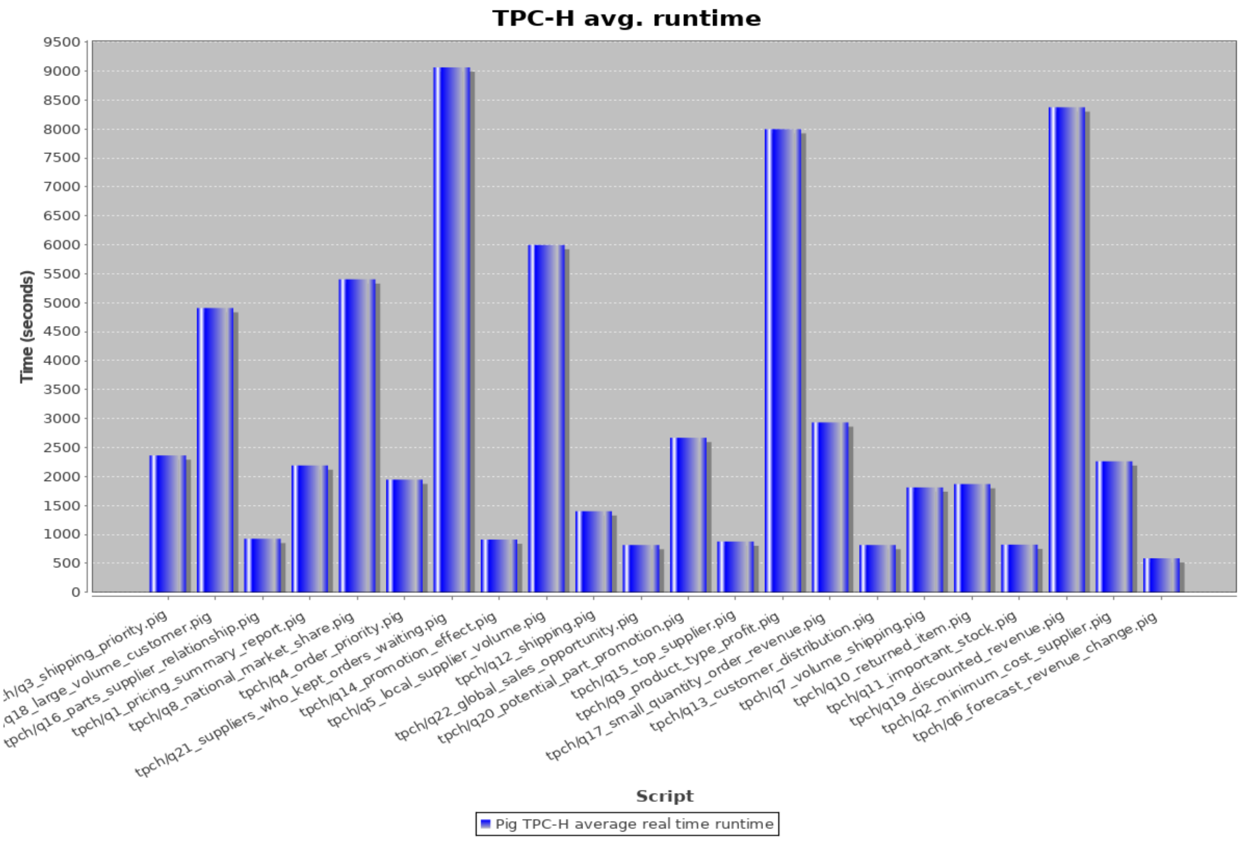


Figure 4.3: Real time runtimes of all 22 TPC-H benchmark scripts for Pig.

4.4 Hive vs Pig (TPC-H)

As shown in figure 4.5, Hive outperforms Pig in the majority of cases (12 to be precise). Their performance is roughly equivalent for 3 cases and Pig outperforms Hive in 6 cases. At first glance, this contradicts all results of the previous ISO experiments[13].

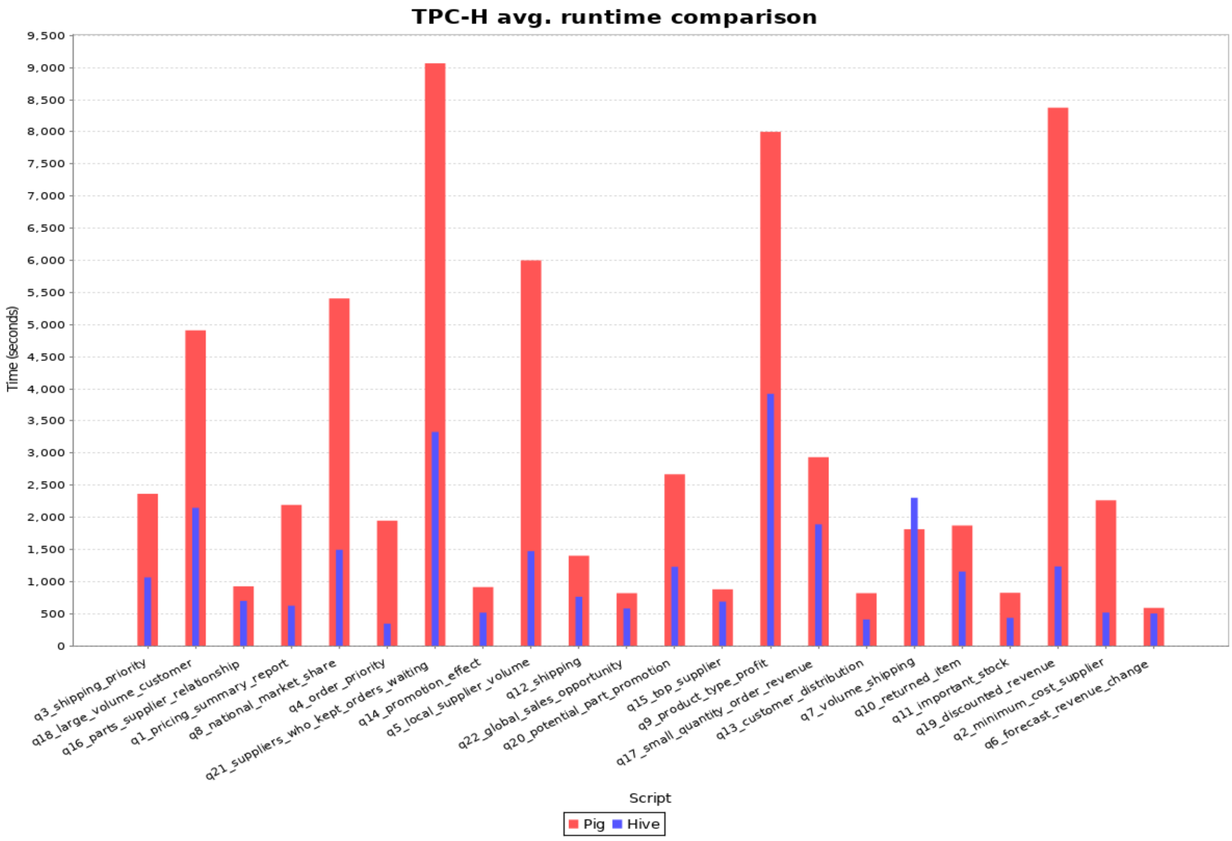


Figure 4.4: Real time runtimes of all 22 TPC-H benchmark scripts contrasted.

Upon examining the TPC-H benchmarks more closely, two issues stood out that explain this discrepancy. The first is that after a script writes results to disk, the output files are immediately deleted using Hadoop's `fs -rmr` command. This process is quite costly and is measured as part the real-time execution of the script (however the fact that this operation is expensive (in terms of runtime) is not catered for). In contrast, the Hive QL scripts merely drop tables at the beginning of the script - dropping tables is cheap as it only involves manipulating the meta-information on the local filesystem - no interaction with the Hadoop filesystem is required. In fact, omitting the recursive delete operation reduces runtime by about 2%. In contrast, removing `DROP TABLE` in Hive does not produce any performance difference.

The aforementioned issue only accounts for a small percent of inequality. What causes the actual performance difference is the heavy usage of the `Group By` operator in all but 3 TPC-H test scripts. Recall from [13] that Pig outperformed Hive in all instances except when using the `Group By` operator: when grouping data Pig was 104% slower than Hive[13].

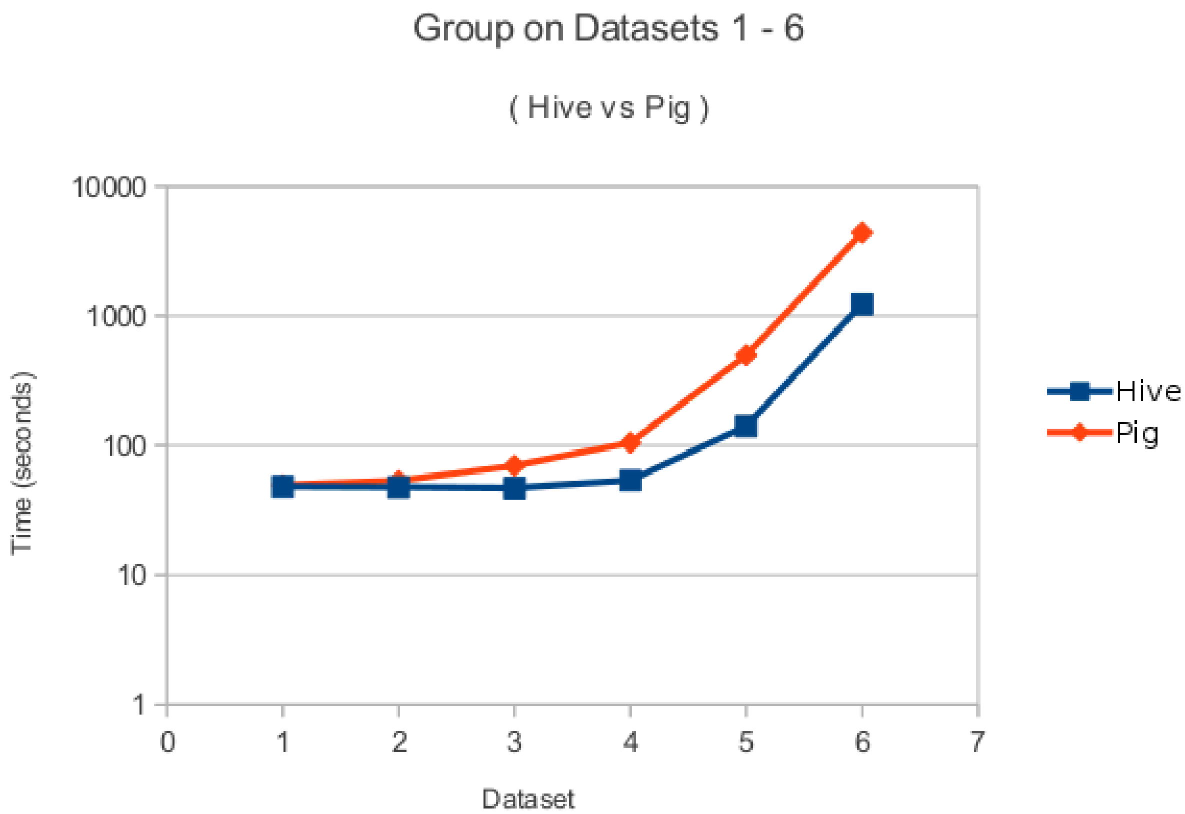


Figure 4.5: The runtime comparison between Pig and Hive (plotted in logarithmic scale) for the Group By operator. Taken from the ISO Report[13].

For example when running the TPC-H benchmarks for Pig, script 21 (q21_suppliers_who_kept_orders_waiting.pig) had a real-time runtime of 9065.29 seconds. 41% (or 3748.32 seconds) were required to execute the first `Group By`. In contrast, Hive only required 1031.23 seconds for the grouping of data. The script grouped data 3 times:

```
-- This Group By took up 41% of the runtime
gl = group lineitem by l_orderkey;

[...]

fo = filter orders by o_orderstatus == 'F';

[...]

ores = order sres by numwait desc, s_name;
```

Consequently, the excessive use of the `Group By` operator skews the benchmark results significantly. Re-running the scripts whilst omitting the the grouping of data produces the expected results. For example, running script 3 (q3_shipping_priority.pig) whilst omitting the `Group By` operator significantly reduces the runtime (to 1278.49 seconds real time runtime or a total of 12,257,630ms CPU time).

The fact that the `Group By` operator skews the TPC-H benchmark in favour of Apache Hive is supported by further experiments: as noted in section 5.2 a benchmark was carried out on a transitive self-join (the datasets consisted of 1,000 and 10,000,000 records. The scripts and dataset generator used for this benchmark were provided by Yu Liu). The former took Pig an average of 45.36 seconds (real time runtime) to execute; it took Hive 56.73 seconds. The latter took Pig 157.97 and Hive 180.19 seconds (again, on average). However adding the `Group By` operator to the scripts turned the tides: Pig is now significantly slower than Hive, requiring an average of 278.15 seconds[2]. Hive on the other hand required only 204.01 to perform the JOIN and GROUP operations.

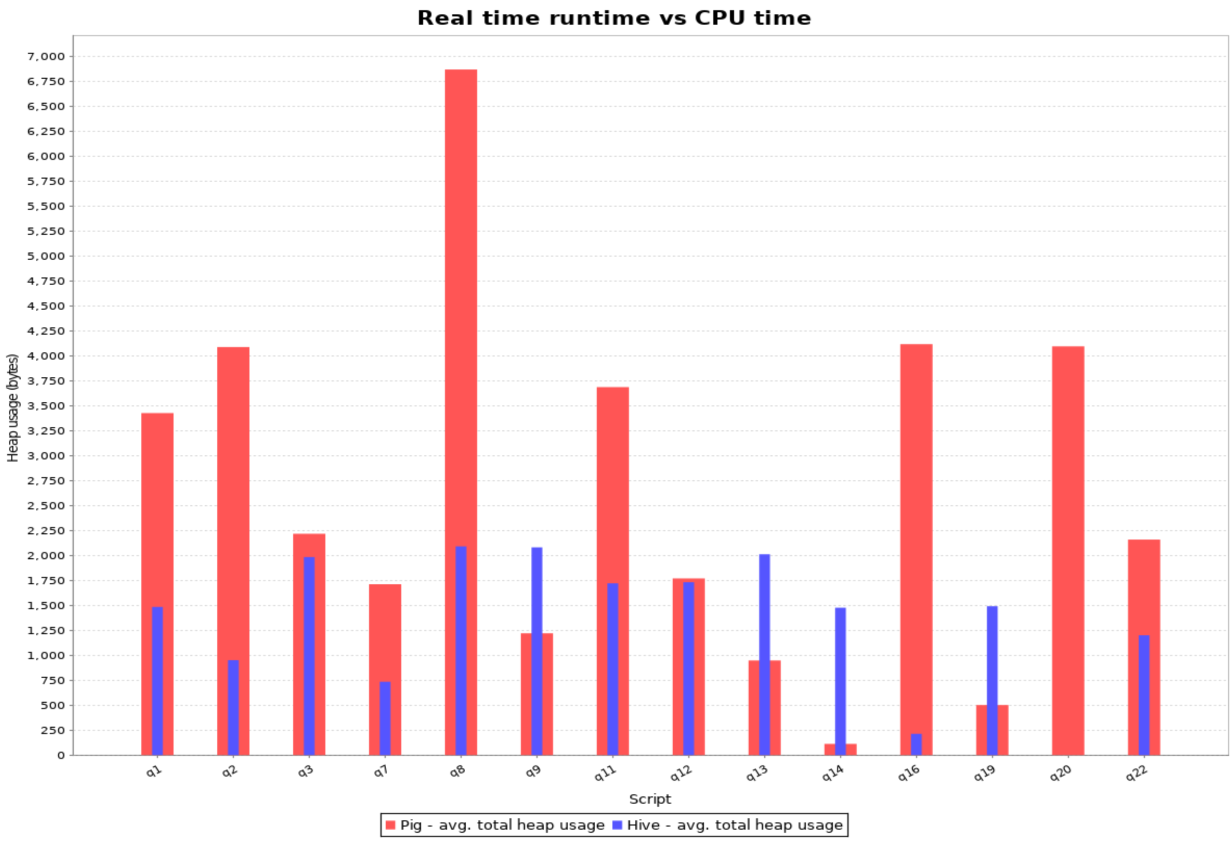


Figure 4.6: The total average heap usage (in bytes) of all 22 TPC-H benchmark scripts contrasted.

Another interesting artefact is exposed by figure 4.6: In all instances, Hive's heap usage is sig-

[2]As always, this refers to real time runtime

nificantly lower than that of Pig. This is explained by the fact that Hive does not need to build intermediary data structure, whilst Pig, being a declarative language, does.

4.5 Configuration

Manipulating the configuration of the original ISO benchmarks in an effort to determine optimal cluster usage produced interesting results.

For one, data compression is important and significantly impacts runtime performance of JOIN and GROUP BY operations in Pig. For example, enabling compression on dataset size 4 (which contains a large amount of random data) produces a 3.2% speed-up in real time runtime.

Compression in Pig can be enabled by setting the `pig.tmpfilecompression` flag to true and then specifying the type of compression `pig.tmpfilecompression.codec` to either gzip or lzo. Note that gzip produces better compression whilst LZO is much faster in terms of runtime.

By editing the entry for `mapred.reduce.slowstart.completed.maps` in Hadoop's conf/mapred-site.xml we can tune the percentage of map tasks that must be completed before reduce tasks can be created. By default, this value is set to 5% which was found to be too low for our cluster. Balancing the ratio of mappers and reducers is critical to optimizing performance: reducers should be started early enough so that data transfer is spread out over time and thus preventing network bottlenecks. On the other hand, reducers shouldn't be started late enough so that they do not use up slots that could be used by map tasks. Performance peaked when reduce tasks were fired after 70% of map jobs completed.

The maximum number of map and reduce tasks per node can be specified using `mapred.tasktracker.map.task` and `mapred.tasktracker.reduce.tasks.maximum`. Naturally care should be taken when configuring these: having a node with a maximum of 20 map slots but a script configured to use 30 map slots will result in significant performance penalties as the first 20 map tasks will run in parallel, but the additional 10 will only be spawned once the first 20 map tasks have completed execution (consequently requiring one extra round of computation). The same goes for the number of reduce tasks: as is illustrated by figure 4.7, performance peaks when a task requires just little below the maximum number of reduce slots per node.

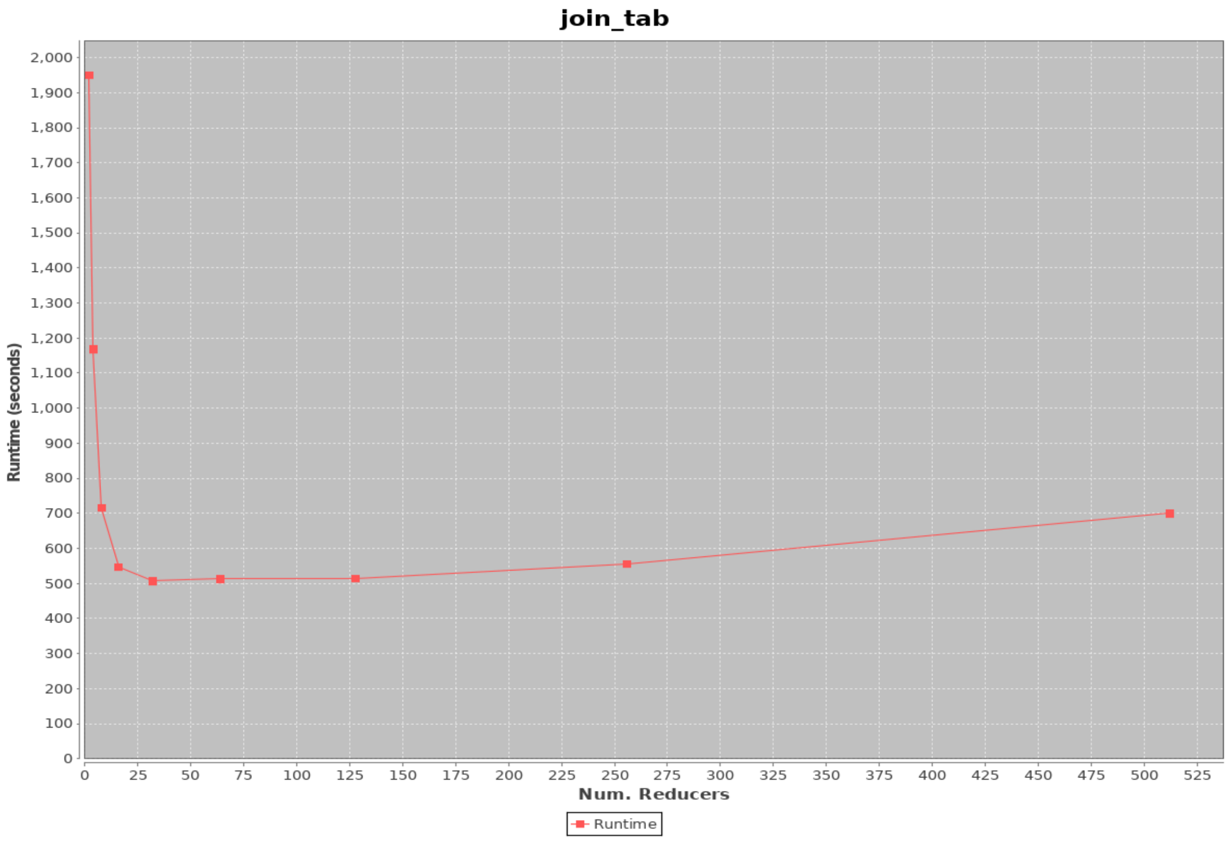


Figure 4.7: Real time runtimes contrasted with a variable number of reducers for join operations in Pig.

4.6 ISO addition - CPU runtimes

One outstanding item of the ISO report[13] was the contrasting between real time runtime and CPU runtime. As expected, cumulative CPU runtime was higher than real time runtime (since tasks are distributed between nodes).

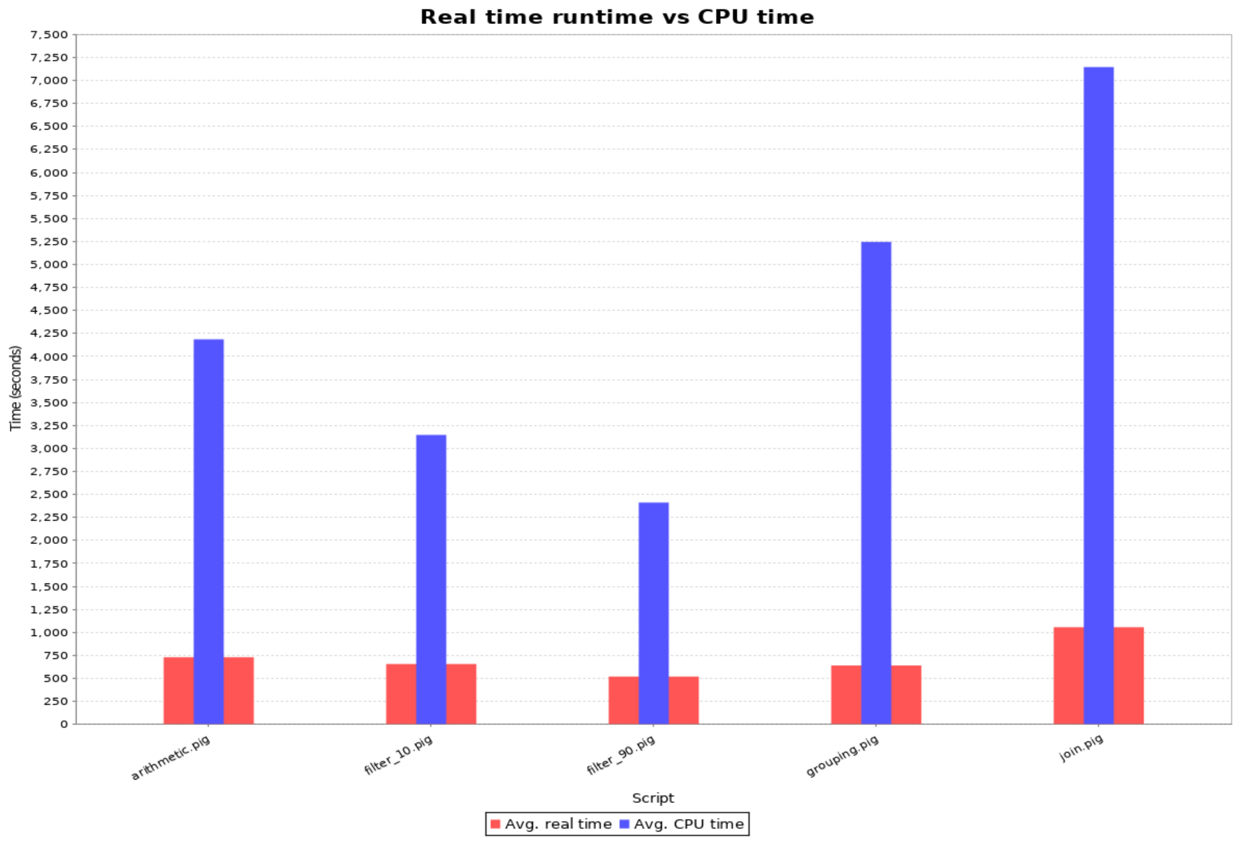


Figure 4.8: Real time runtime contrasted with CPU runtime for the ISO Pig scripts run on dataset size 5.

4.7 Conclusion

Running the discussed experiments allowed for the answering of 5 questions asked in chapter 3

1. How do Pig and Hive perform as other Hadoop properties are varied (e.g. number of map tasks)? Balancing the ratio of mappers and reducers had a big impact on real time runtime and consequently is critical to optimizing performance: reducers should be started early enough so that data transfer is spread out over time and thus preventing network bottlenecks. On the other hand, reducers shouldn't be started late enough so that they do not use up slots that could be used by map tasks. Performance peaked when reduce tasks were fired after X% of map jobs completed.

Care should also be taken when setting the maximum allowable map and reduce slots per node. For example having a node with a maximum of 20 map slots but a script configured to use 30 map slots will result in significant performance penalties as the first 20 map tasks will run in parallel, but the additional 10 will only be spawned once the first 20 map tasks have completed execution (consequently requiring one extra round of computation). The same goes for the number of reduce tasks: as is illustrated by figure 4.7, performance peaks when a task requires just little below the maximum number of reduce slots per node.

2. Do more complex datasets and queries (e.g. TPC-H benchmarks) yield the same results than the ISO? At first glance, running the TPC-H benchmarks contradicts the ISO results - in nearly all instances, Hive outperforms Pig. However closer examination revealed that nearly all TPC-H scripts relied heavily on the `Group By` operator - an operator which appears to be poorly implemented in Pig and which greatly degrades the performance of Pig Latin scripts (as demonstrated by the ISO benchmarks [13]). This leads to the conclusion that TPC-H is not an accurate benchmark as operators are not evenly distributed throughout the scripts: if one operator is poorly implemented, then this will skew the entire result set - as can be seen in section 4.4 with the `Group By` operator.

3. How does real time runtime scale with regards to CPU runtime? As expected given the cluster configuration (9 nodes). The real time runtime was between 15%-20% of the cumulative CPU runtime.

4. What should the ratio of map and reduce tasks be? The ratio for map and reduce tasks can be configured through `mapred.reduce.slowstart.completed.maps` Hadoop's conf/mapred-site.xml. The default value of 0.05 (i.e. 5%) was found to be too low. The optimal for the given cluster was at about 70%.

It should also be noted that the excessive use of the `Group By` operator within the TPC-H benchmarks skew results significantly (recall from [13] that Pig outperformed Hive in all instances except when using the `Group By` operator: when grouping data Pig was 104% slower than Hive[13]). Re-running the scripts whilst omitting the the grouping of data produces the expected results. For example, running script 3 (q3_shipping_priority.pig) whilst ommitting the `Group By` operator significantly reduces the runtime (to 1278.49 seconds real time runtime or a total of 12,257,630ms CPU time).

Additional benchmarks were run - their results support the above claim.

No clear answer can be given with regards to which schedulers "is best"3. Benchmarks run on the FIFO scheduler and fair scheduler prove that they do exactly as intended and that the choice of scheduler really depends on your intentions. Similarily, earlier in this report, the question *"what are the different schedulers good at?"* was posed. Results do not vary from the descriptions presented in the background analysis chapter (although it should be noted that due to time constraints the Deadline Constraint Scheduler and Intelligent Schedulers were not benchmarked).

5. How and based on what should I choose a specific scheduler?
The type of scheduler needed depends on your organisational needs. If you are running a small, single-user cluster then the FIFO scheduler is perfectly appropriate. As the name implies, the fair scheduler schedules jobs in such a way that each receive an equal share of the available resources and ensures that smaller jobs make progress even in the presence of large jobs without actually starving the large job. Thus the fair scheduler is idea sharing a cluster between different users within the same organization and when jobs are of varied / mixed sizes.

The capacity scheduler should be used for large clusters shared between different organizations or third parties. As discussed in chapter 2, the capacity scheduler is a more fine-grained version of the fair scheduler and imposes access restrictions as well as limits the waste of excess capacity . The capacity scheduler also supports pre-emption. Pre-emption with the capacity scheduler differs to fair scheduling pre-emption in that it uses priorities as opposed to time.

The HOD scheduler is no longer actively supported and should consequently not be used within a production environment. The same goes for the deadline constraint scheduler, priority parallel task scheduler and the intelligent schedulers discussed in chapter 2 which are scientific proves of concept rather than production standard schedulers.

The reason for Hive outperforming Pig's `Group By` operator will be discussed in the next chapter.

Chapter 5

Pig and Hive under the hood

5.1 Syntax Trees, Logical and Physical Plans

Recall that the ISO benchmarks showed a significant performance difference between Hive and Pig. For example, as illustrated in table 5.1 below, Pig is, on average, 46% faster than Hive when performing arithmetic operations.

Dataset size	**% Pig being faster**
1	0.061%
2	3%
3	32%
4	72%
5	83%
6	85%
Avg.:	46%

Table 5.1: The percentage (in terms of real time) that Pig is faster than Hive when performing **arithmetic** operations

In an effort to come to the bottom of this difference in performance, the logical plans of the scripts were examined. To this end, the **EXPLAIN** keyword was prepended to the Hive QL query resulting in the parser printing the abstract syntax tree (ABS) and logical plan for the script (as opposed to compiling and executing it) as follows[1]:

```
EXPLAIN
SELECT  (dataset_30000000.age * dataset_30000000.gpa + 3) AS F1,
```

[1]Map and reduce configuration information were omitted and formatting was adjusted to make the ABS easier to read.

```
                (dataset_30000000.age/dataset_30000000.gpa − 1.5) AS F2
FROM dataset_30000000
WHERE dataset_30000000.gpa > 0;




ABSTRACT SYNTAX TREE:
  (TOK_QUERY
    (TOK_FROM (TOK_TABREF (TOK_TABNAME dataset_30000000)))
    (TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE))
    (TOK_SELECT
     (TOK_SELEXPR
       (+
         (*
           (. (TOK_TABLE_OR_COL dataset_30000000) age)
           (. (TOK_TABLE_OR_COL dataset_30000000) gpa)
         ) 3)
       F1)
       (TOK_SELEXPR
         (-
           (/
             (. (TOK_TABLE_OR_COL dataset_30000000) age)
             (. (TOK_TABLE_OR_COL dataset_30000000) gpa)
            ) 1.5)
        F2))
   (TOK_WHERE (> (. (TOK_TABLE_OR_COL dataset_30000000) gpa) 0)))
  )
```

As expected, the ABS references the correct table: `(TOK_TABREF (TOK_TABNAME dataset_30000000))`. Before defining the operations, Hive specifies that the output for the query should be written to a temporary file before it is written to stdout (i.e. the console): `(TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE))`. Next, the order of the arithmetic operations is defined:

$$\left.\begin{array}{l}\text{Extract column "age" from dataset}\\ \text{Extract column "gpa" from dataset}\\ \text{Apply multiplication operator}\\ \text{Add 3 to the result}\end{array}\right\}\text{F1}$$

$$\left.\begin{array}{l}\text{Extract column "age" from dataset}\\ \text{Extract column "gpa" from dataset}\\ \text{Apply division operator}\\ \text{Add 1.5 to the result}\end{array}\right\}\text{F2}$$

Then, the selection constraint is applied to the overall expression:

`TOK_WHERE (> (. (TOK_TABLE_OR_COL dataset_30000000) gpa) 0).`

Consequently, we can conclude that the Abstract Syntax Tree generated by Hive corresponds to the original query and that hence any performance difference between Pig and Hive must be either due to the interpretation of the syntax tree by the Hive logical plan generator, differences between the Pig and Hive optimizers or due to differences in how the physical plans are translated.

Hive logical plans are composed of individual "steps". These "steps" are called stages, each of which may be dependent on another stage. Every stage is either a map job, a reduce job, a merge or sampling job or a limit[2]. The more complex a query, the more stages a logical plan will contain (and consequently the more processing is require to execute the job)[19].

Our logical plan's first stage (see Appendix B for the complete plan) uses a `TableScan` operation to take the entire table, dataset_30000000, as input and produce two output columns, _col0 and _col1. The `Filter Operator` ensures that the `Select Operator` only considers rows in which the gpa column has a value greater than zero. The `Select Operator` then applies two arithmetic expressions to produce _col0 and _col1 (the former being produced by ((age * gpa) + 3) and the latter by ((age / gpa) - 1.5)). All of this is done in the job's map task (indicated on the third line via the string `Alias -> Map Operator Tree)` - nothing is done inside the reduce side of the job (and hence the logical plan does not contain a reduce section for either stages (i.e. the plan contains no reduce operator tree):

[2]There do exist other types of (less common) stages, but to keep things simple this report will only refer to the aforementioned stages.

```
Stage: Stage-1
    Map Reduce
      Alias -> Map Operator Tree:
        dataset_30000000
          TableScan
            alias: dataset_30000000
            Filter Operator
              predicate:
                  expr: (gpa > 0.0)
                  type: boolean
              Select Operator
                expressions:
                      expr: ((age * gpa) + 3)
                      type: float
                      expr: ((age / gpa) - 1.5)
                      type: double
                outputColumnNames: _col0, _col1
```

We can further see that data compression is disabled and that the input is treated as text:

```
compressed: false
  GlobalTableId: 0
    table:
     input format: org.apache.hadoop.mapred.TextInputFormat
        output format: org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
```

Since the script does not make use of the LIMIT clause, Stage-0 is a no-op stage:

```
  Stage: Stage-0

    Fetch Operator

      limit: -1
```

Executing `pig -x local -e 'explain -script arithmetic.pig'` produces both the logical and physical plan for the corresponding Pig Latin script (see Appendix B for the complete logical and physical plans. Pig's logical plan loosely corresponds to Hive's Abstract Syntax Tree). As can be inferred below, Pig's logical plan is slightly more complex and intricate than the one produced by the Hive interpreter.

```
        B: (Name: LOStore Schema: #49:double,#54:double)
        ColumnPrune:InputUids=[38, 43]ColumnPrune:OutputUids=[38, 43]
|
|---B: (Name: LOForEach Schema: #49:double,#54:double)
    |   |
    |   (Name: LOGenerate[false,false] Schema: #49:double,#54:double)
    |   |   |
    |   |   (Name: Add Type: double Uid: 49)
    |   |   |
```

```
|   |   |---(Name: Multiply Type: double Uid: 46)
|   |   |   |
|   |   |   |---(Name: Cast Type: double Uid: 20)
|   |   |   |   |
|   |   |   |   |---age:(Name: Project Type: bytearray Uid: 20 Input: 0 Column: (*))
|   |   |   |
|   |   |   |---(Name: Cast Type: double Uid: 21)
|   |   |       |
|   |   |       |---gpa:(Name: Project Type: bytearray Uid: 21 Input: 1 Column: (*))
|   |   |
|   |   |---(Name: Cast Type: double Uid: 47)
|   |       |
|   |       |---(Name: Constant Type: int Uid: 47)
|   |   |
|   |   (Name: Subtract Type: double Uid: 54)
|   |   |
|   |   |---(Name: Divide Type: double Uid: 52)
|   |   |   |
|   |   |   |---(Name: Cast Type: double Uid: 20)
|   |   |   |   |
|   |   |   |   |---age:(Name: Project Type: bytearray Uid: 20 Input: 2 Column: (*))
|   |   |   |
|   |   |   |---(Name: Cast Type: double Uid: 21)
|   |   |       |
|   |   |       |---gpa:(Name: Project Type: bytearray Uid: 21 Input: 3 Column: (*))
|   |   |
|   |   |---(Name: Constant Type: double Uid: 53)
|   |
|   |---(Name: LOInnerLoad[0] Schema: age#20:bytearray)
|   |
|   |---(Name: LOInnerLoad[1] Schema: gpa#21:bytearray)
|   |
|   |---(Name: LOInnerLoad[0] Schema: age#20:bytearray)
|   |
|   |---(Name: LOInnerLoad[1] Schema: gpa#21:bytearray)
|
|---A: (Name: LOLoad Schema: age#20:bytearray,gpa#21:bytearray)ColumnPrune:RequiredColumns=[1, 2]Co
```

Unlike the one produced by Hive, Pig's logical plan flows from bottom to top with the lines connecting the operators indicating the exact flow path. Each line contains an operator, each of which have an attached schema and, optionally, an expression. Each operator in turn is connected to another operator, and as such, one can loosely interpret an operator as a "stage" or "step" in the plan's execution. Starting from bottom to top, the first such step is the `Load` operation which, as its name implies, tells the compiler what data to use. Next, a for-loop is used to extract the relevant columns (gpa and age) from each row. The extracted fields are cast to their appropriate type (double) and the relevant arithmetic operations are applied (since the plan reads from bottom to top, the addition of the constant is applied last since it appears at the top of the plan). The plan is illustrated in figure 5.6 below.

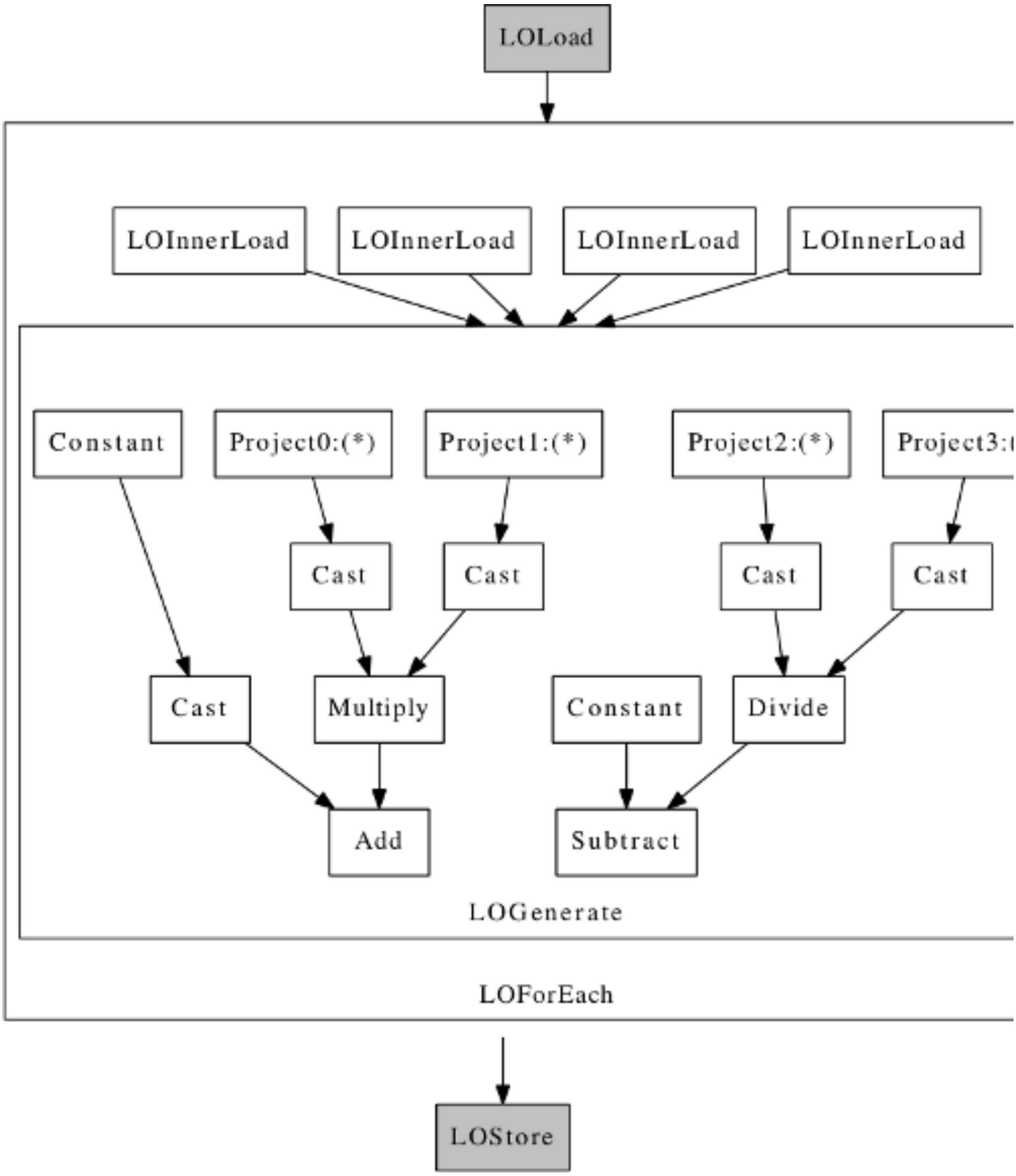


Figure 5.1: Pig logical plan for the script arithmetic.pig

Next, the logical plan is fed to the optimizer, which attempts to make the entire operation more efficient by replacing redundant statements and pushing filters as far up the plan as possible (ensuring that less data needs to be processed in subsequent steps). This optimizer produces the physical plan which it in turn uses to construct the map-reduce plan. Note that unlike Hive, which considers execution in different stages (some of which may be map-reduce) Pig seems to view everything in terms map tasks and reduce tasks.

That is, the difference between the logical and physical plan lie with the operators: the logical plan describes the logical operations that have to be executed by Pig; the physical plan describes the physical operators that are needed to implemented the aforementioned logical operators.

In order to produce a set of map-reduce tasks, the physical plan is scanned sequentially and individual map and reduce operations are identified. Once this is complete, the Pig compiler once again tries to optimize the plan by, for example by identifying sorts and pushing them into the shuffle phase.

```
#--------------------------------------------------
# Map Reduce Plan
#--------------------------------------------------
MapReduce node scope-22
Map Plan
B: Store(hdfs://ebony:54310/user/bj112/dataset_30000000_projection:PigStorage) - scope-21
|
|---B: New For Each(false,false)[bag] - scope-20
    |   |
    |   Add[double] - scope-8
    |   |
    |   |---Multiply[double] - scope-5
    |   |   |
    |   |   |---Cast[double] - scope-2
    |   |   |   |
    |   |   |   |---Project[bytearray][0] - scope-1
    |   |   |
    |   |   |---Cast[double] - scope-4
    |   |       |
    |   |       |---Project[bytearray][1] - scope-3
    |   |
    |   |---Cast[double] - scope-7
    |       |
    |       |---Constant(3) - scope-6
    |   |
    |   Subtract[double] - scope-17
    |   |
    |   |---Divide[double] - scope-15
    |   |   |
    |   |   |---Cast[double] - scope-12
    |   |   |   |
    |   |   |   |---Project[bytearray][0] - scope-11
    |   |   |
    |   |   |---Cast[double] - scope-14
    |   |       |
    |   |       |---Project[bytearray][1] - scope-13
    |   |
    |   |---Constant(1.5) - scope-16
    |
    |---A: Load(/user/bj112/data/4/dataset_30000000:PigStorage('')) - scope-0--------
Global sort: false
----------------
```

We can conclude that there is no difference in how arithmetic operations are treated at a logical level. Both load the table, perform casts where appropriate and then perform multiplication, division, addition and subtraction in line with the rules of mathematics. Similarly, the map-reduce plans for both are exactly the same. These findings are supported when analysing the plans of TPC-H (and consequently more complex) scripts (see appendix B)

These findings lead to the assertion that the performance difference between Pig and Hive can be attributed to the fact that at a bytecode level, Hive's implementation of the map-reduce plans are less efficient than Pig. As will be discussed in the following sections, Pig consists of more high-quality code than Hive.

The partial publication of the ISO results on an online tech discussion board[20] and the Pig/Hive developers mailing list confirmed the suspicion that performance differences between Pig and Hive are attributed to the bytecode level as opposed to the logical interpretation of the Hive QL or Pig Latin scripts. As one user going by the pseudonym *zsxwing* pointed out[20]:

> Hive's `JOIN` implementation is less efficient than Pig. I have no idea why Hive needs to create so many objects (Such operations are very slow and should have been avoided) in the Join implementation. I think that's why Hive does Join more slowly than Pig. If you are interested in it, you can check the CommonJoinOperator code by yourself. So I guess that Pig usually more efficient as its high quality codes.

Indeed, examining the most recent, stable release of both projects (Hive v.0.11.0 and Pig v.0.11.1) yielded interesting results.

5.1.1 General design quality and performance

Although lines of code (LOC) is an imprecise metric for comparing the quality between software projects, an overall rule of thumb is the less code, the better. As such, Pig scores higher than Hive, with a total code (and comment) base of 154,731 LOC (25% comments, 13% blank lines and the rest consists of actual code). Hive, although more sparsely documented, consists of 187,916 LOC

(23% comments, 12% blank lines and the rest consists of actual code). That is, Pig's codebase is nearly 18% smaller than Hive. Furthermore, a static code analysis of both projects showed 134,430 issues in the Hive codebase (that's 71.5 issues per 100 lines of code) and 38,452 issues with the Pig codebase (that's 24.85 issues per 100 lines of code). The code analysed included all test classes and the analysis was concerned with identifying general bad coding practices (such as serializable inner classes, suspicious reference comparisons to constants, confusing method names etc), correctness (an example of incorrect code would be a call to equals(null), infinite loops or an illegal format string), malicious code vulnerabilities and security issues, performance bottlenecks, questionable or redundant code (such as unread fields), overall coding style, duplicate code as well as dependency analysis.

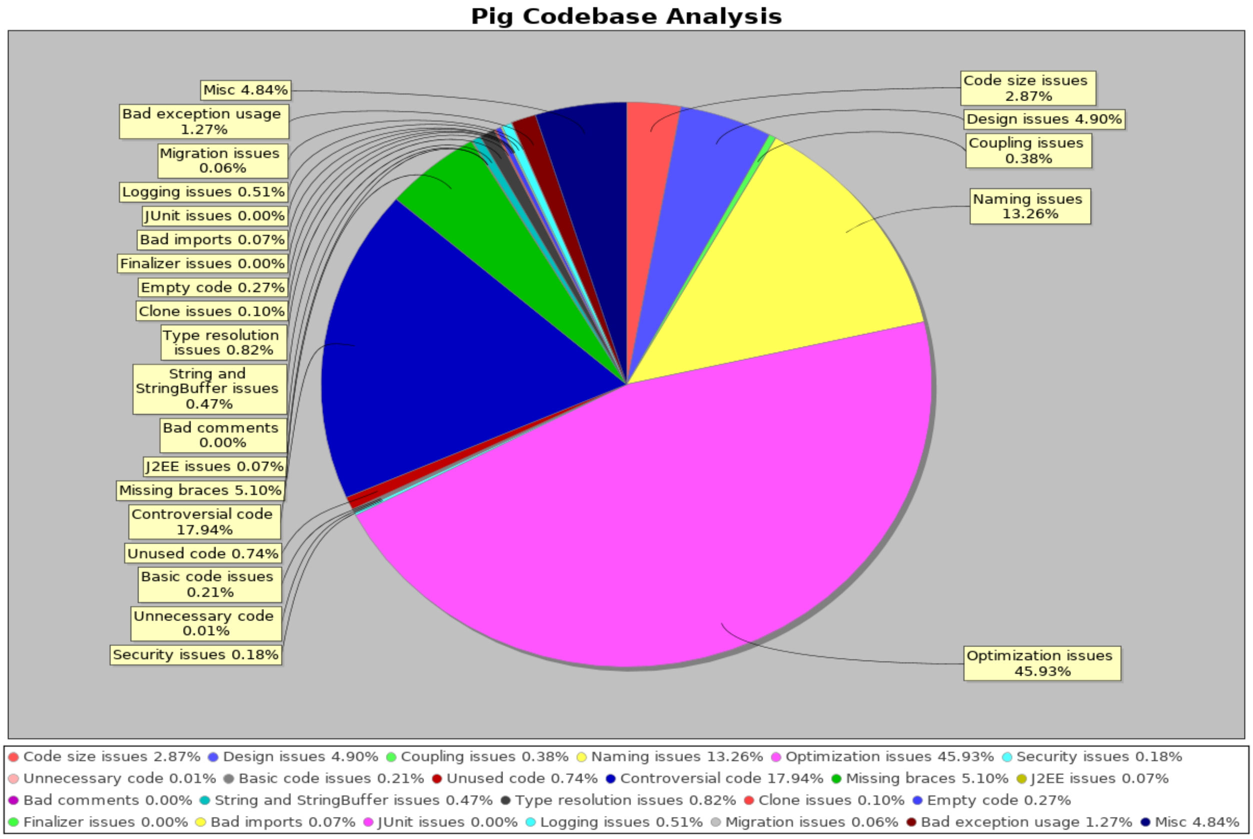


Figure 5.2: Summary of the issues found with the Pig codebase.

For Pig, the issue summary is as follows:

Issue category	**Num. of issues in Pig**	**Num. of issues in Hive**
Code size	1104	3718
Design	1883	6778
Coupling	147	1014
Naming	5100	16764
Optimization	17660	59366
Security vulnurabilities	68	165
Unnecessary code	3	134
Basic	80	430
Unused code	286	1055
Controversial code	6897	27666
Missing braces	1961	4518
J2EE issues	25	140
Bad commenting	0	0
String and StringBuffer issues	179	790
Type resolution issues	316	1891
Clone implementation issues	39	35
Empty code	103	223
Finalizer issues	1	1
Import Stmts	26	37
JUnit issues	0	2687
Logging issues	198	701
Migration issues	22	426
Strict exceptions / Bad exception handling	489	745
Misc	1862	5016

Table 5.2: Summary of issues found within the Pig and Hive codebase.

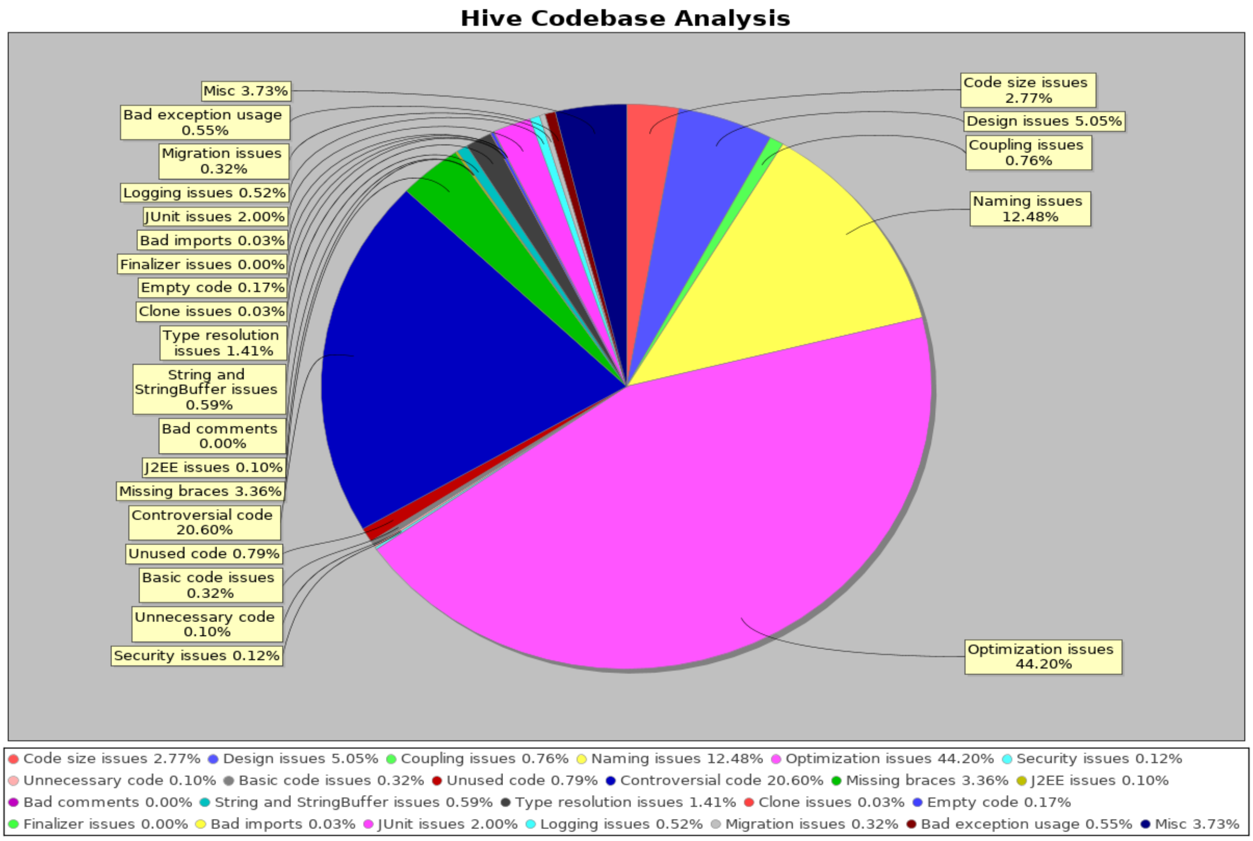


Figure 5.3: Summary of the issues found with the Hive codebase.

Although the Hive codebase contains much more issues, the distribution of the types of code issues are approximately equal in each codebase (as can be seen in figures 5.3 and 5.2): 42.2% of the problems with the Hive codebase were related to code optimization as opposed to 45.93% in the Pig codebase. Similarly, both codebases have roughly the same percentage of design flaws: 5.05% for Hive and 4.9% for Pig. The same goes for codesize issues (2.87% for Pig, 2.77% for Hive), naming issues (13.26% for Pig, 12.48% for Hive) and security vulnurabilities (0.18% for Pig and 0.12% for Hive). Interestingly however, Pig does not exhibit any JUnit test issues (whilst 2% of the issues found in Hive's codebase relate to JUnit tests), indicating that possibly a lot of emphasis has been placed on testing.

The fact that an equal percentage of found issues relate to correct naming of variables, methods and classes may be explained by the fact that inexperienced programmers (students, interns, recent graduates etc) may have contributed to fractions of the codebase. As one can assume that every company or software project would roughly allocate the same amount of rookies to a project the percentage of ”rookie mistakes” may be similar across software projects.

In terms of issues per lines of code, Pig is vastly superior to Hive (see figures 5.4 and 5.5): Hive has 31.59 optimization issues per 100 lines of code (LOC); Pig only 11.41. Furthermore Hive is 3.61 design problems per 100 LOC; Pig only 0.71 (indicating that Pig's codebase is of vastly superior design). Although relatively little unused code (0.56 unused code issues per 100 LOC), Hive's codebase still contains 3 times as much unused code as Pig (0.18 unused code per 100 LOC). Hive's codebase is also far more bloated than Pig's: at 1.98 code size issues for every 100 LOC, Hive code is nearly 3 times as complex as Pig's.

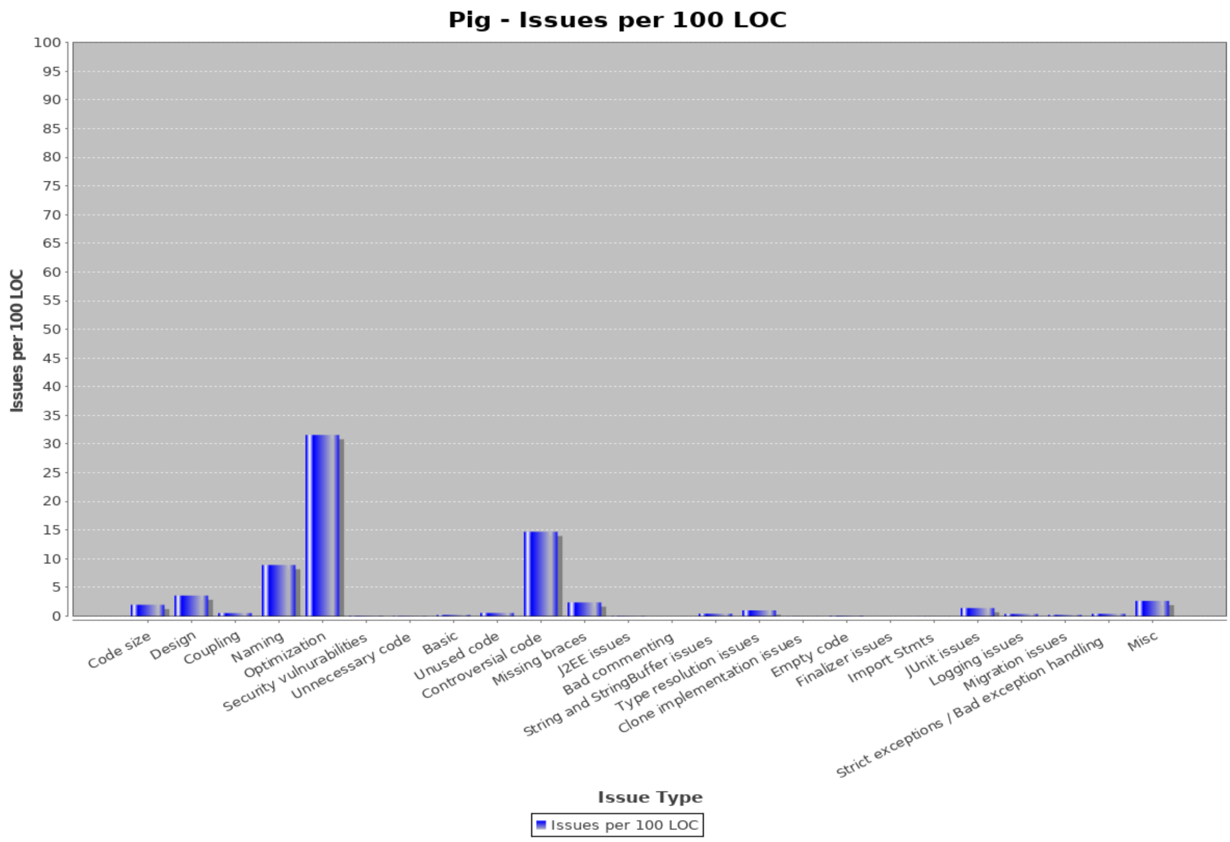


Figure 5.4: Hive - number of issues per 100 lines of code

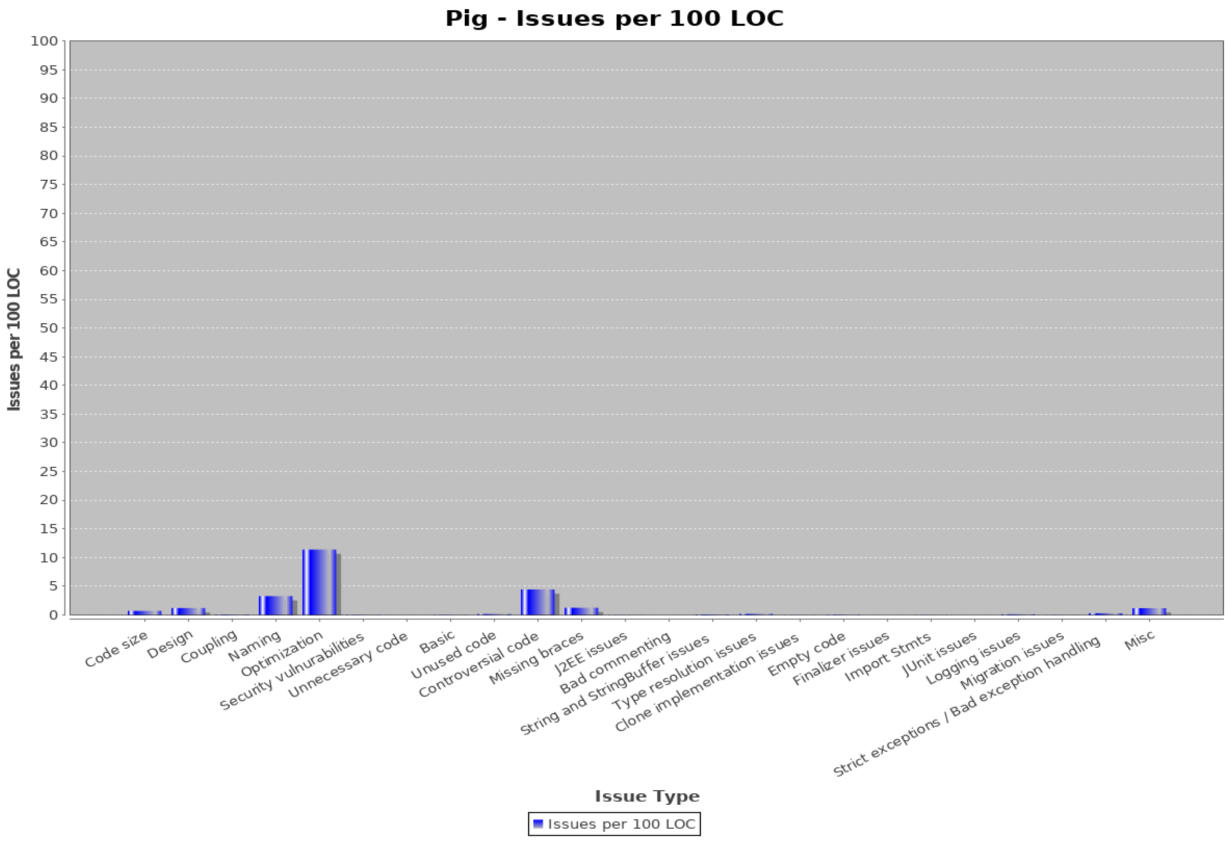


Figure 5.5: Pig - number of issues per 100 lines of code

The optimizations that could be applied to Hive's codebase are as follows:

1. Use StringBuffer when appending strings - In 184 instances, the concatination operator (+=) was used when appending strings. This is inherintly inefficient - instead Java's StringBuffer or StringBuilder class should be used. 12 instances of this optimization can be applied to the `GenMRSkewJoinProcessor` class and another three to the optimizer. `CliDriver` uses the + operator inside a loop, so does the column projection utilities class (`ColumnProjectionUtils`) and the aforementioned skew-join processor. Tests showed that using the StringBuilder when appending strings is 57% faster than using the + operator (using the `StringBuffer` took 122 milliseconds whilst the + operator took 284 milliseconds - see appendix D for the test case). The reason as to why using the `StringBuffer` class is preferred over using the + operator, is because

```
String third = first + second;
```

gets compiled to:

```
StringBuilder builder = new StringBuilder( first );
builder.append( second );
third = builder.toString();
```

Therefore, building complex strings inside loops requires many instantiations (and as discussed below, creating new objects inside loops is inefficient)[21].

2. Use arrays instead of `List` - Java's `java.util.Arrays` class `asList` method is more efficient at creating lists from arrays than using loops to manually iterate over the elements (using `asList` is computationally very cheap, O(1), as it merely creates a wrapper object around the array; looping through the list however has a complexity of O(n) since a new list is created and every element in the array is added to this new list[**?**]).[24] As confirmed by the experiment detailed in Appendix D, the Java compiler does not automatically optimize and replace tight-loop copying with `asList`: the loop-copying of 1,000,000 items took 15 milliseconds whilst using `asList` is instant.

Four instances of this optimization can be applied to Hive's codebase (two of these should be applied to the Map-Join container - `MapJoinRowContainer`) - lines 92 to 98:

```
for (obj = other.first(); obj != null; obj = other.next()) {
    ArrayList<Object> ele = new ArrayList(obj.length);
    for (int i = 0; i < obj.length; i++) {
      ele.add(obj[i]);
    }
    list.add((Row) ele);
  }
```

3. Unnecessary wrapper object creation - In 31 cases, wrapper object creation could be avoided by simply using the provided static conversion methods. As noted in the PMD documentation[**?**], "using these avoids the cost of creating objects that also need to be garbage-collected later."

For example, line 587 of the `SemanticAnalyzer` class, could be replaced by the more efficient `parseDouble` method call:

```
// Inefficient:
Double percent = Double.valueOf(value).doubleValue();
// To be replaced by:
Double percent = Double.parseDouble(value);
```

Our test case in Appendix D confirms this: converting 10,000 strings into integers using `Integer.parseInt(gen.ne` (i.e. creating an unnecessary wrapper object) took 119 on average; using `parseInt()` took only 38. Therefore creating even just one unnecessary wrapper object can make your code up to 68% slower.

4. Converting literals to strings using + "" - Converting literals to strings using `+ ""` is quite inefficient (see Appendix D) and should be done by calling the `toString()` method instead: converting 1,000,000 integers to strings using `+ ""` took, on average, 1340 milliseconds whilst using the `toString()` method only required 1183 milliseconds (hence adding empty strings takes nearly 12% more time).

89 instances of this using `+ ""` when converting literals were found in Hive's codebase - one of these are found in the `JoinUtil`.

5. Avoid manual copying of arrays - Instead of copying arrays as is done in `GroupByOperator` on line 1040 (see below), the more efficient `System.arraycopy` can be used (`arraycopy` is a native method meaning that the entire memory block is copied using `memcpy` or `mmove`).

```
// Line 1040 of the GroupByOperator
for (int i = 0; i < keys.length; i++) {
        forwardCache[i] = keys[i];
}
```

Using `System.arraycopy` on an array of 10,000 strings was (close to) instant whilst the manual copy took 6 milliseconds. 11 instances of this optimization should be applied to the Hive codebase.

6. Avoiding instantiation inside loops - As noted in the PMD documentation, "new objects created within loops should be checked to see if they can created outside them and reused."[**?**].

Referring to the test case in Appendix D, declaring variables inside a loop (i from 0 to 10,000) took 300 milliseconds whilst declaring them outside took only 88 milliseconds (this can be explained by the fact that when declaring a variable outside the loop, its reference will be re-used for each iteration. However when declaring variables inside a loop, new references will be created for each iteration. In our case, 10,000 references will be created by the time that this loop finishes, meaning lots of work in terms of memory allocation and garbage collection). 1623 instances of this optimization can be applied.

7. Making local variables and method arguments final - This optimization is arguable - some authors claim[**?**] that making local variables final improves garbage collection. However as the `final` keyword does not actually appear in class files, they may not impact performance and garbage collection. Regardless, using the `final` keyword is good practice and should be followed (the codebase contains 23,600 instances in which local variables could be made final; 33,815 instances in which method arguments could be final).

8. Replacing `startsWith` with `charAt` - There are 9 instances in which this optimization can be applied.

The Pig codebase suffers from the same performance issues, albeit there are fewer of them:

1. Use StringBuffer when appending strings - This optimization can be applied 62 times.

2. Use arrays instead of `List` - This optimization can be applied once.

3. Unnecessary wrapper object creation - This optimization can be applied 15 times.

4. Converting literals to strings using + "" - This optimization can be applied 12 times.

5. Avoid manual copying of arrays - This optimization can be applied 14 times. Two of these performance bottlenecks are found in the class responsible for implementing a for-each, `POForEach`, and one in `POSkewedJoin`.

6. Avoiding instantiation inside loops - 494 optimizations of this type can be applied. Most of these are found in the plan generator, optimizer and HBase storage classes however several are also found in the relational operator classes: `LOSort`, `LOJoin`, `LOGenerate`, `LOUnion`, `LOCogroup`.

7. Making local variables and method arguments final - Although questionable whether it actually improves performance, 7962 local variables and 9084 method arguments could be final.

8. Replacing `startsWith` with `charAt` - There are 10 instances in which this optimization can be applied.

There is also an 8th optimization that could be applied:

9. Replacing vectors with array lists - Vectors synchronized, making them slower than array lists. Therefore using vectors in circumstances where thread-safety is not an issue will decrease performance. The test code in Appendix D added 9,999,999 integers to a vector: this took 2367 milliseconds. Adding them to an array list on the other hand took only 934 milliseconds.

6 instances of this optimization can be applied.

Furthermore, several optimization issues shared between the two codebases can be attributed to inefficient string operations:

1. Inefficient StringBuffer appends - When using the `StringBuffer`, it is more efficient to append an individual character as a character as opposed to a string (i.e. sb.append('a') instead of sb.append("a")). The test case in Appendix D showed that a string append took 52 milliseconds whilst a char append took 44 milliseconds (the letter 'a' was appended 1,000,000 times). It should be noted that this optimization is is hardly relevant for the Hive codebase: only 4 instances of inefficient appends exist, none of which are inside core classes. Pig on the other hand contains 37 of these instances, 5 of which inside the class used for arithmetic evaluations.

2. Inefficient use of indexOf() - When looking up the index of a single character, the character should be specified as type `char` and not string (i.e. indexOf('a') executes faster than indexOf("a"). The test case presented in Appendix D used a string of 1000 characters and performs a call to `indexOf` 5,000 times for each string and character look-ups: passing a string to `indexOf` took 9 milliseconds whilst passing a character to `indexOf` took 3 milliseconds. Relatively few of these optimizations can be applied to either codebase: 6 for Pig and 17 for Hive.

4. Dublicate strings - In 111 cases, Pig contains dublicate strings that should be replaced by constants; the same goes for 680 cases in the Hive codebase.

5. Unneccesary call to toString() - Calling `toString()` on a `String` object is unnecessary and should be avoided. Fortunately, relative few instances of this bad practice were found in either codebase.

Other minor optimizations include making final fields static (24 in Hive; 1 in Pig) and this decreasing runtime overhead for the given object[**?**].

Another interesting observation is the fact that Pig's MR compiler and logical plan builder contain the most optimization issues. That is, having counted the number of optimization issues per class[3], the classes with the most amount of optimization issues are `MRCompiler`, `LogicalPlanBuilder` and `LogToPhyTranslationVisitor`). For Hive, it is the `ThriftHiveMetastore` and the `SemanticAnalyzer`.

For a complete listing of classes and their number of optimization issues see Appendix E.

[3]Note: This refers to first 8 optimization issues listed above and does **not** include string optimization issues.

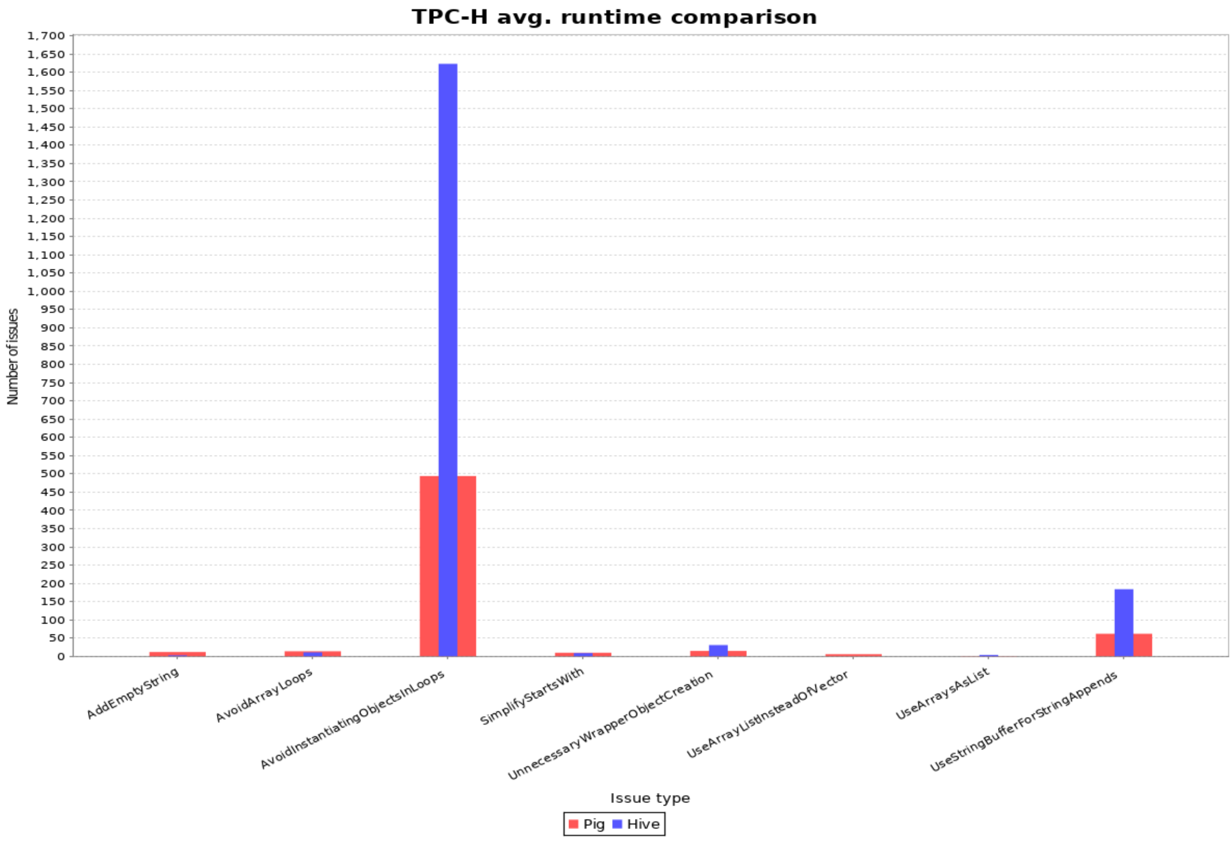


Figure 5.6: Comparison of the number and types of optimization issues found in the Pig and Hive codebases.

Pig and Hive both share many of the same design issues, however the Hive codebase is plagued by many more than Pig. For one, both codebases contain many deeply nested if-statements (this complicates the code unnecessarily and makes it difficult to maintain.). The Hive codebase contains 56 deeply nested conditionals. For example methods in the class responsible for implementing the Group By operator `GroupByOperator.java`, contain if-statements of depth 5:

```
if (sfs.size() > 0) {
    StructField keyField = sfs.get(0);
    if (keyField.getFieldName().toUpperCase().equals(
        Utilities.ReduceField.KEY.name())) {
      ObjectInspector keyObjInspector = keyField.getFieldObjectInspector();
      if (keyObjInspector instanceof StandardStructObjectInspector) {
        List<? extends StructField> keysfs =
          ((StandardStructObjectInspector) keyObjInspector).getAllStructFieldRefs();
        if (keysfs.size() > 0) {
```

```
            // the last field is the union field, if any
            StructField sf = keysfs.get(keysfs.size() - 1);
            if (sf.getFieldObjectInspector().getCategory().equals(
                ObjectInspector.Category.UNION)) {
              unionExprEval = ExprNodeEvaluatorFactory.get(
                new ExprNodeColumnDesc(TypeInfoUtils.getTypeInfoFromObjectInspector(
                sf.getFieldObjectInspector()),
                keyField.getFieldName() + "." + sf.getFieldName(), null,
                false));
              unionExprEval.initialize(rowInspector);
            }
          }
        }
      }
    }
```

Likewise, Pig contains 38 instances of deeply-nested if-statements. For example, `DuplicateForEachColumnRewrite.java`:

```
if (exp.getFieldSchema()!=null) {
                    if (flatten && (exp.getFieldSchema().type == DataType.BAG || exp.
                        getFieldSchema().type == DataType.TUPLE)) {
                        List<LogicalFieldSchema> innerFieldSchemas = null;
                        if (exp.getFieldSchema().type == DataType.BAG) {
                            if (exp.getFieldSchema().schema!=null) {
                                if (exp.getFieldSchema().type == DataType.BAG) {
                                    // assert(fieldSchema.schema.size() == 1 && fieldSchema.
                                        schema.getField(0).type == DataType.TUPLE)
                                    if (exp.getFieldSchema().schema.getField(0).schema!=null
                                        )
                                        innerFieldSchemas = exp.getFieldSchema().schema.
                                            getField(0).schema.getFields();
                                } else {
```

```
                    if (exp.getFieldSchema().schema!=null)
                        innerFieldSchemas = exp.getFieldSchema().schema.
                            getFields();
                }
            }
        }
```

Rookie mistakes such as uncommented empty constructors (641 in Hive; 126 in Pig), uncommented empty methods (161 in Hive; 126 in Pig) and missing braces (in Hive, 4490 if-statements were missing braces, 14 if-else-statements and 14 for-loops missed braces. Pig's codebase contains 1205 if-statements with missing braces 632 if-else-statements with missing braces and 109 for-loops with missing braces) were painstakingly common. Whilst other design issues are easily forgiveable, such basic mistakes should not be in release versions of any software. Other instances of rather basic bad practices include:

- Unnecessary local variable assignments before returns (for example int x = something(); return x; instead of just return doSomething()).
- Missing breaks inside switch statements.
- Missing static methods in non-instantiatable classes (this means that the given methods cannot actually be used since the class constructors themselves are private and there is no way of calling the non-static methods. Fortunately only one such class exists in the Pig codebase and two in Hive (indicating incomplete or obsolete code).
- Empty methods in abstract classes are not tagged as abstract (making methods abstract helps improve readability and prevent inappropriate usage).
- Unclear object comparison - in 23 instances in both the Pig and Hive codebase, objects were compared using the `==` operator. However using this operator only checks for reference equality - it does not check for . A better practice would be to use the `equals` method.
- Not making classes with private constructors final. As noted in the official Java documentation, declaring a class as final makes it immutable (meaning that it cannot be subclassed). Since the classes' constructors are private, cannot be subclassed. Consequently declaring the class final would be good practice.

- Method-level synchronization (it is generally considered better practice to use block-level synchronization).
- Re-assignment of parameters (again, this is generally considered bad practice).
- Using interfaces as containers for constants (Hive uses 5 interfaces to store constants; Pig violates this usage pattern only once).
- Default keyword not being at the end of switch statements.
- Abstract classes without abstract methods (by convention, abstract classes should contain abstract methods. The lack of abstract methods suggests incomplete development. The Hive codebase contains 7 such classes; Pig 6).
- A high ratio of labels inside switch statements (i.e. high switch density) indicate poor readability and unnecessarily complex code - a better approach would be to introduce methods and/or subclasses.
- 1019 switch-statements in Hive and 27 in Pig do not include a `default` option to catch unspecified cases. This is generally considered bad practice and may prevent the application from failing gracefully.

Furthermore, in both codebases, there were many instances in which boolean expressions, conditionals and boolean returns could have been simplified. For example the unnecessary comparison on lines 1455 - 1456 in Hive's `SemanticAnalyzer.java` could be removed:

```
if (joinTree.getNoSemiJoin() == false
            && condn.getToken().getType() == HiveParser.DOT) {
```

In addition, both codebases contained many confusing ternaries (that is, negations within if-else-statements). Other, more minor design issues shared by the two codebases included:

- Missing factory patterns - In 514 instances within the Hive codebase and 2 instances in Pig, factory patterns could be introduced to avoid duplication of code and to provide a sufficient level of abstraction.
- Unsafe usages of static fields.

- Instance checks in catch-causes - Instead of using the `instanceof` keyword in catch-clauses, exceptions should be handled by their own clauses. For example, lines 1336 - 1355 in Hive's `HiveMetaStore.java`:

```
try {
    drop_table_core(getMS(), dbname, name, deleteData, envContext);
    success = true;
  } catch (IOException e) {
    ex = e;
    throw new MetaException(e.getMessage());
  } catch (Exception e) {
    ex = e;
    if (e instanceof MetaException) {
      throw (MetaException) e;
    } else if (e instanceof NoSuchObjectException) {
      throw (NoSuchObjectException) e;
    } else {
      MetaException me = new MetaException(e.toString());
      me.initCause(e);
      throw me;
    }
  } finally {
    endFunction("drop_table", success, ex, name);
  }
```

...should be replaced with:

```
try {
    drop_table_core(getMS(), dbname, name, deleteData, envContext);
    success = true;
  } catch (IOException e) {
    ex = e;
    throw new MetaException(e.getMessage());
```

```
} catch (MetaException me) {
        // do stuff
} catch (NoSuchObjectException noe) {
        // do more stuff
}
} finally {
  endFunction("drop_table", success, ex, name);
}
```

- Calling overridable methods within a class' constructor. As noted by the PMD documentation: "Calling overridable methods during construction poses a risk of invoking methods on an incompletely constructed object and can be difficult to debug. It may leave the sub-class unable to construct its superclass or forced to replicate the construction process completely within itself, losing the ability to call super()."[**?**]
- Fields that should be final but arent'.
- Singletons that are not thread-safe (this can be resolved by synchronizing the instantiation method).
- Calls to `toArray` that do not specify array sizes.
- Loosing the stack trace when throwing exceptions.
- Returning `null` values when other values may be more appropriate. For example, returning null when instead an empty array could be returned makes the application susceptible to null pointer exceptions.
- Not specifying a locale when using date objects.
- Unnecessary fields - That is, fields whose scope is limited to one method do not need to be fields - they could just be local variables.

The Hive codebase is also plagued by additional design issues not shared with Pig. In 93 instances[4], streams were not being closed which may result in the loss of buffered data or the failure to release unused resources. In several cases, switch-statements also contained none-case labels which

[4]A majority of these were in test classes. Nevertheless, resources also remained open in core classes, such as `HiveConnection.java`

was confusing (albeit correct). In 808 instances, switch-statements should have been replaced with if-statements (since they contained too few branches). Few instances also used incorrect null comparisons (comparing null by calling `equals()` as opposed to `==`).

By dividing the aforementioned code issues into categories according to the experience level required to notice or avoid them, we get three categories: rookie mistakes, intermediate mistakes and expert mistakes.

Rookie mistakes indicate lack of basic understanding of Java's coding principles, rushed and messy implementations or general lack of effort. Any Java 101 class should provide the programmer with sufficient knowledge to avoid these mistakes. The category contains the following issues: jumbled incrementers, for-loops that should be while loops, overriding both `equal()` and `hashcode()`, returning from a method inside a `finally` block, unconditional if-statements (e.g. if (true) {}), boolean instantiation, collapsible if-statements, misplaced null checks, using hardcoded octal values, explicitly extending `Object`, `BigInteger` instantiation, missing braces, uncommented methods and constructors, empty code, unusued code, bad naming practices and unnecessary imports.

Intermediate mistakes indicate that these mistakes were possibly made by intermediate programmers (i.e. those with a good knowledge of the language but possibly little practical experience). Any knowledgable programmer should be able to spot unnecessary code and general design issues. Intermediate mistakes include: unnecessary code, violating security Code guidelines, n-path complexity of 200 or more, excessive method length, methods with too many parameters, excessively large classes, high cyclomatic complexity, disproportionately many public methods, disproportionately many fields, high NCSS (Non-Commenting Source Statements) method count, high NCSS type count, high NCSS constructor count, classes with too many methods, high degree of coupling, excessive imports, violating the law of Demeter (again, this leads to a high degree of coupling), the aforementioned optimization (instantiating objects inside loops, unnecessary wrapper object creation, inefficient string appends etc) and design (e.g. deeply nested conditionals, unsimplified boolean expressions, missing `break` and `default` statements, high switch density, empty methods etc) issues as well as general bad practices involving the use of `String` and `StringBuffer` objects.

Expert mistakes indicate mistakes made despite possible expert knowledge. These include type resolution issues (e.g. implementing the `clone()` method without having the class implement `Cloneable`, throwing `Exception` as opposed to the precise exception (e.g. `IOException`)), JUnit mistakes (e.g. failing to include asserts in JUnit tests, assertions with missing messages, empty test classes, unnecessary boolean assertions, etc), general controversial code (e.g. unnecessary constructors, `null` assignments to variables outside of their declaration, methods with more than one exit point, missing constructors, imports from the `sun.*` packages, unnecessary parentheses, dataflow anomalies, explicit garbage collection etc), bad Java Bean practices, violation of JDK migration rules, catching `throwable`, using exceptions as flow control, throwing `NullPointerException`, throwing raw exceptions, re-throwing exceptions, extending `java.lang.Error` (because `Error` is meant only for system exceptions), catching generic exceptions, violating rules related to J2EE implementations, controversial error logging and questionable usages of finalizers.

As illustrated in figures 5.7 and 5.8, a larger percentage of Hive's code issues are composed of expert mistakes; Pig on the other hand contains more intermediate mistakes. Although this may make the Hive codebase appear superior at first, the Hive contains many more issues than Pig (as discussed above Hive contains 71.5 issues per 100 lines of code whilst Pig only consists of 24.85 issues per 100 lines of code):

- 12 rookie mistakes per 100 LOC (a total of 23027 rookie mistakes).

- 38 intermediate mistakes per 100 LOC (a total of 71965 intermediate mistakes).

- 18 expert mistakes per 100 LOC (a total of 34292 expert mistakes).

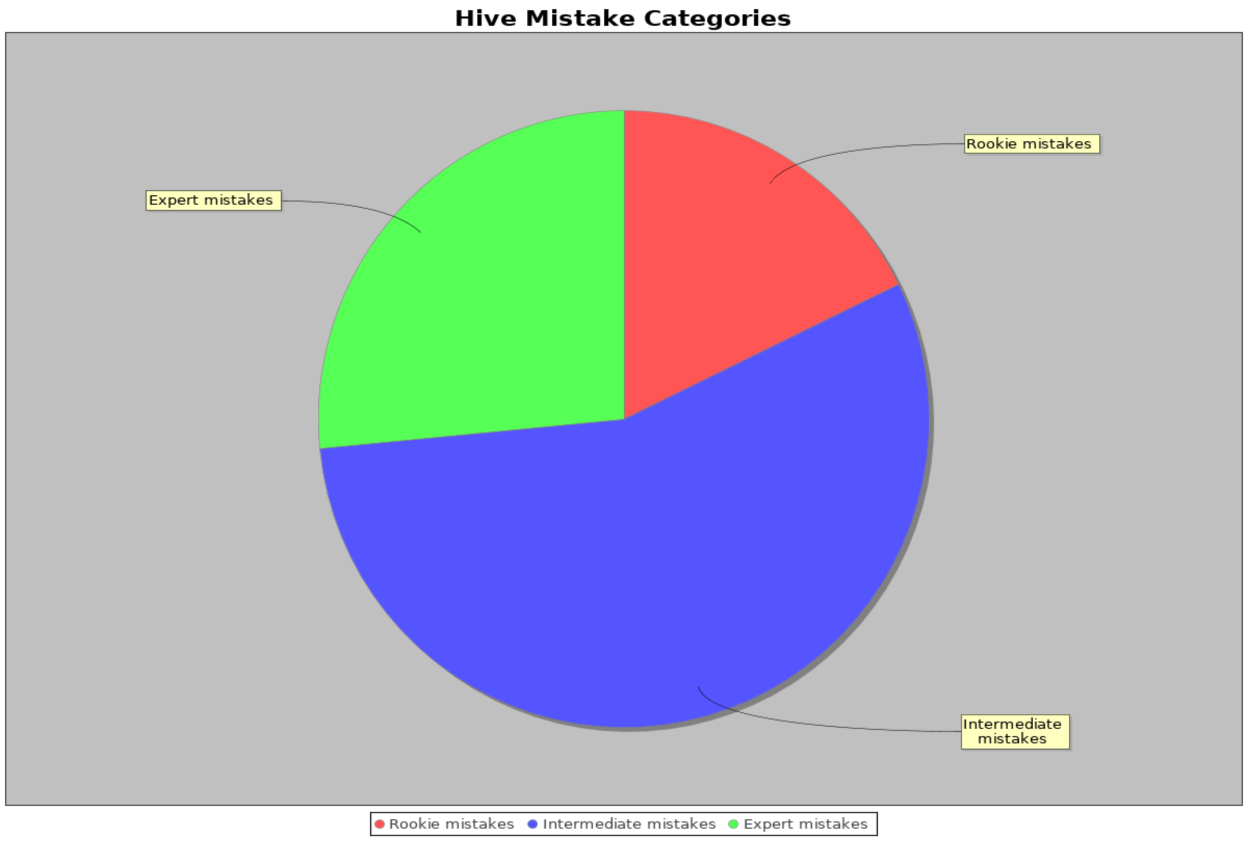


Figure 5.7: Hive codebase mistake categories

Pig on the other hand:

- 5 rookie mistakes per 100 LOC (a total of 7556 rookie mistakes).
- 14 intermediate mistakes per 100 LOC (a total of 21044 intermediate mistakes).
- 5 expert mistakes per 100 LOC (a total of 7987 expert mistakes).

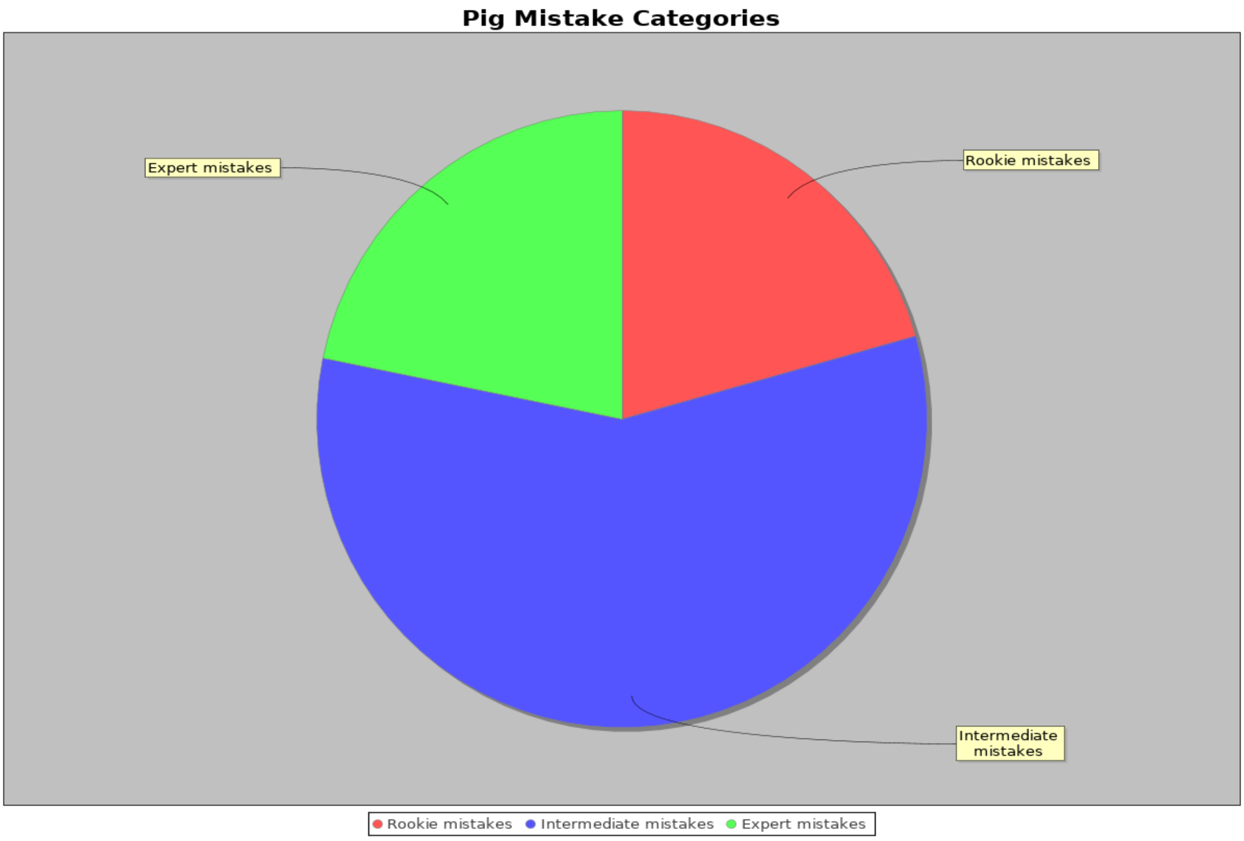


Figure 5.8: Pig codebase mistake categories

5.1.2 Naming conventions

The Pig codebase exposes a wider range of naming convention abuse (albeit less than Hive): a total of 5,100 issues were found. Hive on the other hand contains 16,764 violations of 3 types:

- Abstract class naming - Abstract classes should always be prefixed "Abstract"[25]. 97 classes in the Hive codebase violate this convention.
- Methods with the same field name - In 131 instances, methods had the same name than their corresponding field (for example private int foobar = 0; public void foobar() { ... }).
- Field name matching class name - 6 classes contain fields that had a matching class name.

Pig's violations consist of:

- Abstract class naming (as described above) - 94 violations.
- Methods with the same field name (as described above) - 59 violations.
- Field name matching class name (as described above) - 4 violations.

- BooleanGetMethodName - As stated in the PMD documentation: ”methods that return boolean results should be named as predicate statements to denote this. I.e, ’isReady()’, ’hasValues()’, ’canCommit()’, ’willFail()’, etc. Avoid the use of the ’get’ prefix for these methods.”[?]. There exist 15 instances of this violation.
- Long field and variable names - Names exceeding 17 characters tend to impact readability.
- Not adhereing to Java method naming conventions - As stated in the official Java specification, method names should[26]:
  - Begin in lowercase.
  - Be a verb or a multi-word name that begins with a verb.
  - Words of multi-word names should begin with a capital letter (except for the first word).
- Not adhereing to Java package naming conventions
- Not adhereing to variable naming conventions - As stated in the official Java specification, variable names should[27]:
  - Begin with a letter (not a number or underscore).
  - Not use abbreviations.
  - Be one-word or multi-word. In the case of the latter, the first letter of each subsequent word should be capitalized.
  - Be capitalized if the variable is a constant.
- Short method and variable names - This may indicate that they are not for meaningful and may impact readability.
- ShortVariable
- Suspicious constant field name
- Suspicious equals method name

5.1.3 Codesize, coupling and complexity

The codesize and the code complexity of both codebases could be improved upon (again, Hive could do with many more improvements than Pig). Thomas J. McCabe’s ”cyclomatic complexity” metric is used to measure the independent paths through both programs[28]. Typically, the lower

the cyclomatic complexity the better. It should be stressed that lowering cyclomatic complexity only moves the complexity around to other parts of the program - it does not actually remove complexity. Consequently, lowering cyclomatic complexity results in additional classes, interfaces and methods and hence may mean finding trade-offs between other undesireable design properties such as increasing code size or lowering cohesion. Analysis of the Hive codebase using PMD resulted in the discovery of 1,955 classes and methods which exceeded the cyclomatic complexity threshold (one class that stood out was `CommonJoinOperator.java`: it contained 4 instances of high cyclomatic complexity, one of which was the `genObject` method with a complexity of 27). Overall, the average cyclomatic complexity is very low: 2.3. Methods with the highest cyclomatic complexity are `SemanticAnalyzerFactory::get` (complexity of 69), `DDLSemanticAnalyzer::analyzeInternal` (complexity of 67) `DDLTask::alterTable` (complexity of 67), `SemanticAnalyzer::doPhase1` (complexity of 52)
`SemanticAnalyzer::genFileSinkPlan` (complexity of 44).

Cyclomatic complexity analysis of the Pig codebase identified 606 classes and methods that exceeded the threshold. Interestingly, the average cyclomatic complexity is close to Hive's 2.4 some methods have a staggering complexity: for example `POCast::convertWithSchema` has a complexity of 102. Ranked just below it are `Main::run` (complexity of 64), `JobControlCompiler::getJob` (complexity of 53), `GroovyScriptEngine::registerFunctions` (complexity of 50) and `BinInterSedes::readDatum` (complexity of 49).

The n-path complexity refers to the number of acyclic execution paths through that method. The more paths there are, the more complex the method is. Using a threshold of 200, running the PMD analyzer over the Hive codebase allowed for the identification of 677 methods whose complexity should be reduced and whose readability should be increased. Pig on the other hand contains only 198 methods which exceed this threshold, allowing us to safely conclude that the Pig codebase is generally easier to read and understand than Hive. This assertion is supported the fact that Hive contains a lot more excessively large classes than Pig (61 vs 19). The same goes for excessive method lengths (161 in Hive vs 89 in Pig) and the amount of methods and fields in different classes: in the Pig codebase, only 134 classes were found to contain an excessive amount of methods. In Hive on the other hand it is 586. Numerous classes in Hive were also found to contain too many fields, indicating that the developers did a bad job at grouping related fields.

Unlike the Pig codebase, Hive also suffers from a high degree of coupling: by counting the number of fields, local variables and return types, 22 classes were found to have a high degree of coupling ("high" referring to over 20 references to other classes). This assertion is confirmed by examining imports: 62 classes in the Pig codebase seem to import an excessive amount of classes; contrast this to 120 instances in Hive! Furthermore, the Hive codebase does badly on loose coupling: In 872 cases, implementation types were used over interface types which can make code maintaince / change introduction difficult. In contrast, Pig only used implementation types in 81 instances (although this is still quite a lot).

Hive's codebase also contains methods and constructors with large amounts of parameters, making their usage rather complex and difficult. Take Hive's `CreateIndexDesc` constructor for example:

```
public CreateIndexDesc(String tableName, String indexName,
    List<String> indexedCols, String indexTableName, boolean deferredRebuild,
    String inputFormat, String outputFormat, String storageHandler,
    String typeName, String location, Map<String, String> idxProps, Map<String,
        String> tblProps,
    String serde, Map<String, String> serdeProps, String collItemDelim,
    String fieldDelim, String fieldEscape, String lineDelim,
    String mapKeyDelim, String indexComment)
```

5.1.4 Controversial

Controversial issues are bad practices that are not derived from the official Java specification. Instead they are subjective interpretations of what "good code" should look like. The following controversial practices were found in both the Pig and Hive codebase:

1. Making assignments inside operands - For example: if ((time = getTime()) == 8000). Making such assignments impacts readability.

2. Missing constructors - Every class should contain at least one constructor. **3. Final local variables** - Where possible, it may make sense to turn final local variables into fields. **4. Data flow anomalies** - As stated in the PMD documentation[**?**]

> The dataflow analysis tracks local definitions, undefinitions and references to variables on different paths on the data flow. From those informations there can be found various problems. 1. UR - Anomaly: There is a reference to a variable that was not defined before. This is a bug and leads to an error. 2. DU - Anomaly: A recently defined variable is undefined. These anomalies may appear in normal source text. 3. DD - Anomaly: A recently defined variable is redefined. This is ominous but don't have to be a bug.

5. **Failing to call super() inside constructor**
6. **NullAssignment** - As stated in the PMD documentation[**?**]:

> Assigning a "null" to a variable (outside of its declaration) is usually bad form. Sometimes, this type of assignment is an indication that the programmer doesn't completely understand what is going on in the code. NOTE: This sort of assignment may used in some cases to dereference objects and encourage garbage collection.

For a full table of issues see Appendix C.

5.2 Concrete example - JOIN

As described in [13], Hive's join operations are significantly slower than in Pig. These results were confirmed by running additional benchmarks, including two transitive self-join experiments on datasets consisting of 1,000 and 10,000,000 records (the scripts and dataset generator used for this benchmark were provided by Yu Liu). The former took Pig an average of 45.36 seconds (real time runtime) to execute; it took Hive 56.73 seconds. The latter took Pig 157.97 and Hive 180.19 seconds (again, on average). The fact that Hive's join implementation is less efficient than Pig's can be explained by examining Hive's join operator implementation `.org.apache.hadoop.hive.ql.exec.CommonJoinOp` in essence, the methods responsible for the join recursively produce arrays of bitvectors. Each entry in the bitvector denotes whether an element is to be used in the join. (if the element is null, then it won't be used). Not only does recursion result in a large memory footprint, but it also means that an unnecessary amount of objects are created for each join operation (object creation in Java is expensive)[5], making the entire join operations a lot less efficient than Pig's.

[5] Smaller optimization issues, object instantiation inside loops, contribute to this performance difference. An example of this can be found inside the `checkAndGenObject()` method on line 666.

5.3 Evolution over time

As is evident from figures 5.9 and 5.10, both codebases evolved quite differently. Initially the Hive codebase was much smaller than Pig's, and so were the number of optimization issues per 100 LOC. Hive version 0.10.0 saw a sudden explosion in size as many new features were introduced (such as fixes to the ZooKeeper driver (HIVE-3723), ability to group sets (HIVE-3471) or the addition of the "explain dependency" command (HIVE-3610)). The focus with Pig on the other hand seems to have been efficiency: the number of optimization issues in version 0.10.0+ dropped and then stabilized. This supports the argument presented by the author in [13] that fixes to the Pig codebase in 2011 accounted for the results by some studies that showed Hive to initially outperform Pig.

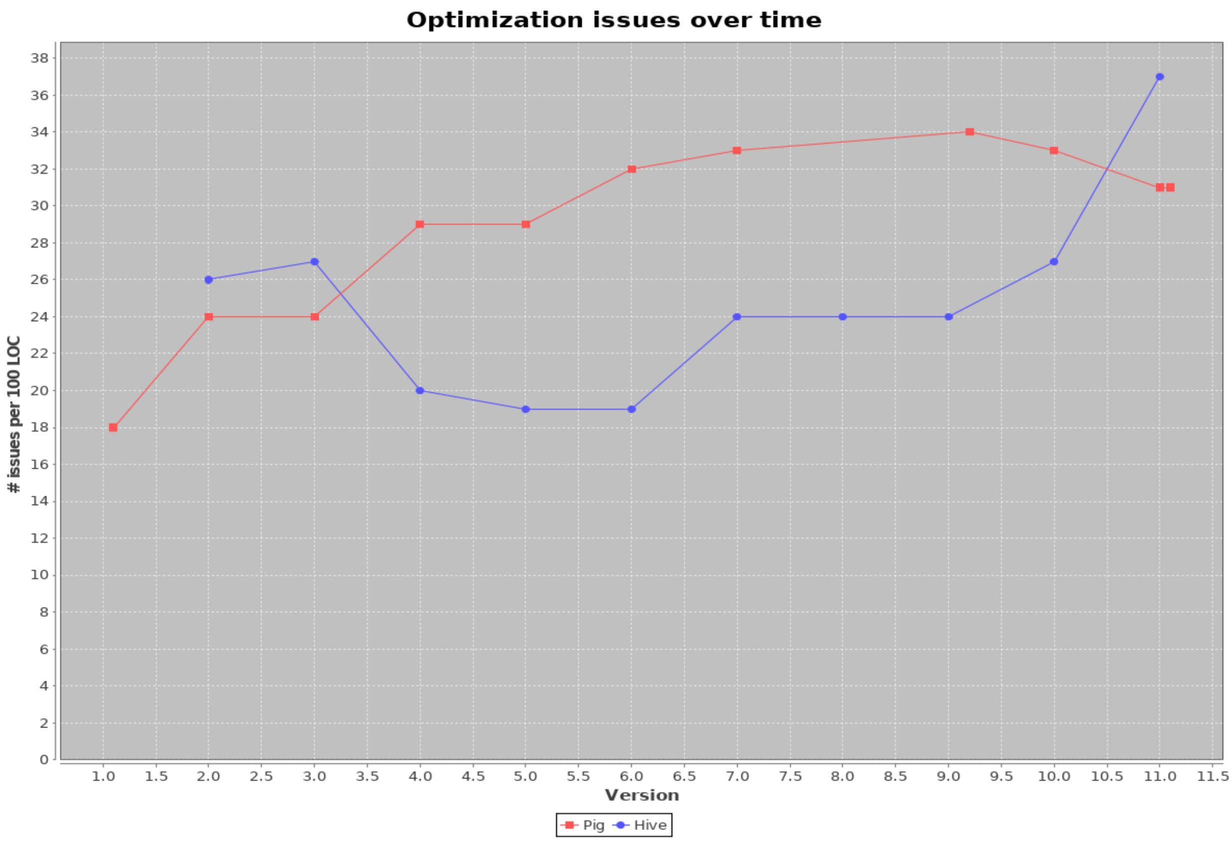


Figure 5.9: The number of optimization issues in the Pig and Hive codebases over time. Note that the x-axis should be read as version numbers. For example, 1.0 refers to version 0.1.0, 11.1 refers to version 0.11.1

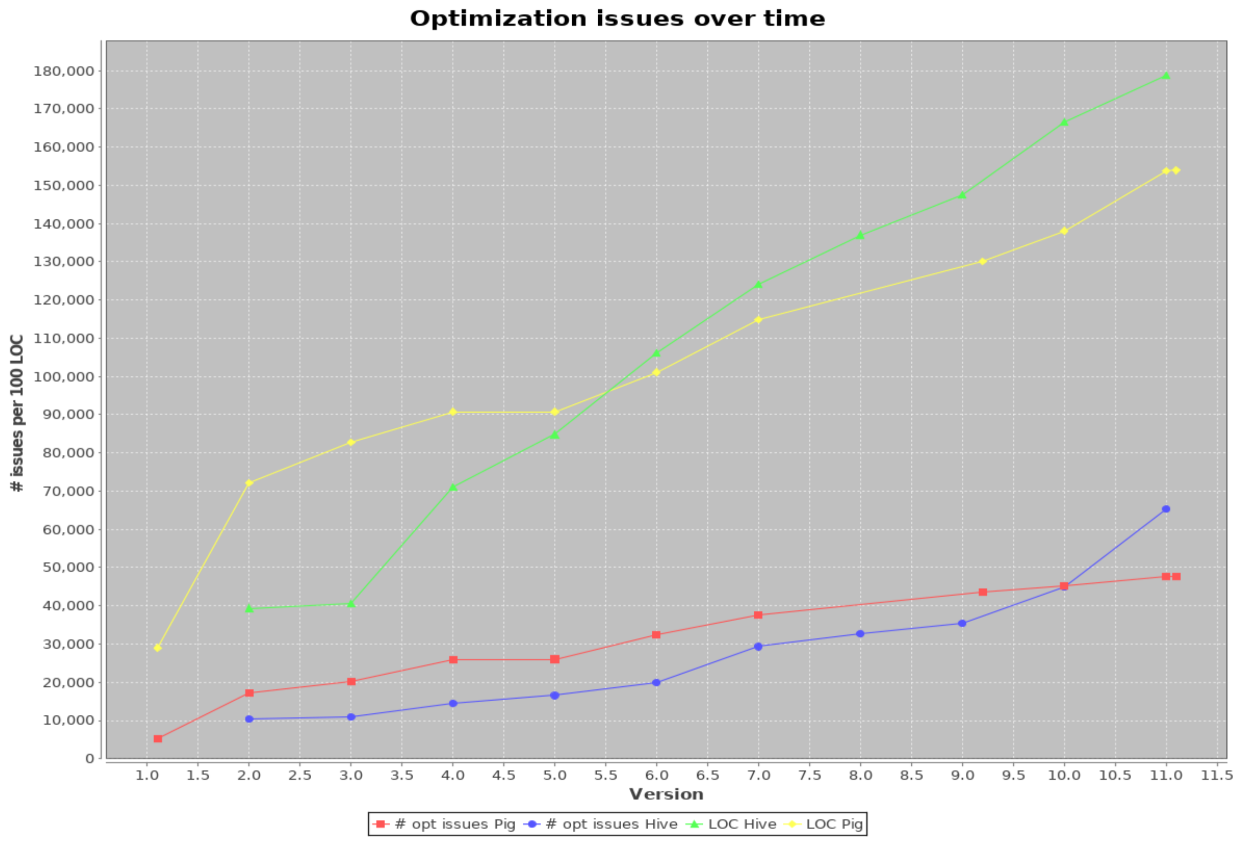


Figure 5.10: The number of optimization issues as well as the number of lines of code (LOC) in the Pig and Hive codebases over time. Note that the x-axis should be read as version numbers. For example, 1.0 refers to version 0.1.0, 11.1 refers to version 0.11.1

5.4 The Group By operator

As was discussed in chapter 4, runtime numbers show that Pig's `Group By` operator is outperformed by Hive. CPU times show that the mappers for this operator are very slow[13].

Reviewing the original query we see an algebraic UDF function: `COUNT`:

```
a = LOAD '$input/dataset' using PigStorage('\t') AS (name, age, gpa)
PARALLEL $reducers;
b = GROUP a BY name PARALLEL $reducers;
c = FOREACH b GENERATE flatten(group), COUNT(A.age) PARALLEL $reducers;
STORE c INTO '$output/dataset_group' using PigStorage() PARALLEL $reducers;
```

Pig is a procedural language, and being younger and less popular within the database community, is less optimized than SQL-like languages. As such, Pig creates aliases (intermediary datastructures) with each step of the query. That is, every time a mapper finishes, it writes this data to disk. As noted by core developer Cheolsoo Park, the way Hadoop works is that when data comes out of a map task, it is being serialized so that the size of the output buffer can be determined quickly (Java lacks a sizeof operator). Next, a special Hadoop process, called a "combiner" reads these datastructures (byte streams) back into memory and deserializes them. Consequently, for each "stage" in the query, one requires disk I/O operations as well as serialization/deserialization; which is expensive and not worth it unless the data reduction is significant.

Looking at the script used to generate the test dataset (https://issues.apache.org/jira/browse/PIG-200), we see that a lot of random keys are generated. Consequently, one will end up with a large number of small bags rather than a small number of large bags. If that's the case, the combiner will only add overhead to mappers and data reduction per bag will be insignificant and will be outweighed by the cost of serialization and disk I/O. Therefore disabling the combiner (using `set pig.exec.nocombiner true;`) produces signficant performance advantages (although the algorithmic problem of having too many stages will persist. For example, Pig creates 99 map jobs for the given script and dataset; Hive only 8). The number of bags can be computed via:

$$\frac{total\ number\ of\ input\ records}{(reduce\ input\ groups\ .\ number\ of\ reducers)}$$

Additionally, Pig 0.10+ allows for in-memory aggregation[6]. If enabled, Pig will buffer map outputs in memory and apply combiners without having to serialize/deserialize and perform disk read-/writes.

It should also be noted that `DISTINCT` should be used in place of `GROUP BY` whenever possible as the former is more efficient (the output of a `DISTINCT` contains only the column(s) to which the operation was applied, whilst the output of the `GROUP BY` relation is a key and bag which contains all of the tuples that have the same group key).

[6]In-memory aggregation can be enabled/disabled by setting the `pig.exec.mapPartAgg` flag.

5.5 Hive patch implementation

Two patches were developed and submitted: HIVE-5018 (implementing recommendation 1 from section 5.1.1 and resulting in a 2.6% performance increase when it comes to arithmetic operations) and HIVE-5019 (implementing recommendation 6 in section 5.1.1).

5.6 Conclusion

In section 3.1 the question **"what initially caused Hive to outperform Pig[17][15][14]"** was posed. The data presented in this and the previous chapter answers this question by illustrating that the results presented by [15] are questionable: Hive most likely did not outperform Pig - the TPC-H benchmarks are flawed as one operator dominates the outcome of the entire script. Consequently they are not a realistic assessment as in reality not every query would be relying on a dominant set of operators. Furthermore, as noted in section **??**, the differences presented by [17] and [14] can be explained by initial problems with the Pig compiler as well as issues with the compiler's logical plan[13]. Recall that upon examining Apache's Pig repository, two releases stood out:

29 July, 2011: release 0.9.0
This release introduces control structures, changes query parser, and performs semantic cleanup.

24 April, 2011: release 0.8.1
This is a maintenance release of Pig 0.8, contains several critical bug fixes.

Closer inspection found that the following tickets PIG-1775 (removal of old logical plan), PIG-1787 (error in logical plan generated), PIG-1618 (switch to new parser generator technology), PIG-1868 (new logical plan fails when I have complex data types), PIG-2159 (new logical plan uses incorrect class for SUM causing) appeared to account for the aforementioned problems.

Analysis of the Pig and Hive codebases revealed that overall, Pig's source code is of higher quality than that of Hive:

- Pig's codebase is nearly 18% smaller than Hive: Pig consists of a total code (and comment) base of 154,731 LOC (25% comments, 13% blank lines and the rest consists of actual code). Hive, although more sparsely documented, consists of 187,916 LOC (23% comments, 12% blank lines and the rest consists of actual code).

- The Hive codebase contains 71.5 issues per 100 lines of code[7]; Pig contains 24.85 issues per 100 lines of code.
- In terms of issues per lines of code, Pig is vastly superior to Hive: Hive has 31.59 optimization issues per 100 lines of code (LOC); Pig only 11.41.
- Both codebases have roughly the same percentage of design flaws: 5.05% for Hive and 4.9% for Pig.
- The Pig codebase exposes a wider range of naming convention abuse (albeit less than Hive): a total of 5,100 issues were found.
- The Pig codebase seems more professional: 5 rookie mistakes per 100 LOC (as opposed to 12 rookie mistakes per 100 LOC for Hive).
- 14 intermediate mistakes per 100 LOC (as opposed to 38 intermediate mistakes per 100 LOC for Hive).
- 5 expert mistakes per 100 LOC (18 expert mistakes per 100 LOC for Hive).

In terms of cyclomatic complexity however, both codebases are the same. Although Pig has a much lower n-path complexity than Hive (supporting the argument that Pig's codebase is much easier to understand and maintain).

Any performance differences between Pig and Hive should be attributed to code quality: on a logical level, translation of scripts into map-reduce jobs are the same.

As a concluding remark, it should be noted that both codebases are still far from mature and both could be greatly improved upon. They are not nearly as mature as Apache Ant for example (which contains 16.84 issues per 100 LOC[8]).

[7]"Issues" refer to either basic violations of good practices, problems related to code size or complexity, bad commenting / bad code documentation, code that is deemed controversial, high or inappropriate coupling between classes, general bad design practices, empty or redundant code, violations related to naming of variables, classes and methods, various optimization issues, bad exception handling, unused code, security vulnerabilities, problems related to type resolution as well as suboptimal usage of strings and string buffers.

[8]The analysis of Apache Ant v.1.9.2 (consisting of 259,711 LOC) resulted in 43,756 issues

Chapter 6

Developing an IDE

IDEs are development tools designed to aid software development and as such are an essential aspect of maximizing programmer productivity and development speed. IDEs tend to have similar user interfaces and provide a wide array of features for debugging, compilation, configuration, authoring and software deployment.

As noted in section 2.2.1, no cohesive development tool for writing big data scripts in Pig Latin or Hive QL exists to date. Having tried a variety of different tools throughout the ISO, local development with each tool still proved tedious. For one, no existing IDE allowed for easy deployment of scripts: one had to always either transfer them using SSH commands, SFTP transfer clients such as FileZilla or write deployment scripts. Furthermore, browsing the Hadoop filesystem and transferring files to and from is time consuming when using only a terminal: for example if one works from at home and requires access to the Hadoop cluster at Imperial College London, one first needs to SSH into shell1.doc.ic.ac.uk and from there onto ebony.doc.ic.ac.uk. On ebony one can then use the hadoop fs command line utility to explore the file system (there is no reason why an editor should not be able to do this automatically).

Furthermore, version control of the scripts was inconvenient: the Git plugin for Eclipse is faulty and difficult to setup (and under less popular Linux distributions, such as Sabayon, seemingly impossible to configure) and stand-alone Pig/Hive development tools included no form of version control at all.

Quite some time was also lost when running batches of benchmarks, only to discover that the path to some data files in the Hadoop filesystem was incorrect (this happened quite a lot, especially when administrators reconfigured various nodes within the cluster). No existing big data editors contain features that automatically check the Hadoop filesystem for the existence of datafiles upon encountering a LOAD statement in the code.

As already discussed, specifying configuration settings, such as the number of map and reduce jobs is rather awkward and to date no editor contains features that make this easier.

Scripts processing big data often take many hours (if not days) to execute. The obvious, and rather tedious way of determining when a script has finished is to regularly check the job tracker. Alternatively, one could write his/her own script that periodically check the tracker and notify the user when a task (or set of tasks) has successfully completed execution. Both methods seem archaic - why not include a notification engine with the IDE? Such a notification engine could easily be coupled with a runtime manager which would submit new tasks to the cluster as others complete their execution. Furthermore, a result analyzer could automatically analyse task runtimes and compute elementary statistics such as mean, median and mode of real time runtimes, total CPU runtime, CPU map time and CPU reduce time as well as their variance and standard deviation and plot these results as line, bar and pie charts.

Of course an IDE's defining feature is the script editor which allows for the authoring, display and editing of Hive and Pig scripts and possibly containing undo / redo functions, as well as a syntax checker and code completion features. The next chapter will discuss all these features in more detail.

Last but not least the question arises as to whether modify an existing IDE (for example Netbeans or Eclipse) or to write one from scratch. The latter option was chosen for the following reasons:

- Existing IDEs such as Netbeans (204MB) and Eclipse (244MB) are bloated and contain many features not needed when developing Pig/Hive scripts. Consequently, why require the user to install either one of these if most of their features are not needed?
- A stand-alone IDE is faster. Due to their size and complexity, Netbeans and Eclipse take some time to load and can be sluggish and difficult to use at times.

- Bigger challenge. On a personal note, I found it fulfilling to be able to claim that "I wrote my own IDE".

- Ease of use - the user interfae Eclipse and Netbeans IDE can be quite overwhelming at first and take some time to get used to. Keeping the UI minimal and only including the necessary controls makes the software much easier to use.

However by design, AutoPig's code is modular and any component can be turned into a Netbeans Platform module. As proof of this, the HDFS file manager was turned into a Netbeans plugin (see chapter 8.4).

Chapter 7

Architecture

7.1 Benchmarking Application Design

The previous sections reviewed the existing state of big data development tools introduced the main problems and constraints faced by the application's development. This section aims to describe the architecture that makes the implementation of these solutions possible. Only core components and core design features will be discussed.

AutoPig consists of 7 distinct components (as illustrated below).

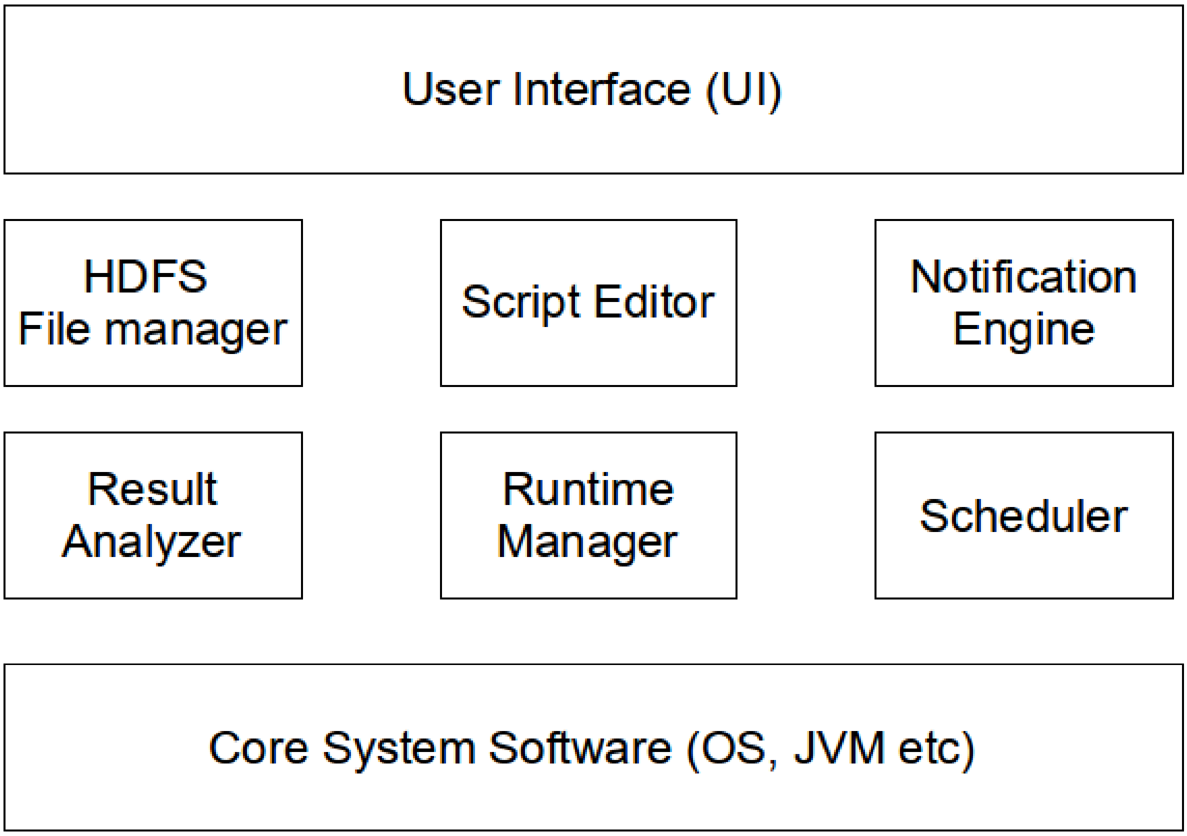


Figure 7.1: *AutoPig* component structure.

7.2 HDFS file manager

The HDFS file manager is, as its name implies, a module for managing files within the (remote) HDFS filesystem. That is, it is a front-end for the `hadoop fs` command set and supports:

- Copying files to and from the Hadoop filesystem and an external file system. Equivalent to `hadoop fs copyFromLocal` and `hadoop fs -copyToLocal`.
- Transferring files across a network using `scp` as well as automatic deployment and execution of scripts.
- Change group association of files. Equivalent to `hadoop fs -chgrp`.
- Change the permissions of files. Equivalent to `hadoop fs -chmod`
- Change the owner of files. Equivalent to `hadoop fs -chown`
- Copy files from source to destination. Equivalent to `hadoop fs -cp`
- Displays aggregate length of files. Equivalent to `hadoop fs -du`.

- Empty the trash. Equivalent to `hadoop fs -expunge`.
- Display stats on a file. Equivalent to `hadoop fs -ls`.
- Takes path uri's as argument and creates directories. Equivalent to `hadoop fs -mkdir`
- Moves files from source to destination. Equivalent to `hadoop fs -mv`.
- Delete files. Equivalent to `hadoop fs -rm`.
- Recursively delete files. Equivalent to `hadoop fs -rmr`.
- Outputs the file in text format. Equivalent to `hadoop fs -text`.
- Test dataset generation.

7.3 Unix file manager

Just as the HDFS file manager above, the Unix file manager is a module for managing files within standard Unix / POSIX compliant remote hosts. That is, it is a front-end for executing filesystem-related shell commands and supports:

- Copying files to and from the remote host using `scp`.
- Automatic deployment and execution of scripts.
- Change the permissions and ownership of files using `chmod` and `chown`.
- Copying and moving files from source to destination using `cp` and `mv`.
- Display disk usage information using `du -h`.
- Listing files and changing directories using `ls` and `cd`.
- Creating directories and deleting directories using `mkdir` and `rm -r`.
- File deletion using `rm`.

7.4 Script editor

The script editor will allow for the creation, display and editing of Hive and Pig scripts. Specifically, the editor supports:

- Creation of new files. Saving of files.
- Editing of existing files.
- Pig and Hive QL syntax highlighting.
- Easy viewing of datasets referenced in the script.
- Path checking. That is, if the user accesses a file (or table, as is the case in Hive QL), the script editor will interface with the Hadoop file manager and check whether the actual data file (and table) exist within the Hadoop file system. If not, a warning will be displayed.
- Undo / redo functions.
- Possibly auto-complete and syntax validity checker.
- Inbuilt Git version control.

7.5 Notification engine

The notification engine interfaces with the Hadoop job tracker and notifies the user (by email or by displaying a message on screen) when a scheduled job completes execution. It also reacts to error messages and advises the user on a possible course of action.

7.6 Result analyzer

This module allows for the fast analysis of benchmarks. Specifically it allows the user to:

- Calculate mean, median and mode of real time runtimes, total CPU runtime, CPU map time and CPU reduce time as well as their variance and standard deviation.
- Plotting of the given results as line, bar and pie charts.
- Export of the given results into CVS and PDF formats.

7.7 Runtime-manager

Exact features of this module are still to be determined and will become clearer as development progresses. However in essence it should monitor the execution of scripts and interface with the scheduler and notification engine. It should counter-act errors where possible. For example if an out-of-disk space error seems imminent, it should try and remove temporary files or move data files that are currently not needed to another server until the script terminates.

Furthermore, the IDE should support local execution of scripts as well as a debugger and dataset viewer.

7.7.1 Scheduler

Allows for simple scheduling of scripts.

7.8 User Interface

The User Interface Component's sole responsibility is the presentation and visualization of data. These can be summarized as:

1. Graphical file system representation and a graphical method to interface with the Hadoop file manager.
2. Visualization of the script editor and its accompanying features.
3. The construction and visualization of various user interface controls such as windows, buttons and text fields.
4. The conversion of UI events into interactions between the remaining system components.

To serve this purpose, the UI component is divided up into three sub-components, one for each of the other 6 system components.

1. The `file_manager` package handles visualization of the hadoop file system.
2. The `editor` package handles the visualization of the script editor.
3. The `notification` package provides auxiliary services to the `notification engine` component for the visualization messages and customization of emails.
4. The `analysis` package handles visualization of the `result analyzer`.

5. The `scheduler` package handles visualization of the `scheduler`.
6. The `runtime` package handles visualization of the `runtime manager`.

7.9 Package Structure

The *Benchmark Application* project tree is organized into a set of 10 packages, each of which can contain sub-packages. Navigating from the source node downwards, the project tree's package structure is as follows:

- `hadoopdevtool` - Contains the application's `Main` class as well as the `Conf` class which holds runtime configuration variables for the entire application.
- `assets` - Contains application assets such as icons and configuration files.
- `exception` - Contains application specific exceptions (see section 8.11).
- `remote` - Composes the HDFS and Unix file manager (see section 7.2).
- `ui` - Composes the UI Component (see section **??**
- `editor` - Composes the script editor (see section 7.4).
- `notification` - Composes the notification engine (see section 7.5).
- `analysis` - Composes the result analyser (see section 7.6).
- `runtime` - Composes the runtime manager (see section 7.7).
- `scheduler` - Composes the scheduler (see section 7.7.1).

7.10 Architectural Strategies

7.10.1 Policies and Tactics

Design policies and tactics that do not have sweeping architectural implications, but which nonetheless affect the system's implementation are as follows:

1. All development constraints outlined in section **??** are met.

2. `JAVAC` is the Java bytecode compiler of choice. The latest stable release (JAVAC 1.6.014, May 28, 2009) is used for all development and deployment. The compiler is licensed under a GNU General Public License.

3. Where possible, design patterns are used (see section 7.10.2 for details).

4. Traceability matrices are used to ensure that development meets the specified requirements / problem solutions.

5. Deliverables are built using the default Netbeans 7.3 and Ant Scripts.

7.10.2 Design Patterns

Design patterns are applied wherever possible for the following reasons[11]:

1. **Decoupling** - The purpose of decoupling is to divide the system into components in such a way that individual parts can be built, changed, replaced, and reused independently (i.e. the aim is to minimize the cost of change).

2. **Integration** - This is closely related to decoupling in that integration patterns ensure that independently developed components work together. Most patterns promoting integration also promote decoupling.

3. **Control** - The purpose of control patterns lies with managing object access and execution control flow.

4. **Convenience** - Miscellaneous patterns whose sole objective is to simplify code.

The most prevalent patterns within the application are:

1. **Abstract Data Type (Class)**

   Category: Decoupling.

   Purpose: To hide algorithm implementations behind a change-insensitive interface.

   Implementation: Use of Java's `interface` or `abstract` functionality.

2. **Manager (Collection)**

   Category: Decoupling.

Purpose: To aggregate collection-related methods (such as creation/deletion, registration e.t.c.) into a single class and controls all collection access.

Implementation: Abstracting data access into one class.

3. **Module**

Category: Decoupling.

Purpose: The grouping of components that work towards a common goal into a single class, effectively hiding their workings behind a change-insensitive, public interface.

Implementation: Aggregation of classes so that they can be hidden behind a public interface.

4. **Singleton**

Category: Control.

Purpose: Limiting the maximum number of instances of a class. Also used to ensure that data is shared between consumers.

Implementation: Declaring a class' constructor to be private and creating a public accessor that creates and returns instances of the class based on a counter.

5. **Convenience Patterns**

Category: Convenience.

Purpose: The simplification of method invocations.

Implementation: Defining specialized methods that call the general methods, supplying frequently used parameter combinations.

7.11 User Interface Design

7.11.1 Main Screen

The main screen (as seen in figure **??** and implemented by the `MainUI` class) is the central hub for the entire application.

It is from this screen where the user can access all of the application's different features. Essentially these features are made accessible by four main components:

1. A menu bar that contains a set of menu items that group features according to their overall functionality / purpose.

2. A tabbed pane that contains panels for holding the script editor window, the log viewer, project file manager and the HDFS file manager. By selecting individual tabs, the user can switch between the editor and other components.

3. A side panel displayed at the left hand side of the screen presenting project structures.

4. A `JFrame` container in which the above two components are held.

Netbeans' "Matisse GUI Builder" is used to design and construct the user interface.

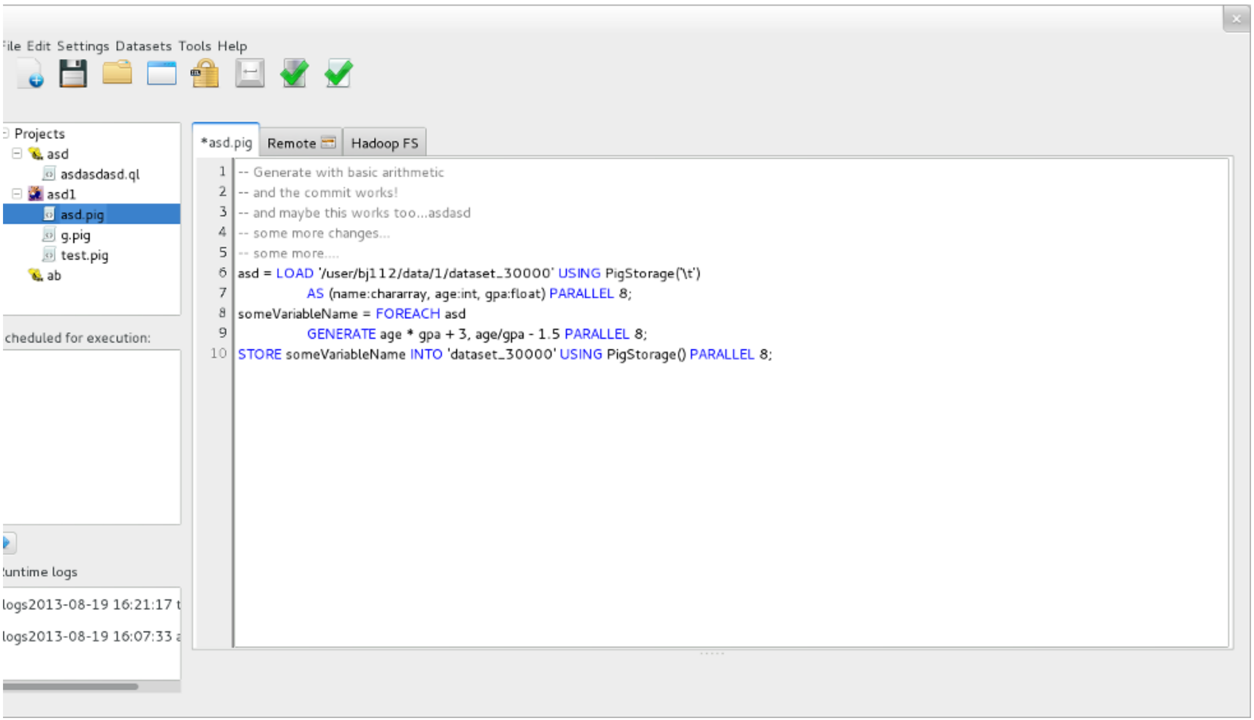


Figure 7.2: *AutoPig's* user interface.

7.12 Summary

This chapter explained the application's architecture, its internal component and package structure, the visual design decisions. To summarize:

- The application consists of 7 distinct components, each dedicated to a distinct set of tasks.
- Error detection, error recovery and concurrency have all been put under careful consideration.

- Various design patterns have been employed to address the issues of decoupling, integration, control and convenience.
- The user interface has been implemented according to best industry practices.

The next chapter will focus on how the core problems faced by the system are implemented following the design and architecture described in this chapter.

Chapter 8

Implementation

This section focuses on how *AutoPig* came about. The tools, methodology, algorithms and their implementation are discussed in detail in the hope of giving the reader a complete understanding of the application's inner workings.

8.1 Language Choice

Java is the development technology of choice based on three factors:

1. **Support on platforms/devices** - Given that the application serves as a complete, functional IDE, specific device requirements are unknown. Given that the Java VM is supported on all major platforms, it proved to be the best choice when considering availability and deployment effort.

2. **Productivity** - Due to the project's tight deadline, a relatively high-level language that would offer good support for visualization and networking operations was required. Again, Java proved to be the best choice given that:

   - The author (Benjamin Jakobus) was already familiar with the language (and hence would not be required to spend time learning a new technology).
   - The Java platform comes with a rich set of APIs and is maintained by a large community of developers.
   - The language has stood the test of time - Java has matured nicely and has been tested and deployed in many industries.

- Java has a large user base and therefore many third party APIs and development tools are readily available.

3. **Performance** - The resulting application should be of reasonable computational performance. Although faster solutions (such as C) exist, Java's performance is acceptable.

8.2 Tools and Technologies

As noted above, the system is developed in Java (with Netbeans 7.3 being the IDE of choice), relying Swing/AWT for user interface and 2D visualization.

Git is used as a version control system, with local version control being handled by the Netbeans IDE. A remote Git repository was created and is hosted by Bitbucket.com. The repository is private, so only a dedicate set of users may access the codebase.

8.3 The Script Editor

As already stated in section 7.4, the logic for the script editor is contained in the `editor` package. This package contains 23 classes and 2 interfaces, the most notable of which the `Workspace` class. The workspace is responsible for initializing and managing the development environment and it interfaces directly with the user interface, receiving and responding to action events and controlling the project workspace. It is one of the "controllers" that form the application's MVC model; user interface components that form the script editor (e.g. editor buttons such as "Save Script") delegate their work to this class.

To initialize the user interface, a `Workspace` object must first be created and then passed to the user interface object:

```
// Create a new workspace
Workspace ws = new Workspace();

// Create application user interface
```

```
IMainUI mainUI = new MainUI(ws);
mainUI.setVisible(true);
```

8.3.1 Syntax highlighting

Syntax highlighting is accomplished using only the 3 classes contained inside `editor.syntax`. The syntax checker for each project type (currently the editor supports only Hive and Pig projects) is represented by a separate class (`PigSyntaxChecker.java` and `HiveSyntaxChecker.java`), all of which inherit the syntax checker's core from `SyntaxChecker.java`. Stating that the individual syntax checkers for the different projects inhert "the syntax checker's core" means that they the logic for identifying and highlighting keywords is contained inside this superclass. The subclasses only define the keywords that are unique to the language which they represent.

The two key methods within this context are `highlightSyntax()` and `highlightCurrentString()`. The former takes the entire script and walks through it line by line, applying the appropriate syntax highlights as necessary. That is, for each word on every line, it checks the `reservedKeywords` array whether this word is contained in the array. If it is, then the word's font colour is changed (to blue). If it isn't, it is ignored.

`highlightCurrentString` on the other hand only checks whether the word that the user has last typed (i.e. the "current string that the user is working on") is a reserved keyword; if it is, highlighting is applied in the same manner as above.

The `commentIdentifier` is, as its name implies, the string that identifies comments and is set accordingly by the syntax checker's subclasses. For example, the comment identifier for a Pig Latin syntax checker is "–".

8.3.2 Search and replace

The script editor supports three search modalities: *find all*, *find next* and *find previous*. Search can be either case sensitive or case insensitive and the user is also presented with the option of replacing all occurrences of a given string with a new, user specified, string.

When searching for a string, matches are highlighted in yellow. The actual search capabilities are implemented by the `SearchEngine` class contained inside the `editor` package. The class exposes six public methods (all methods are static):

- `clearHighlights` - Removes all highlighters added by the search engine. As a match is found, the search engine adds a highlighter to the `JTextPane` on which the match was found, using the offset of the match (i.e. it highlights the match). This method removes all these added highlighters (it is used when the user cancels a search, or when the user starts a new search. Alternatively, it is also used by `findNext` and `findPrevious` to clear the highlights of the previous matches).
- `findAll` - Finds and highlights all occurrences of a given word.
- `findNext` - Finds and highlights the occurrence of the next match.
- `findPrevious` - Finds and highlights the occurrence of the previous match.
- `replaceAll` - Finds and replace all occurrences of a given keyword.
- `reset` - Resets the search engine's private members. These members keep track of the previous match offsets (so as to allow for the *find next* and *find previous* search capability) and of the added highlighters (so that we can efficiently remove them after).

At the core of the search engine lies the following snippet (note: it varies slightly depending on the search modality used):

```
while ((lastIndex = content.indexOf(keyword, lastIndex)) != -1) {
    int endIndex = lastIndex + wordLength;
    try {
        highlighter.addHighlight(lastIndex, endIndex, painter);

    highlighters.add(painter);

    // Increment the number of matches
```

```
        matchCount++;
        } catch (BadLocationException e) {
                e.printStackTrace();
        }
        if (firstOffset == −1) {
                firstOffset = lastIndex;
        }
        lastIndex = endIndex;
}
```

The above code is self-explanatory: we search the text pane's contents for the occurrence of the given search term, starting off at the offset of the last found term, and incrementing this offset with each new match. The loop terminates once no more matches have been found (i.e. as soon as the `indexOf` function returns -1).

8.3.3 Refactoring

At the time of writing, the refactoring capabilities are limited to renaming variables within a script. Selecting a string within your script and then using the key combination `CTRL + R` allows the user to rename all occurrences of the string within the script at once (see figures 8.1 and **??**) whilst seeing all highlighted occurrences of the given string. This feature is identical to those of major IDE's such as Netbeans whose "Refactor - Rename" capability results in Netbeans updating the source code within a project to reference the element by its new name.

To exit the renaming mode, the user has to simply press enter. To this end, the `JTextPane` used as the script editor (`txtScript`) no longer uses Java's default `DefaultStyledDocument` class; instead it uses a subclass, `StyledScriptDocument`, that catches return key events if the user is performing a refactor rename.

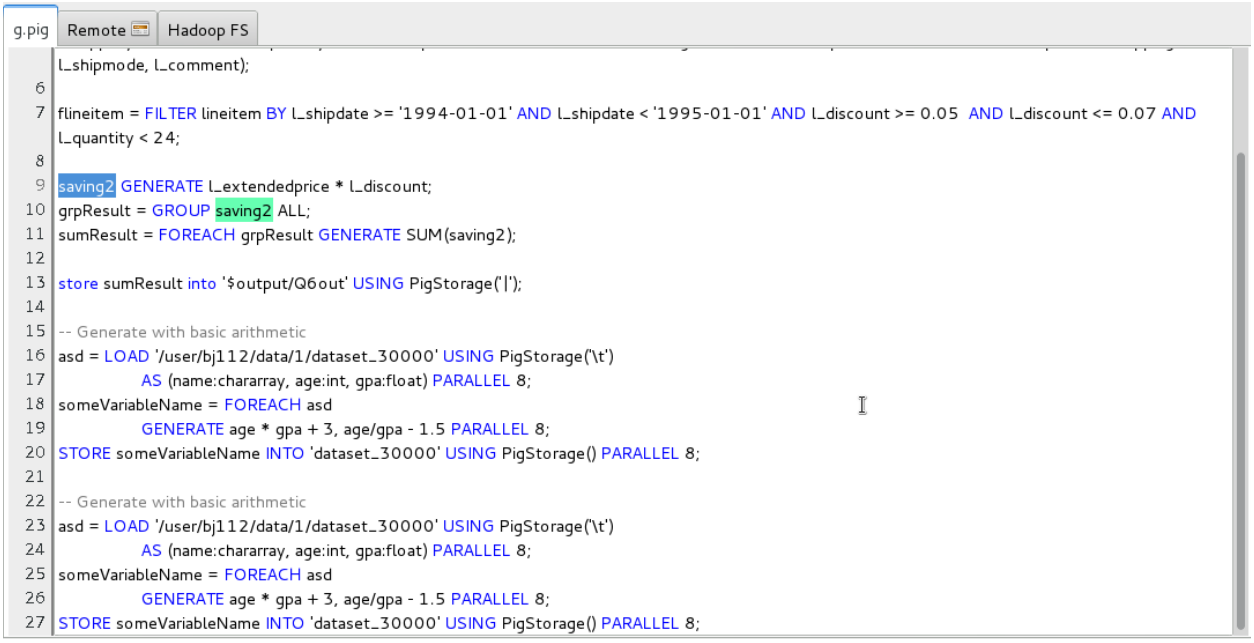


Figure 8.1: Renaming a variable.

The overall logic behind this is contained inside the `MainUI` class. `SearchEngine` is used to find an highlight all occurrences of the selected string.

8.3.4 Workspace management

Workspace management concerns the management of projects and their associated script files. A (script) project consists of a directory (called the project directory) inside the workspace directory. The project directory carries the same name than the actual project and contains a collection of script files as well as one project identifier file. The project identifier file is an empty file called `project.hbt.ql` (the project identifier files for Pig Latin projects end with ".pig") and are used by the application to identify projects (i.e. consider that a workspace may be a user's home directory and hence might contain other, non-project related, directories. The project identifier hence allows the application to determine which directory is a project directory and which isn't).

Workspace management revolves around 4 central classes: `editor.Workspace.java` handles project creation, script creation, relevant UI updates as well as data persistence. It receives messages solemnly from `MainUI.java` and updates its `JTree` component which represents the workspace tree (see figure ??).

The workspace object keeps track of open projects by maintaining a list of `ScriptProject` instances - the `ScriptProject` encapsulates the notion of either a Pig Latin or Hive QL project. Each `ScriptProject` consists of a collection of `Script` objects (subclassed into `HiveScript` and `PigScript`) and is responsible for managing these.

[H]

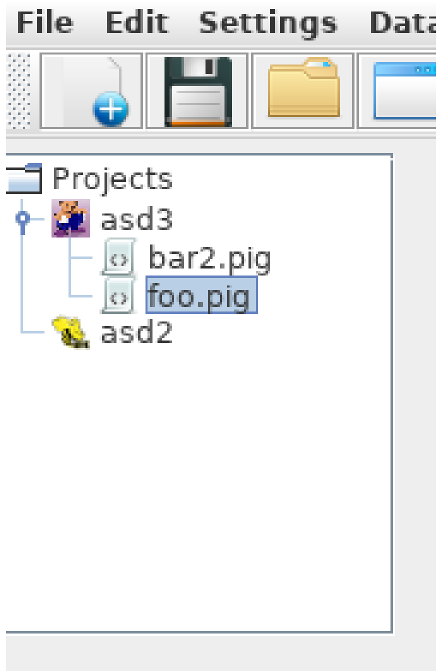


Figure 8.2: The workspace tree displays projects and their contents.

Each script instance contains an undo log which allows for the recovery of changes made to the script as the user is working on his/her project. The undo log is a stack: before a change is applied to the script, its current version is pushed onto the stack. Only then is the change applied to the script itself. An "undo" action simply involves popping the top of this stack: the top element is displayed in the script editor and is also pushed on top of the redo stack (which allows the user to redo (i.e. "undo undo") a change).

The user can interact with the workspace tree in order to move, rename or delete files and projects. These operations are handled by the `ui.com.mnu` package which contains the logic and interface code for the project tree's pop-up menu. File movement is handled through the workspace tree's drag events (users can drag script files into other projects) using the `TreeTransferHandler.java` (an original version of the tree transfer handler was written and published by Craig Wood on coderanch.com).

8.4 Remote File Manager

The `remote` package contains the three classes that are responsible for interfacing with remote filesystems (either the remote HDFS filesystem or a standard Unix filesystem). Both `FileManager` and `HadoopFileManager` rely on `RemoteServer` for connectivty and session handling (`RemoteServer` on the other hand relies on the the Java `JSch SSH library`). `RemoteServer` exposes three different ways of connecting the the remote filesystem: calling `connect()` will establish a directl SSH session with the project server; `tunnelProjectServerConnect()` will tunnel through an intermediary host and then establish a connection with the project server (that is, assuming that shell1.doc.ic.ac.uk is your intermediary and ebony.doc.ic.ac.uk is your project server, calling `tunnelProjectServerConnect()` will be equivalent to first SSHing into shell1 and from there executing the command ssh user@ebony.doc.ic.ac.uk). Similarily, `tunnelHDFSConnect()` will connect to the Hadoop server through an intermediary host.

Once connected, the following methods can be executed:

`sendFile` - Transfers a file from the localhost to the remote host.
`downloadFile` - Downloads file from remote host to the localhost.
`sendCommand` - Executes a given command on the remote host.
`disconnect` - Closes the SSH session(s).

As implied by its name, the `HadoopFileManager` class encapsulates the Hadoop file system (in essence it executes Hadoop shell commands by calling `sendCommand` in `RemoteServer`. Likewise, `FileManager` encapsulates the Unix filesystem on the remote host. It should be noted that Windows or other none POSIX compliant operating systems are currently not supported (the reason for this is two-fold: firstly, Hadoop is generally not run on Windows servers and secondly ease of implementation).

As a proof of AutoPig's modularity, the Hadoop file system manager was also turned into a Netbeans plugin (see figures 8.3 and 8.4).

8.5 Runtime configuration variables

The runtime configuration variables are contained in `hadoopdevtool.conf` and are as follows:

`projectFolder` - Refers to the absolute path of the project folder. That is, the folder in which all project directories will be created. e.g. /home/benjamin.

`currentProjectNode` - The application uses Swing's `JTree` component to display the project structure in the form of a tree. With "Projects" being the root node and any project being its childr (whose children in turn are .the individual script files that form the project). For example, a project called "FooBar" that contains 2 files, A1.pig and A2.pig, would take the form of: Projects - FooBar - {child 1: A1, child 2: A2}.

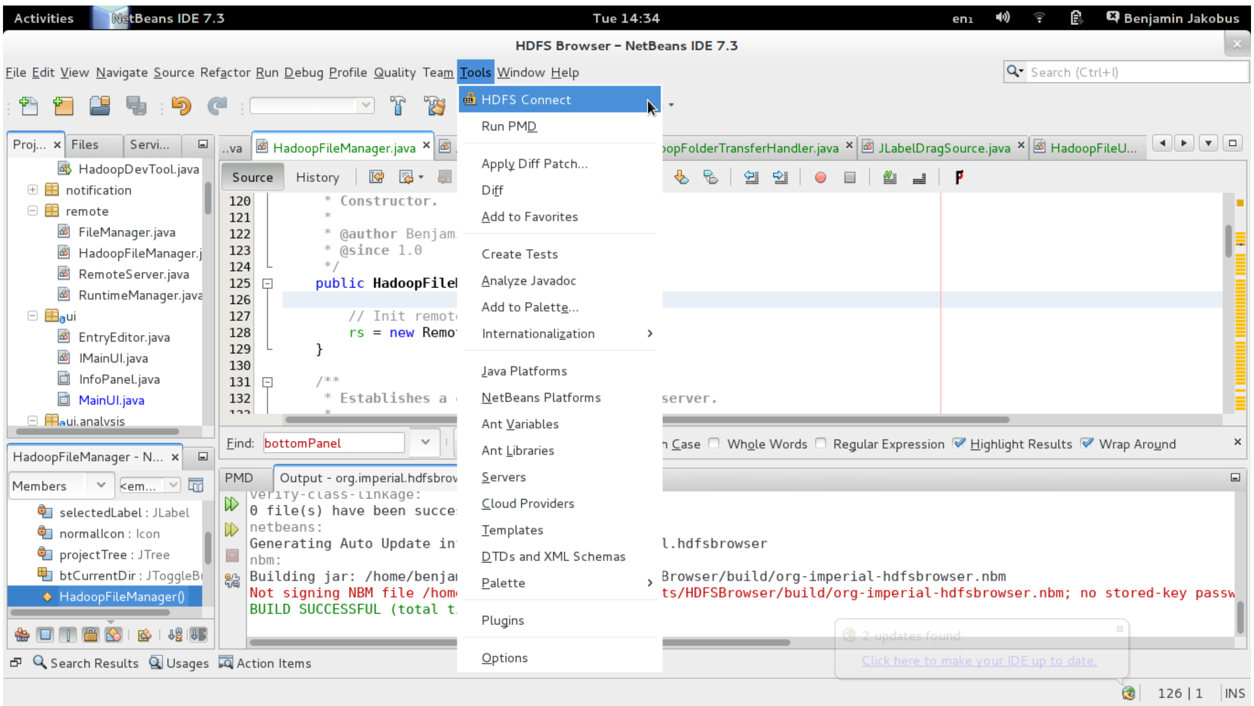
1.png 

Figure 8.3: The HDFS file manager as a Netbeans plugin.

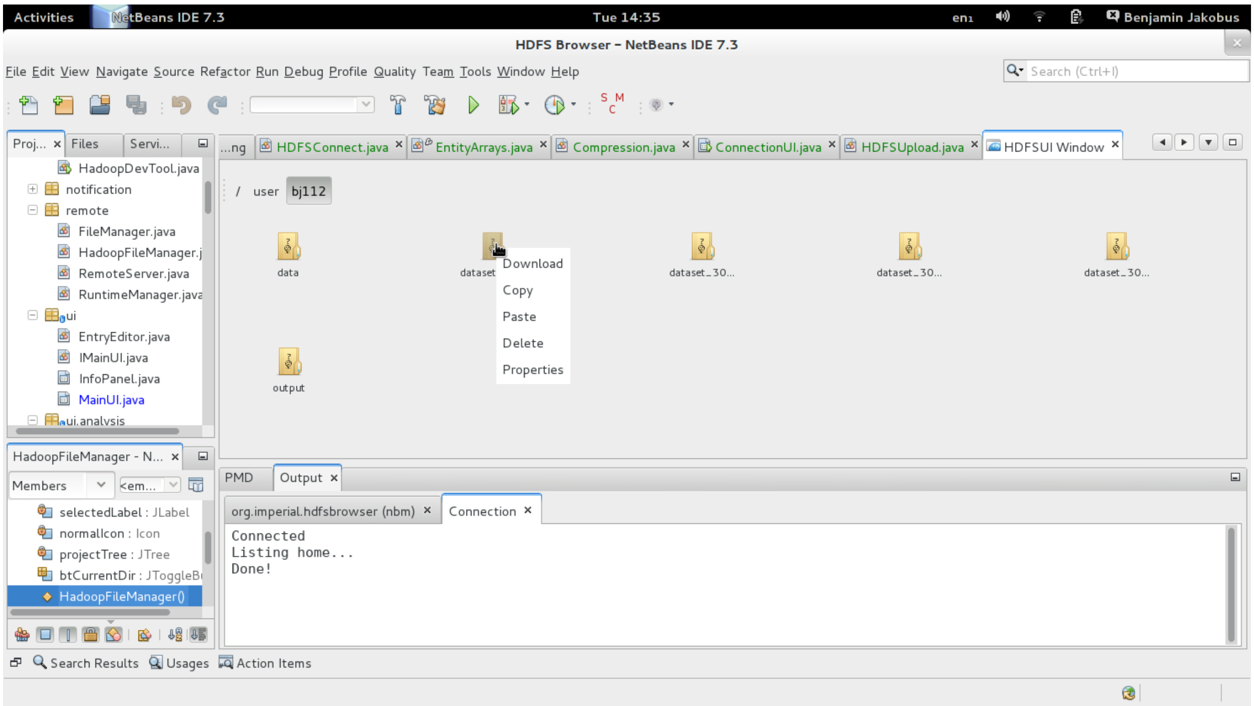
4.png 

Figure 8.4: The HDFS file manager as a Netbeans plugin.

The `currentProjectNode` refers to the node in the tree that is currently selected by the user (i.e. the node which the user last clicked). This is necessary to keep track of what the user is currently working on, hence allowing us to save changes to the correct script file, etc. This configuration variable is set inside the `ui.MainUI` class.

`currentProject` - By default null, this variable is the name of the project that is currently loaded into the workspace (i.e. it keeps track of the project on which the user is currently working on). Its purpose is similar to the above mentioned `currentProjectNode` variable in that it allows changes to be made to the correct project.

`currentScript` - This variable references the script that is currently loaded into the editor. As should be obvious to the reader, this variable is needed in order to determine which file the user is currently editing and to consequently allow the program to make changes to this file or to apply various other actions (such as refactoring or searching the script for a given keyword). Null by default.

`unsavedScripts` - A hashmap mapping strings to strings. Specifically, it maps the name of the unsaved script to its contents (the contents being the draft and **not** the actual saved contents of the script). Again, it is rather obvious that this is used in order to allow the user to switch between scripts without having to save them (the editor displays the contents from an unsaved script by querying this hashmap; otherwise the contents is loaded from the actual script object by calling `getContents()`. It should be noted that initially all scripts of open projects are loaded from file into memory). Furthermore, the `unsavedScripts` map is queried as the user closes the program: if it is not empty, then the user is prompted with a message dialog asking him/her whether he/she would like to save the unsaved changes. If so, the contents of `unsavedScripts` is written to the script denoted by its corresponding key.

`currentNodeInFocus` - Initially an empty string, this variable contains the name of the node from the project tree that is currently in focus (i.e. the name of the node from the `JTree` that the user clicked on last). The variable is only accessed by the `MainUI` and `EntryEditor` class and is used when the user wishes to rename a node: as the renaming event is triggered, the original name of the node is lost. Hence we access `currentNodeInFocus` in order to rename the file represented by this node to the new name specified by the user.

`deployDir` - A string that points to a directory on the remote host to which script files should be deployed and run (empty by default). Used by the auto-deploy feature.

`useTunnel` - A flag indicating whether or not to use the SSH tunnel settings when connecting to remote hosts (i.e. whether to tunnel through a given host if connecting to the project server or Hadoop filesystem).

`projectServerHost` - The hostname of the machine on which the project files are located. Auto-deployments of project files will be made on this host.

`projectServerUsername` - The username used to access the machine on which the project files are located.

`projectServerPassword` - The password used to access the machine on which the project files are located.

`sshTunnelHost` - The host which to use as a SSH tunnel when connecting to the Hadoop filesystem or the project server.

`sshTunnelUsername` - The username used to access the machine which to use as an SSH tunnel.

`sshTunnelPassword` - The password used to access the machine which to use as an SSH tunnel.

`hadoopHost` - The hostname of the machine on which the Hadoop filesystem is to be accessed.

`hadoopUsername` - The username used to access the machine on which the Hadoop filesystem is to be accessed.

`hadoopPassword` - The password used to access the machine on which the Hadoop filesystem is to be accessed.

`remoteWD` - Working directory on the remote host (used by the remote file manager). This variable is not saved to the configuration file upon exit (since the file manager is stateless, this variable is needed when transferring files to the server using the UI since the UI may represent a state different to the default state (the default state being the home directory)).

`clipBoard` - The clipboard is a non-persistent global variable that holds a reference to the file that is currently being copied (this information is used by the remote file managers).

`gitRemotePaths` - Maps remote repository URLs to local repositories. Used by the Git interface.

8.6 Git interface

The Git interface uses the local Git installation to provide the user with version control. The interface implements the most basic Git commands: init, add, push, pull, diff and commit. Two packages (a total of 13 classes) contain the code necessary for this feature: `ui.com.git` (contains the "front-end" i.e. Swing UI components for visualizing the actual Git interface) and `ui.com.mnu.git` (contains the action listeners and console wrapper). The interface is "plugged" into the main application through the project tree's pop-up menu: `ui.com.mnu.ProjectPopupMenu`.

The `committedFiles` hashmap inside the project tree renderer is used to color committed (black) and uncommitted (blue) entries.

8.7 Code auto-completion

Code auto-completion can be triggered by pressing CTRL-Space as one types a word into the script editor. Three classes, `editor.syntax.PigAutoComplete`, `editor.syntax.HiveAutoComplete` and `editor.syntax.CodeAutocomplete` implement this feature, whereby `CodeAutocomplete` is the abstract superclass implementing the code-completion logic (`PigAutoComplete` and `HiveAutoComplete` are subclasses that merely instantiate the keyword array used to create code suggestions).

8.8 Script configuration

Scripts are configured on a local basis (there is no global script configuration) through the project's file menu. The configuration UI is a front-end (`ui.com.ScriptConfigurationUI` that detects existing configurations or applies new configurations to a script file without the user having to actually modify any code. The interface is "plugged into" the main application through the project tree's pop-up menu: `ui.com.mnu.ProjectPopupMenu`.

The configuration UI allows for the setting of:

- mapred.min.split.size
- mapred.max.split.size

- mapred.reduce.tasks
- mapred.max.jobs.per.node

8.9 Remote path checker

The remote path checker is implemented by `editor.syntax.PathChecker` and scans the script for references to files on the remote HDFS using the remote file manager (see section 8.4). Custom highlighters were implemented to underline references to files in the code that do not exist on the remote server.

8.10 Auto-deployment, local execution and debugging

These three features are an extension of the file manager and console interface and as such are self-explanatory (see section 8.4). The debugger re-writes Pig scripts temporarily: it identifies all aliases in the script and adds an `ILLUSTRATE` statement to them. Similarly, Hive scripts are re-written as to include `EXPLAIN` statements.

8.11 Error Detection and Recovery

Errors are reports of the applications' inability to respond to an action request. Within the context of this document, the term "error" and "exception" may be used interchangeably.

The *AutoPig's* error system is designed in such a way that errors produced by individual components do not destabilize or halt the entire application. Each object processes internal errors or, if necessary, propagates these errors to other objects / components.

Users are notified of errors that occur as a result of invalid data input.

Depending on the error severity, errors are either silently discarded (this is done when error severity is extremely low and does not influence future operations), are displayed to the user via the UI (if the error is due to a bad command or input) or are written to standard output. Stack traces are included as part of every error report written to standard output.

All errors take the form of Java Exceptions

Once an exception is thrown, the current operation/service is aborted to prevent the error from migrating to other levels of the system. In the case of errors that are reported to the user, the service's restart will depend on the user. On the other hand, exceptions that are caught but that are not logged (i.e. low severity errors due to, for example, a very short interruption in the user's network connectivity) will result in the automatic restart of the affected service.

8.12 Data Persistence

Data persistence is handled by the `Conf` class in which the runtime configuration variables are located (see section 8.5). Upon termination of the program, the contents of the configuration variables are encrypted using 256-bit AES (CBC and padding) and are written to a file, `conf`, inside the application's root directory. The same file is decrypted and read when the application starts and the `Conf` object is updated respectively.

8.13 Concurrency and Synchronization

To prevent race conditions or other concurrency related problems, Java's in-built concurrency safeguards are used whenever threads are forked / new processes are being spawned. This means that:

1. All threads are derived using `java.lang.Thread`.
2. Methods shared by two or more threads are `synchronized` upon declaration to prevent race conditions.
3. Possible occurrences of `InterruptedException` are dealt within the thread's implementation of `Runnable`.
4. Data structures that are shared by two or more threads are made thread-safe by defining them using Java's `Collections` class which offers thread-safe implementations of all common data structures.

Chapter 9

Testing

This chapter details the testing procedures applied to the development of the AutoPig IDE as well as the Hive patches. IDE development follows the IEEE 892 standard for software testing documentation. The chapter begins by introducing the types of tests used as part of the system's development process, and then moves on to discuss the individual test specifications and test executions. The chapter concludes by presenting the results for each testing process and briefly highlights remedies applied to the reported faults / incidents.

The IDE underwent three levels of testing: unit testing, system testing, and usability testing. The details for each type are addressed in their appropriate section.

9.1 IDE

9.1.1 Test Goals

The testing of the AutoPig IDE aims to achieve the following quality levels:

1. No outstanding high severity faults prior to software release.
2. No outstanding product requirements prior to software release.
3. Highest possible quality of user interface, intuitiveness and ease of use.
4. Not more than one fault of the highest severity per 1000 lines of code.

Furthermore, testing aims to identify strengths and opportunities for future improvement.

9.1.2 Unit Testing

Unit testing is defined as "a method by which individual units of source code are tested to determine if they are fit for use"[11]. JUnit (a Java Unit Testing Framework) is used to conduct the unit tests undertaken by the author. Using a testing framework allows for the execution of the code body outside of its natural environment. That is, every unit of code on which a test is performed, is executed outside of the calling context for which it was originally created. This approach has the advantage that it allows for the identification of unnecessary dependencies whilst at the same time making it easy to construct test cases.

9.1.3 System Testing

System testing is "conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements". A black box testing approach is undertaken, whereby all aspects of the system are tested in an effort to detect any inconsistencies between individual components or between the system and the requirements specification. System testing is undertaken by Benjamin Jakobus.

9.1.4 Usability Testing

Usability testing is performed by and independent test team whereby each team member is asked to complete a set of scenarios contained on a task sheet. Upon completion, the participants are asked for feedback by completing a questionnaire.

9.1.5 Test Specification

Unit Test Design Specification

Unit testing utilizes the JUnit testing framework and is carried out by Benjamin Jakobus.

Encountered problems as well as their solutions are discussed in section 9.2.

Unit tests are divided into three phases:

Pre-test: Prior to starting the test, the test cases for each individual class must be recorded in a text file following the naming scheme of ≤class name.unitversion.txt≥ (for example, the test case for the first unit test of class `MainUI` must be recorded in a file called `MainUI.1.txt`). Test cases must include input values and expected output values.

Test execution: As the unit test is executed, the output values are recorded in the test case report file.

Post-test: The record is examined to identify potential errors.

All unit tests must be performed using JUnit and every class of the system must be tested. A test case passes when:

1. No errors have occurred.
2. The test's produced output matches the expected output.

If the above criteria are not met, then the test fails.

Due to the system's size, test cases are not included as part of this report, but are available upon request.

All unit tests are run on the hardware listed in table 9.1.5.

Model Name:	MacBook
Model Identifier	MacBook 6,1
Processor Name:	Intel Core 2 Duo
Processor Speed:	2.26 GHz
Number Of Processors:	1
Total Number Of Cores:	2
L2 Cache:	3 MB
Memory:	4 GB
Bus Speed:	1.07 GHz
Graphics Chipset Model:	NVIDIA GeForce 9400M
Graphics Chipset Type:	GPU
Graphics Chipset Bus:	PCI
Graphics Chipset VRAM (Total):	256 MB
Operating System:	Mac OS X (Snow Leopard)

Table 9.1: Test Hardware Configuration.

System Test Design Specification

System testing falls within the scope of black-box testing; a method of software testing that tests the functionality of an application without consideration for or knowledge of the internal code.

Therefore all of the system's features were tested over a period of 2 days without regards for the internal structure of the application.

Encountered problems as well as their solutions are discussed in section **??**.

The system test requires all aspects of the system to be tested without exceptions.

The test builds will be delivered using Netbeans 7.3 and Ant.

Bitbucket's bug tracker will be used to track bugs.

The pass criteria for the system test is as follows:

1. All processes will execute with no unexpected errors.
2. All processes will finish update/execution in an acceptable amount of time[1].

The system test may be suspended partially or fully on a given build if any of the following criteria are met:

1. Files are missing from the new build.
2. There is a fault with a feature that prevents its testing.
3. An excessive amount of bugs that should have been caught during the component/unit test phase are found during more advanced phases of testing.
4. A severe problem has occurred that does not allow testing to continue.
5. Development has not corrected the problem(s) that previously suspended testing.
6. A new version of the software is available to test.

The system test should verify all of the following:

[1] The precise time is be based on subjective experience.

1. Syntax highlighting.
2. Auto-deployment.
3. HDFS file manager.
4. Remote file manager.
5. Secure setting storage and data persistence.
6. Tunneling.
7. Git interface.
8. Dataset viewer.
9. Data analysis.
10. Script scheduling and notification.
11. Syntax checker.
12. Code auto-complete.
13. Code auto-formatting.
14. HDFS path checker.
15. Script configuration wizard.

System testing was completed twice, once on each of the hardware configurations outlined in table 9.2.

Usability Test Design Specification

Usability testing is carried out by an independent team of 5 participants over a period of 5 days whereby each test session is one hour in length.

The usability test aims to identify the application's:

- Ease of use.
- Overall appeal.

Model Name:	MacBook
Model Identifier	MacBook 6,1
Processor Name:	Intel Core 2 Duo
Processor Speed:	2.26 GHz
Number Of Processors:	1
Total Number Of Cores:	2
L2 Cache:	3 MB
Memory:	4 GB
Bus Speed:	1.07 GHz
Graphics Chipset Model:	NVIDIA GeForce 9400M
Graphics Chipset Type:	GPU
Graphics Chipset Bus:	PCI
Graphics Chipset VRAM (Total):	256 MB
Operating System:	Mac OS X (Snow Leopard)

Model Name:	iMac
Model Identifier	iMac 11,3
Processor Name:	Intel Core 2 Duo
Processor Speed:	2.8Ghz
Number Of Processors:	1
Total Number Of Cores:	2
L2 Cache:	3 MB
Memory:	16 GB
Bus Speed:	1.07 GHz
Graphics Chipset Model:	ATI Radeon HD 5750
Graphics Chipset Type:	GPU
Graphics Chipset Bus:	PCI
Graphics Chipset VRAM (Total):	256 MB
Operating System:	Mac OS X (Snow Leopard)

Table 9.2: Test Hardware Configuration.

- Degree of immersion.
- Faults that remained undiscovered during previous testing levels.
- Strong and weak points.

Furthermore, the test aims to:

- Identify general usability problems.
- Establish benchmarks for future comparisons.

The ideal target participants are as follows:

Gender: Males and Females

Age: 18 - 65

Professional Background: Students / IT

All participants met the target participant specification outlined in sub-section **??**.

Test subject demographic is as follows:

Gender: 4 males, 1 female.

Age: 22 - 30.

Professional Background: 3 participants are master students and 2 participants are professional software developers.

Comfort with the IT: All 5 participants reported a high level of comfort with IT.

All test participants were presented with three scenarios (refer to Appendix G for the worksheet) that had to be completed within a one hour time-frame.

The usability test scenarios can be summarized as follows:

Test Scenario 1: Familiarization with the system. The user should:

1. Adjust various application parameters.
2. Create new Hive project.
3. Add script files to the project.
4. Use the editor to write simple scripts.
5. Make use of code auto-completion and the syntax checker.
6. Deploy the project.
7. Run the scripts on the remote Hadoop server.
8. Exit the application.

Test Scenario 2: The same than scenario 1, however using Pig (as opposed to Hive).

Test Scenario 3: Using the remote file manager and scheduler. The user should:

1. Configure project server connection settings.
2. Use the remote file manager to connect to the project server (both using tunnelling and direct connection).
3. Use the file manager to upload, download and create files on the remote server.
4. Use the file manager to delete files.
5. Mark the auto-deployment directory.
6. Use the HDFS manager and repeat steps 1-4.

7. Use the script scheduler to deploy and run files.
8. Exit the application.

Upon completion of the three test scenarios, each participant is asked to complete a feedback questionnaire (see Appendix H).

Usability testing was performed on the hardware configurations outlined in table9.3.

Model Name:	MacBook
Model Identifier	MacBook 6,1
Processor Name:	Intel Core 2 Duo
Processor Speed:	2.26 GHz
Number Of Processors:	1
Total Number Of Cores:	2
L2 Cache:	3 MB
Memory:	4 GB
Bus Speed:	1.07 GHz
Graphics Chipset Model:	NVIDIA GeForce 9400M
Graphics Chipset Type:	GPU
Graphics Chipset Bus:	PCI
Graphics Chipset VRAM (Total):	256 MB
Operating System:	Mac OS X (Snow Leopard)

Model Name:	Dell Precision Workstations
Model Identifier	Dell Precision T1500 Tower
Processor Name:	Intel CoreTM
Processor Speed:	3.1 GHz
Number Of Processors:	1
Total Number Of Cores:	2
L2 Cache:	3 MB
Memory:	16 GB
Bus Speed:	1.07 GHz
Graphics Chipset Model:	ATI FirePro V4800
Graphics Chipset Type:	GPU
Graphics Chipset Bus:	PCI
Graphics Chipset VRAM (Total):	512 MB
Operating System:	1x Windows 7 Professional, 1x Ubuntu Linux 10.0

Table 9.3: Test Hardware Configuration.

9.2 Test Results

9.2.1 Unit Test Results

Unit testing occurred throughout the application's development. However due to the size of the result set, only the results of the system's final unit test are presented in this section.

Unit test results are listed in table F.1. All classes passed their test cases, although in some cases, several test iterations were required. Each test carries a unique ID, whereby the ID is a number ranging from 1 to n (where n denotes the total number of classes). Repeated tests for any single class are composed of the class's test ID and the ID of the repeated separated by a period. e.g. Performing three tests for the class `FooBar` would produce an ID range of 1.0, 1.1 and 1.2 etc.

The overall unit test results for the application are extremely positive: All classes passed with only 5 classes (out of a total of 83) requiring repeated tests. That is, 6% of all classes required repeated testing to eliminate identified bugs. This means that the quality level set out to be achieved by the testing process has been exceeded: A 6% error rate indicates 1 bug per 3,000 lines of code (given that the application consists of an approximate total of 16,000 lines of code) whilst the initial QoL assumed an error rate of 1 bug for every 1,000 lines of code.

9.2.2 Usability Test Results

Five individuals participated in the usability test which aimed to assess AutoPig's overall appeal and ease of use. The questionnaire's responses are summarized by figures **??** to **??** below (the figure's labels indicate the statement with which the subject was presented. The figure's legend denotes the choice of possible answers).

Feedback is mostly very positive. The only criticism voiced lies with the responsiveness of the remote file manager: All participants agreed that the HDFS file manager was somewhat slow. This however was to be expected, given network overhead and communication overhead with HDFS.

The application's strong point lies with its consistent, easy-to-use interface: All participants agreed with the statement that "it was easy to learn to use this system" and believed that they could become productive quickly using this system. Furthermore, the participants indicated that the system recovered quickly from errors, presented all information in a manner that was easy to understand and that it was very easy to modify application settings.

Another notable point is the application's stability: Over the period of trials, the application never crashed or became unusable.

Concluding; the usability test's overall verdict is extremely positive and exceeds initial expectations.

9.3 Hive Patches

Patch testing and development consists of 3 stages:

1. Proof of concept: Experiments are carried out to prove that the patch does indeed lead to a performance improvement (see chapter 5).
2. Patch is implemented. For each implementation stage, Hive is re-compiled from scratch (meaning that the project is cleaned and maven and ivy dependencies are downloaded and integrated) against Hadoop version 1.2.1.
3. Checkstyle tests are run locally to ensure that the produced source code adheres to Apache Hive coding guidelines.
4. A `git diff` is performed to produce the patch. Submitting only a difference ensures that "patches" only the changes to the source code and hence avoids issues of change integration where several people are working on the same file at the same time.
5. Once submitted, the Apache Hive development team compiles the patch, applies checkstyle tests and runs various JUnit tests. If the test does not pass with a score of +1 then the patch is rejected.
6. If the test passes, a core developer will review the submitted ticket. If the ticket passes the review, then the patch is accepted.

9.4 Summary

This chapter has analysed the application in terms of correctness, validity and usability using three levels of testing: unit testing, system testing, and usability testing. Each level of testing successfully identified bugs and short-comings, all of which have been resolved. Overall test results were very positive, with a small number of flaws having been identified. Identified strong points however include a good degree of usability, an extremely low bug rate and a highly efficient 2D visualization process.

Chapter 10

Conclusion

This chapter presents an overview of the project, reviews its accomplishments and discusses future development plans.

The project's aim was to improve the "big data user experience". This meant answering a range of questions: what are the challenges of dealing with big data? How can we deal with big data more effectively? And, last but not least, how can we improve the big data experience both in terms of usability and performance? To this end, several different benchmarks were run, the Pig and Hive codebases were analysed and findings were discussed with Apache developers. The gathered knowledge was utilized to produce patches for Hive, recommendations for both the Pig and Hive codebases as well as cluster configuration recommendations. Optimizations to the Hive codebase were demonstrated to be effective and a complete IDE, consisting of over 16,000 lines of code was implemented from scratch. The IDE's codebase was modularized to allow for easy integration into existing IDEs (as a proof of concept, the Hadoop file system manager was integrated into Netbeans 7.3).

10.1 Overview

From the perspective of the software developer, a primary problem associated with using or working with a big data context is the lack of development tools. Of the few available tools, none provide the capabilities needed to effectively work on industry standard applications. Using existing IDEs and text editors, the deployment of scripts was still tedious: one had to always either transfer them using SSH commands, SFTP transfer clients such as FileZilla or write deployment scripts. Browsing the Hadoop filesystem and transferring files to and from is time consuming when using only a terminal and none of the surveyed tools provided support for version control. The consequence of this and other missing features was the development of a stand-alone Pig/Hive IDE containing a syntax highlighter, auto-deployment functionality, file managers for interacting with remote file systems (HDFS and ”normal”), tunnelling, a Git interface, data analysis capability, script scheduling, syntax checkers, code auto-completion, HDFS path checking as well as a script configuration utility. As a proof of concept, a Netbeans plugin was developed to demonstrate the code's modularity and portability.

The application of further benchmarks produced several interesting results. For one, it answered the question as to how to best balance the ratio of mappers and reducers and demonstrated the impact that this ratio has on performance. It showed that reducers should be started early enough so that data transfer is spread out over time and thus preventing network bottlenecks but should not be started too early as to not use up slots that could be used by map tasks.

The experiments also found that care must be taken when specifying the maximum allowable map and reduce slots per node. For example, having a node with a maximum of 20 map slots but a script configured to use 30 map slots will result in significant performance penalties as the first 20 map tasks will run in parallel, but the additional 10 will only be spawned once the first 20 map tasks have completed execution (consequently requiring one extra round of computation). The same goes for the number of reduce tasks.

At first glance, the TPC-H benchmarks seemed to contradict earlier results in which Pig outperformed Hive. However closer examination revealed that nearly all TPC-H scripts relied heavily on the `Group By` operator - an operator which appears to be poorly implemented in Pig and which greatly degrades the performance of Pig Latin scripts (as demonstrated by the ISO benchmarks [13]). This supports the argument that TPC-H is not an accurate benchmark as operators are not evenly distributed throughout the scripts: if one operator is poorly implemented, then this will skew the entire result set - as can be seen in section 4.4 with the `Group By` operator. The excessive use of this operator within the TPC-H benchmarks skewed results significantly (recall from [13] that

Pig outperformed Hive in all instances except when using the `Group By` operator: when grouping data Pig was 104% slower than Hive[13]). Re-running the scripts whilst omitting the the grouping of data produces the expected results. For example, running script 3 (q3_shipping_priority.pig) whilst omitting the `Group By` operator significantly reduces the runtime (to 1278.49 seconds real time runtime or a total of 12,257,630ms CPU time).

Furthermore, the benchmarks confirmed that the CPU runtime scaled with real time runtime as expected.

Analysis of the Pig and Hive codebases resulted in the development of optimizations for Apache Hive (HIVE-5018.1.patch.txt (avoiding object instantiation in loops) speeds arithmetic operations up by approximately 2.6% (tested on dataset size 4, standalone mode, ISO benchmark script arithmetic.ql - the patched version of Hive took, on average, 303.39 seconds of real time runtime to complete the operations; the unpatched version took X seconds of real time runtime) Running the patched version of Hive on a small dataset consisting of 30,000,000 records (standalone mode) showed that the patched version was 2.6% faster than the unpatched version.") and revealed that overall, Pig's source code is of higher quality than that of Hive:

- Pig's codebase is nearly 18% smaller than Hive: Pig consists of a total code (and comment) base of 154,731 LOC (25% comments, 13% blank lines and the rest consists of actual code). Hive, although more sparsely documented, consists of 187,916 LOC (23% comments, 12% blank lines and the rest consists of actual code).
- The Hive codebase contains 71.5 issues per 100 lines of code[1]; Pig contains 24.85 issues per 100 lines of code.
- In terms of issues per lines of code, Pig is vastly superior to Hive: Hive has 31.59 optimization issues per 100 lines of code (LOC); Pig only 11.41.
- Both codebases have roughly the same percentage of design flaws: 5.05% for Hive and 4.9% for Pig.

[1]"Issues" refer to either basic violations of good practices, problems related to code size or complexity, bad commenting / bad code documentation, code that is deemed controversial, high or inappropriate coupling between classes, general bad design practices, empty or redundant code, violations related to naming of variables, classes and methods, various optimization issues, bad exception handling, unused code, security vulnerabilities, problems related to type resolution as well as suboptimal usage of strings and string buffers.

- The Pig codebase exposes a wider range of naming convention abuse (albeit less than Hive): a total of 5,100 issues were found.
- The Pig codebase seems more professional: 5 rookie mistakes per 100 LOC (as opposed to 12 rookie mistakes per 100 LOC for Hive).
- 14 intermediate mistakes per 100 LOC (as opposed to 38 intermediate mistakes per 100 LOC for Hive).
- 5 expert mistakes per 100 LOC (18 expert mistakes per 100 LOC for Hive).

In terms of cyclomatic complexity however, both codebases are the same. However Pig has a much lower n-path complexity than Hive (supporting the argument that Pig's codebase is much easier to understand and maintain).

Any performance differences between Pig and Hive should be attributed to code quality: on a logical level, translation of scripts into map-reduce jobs are the same.

Investigation into schedulers did not produce any interesting or conclusive results.

10.2 Project Outcome

Over the course of its development, *AutoPig* achieved significant real-world contributions in three areas: open source, science and industry:

1. **Open Source** - Based on the analysis performed as part of this project, several patches have been developed and accepted as a contribution to Apache Hive. Patches HIVE-5018 and HIVE-5019 are available as part of the core Hive codebase and are available for download via https://issues.apache.org. Benjamin Jakobus is listed as an official contributor to Apache Hive. Furthermore the developed IDE
2. **Industry** - Apache Hive has a wide range of adopters, including Netflix and Amazon. As the author's patches form part of Apache Hive 12.0, the contributed optimizations and performance improvements have a real-world value.

10.3 Future Work

Whilst much has been achieved given the tight development schedule, development will continue in order to exploit the project's full potential. From September 2013 onwards, AutoPig will be extended to include:

- Auto-formatter for AutoPig. To date no official style guide for either Hive or Pig exist so a style needs to be determined upon.
- Code refactoring.
- Spell-checker.
- Code templates.

Further patches for Hive will be implemented following the recommendations made as part of this report. A fix for Pig's `Group By` operator will be explored.

10.4 Summary of Thesis Achievements

To summarize, research over the past three months resulted in the following achievements:

- Apache Hive patchs.
- AutoPig - A complete IDE for Pig/Hive.
- A Netbeans plugin for remote file management in HDFS.
- Benchmark results.
- Cluster configuration knowledge.

Appendices

Appendix A

Legend: script abbreviations

Mapping of script abbreviations to script names for the TPC-H benchmarks (Hive).

Abbreviation	Script names
q1	q1_pricing_summary_report.hive
q2	q2_minimum_cost_supplier.hive
q3	q3_shipping_priority.hive
q4	q4_order_priority.hive
q5	q5_local_supplier_volume.hive
q6	q6_forecast_revenue_change.hive
q7	q7_volume_shipping.hive
q8	q8_national_market_share.hive
q9	q9_product_type_profit.hive
q10	q10_returned_item.hive
q11	q11_important_stock.hive
q12	q12_shipping.hive
q13	q13_customer_distribution.hive
q14	q14_promotion_effect.hive
q15	q15_top_supplier.hive
q16	q16_parts_supplier_relationship.hive
q17	q17_small_quantity_order_revenue.hive
q18	q18_large_volume_customer.hive
q19	q19_discounted_revenue.hive

Abbreviation	Script names
q20	q20_potential_part_promotion.hive
q21	q21_suppliers_who_kept_orders_waiting.hive
q22	q22_global_sales_opportunity.hive

Appendix B

Scripts, Logical Plans, Physical Plans, MR Plans

The Pig Latin script for performing arithmetic operations on dataset size 4 (map and reduce configuration information omitted).

```
A = load '/user/bj112/data/4/dataset_30000000' using PigStorage('\t')
 as (name, age, gpa) PARALLEL 8;
B = foreach A generate age * gpa + 3, age/gpa - 1.5 PARALLEL 8;
store B into 'dataset_30000000_projection' using PigStorage() PARALLEL 8;
```

The Abstract Syntac Tree and Logical Plan generated for the Hive QL script used to perform arithmetic operations on dataset size 4 (ISO benchmarks):

```
ABSTRACT SYNTAX TREE:
  (TOK_QUERY (TOK_FROM (TOK_TABREF (TOK_TABNAME dataset_30000000)))
  (TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE))
  (TOK_SELECT (TOK_SELEXPR (+ (* (. (TOK_TABLE_OR_COL dataset_30000000) age)
  (. (TOK_TABLE_OR_COL dataset_30000000) gpa)) 3) F1)
  (TOK_SELEXPR (- (/ (. (TOK_TABLE_OR_COL dataset_30000000) age)
  (. (TOK_TABLE_OR_COL dataset_30000000) gpa)) 1.5) F2))
  (TOK_WHERE (> (. (TOK_TABLE_OR_COL dataset_30000000) gpa) 0))))
```

```
STAGE DEPENDENCIES:
  Stage-1 is a root stage
  Stage-0 is a root stage

STAGE PLANS:
  Stage: Stage-1
    Map Reduce
      Alias -> Map Operator Tree:
        dataset_30000000
          TableScan
            alias: dataset_30000000
            Filter Operator
              predicate:
                  expr: (gpa > 0.0)
                  type: boolean
              Select Operator
                expressions:
                      expr: ((age * gpa) + 3)
                      type: float
                      expr: ((age / gpa) - 1.5)
                      type: double
                outputColumnNames: _col0, _col1
                File Output Operator
                  compressed: false
                  GlobalTableId: 0
                  table:
                      input format: org.apache.hadoop.mapred.TextInputFormat
                      output format: org.apache.hadoop.hive.ql.
                       io.HiveIgnoreKeyTextOutputFormat

  Stage: Stage-0
    Fetch Operator
      limit: -1
```

The Logical Plan generated for the Pig Latin script used to perform arithmetic operations on dataset size 4 (ISO benchmarks):

```
#-----------------------------------------------
# New Logical Plan:
#-----------------------------------------------
B: (Name: LOStore Schema: #49:double,#54:double)ColumnPrune:InputUids=[38, 43]
ColumnPrune:OutputUids=[38, 43]
|
|---B: (Name: LOForEach Schema: #49:double,#54:double)
    |   |
    |   (Name: LOGenerate[false,false] Schema: #49:double,#54:double)
    |   |   |
    |   |   (Name: Add Type: double Uid: 49)
    |   |   |
    |   |   |---(Name: Multiply Type: double Uid: 46)
    |   |   |   |
    |   |   |   |---(Name: Cast Type: double Uid: 20)
    |   |   |   |   |
    |   |   |   |   |---age:(Name: Project Type: bytearray Uid:
    20 Input: 0 Column: (*))
    |   |   |   |
    |   |   |   |---(Name: Cast Type: double Uid: 21)
    |   |   |       |
    |   |   |       |---gpa:(Name: Project Type: bytearray Uid:
    21 Input: 1 Column: (*))
    |   |   |
    |   |   |---(Name: Cast Type: double Uid: 47)
    |   |       |
    |   |       |---(Name: Constant Type: int Uid: 47)
    |   |   |
    |   |   (Name: Subtract Type: double Uid: 54)
    |   |   |
    |   |   |---(Name: Divide Type: double Uid: 52)
```

```
    |   |   |   |
    |   |   |   |---(Name: Cast Type: double Uid: 20)
    |   |   |   |   |
    |   |   |   |   |---age:(Name: Project Type: bytearray Uid:
    20 Input: 2 Column: (*))
    |   |   |   |
    |   |   |   |---(Name: Cast Type: double Uid: 21)
    |   |   |       |
    |   |   |       |---gpa:(Name: Project Type: bytearray Uid:
    21 Input: 3 Column: (*))
    |   |   |
    |   |   |---(Name: Constant Type: double Uid: 53)
    |   |
    |   |---(Name: LOInnerLoad[0] Schema: age#20:bytearray)
    |   |
    |   |---(Name: LOInnerLoad[1] Schema: gpa#21:bytearray)
    |   |
    |   |---(Name: LOInnerLoad[0] Schema: age#20:bytearray)
    |   |
    |   |---(Name: LOInnerLoad[1] Schema: gpa#21:bytearray)
    |
    |---A: (Name: LOLoad Schema: age#20:bytearray,gpa#21:bytearray)
    ColumnPrune:RequiredColumns=[1, 2]ColumnPrune:InputUids=[21, 20]
    ColumnPrune:OutputUids=[21, 20]RequiredFields:[1, 2]

#-----------------------------------------------
# Physical Plan:
#-----------------------------------------------
B: Store(hdfs://ebony:54310/user/bj112/dataset_30000000_projection:PigStorage)
- scope-21
|
|---B: New For Each(false,false)[bag] - scope-20
    |   |
```

```
    |   Add[double] - scope-8
    |   |
    |   |---Multiply[double] - scope-5
    |   |   |
    |   |   |---Cast[double] - scope-2
    |   |   |   |
    |   |   |   |---Project[bytearray][0] - scope-1
    |   |   |
    |   |   |---Cast[double] - scope-4
    |   |       |
    |   |       |---Project[bytearray][1] - scope-3
    |   |
    |   |---Cast[double] - scope-7
    |       |
    |       |---Constant(3) - scope-6
    |   |
    |   Subtract[double] - scope-17
    |   |
    |   |---Divide[double] - scope-15
    |   |   |
    |   |   |---Cast[double] - scope-12
    |   |   |   |
    |   |   |   |---Project[bytearray][0] - scope-11
    |   |   |
    |   |   |---Cast[double] - scope-14
    |   |       |
    |   |       |---Project[bytearray][1] - scope-13
    |   |
    |   |---Constant(1.5) - scope-16
    |
    |---A: Load(/user/bj112/data/4/dataset_30000000:PigStorage('')) - scope-0

#--------------------------------------------------
```

```
# Map Reduce Plan
#--------------------------------------------------
MapReduce node scope-22
Map Plan
B: Store(hdfs://ebony:54310/user/bj112/dataset_30000000_projection:PigStorage)
- scope-21
|
|---B: New For Each(false,false)[bag] - scope-20
    |   |
    |   Add[double] - scope-8
    |   |
    |   |---Multiply[double] - scope-5
    |   |   |
    |   |   |---Cast[double] - scope-2
    |   |   |   |
    |   |   |   |---Project[bytearray][0] - scope-1
    |   |   |
    |   |   |---Cast[double] - scope-4
    |   |       |
    |   |       |---Project[bytearray][1] - scope-3
    |   |
    |   |---Cast[double] - scope-7
    |       |
    |       |---Constant(3) - scope-6
    |   |
    |   Subtract[double] - scope-17
    |   |
    |   |---Divide[double] - scope-15
    |   |   |
    |   |   |---Cast[double] - scope-12
    |   |   |   |
    |   |   |   |---Project[bytearray][0] - scope-11
    |   |   |
```

```
    |   |   |---Cast[double] - scope-14
    |   |       |
    |   |       |---Project[bytearray][1] - scope-13
    |   |
    |   |---Constant(1.5) - scope-16
    |
    |---A: Load(/user/bj112/data/4/dataset_30000000:PigStorage(''))
    - scope-0--------
Global sort: false
----------------
```

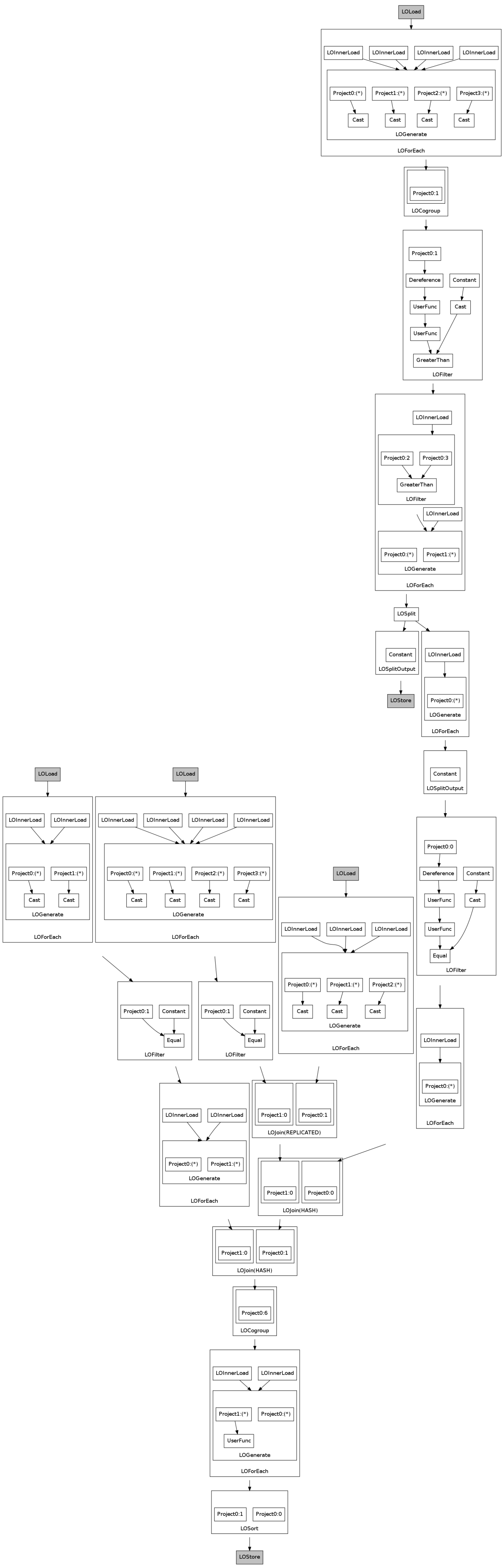


Figure B.1: Explanation of Pig TPC-H script q21_suppliers_who_kept_orders_waiting.pig

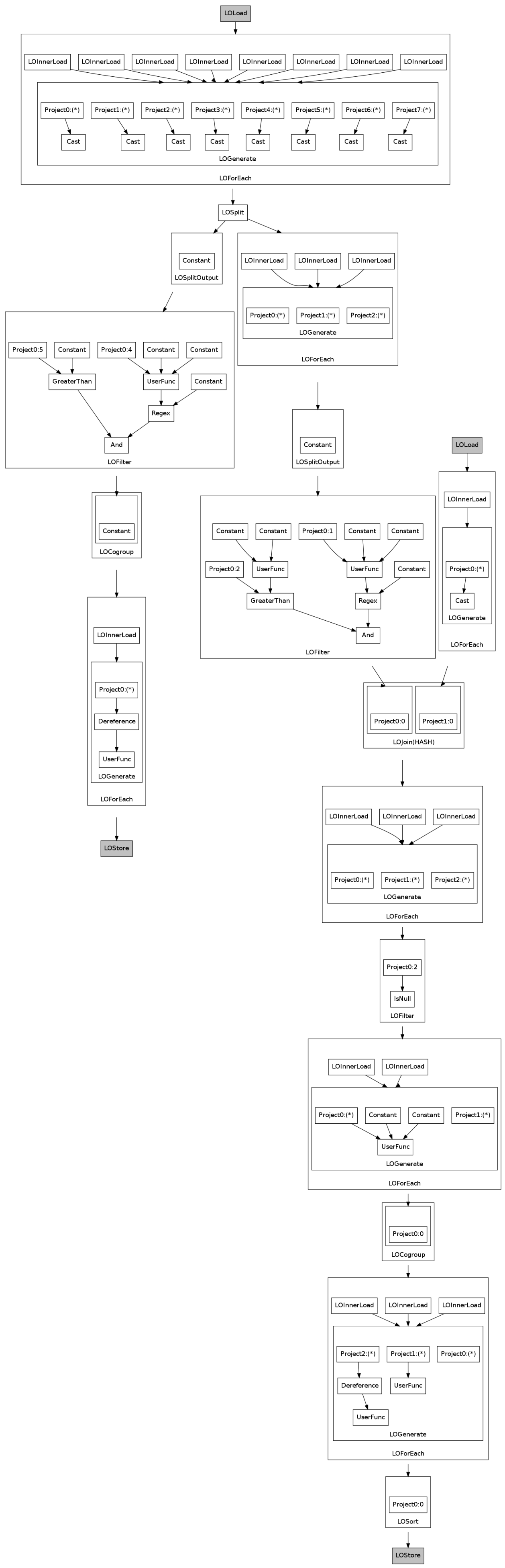


Figure B.2: Explanation of Pig TPC-H script q22_global_sales_opportunity.pig

Appendix C

Hive codebase issues

Issues found in the Hive codebase:

Issue	**Num. of issues**	**Issue category**
AbstractClassWithoutAbstractMethod	7	Design
AbstractClassWithoutAnyMethod	2	Design
AbstractNaming	97	Naming
AccessorClassGeneration	514	Design
AddEmptyString	89	Optimization
AppendCharacterWithChar	4	String and String-Buffer issues
ArrayIsStoredDirectly	98	Security vulnura-bilities
AssignmentInOperand	62	Controversial code
AssignmentToNonFinalStatic	6	Design
AtLeastOneConstructor	19	Controversial code
AvoidAccessibilityAlteration	1	Controversial code
AvoidArrayLoops	11	Optimization
AvoidCatchingNPE	5	Strict exceptions / Bad exception handling
AvoidCatchingThrowable	176	Strict exceptions / Bad exception handling
AvoidConstantsInterface	5	Design
AvoidDeeplyNestedIfStmts	56	Design
AvoidDuplicateLiterals	680	String and String-Buffer issues
AvoidFieldNameMatchingMethodName	131	Naming
AvoidFieldNameMatchingTypeName	6	Naming
AvoidFinalLocalVariable	79	Controversial code
AvoidInstanceofChecksInCatchClause	212	Design
AvoidInstantiatingObjectsInLoops	1623	Optimization
AvoidPrintStackTrace	245	Logging issues
AvoidProtectedFieldInFinalClass	6	Design
AvoidReassigningParameters	393	Design
AvoidRethrowingException	64	Strict exceptions / Bad exception handling
AvoidSynchronizedAtMethodLevel	112	Design
AvoidThrowingNullPointerException	72	Strict exceptions / Bad exception handling
AvoidThrowingRawExceptionTypes	410	Strict exceptions / Bad exception handling
AvoidUsingHardCodedIP	7	Basic
AvoidUsingOctalValues	7	Basic

Table C.1: All issues found within the Hive codebase.

Issue	Num. of issues	Issue category
AvoidUsingShortType	3162	Controversial code
AvoidUsingVolatile	3	Controversial code
BeanMembersShouldSerialize	4392	Misc
BooleanGetMethodName	128	Naming
BooleanInstantiation	7	Basic
BooleanInversion	4	Controversial code
ByteInstantiation	3	Migration issues
CallSuperInConstructor	416	Controversial code
CheckResultSet	22	Basic
ClassNamingConventions	22	Naming
ClassWithOnlyPrivateConstructorsShouldBeFinal	9	Design
CloneMethodMustImplementCloneable	34	Type resolution issues
CloneThrowsCloneNotSupportedException	17	Clone implementation issues
CloseResource	93	Design
CollapsibleIfStatements	171	Basic
CompareObjectsWithEquals	23	Design
ConfusingTernary	571	Design
ConstructorCallsOverridableMethod	219	Design
CouplingBetweenObjects	22	Coupling
CyclomaticComplexity	1955	Code size
DataflowAnomalyAnalysis	7181	Controversial code
DefaultLabelNotLastInSwitchStmt	16	Design
DefaultPackage	2325	Controversial code
DoNotCallSystemExit	52	J2EE issues
DoNotExtendJavaLangError	1	Strict exceptions / Bad exception handling
DoNotThrowExceptionInFinally	5	Strict exceptions / Bad exception handling
DoNotUseThreads	74	J2EE issues
DontImportJavaLang	6	Import Stmts
DoubleCheckedLocking	2	Basic
DuplicateImports	4	Import Stmts
EmptyCatchBlock	134	Empty code
EmptyFinallyBlock	3	Empty code
EmptyIfStmt	18	Empty code
EmptyMethodInAbstractClassShouldBeAbstract	73	Design
EmptyStatementNotInLoop	38	Empty code
EmptySwitchStatements	30	Empty code
EqualsNull	3	Design

Table C.2: All issues found within the Hive codebase.

Issue	**Num. of issues**	**Issue category**
ExceptionAsFlowControl	12	Strict exceptions / Bad exception handling
ExcessiveClassLength	61	Code size
ExcessiveImports	120	Coupling
ExcessiveMethodLength	186	Code size
ExcessiveParameterList	29	Code size
ExcessivePublicCount	87	Code size
FinalFieldCouldBeStatic	24	Design
FinalizeShouldBeProtected	1	Finalizer issues
ForLoopShouldBeWhileLoop	2	Basic
ForLoopsMustUseBraces	14	Missing braces
IfElseStmtsMustUseBraces	14	Missing braces
IfStmtsMustUseBraces	4490	Missing braces
ImmutableField	464	Design
ImportFromSamePackage	16	Import Stmts
InefficientEmptyStringCheck	3	String and String-Buffer issues
InefficientStringBuffering	1	String and String-Buffer issues
IntegerInstantiation	25	Migration issues
JUnit4TestShouldUseAfterAnnotation	29	Migration issues
JUnit4TestShouldUseBeforeAnnotation	37	Migration issues
JUnit4TestShouldUseTestAnnotation	310	Migration issues
JUnitAssertionsShouldIncludeMessage	2476	JUnit issues
JUnitSpelling	6	JUnit issues
JUnitTestsShouldIncludeAssert	86	JUnit issues
JUnitUseExpected	13	Migration issues
LocalVariableCouldBeFinal	23600	Optimization
LongInstantiation	6	Migration issues
LongVariable	4633	Naming
LooseCoupling	872	Coupling
MethodArgumentCouldBeFinal	33815	Optimization
MethodNamingConventions	2093	Naming
MethodReturnsInternalArray	67	Security vulnura-bilities
MissingBreakInSwitch	85	Design
MissingSerialVersionUID	337	Misc
MissingStaticMethodInNonInstantiatableClass	2	Design
NPathComplexity	677	Code size
NcssConstructorCount	1	Code size
NcssMethodCount	64	Code size
NcssTypeCount	12	Code size
NonCaseLabelInSwitchStatement	1	Design
NonStaticInitializer	3	Design
NonThreadSafeSingleton	4	Design
NullAssignment	2799	Controversial code

Table C.3: All issues found within the Hive codebase.

Issue	Num. of issues	Issue category
OnlyOneReturn	11181	Controversial code
OptimizableToArrayCall	28	Design
OverrideBothEqualsAndHashcode	21	Basic
PositionLiteralsFirstInComparisons	111	Design
PreserveStackTrace	225	Design
ProperCloneImplementation	18	Clone implementation issues
ProperLogger	187	Misc
ReplaceHashtableWithMap	1	Migration issues
ReturnEmptyArrayRatherThanNull	10	Design
ReturnFromFinallyBlock	8	Basic
ShortInstantiation	2	Migration issues
ShortMethodName	4	Naming
ShortVariable	7123	Naming
SignatureDeclareThrowsException	304	Type resolution issues
SimpleDateFormatNeedsLocale	33	Design
SimplifyBooleanAssertion	1	JUnit issues
SimplifyBooleanExpressions	63	Design
SimplifyBooleanReturns	7	Design
SimplifyConditional	9	Design
SimplifyStartsWith	9	Optimization
SingularField	104	Design
StringInstantiation	43	String and StringBuffer issues
StringToString	16	String and StringBuffer issues
SuspiciousConstantFieldName	98	Naming
SuspiciousEqualsMethodName	331	Naming
SuspiciousOctalEscape	20	Controversial code
SwitchDensity	15	Design
SwitchStmtsShouldHaveDefault	1019	Design
SystemPrintln	456	Logging issues
TestClassWithoutTestCases	1	JUnit issues
TooFewBranchesForASwitchStatement	808	Design
TooManyFields	60	Code size
TooManyMethods	586	Code size
TooManyStaticImports	11	Import Stmts
UncommentedEmptyConstructor	641	Design
UncommentedEmptyMethod	161	Design
UnconditionalIfStatement	183	Basic
UnnecessaryBooleanAssertion	32	JUnit issues
UnnecessaryCaseChange	17	String and StringBuffer issues
UnnecessaryConstructor	135	Controversial code

Table C.4: All issues found within the Hive codebase.

Issue	**Num. of issues**	**Issue category**
UnnecessaryConversionTemporary	3	Unnecessary code
UnnecessaryFinalModifier	66	Unnecessary code
UnnecessaryLocalBeforeReturn	77	Design
UnnecessaryParentheses	279	Controversial code
UnnecessaryReturn	48	Unnecessary code
UnnecessaryWrapperObjectCreation	31	Optimization
UnsynchronizedStaticDateFormatter	16	Design
UnusedFormalParameter	112	Unused code
UnusedImports	1553	Type resolution issues
UnusedLocalVariable	259	Unused code
UnusedModifier	668	Unused code
UnusedPrivateMethod	16	Unused code
UseArraysAsList	4	Optimization
UseAssertEqualsInsteadOfAssertTrue	14	JUnit issues
UseAssertNullInsteadOfAssertTrue	16	JUnit issues
UseAssertSameInsteadOfAssertTrue	55	JUnit issues
UseCollectionIsEmpty	247	Design
UseCorrectExceptionLogging	100	Misc
UseEqualsToCompareStrings	4	String and StringBuffer issues
UseIndexOfChar	17	String and StringBuffer issues
UseLocaleWithCaseConversions	238	Design
UseProperClassLoader	14	J2EE issues
UseSingleton	63	Design
UseStringBufferForStringAppends	184	Optimization
UselessOverridingMethod	17	Unnecessary code
UselessStringValueOf	5	String and StringBuffer issues
VariableNamingConventions	2098	Naming

Table C.5: All issues found within the Hive codebase.

Issue	Num. of issues	Issue category
AbstractClassWithoutAbstractMethod	6	Design
AbstractNaming	94	Naming
AccessorClassGeneration	2	Design
AddEmptyString	12	Optimization
AppendCharacterWithChar	37	String and StringBuffer issues
ArrayIsStoredDirectly	41	Security vulnurabilities
AssignmentInOperand	43	Controversial code
AssignmentToNonFinalStatic	3	Design
AtLeastOneConstructor	2	Controversial code
AvoidArrayLoops	14	Optimization
AvoidCatchingNPE	7	Strict exceptions / Bad exception handling
AvoidCatchingThrowable	10	Strict exceptions / Bad exception handling
AvoidConstantsInterface	1	Design
AvoidDeeplyNestedIfStmts	38	Design
AvoidDuplicateLiterals	111	String and StringBuffer issues
AvoidFieldNameMatchingMethodName	59	Naming
AvoidFieldNameMatchingTypeName	4	Naming
AvoidFinalLocalVariable	9	Controversial code
AvoidInstanceofChecksInCatchClause	8	Design
AvoidInstantiatingObjectsInLoops	494	Optimization
AvoidPrintStackTrace	32	Logging issues
AvoidReassigningParameters	117	Design
AvoidRethrowingException	122	Strict exceptions / Bad exception handling
AvoidStringBufferField	1	String and StringBuffer issues
AvoidSynchronizedAtMethodLevel	15	Design
AvoidThrowingRawExceptionTypes	347	Strict exceptions / Bad exception handling
AvoidUsingShortType	15	Controversial code
AvoidUsingVolatile	8	Controversial code
BeanMembersShouldSerialize	1727	Misc
BooleanGetMethodName	15	Naming
BooleanInstantiation	8	Basic
BooleanInversion	3	Controversial code
CallSuperInConstructor	207	Controversial code
ClassWithOnlyPrivateConstructorsShouldBeFinal	11	Design
CloneMethodMustImplementCloneable	35	Type resolution issues
CloneThrowsCloneNotSupportedException	3	Clone implementation issues
CollapsibleIfStatements	61	Basic
CompareObjectsWithEquals	23	Design
ConfusingTernary	438	Design
ConstructorCallsOverridableMethod	27	Design
CouplingBetweenObjects	4	Coupling
CyclomaticComplexity	606	Code size

Table C.6: All issues found within the Hive codebase.

Issue	Num. of issues	Issue category
DataflowAnomalyAnalysis	3085	Controversial code
DefaultLabelNotLastInSwitchStmt	1	Design
DefaultPackage	666	Controversial code
DoNotCallGarbageCollectionExplicitly	2	Controversial code
DoNotCallSystemExit	1	J2EE issues
DoNotUseThreads	16	J2EE issues
DontImportJavaLang	9	Import Stmts
DuplicateImports	3	Import Stmts
EmptyCatchBlock	35	Empty code
EmptyIfStmt	39	Empty code
EmptyMethodInAbstractClassShouldBeAbstract	69	Design
EmptyStatementNotInLoop	25	Empty code
EmptyWhileStmt	4	Empty code
ExceptionAsFlowControl	3	Strict exceptions / Bad exception handling
ExcessiveClassLength	19	Code size
ExcessiveImports	62	Coupling
ExcessiveMethodLength	89	Code size
ExcessiveParameterList	2	Code size
ExcessivePublicCount	12	Code size
FinalFieldCouldBeStatic	1	Design
FinalizeDoesNotCallSuperFinalize	1	Finalizer issues
ForLoopShouldBeWhileLoop	3	Basic
ForLoopsMustUseBraces	109	Missing braces
IfElseStmtsMustUseBraces	632	Missing braces
IfStmtsMustUseBraces	1205	Missing braces
ImmutableField	358	Design
ImportFromSamePackage	14	Import Stmts
InefficientStringBuffering	13	String and String-Buffer issues
InstantiationToGetClass	6	Design
InsufficientStringBufferDeclaration	2	String and String-Buffer issues
IntegerInstantiation	4	Migration issues
JUnit4TestShouldUseAfterAnnotation	4	Migration issues
JUnit4TestShouldUseBeforeAnnotation	2	Migration issues
LocalVariableCouldBeFinal	7962	Optimization
LongInstantiation	3	Migration issues
LongVariable	478	Naming
LooseCoupling	81	Coupling

Table C.7: All issues found within the Hive codebase.

Issue	Num. of issues	Issue category
MethodArgumentCouldBeFinal	9084	Optimization
MethodNamingConventions	29	Naming
MethodReturnsInternalArray	27	Security vulnurabilities
MissingBreakInSwitch	41	Design
MissingSerialVersionUID	2	Misc
MissingStaticMethodInNonInstantiatableClass	1	Design
NPathComplexity	198	Code size
NcssMethodCount	26	Code size
NcssTypeCount	1	Code size
NonThreadSafeSingleton	9	Design
NullAssignment	279	Controversial code
OnlyOneReturn	2497	Controversial code
OptimizableToArrayCall	33	Design
OverrideBothEqualsAndHashcode	8	Basic
PackageCase	161	Naming
PositionLiteralsFirstInComparisons	31	Design
PreserveStackTrace	98	Design
ProperCloneImplementation	36	Clone implementation issues
ProperLogger	109	Misc
ReplaceHashtableWithMap	1	Migration issues
ReplaceVectorWithList	7	Migration issues
ReturnEmptyArrayRatherThanNull	8	Design
ShortInstantiation	1	Migration issues
ShortMethodName	5	Naming
ShortVariable	4040	Naming
SignatureDeclareThrowsException	13	Type resolution issues
SimpleDateFormatNeedsLocale	1	Design
SimplifyBooleanExpressions	38	Design
SimplifyBooleanReturns	10	Design
SimplifyConditional	52	Design
SimplifyStartsWith	10	Optimization
SingularField	19	Design
StringInstantiation	1	String and StringBuffer issues
StringToString	2	String and StringBuffer issues
SuspiciousConstantFieldName	6	Naming
SuspiciousEqualsMethodName	1	Naming
SwitchDensity	2	Design

Table C.8: All issues found within the Hive codebase.

Issue	Num. of issues	Issue category
SwitchStmtsShouldHaveDefault	27	Design
SystemPrintln	166	Logging issues
TooManyFields	17	Code size
TooManyMethods	134	Code size
UncommentedEmptyConstructor	70	Design
UncommentedEmptyMethod	126	Design
UnnecessaryCaseChange	3	String and String-Buffer issues
UnnecessaryConstructor	10	Controversial code
UnnecessaryLocalBeforeReturn	23	Design
UnnecessaryParentheses	71	Controversial code
UnnecessaryReturn	3	Unnecessary code
UnnecessaryWrapperObjectCreation	15	Optimization
UnusedFormalParameter	42	Unused code
UnusedImports	268	Type resolution issues
UnusedLocalVariable	16	Unused code
UnusedModifier	190	Unused code
UnusedPrivateField	30	Unused code
UnusedPrivateMethod	8	Unused code
UseArrayListInsteadOfVector	6	Optimization
UseArraysAsList	1	Optimization
UseCollectionIsEmpty	117	Design
UseCorrectExceptionLogging	16	Misc
UseIndexOfChar	6	String and String-Buffer issues
UseLocaleWithCaseConversions	11	Design
UseProperClassLoader	8	J2EE issues
UseSingleton	42	Design
UseStringBufferForStringAppends	62	Optimization
UseStringBufferLength	2	String and String-Buffer issues
UselessOverridingMehod	8	Misc
UselessStringValueOf	1	String and String-Buffer issues
VariableNamingConventions	208	Naming
WhileLoopsMustUseBraces	15	Missing braces

Table C.9: All issues found within the Pig codebase.

Appendix D

Java code optimization test cases.

```
/**
 * String concatination – comparing + to StringBuffer
 */
public static void main(String[] args) {
String str1 = "asdasdasfdfgfgasdasdasfdfgfgasdasdasfdfgfgasdasda"
    + "sfdfgfgasdasdasfdfgfgasdasdasfdfgfg"
    + "sfdfgfgasdasdasfdfgfgasdasdasfdfgfg"
    + "sfdfgfgasdasdasfdfgfgasdasdasfdfgfg"
    + "sfdfgfgasdasdasfdfgfgasdasdasfdfgfg"
    + "sfdfgfgasdasdasfdfgfgasdasdasfdfgfg"
    + "sfdfgfgasdasdasfdfgfgasdasdasfdfgfg"
    + "sfdfgfgasdasdasfdfgfgasdasdasfdfgfg"
    + "sfdfgfgasdasdasfdfgfgasdasdasfdfgfg"
    + "sfdfgfgasdasdasfdfgfgasdasdasfdfgfg"
    + "sfdfgfgasdasdasfdfgfgasdasdasfdfgfg";
String str2 = "vcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvc"
    + "vbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcv"
    + "vbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcv"
    + "vbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcv"
    + "vbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcv"
    + "vbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcv"
```

```
            + "vbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcv"
            + "vbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcv"
            + "vbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcvvcvbcv";
    // Concatinate
    long start = System.currentTimeMillis();
    String result = "";

    for (int i = 0; i < 500; i++) {
        result += str1 + str2;
    }

    long end = System.currentTimeMillis();
    System.out.println("+ operation took " + (end - start) + " milliseconds");

    StringBuffer sb1 = new StringBuffer(str1);
    StringBuffer sb2 = new StringBuffer(str2);
    StringBuffer rsb = new StringBuffer("");
    start = System.currentTimeMillis();
    for (int i = 0; i < 500; i++) {
        rsb.append(sb1.append(sb2).toString());
    }

    end = System.currentTimeMillis();
    System.out.println("SB operation took " + (end - start) + " milliseconds");
}



            /**
             * Comparing the performance of Java's asList method to tight-loop copying.
             */
             public static void main(String[] args) {
    // Create array of 10,000 random strings
    SessionIdentifierGenerator gen = new SessionIdentifierGenerator();
```

```
        String[] array = new String[1000000];
        for (int i = 0; i < array.length; i++) {
            array[i] = gen.nextSessionId();
        }

        List<String> copy = new ArrayList<>();
        long start = System.currentTimeMillis();
        for (int i = 0; i < array.length; i++) {
            copy.add(array[i]);
        }
        long end = System.currentTimeMillis();
        System.out.println("loop copy took " + (end - start) + " milliseconds");

        start = System.currentTimeMillis();
        List<String> copy2 = Arrays.asList(array);
        end = System.currentTimeMillis();
        System.out.println("asList copy " + (end - start) + " milliseconds");
    }

     public static final class SessionIdentifierGenerator {

        private SecureRandom random = new SecureRandom();

        public String nextSessionId() {
            return new BigInteger(130, random).toString(32);
        }
    }



        /**
         * Examine the effect of unnecessary wrapper object creation.
         */
        public static void main(String[] args) {
```

```
        int j;
        SessionIdentifierGenerator gen = new SessionIdentifierGenerator();
        long start = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            j = Integer.valueOf(gen.nextSessionId()).intValue();
        }
        long end = System.currentTimeMillis();
        System.out.println("unnecessary wrapper object creation took " + (end - start) + "
            milliseconds");

        start = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            j = Integer.parseInt(gen.nextSessionId());
        }
        end = System.currentTimeMillis();
        System.out.println("using parseInt() took " + (end - start) + " milliseconds");



    }

    public static final class SessionIdentifierGenerator {

        private SecureRandom random = new SecureRandom();

        public String nextSessionId() {
            return new String("" + random.nextInt());
        }
    }



        /**
         * Examine the performance difference between declaring variables inside loops
         * and declaring them outside of loops.
```

```
    */
    public class InLoopInstantiationTest {

public InLoopInstantiationTest() {
    long start = System.currentTimeMillis();
    SessionIdentifierGenerator gen = new SessionIdentifierGenerator();
    for (int i = 0; i < 10000; i++) {
        FooBar f = new FooBar();
        Integer i1 = new Integer(i);
        String s = gen.nextSessionId();
    }
    long end = System.currentTimeMillis();
    System.out.println("in loop instantiation took " + (end - start) + " milliseconds");

    start = System.currentTimeMillis();
    FooBar f;
    Integer i1;
    String s;
    for (int i = 0; i < 10000; i++) {
        f = new FooBar();
        i1 = new Integer(i);
        s = gen.nextSessionId();
    }
    end = System.currentTimeMillis();
    System.out.println("avoiding in loop instantiation took " + (end - start) + "
        milliseconds");

}

public static void main(String[] args) {
    new InLoopInstantiationTest();
}
```

```
    private class FooBar {

        private String foo = "asdasdasdasdasdasdasdasdasdasdasdasdasdasd"
                + "asdasdasdasdasdasdasdasdasdasdasdasdasdasdasdasd"
                + "asdasdasdasdasdasdasdasdasdasdasdasdasdasdasdasd";
    }

    public final class SessionIdentifierGenerator {

        private SecureRandom random = new SecureRandom();

        public String nextSessionId() {
            return new BigInteger(130, random).toString(32);
        }
    }
}
```

```
        /**
         * Contrasting the performance between Vector and ArrayList.
         */
    public static void main(String[] args) {
        // Create array of 10,000 random strings



        List<Integer> arrayList = new ArrayList<>();
        Vector<Integer> vector = new Vector();
        long start = System.currentTimeMillis();
        for (int i = 0; i < 9999999; i++) {
            vector.add(i);
        }
        long end = System.currentTimeMillis();
```

```
        System.out.println("vector took " + (end - start) + " milliseconds");

        start = System.currentTimeMillis();
        for (int i = 0; i < 9999999; i++) {
            arrayList.add(i);
        }
        end = System.currentTimeMillis();
        System.out.println("ArrayList copy " + (end - start) + " milliseconds");
    }

        /**
         * Contrasting the performance between string and character appends.
         */
        public static void main(String[] args) {
        StringBuffer sb = new StringBuffer();
        long start = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            sb.append("a");
        }
        long end = System.currentTimeMillis();
        System.out.println("string append took " + (end - start) + " milliseconds");

        sb = new StringBuffer();
        start = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            sb.append('a');
        }
        end = System.currentTimeMillis();
        System.out.println("char append took " + (end - start) + " milliseconds");
    }

        /**
```

```
    * Contrasting the performance between string and character indexOf().
    */
public static void main(String[] args) {
    ASessionIdentifierGenerator gen = new SessionIdentifierGenerator();
    String str = new String();
    for (int i = 0; i < 1000; i++) {
        str += gen.nextSessionId();
    }

    long start = System.currentTimeMillis();
    for (int i = 0; i < 5000; i++) {
        int index = str.indexOf("d");
    }
    long end = System.currentTimeMillis();
    System.out.println("string indexOf took " + (end - start) + " milliseconds");

    start = System.currentTimeMillis();
    for (int i = 0; i < 5000; i++) {
        int index = str.indexOf('d');
    }
    end = System.currentTimeMillis();
    System.out.println("char indexOf " + (end - start) + " milliseconds");
}
```

Appendix E

Static Analysis Results

Class	**Num. of optimization issues**
FileInputLoadFunc.java	1
AccumulatorEvalFunc.java	2
ComparisonFunc.java	3
IllustrateDummyReporter.java	4
SortColInfo.java	5
PrimitiveEvalFunc.java	6
Expression.java	7
PigHadoopLogger.java	8
ConfigurationUtil.java	9
ColumnInfo.java	10
AlgebraicEvalFunc.java	11
HSeekableInputStream.java	12
StoreFuncWrapper.java	13
PigBooleanRawComparator.java	14
StoreFunc.java	15
NoopStoreRemover.java	16
EvalFunc.java	17
AccumulatorOptimizer.java	18
ResourceStatistics.java	19
InputSizeReducerEstimator.java	20
POJoinPackage.java	21
TypedOutputEvalFunc.java	22
ConstantExpression.java	23
MergeJoinIndexer.java	24
PODemux.java	25
LoadFunc.java	26
hadoop/HDataType.java	27
FuncSpec.java	28
POStream.java	29
POUserComparisonFunc.java	30
DataByteArray.java	31
POCounter.java	32
SampleOptimizer.java	33

Table E.1: The Pig codebase: classes mapped to the number of optimization issues (in ascending order).

Class	Num. of optimization issues
BackendException.java	34
POBinCond.java	35
PigGenericMapBase.java	36
POCollectedGroup.java	37
HPath.java	38
AppendableSchemaTuple.java	39
SchemaTupleBackend.java	40
HDataStorage.java	41
data/SchemaTupleFrontend.java	42
ResourceSchema.java	43
PigSplit.java	44
PhysicalOperator.java	45
HExecutionEngine.java	46
DataReaderWriter.java	48
PigException.java	50
COR.java	51
PlanPrinter.java	52
PigJrubyLibrary.java	53
PhyPlanSetter.java	54
MapReduceOper.java	55
JsonMetadata.java	56
PhysicalPlan.java	57
JarManager.java	58
FrontendException.java	59
PushDownForEachFlatten.java	60
shock/SSHSocketImplFactory.java	61
scripting/js/JsFunction.java	63
POPartialAgg.java	64
PigOutputCommitter.java	65
PigInputFormat.java	67
TextLoader.java	68
POMergeJoin.java	69
PigGenericMapReduce.java	70
DryRunGruntParser.java	71
POUserFunc.java	72
PigStorage.java	73
Utf8StorageConverter.java	74
POProject.java	79
DNFPlanGenerator.java	82
Main.java	85
POForEach.java	88
LineageTrimmingVisitor.java	94
Storage.java	96
JobStats.java	97
Launcher.java	99
SecondaryKeyOptimizer.java	100
MapRedUtil.java	101
QueryParserDriver.java	105
PigMacro.java	106
ExpToPhyTranslationVisitor.java	107
ProjectStarExpander.java	111

Table E.2: The Pig codebase: classes mapped to the number of optimization issues (in ascending order).

Class	**Num. of optimization issues**
MapReduceLauncher.java	113
FileLocalizer.java	119
LineageFindRelVisitor.java	121
ColumnPruneHelper.java	138
HBaseStorage.java	140
GroovyAlgebraicEvalFunc.java	147
RubySchema.java	149
SchemaTupleClassGenerator.java	154
CombinerOptimizer.java	160
DataType.java	162
BinInterSedes.java	167
GruntParser.java	168
AugmentBaseDataVisitor.java	193
TypeCheckingRelVisitor.java	211
JobControlCompiler.java	219
PigServer.java	248
MultiQueryOptimizer.java	250
POCast.java	254
Schema.java	260
OperatorPlan.java	277
SchemaTuple.java	345
LogToPhyTranslationVisitor.java	364
LogicalPlanBuilder.java	457
MRCompiler.java	528

Table E.3: The Pig codebase: classes mapped to the number of optimization issues (in ascending order).

Class	**Num. of optimization issues**
GetVersionPref.java	4
ReflectiveCommandHandler.java	5
ObjectPair.java	6
BufferedRows.java	7
DistinctElementsClassPath.java	8
AbstractCommandHandler.java	9
TableOutputFormat.java	10
SQLCompletor.java	11
Rows.java	12
TestCliDriverMethods.java	13
Base64TextOutputFormat.java	14
HCatException.java	15
HCatDriver.java	17
TestHiveLogging.java	18
DefaultHCatRecord.java	19
TypedBytesRecordReader.java	20
MetricsMBeanImpl.java	21
Metrics.java	22
Reflector.java	23
DataType.java	24
DatabaseConnection.java	25
MiniCluster.java	26
HBaseTestSetup.java	27
QFileClient.java	28
RCFileCat.java	29
TypedBytesWritableOutput.java	30
HCatSemanticAnalyzer.java	31
hcatalog/cli/HCatCli.java	33
ColorBuffer.java	34
TestHCatRecordSerDe.java	35
TestDefaultHCatRecord.java	36
HCatRecordSerDe.java	37
index/AggregateIndexHandler.java	38
HiveStatement.java	39
TypedBytesSerDe.java	40
InternalUtil.java	41
AlreadyExistsException.java	42
QTestGenTask.java	43
HCatMapReduceTest.java	44
SymlinkTextInputFormat.java	45
HBaseStorageHandler.java	46
TestHCatLoaderComplexSchema.java	47
Version.java	48
TestHCatMultiOutputFormat.java	49
SkewedValueList.java	50
TableAccessAnalyzer.java	51
HBaseRevisionManagerUtil.java	52
BinaryColumnStatsData.java	53
HCatBaseInputFormat.java	54

Table E.4: The Hive codebase: classes mapped to the number of optimization issues (in ascending order).

Class	**Num. of optimization issues**
EnvironmentContext.java	55
SerDeUtils.java	56
HiveServer.java	57
ColumnStatistics.java	58
JsonSerDe.java	59
HiveHBaseTableInputFormat.java	60
BeeLineOpts.java	61
HiveObjectPrivilege.java	62
HCatRecord.java	63
IndexUtils.java	64
HiveConnection.java	65
QueryPlan.java	66
PrivilegeGrantInfo.java	67
Type.java	68
TempletonControllerJob.java	69
HCatBaseStorer.java	70
PTFDeserializer.java	71
metastore/api/Schema.java	72
PigHCatUtil.java	73
HiveObjectRef.java	74
ThriftCLIServiceClient.java	75
TRowSet.java	76
HiveMetaTool.java	77
HBaseHCatStorageHandler.java	78
ZKUtil.java	79
HadoopJobExecHelper.java	80
TestLazyHBaseObject.java	81
OrcStruct.java	82
StatsTask.java	83
CommonJoinOperator.java	85
QBParseInfo.java	86
HCatUtil.java	87
MapredLocalTask.java	88
HBaseSerDe.java	89
SessionState.java	90
TestRevisionManager.java	92
AvroSerializer.java	93
ColumnStatsSemanticAnalyzer.java	94
HCatClientHMSImpl.java	95
metastore/api/SkewedInfo.java	97
TestHCatStorer.java	98
Operator.java	99
BinarySortableSerDe.java	100
AbstractBucketJoinProc.java	104
GenMRSkewJoinProcessor.java	105
HiveStringUtils.java	106
metastore/api/Partition.java	107
QueryPlan.java	108
HiveDatabaseMetaData.java	109
HiveConf.java	110
Index.java	111

Table E.5: The Hive codebase: classes mapped to the number of optimization issues (in ascending order).

Class	**Num. of optimization issues**
Complex.java	113
MultiOutputFormat.java	115
Stage.java	116
FileOutputCommitterContainer.java	117
CliDriver.java	118
TestHBaseDirectOutputFormat.java	119
Task.java	121
TestHBaseInputFormat.java	123
OpProcFactory.java	125
WindowingTableFunction.java	126
CubeQueryContext.java	127
TestHBaseSerDe.java	128
ExecDriver.java	129
PrincipalPrivilegeSet.java	132
GenMRFileSink1.java	133
BaseSemanticAnalyzer.java	134
TestOrcFile.java	135
TestAvroDeserializer.java	138
PTFPersistence.java	139
Server.java	140
TestHBaseBulkOutputFormat.java	142
NPath.java	143
Operator.java	144
PlanUtils.java	147
HiveDatabaseMetaData.java	152
Commands.java	156
HivePreparedStatement.java	158
Driver.java	160
HivePreparedStatement.java	161
BeeLine.java	166
DDLWork.java	168
CubeMetastoreClient.java	171
ReduceSinkDeDuplication.java	172
GenMapRedUtils.java	173
DummyRawStoreControlledCommit.java	185
DummyRawStoreForJdoConnection.java	186
RecordReaderImpl.java	188
ObjectInspectorUtils.java	190
RCFile.java	193
FunctionRegistry.java	194
MetaStoreUtils.java	200
TestLazyBinarySerDe.java	204
QTestUtil.java	208
WriterImpl.java	220
HcatDelegator.java	225
MapJoinProcessor.java	230
ColumnPrunerProcFactory.java	259
HiveBaseResultSet.java	285

Table E.6: The Hive codebase: classes mapped to the number of optimization issues (in ascending order).

Class	**Num. of optimization issues**
PTFTranslator.java	287
HiveMetaStoreClient.java	296
HiveBaseResultSet.java	297
HiveCallableStatement.java	332
HiveCallableStatement.java	336
TestHiveMetaStore.java	338
HiveMetaStore.java	411
Utilities.java	486
DDLSemanticAnalyzer.java	543
DDLTask.java	592
OrcProto.java	863
ObjectStore.java	891
ThriftHive.java	962
TCLIService.java	1696
SemanticAnalyzer.java	2037
ThriftHiveMetastore.java	11526]

Table E.7: The Hive codebase: classes mapped to the number of optimization issues (in ascending order).

Appendix F

Unit Test Results

Table F.1: Unit Test Results.

Test ID	Class	Test Suite	Result	Notes
1.0	BenchmarkItem	JUnit	Passed	-
2.0	HiveTPCHParser	JUnit	Passed	-
3.0	Job	JUnit	Passed	-
4.0	PigTPCHParser	JUnit	Passed	-
5.0	ScreenImage	JUnit	Passed	-
6.0	ScriptExecution	JUnit	Failed	Err. removing DS.
6.1	TPCHParser	JUnit	Passed	Resolved problem from 6.0. Test re-run.
7.0	ScriptDocumentTransferHandler	JUnit	Failed	- Clipboard problem.
7.1	SearchEngine	JUnit	Passed	- Resolved problem from 7.0. Test re-run.
8.0	StyledScriptDocument	JUnit	Failed	Test re-run.
8.1	TextLineNumber	JUnit	Failed	Test re-run.

8.2	Workspace	JUnit	Passed	Resolved problem from 8.1. Test re-run.
9.0	LightBlueHighlighter	JUnit	Passed	-
10.0	YellowHighlighter	JUnit	Passed	-
11.0	HiveProject	JUnit	Passed	-
12.0	PigProject	JUnit	Passed	-
13.0	ProjectTreeRenderer	JUnit	Passed	-
14.0	SchedulerCellRenderer	JUnit	Passed	-
15.0	ScriptProject	JUnit	Passed	-
16.0	HiveScript	JUnit	Passed	-
17.0	PigScript	JUnit	Passed	-
18.0	Script	JUnit	Passed	-
19.0	CodeAutoComplete	JUnit	Passed	-
20.0	HiveAutoComplete	JUnit	Passed	-
21.0	HiveSyntaxChecker	JUnit	Passed	-
22.0	PathChecker	JUnit	Failed	Err. HDFS connect.
22.1	PigAutoComplete	JUnit	Failed	Err. word distance.
22.2	PigAutoComplete	JUnit	Failed	Err. word distance.
22.3	PigAutoComplete	JUnit	Failed	Err. word distance.
22.4	PigAutoComplete	JUnit	Passed	Resolved problem from 22.3. Test re-run.
23.0	PigSyntaxChecker	JUnit	Passed	-
24.0	SyntaxChecker	JUnit	Passed	-
25.0	LineHighlightPainter	JUnit	Passed	-
26.0	Conf	JUnit	Passed	-
27.0	HadoopDevTool	JUnit	Passed	-

28.0	NotificationEngine	JUnit	Passed	-
29.0	FileManager	JUnit	Passed	-
30.0	HadoopFileManager	JUnit	Passed	-
31.0	RemoteServer	JUnit	Passed	-
32.0	RuntimeManager	JUnit	Passed	-
33.0	EntryEditor	JUnit	Passed	-
34.0	InfoPanel	JUnit	Passed	-
35.0	MainUI	JUnit	Passed	-
36.0	BenchmarkAnalysisUI	JUnit	Passed	-
37.0	FileStatsUI	JUnit	Passed	-
38.0	HadoopFileStatsUI	JUnit	Passed	-
39.0	ProjectCreationUI	JUnit	Passed	-
40.0	ScriptConfigurationUI	JUnit	Passed	-
41.0	ShellUI	JUnit	Failed	Problem with console
41.1	ShellUI	JUnit	Passed	Resolved problem from 41.0. Test re-run.
42.0	TextAreaOutputStream	JUnit	Passed	-
43.0	DnDUtils	JUnit	Passed	-
44.0	FileUploadTransferHandler	JUnit	Passed	-
45.0	FolderTransferHandler	JUnit	Passed	-
46.0	HadoopFileUploadTransferHandler	JUnit	Passed	-
47.0	HadoopFolderTransferHandler	JUnit	Passed	-
48.0	JLabelDragSource	JUnit	Passed	-
49.0	JLabelTransferable	JUnit	Passed	-
50.0	TreeTransferHandler	JUnit	Passed	-
51.0	GitCommitMessageUI	JUnit	Passed	-
52.0	GitDiffUI	JUnit	Passed	-
53.0	GitPullUI	JUnit	Passed	-
54.0	GitPushUI	JUnit	Passed	-

55.0	GitRemoteRepoUI	JUnit	Failed	Problem with remote.
55.1	GitRemoteRepoUI	JUnit	Failed	Problem with remote.
55.2	GitRemoteRepoUI	JUnit	Failed	Problem with remote.
55.3	GitRemoteRepoUI	JUnit	Passed	Resolved problem from 55.2. Test re-run.
56.0	ConfigurationActionListener	JUnit	Passed	-
57.0	FileDeleteActionListener	JUnit	Passed	-
58.0	FileManagerPopupMenu	JUnit	Passed	-
59.0	HadoopFileManagerPopupMenu	JUnit	Passed	-
60.0	LogPopupMenu	JUnit	Passed	-
61.0	NewFileActionListener	JUnit	Passed	-
62.0	ProjectPopupMenu	JUnit	Passed	-
63.0	SchedulerPopupMenu	JUnit	Passed	-
64.0	ScriptEditorPopupMenu	JUnit	Passed	-
65.0	AddActionListener	JUnit	Passed	-
66.0	CommitActionListener	JUnit	Passed	-
67.0	Console	JUnit	Passed	-
68.0	DiffActionListener	JUnit	Passed	-
69.0	Git	JUnit	Passed	-
70.0	InitActionListener	JUnit	Passed	-
71.0	PullActionListener	JUnit	Passed	-
72.0	PushActionListener	JUnit	Passed	-
73.0	SearchBox	JUnit	Passed	-
74.0	HadoopSettingsUI	JUnit	Passed	-
75.0	ProjectServerSettingsUI	JUnit	Passed	-
76.0	ProjectSettings	JUnit	Passed	-
77.0	TunnelSettingsUI	JUnit	Passed	-

Appendix G

Usability Test Scenarios

18-08-2013

AutoPig Usability Test Task Sheet Author: Benjamin Jakobus

This task sheet is to be used testing purposes only. Please complete all three scenarios in order and provide your feedback via the attached answer sheet.

Scenario 1: Familiarization The purpose of this scenario is to familiarize yourself with the application.

1. Start the application by either double-clicking on `AutoPig.jar` or typing `java -jar AutoPig.jar` into your console.
2. Wait for the UI to initialize.
3. Familiarize yourself with the UI by navigating it using your mouse and keyboard.
4. Adjust various application parameters such as your project directory / workspace.
5. Create a new Hive project.
6. Add three scripts to this project.
7. Solve supplied excercises.
8. Save and run the scripts when ready.

9. Exit the application.

Test Scenario 2: Working on Pig projects The purpose of this scenario is to create and work with a Pig Latin project.

1. Start the application by either double-clicking on `AutoPig.jar` or typing `java -jar AutoPig.jar` into your console.
2. Wait for the UI to initialize.
3. Create a new Pig project.
4. Add three scripts to this project.
5. Solve supplied exercises.
6. Save and run the scripts when ready.
7. Exit the application.

Test Scenario 3: Using the remote file manager The purpose of this scenario is to work with the remote file managers.

1. Start the application by either double-clicking on `AutoPig.jar` or typing `java -jar AutoPig.jar` into your console.
2. Wait for the UI to initialize.
3. Edit the project server connection settings.
4. Connect to the project server using no tunnel.
5. Connect to the project server using a tunnel.
6. Auto-deploy and run your project files.
7. Play with the scheduler - pause execution, resume execution, move scripts up and down the queue.
8. Edit the HDFS connection settings.
9. Download a file from the remote HDFS.

10. Upload a file.

11. Create a new directory.

12. Copy files.

13. Exit the application.

Appendix H

Usability Questionnaire

18-08-2013

AutoPig Usability Report Form Name: ______________________________

Once you are confident that you have completed the *AutoPig* testing process, then please fill this report form. The author, Benjamin Jakobus, would like to thank you for generously volunteering your time to participate in this usability testing.

1. I feel that I have successfully completed all the tasks on the task sheet.

2. In relation to other software I have used, I found the *AutoPig* to be: (Tick one box only)

1. Very easy to use
2. Easy to use
3. OK to use
4. Difficult to use
5. Very difficult to use

3. I found the script editor very easy to use: (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

3. I found the code auto-complete feature very easy to use: (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

3. I found the script editor's responsiveness to user input to be: (Tick one box only)

1. Very good
2. Good
3. OK
4. Bad
5. Unacceptable

4. I found the HDFS file manager's responsiveness to user input to be: (Tick one box only)

1. Very good
2. Good
3. OK

4. Bad
5. Unacceptable

5. I found the remote file manager's responsiveness to user input to be: (Tick one box only)

1. Very good
2. Good
3. OK
4. Bad
5. Unacceptable

6. The controls were well organized and easy to find. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

7. I found the remote file manager very easy to use. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

8. I immediately understood the function of each feature. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

9. All of the functions I expected to find in an industry standard IDE were present. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

10. I found it very easy to modify application settings. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

11. The system never crashed or froze during the time that I used the system. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree

4. Disagree

5. Strongly Disagree

12. I would buy and use this system software. (Tick one box only)

1. Strongly agree

2. Agree

3. Neither agree nor disagree

4. Disagree

5. Strongly Disagree

13. I could effectively complete the tasks and scenarios using this system. (Tick one box only)

1. Strongly agree

2. Agree

3. Neither agree nor disagree

4. Disagree

5. Strongly Disagree

14. I felt comfortable using this system. (Tick one box only)

1. Strongly agree

2. Agree

3. Neither agree nor disagree

4. Disagree

5. Strongly Disagree

15. It was easy to learn to use this system. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

16. I believe I could become productive quickly using this system. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

17. The system gave error messages that clearly told me how to fix problems. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

18. Whenever I made a mistake using the system, I could recover easily and quickly. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree

4. Disagree
5. Strongly Disagree

19. Using the system enhanced my productivity. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

20. The information provided by the system was easy to understand. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

21. I felt the system difficult to use. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

22. I was able to complete the tasks and scenarios quickly using this system. (Tick one box only)

1. Strongly agree
2. Agree
3. Neither agree nor disagree
4. Disagree
5. Strongly Disagree

Bibliography

[1] Hadoop Apache *"Fair Scheduler,"*. http://hadoop.apache.org/docs/stable/fair_scheduler.html, Visited 30/05/2013

[2] Jones T. M., IBM *"Scheduling in Hadoop - An introduction to the pluggable scheduler framework,"*. http://www.ibm.com/developerworks//library/os-hadoop-scheduling/, Visited 30/05/2013

[3] Hadoop Apache *"Capacity Scheduler Guide,"*. http://hadoop.apache.org/docs/stable/capacity_scheduler.html, Visited 30/05/2013

[4] Zaharia M., Konwinski A., Joseph A. D., Katz R., Stoica I. (2008) *"Improving MapReduce performance in heterogeneous environments"*,. Proceedings of the 8th USENIX conference on Operating systems design and implementation San Diego, California.

[5] Apache (2009) *"Fair Scheduler Design Document"*,.

[6] Zaharia M., Borthakur D., Sen Sarma J., Elmeleegy K., Shenker S., Stoica I. (2010) *"Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling"*,. In EuroSys 2010.

[7] Seo S., Jang I., Woo K., Kim I., Kim J-S., Maeng S. (2009) *"HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment"*,. IEEE International Conference on Cluster Computing and Workshops, 2009.

[8] Thirumala Rao B., Reddy L.S.S. (2011) *"Survey on Improved Scheduling in Hadoop MapReduce in Cloud Environments"*,. International Journal of Computer Applications (0975 - 8887) Volume 34 - No.9.

[9] Kc K., Anyanwu K. (2010) *"Scheduling Hadoop Jobs to Meet Deadlines"*,. Proc. CloudCom, 2010, pp.388-392.

[10] Sandholm, Thomas, Lai, Kevin (2010) *"Dynamic proportional share scheduling in Hadoop",*. Proceedings of the 15th international conference on Job scheduling strategies for parallel processing.

[11] Sommerville, I., (2010), *"Software Engineering",*. 8th Edition, Pearson Publishing.

[12] Transaction Processing Council *"Transaction Processing Council Website,"*. http://www.tpc.org/, Visited 18/06/2013

[13] Jakobus B. (2013) *"Data Managment in Big Data,"*. Independent Study Option, Imperial College London

[14] Apache Software Foundation. (2009), *"Hive, PIG, Hadoop benchmark results",*. https://issues.apache.org/jira/secure/attachment/12411185/hive_benchmark_2009-06-18.pdf, Visited 03/01/2013

[15] Moussa, R. (2012), *"TPC-H Benchmarking of Pig Latin on a Hadoop Cluster",*. Communications and Information Technology (ICCIT), 2012 International Conference, pages 85 - 90.

[16] Loebman S.; Nunley D.; Kwon Y.; Howe B.; Balazinska M.; Gardner. J.P. (2012), *"Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help?",*. Cin Proc. of CLUSTER. 2009, pages 1 - 10.

[17] Stewart Robert J.; Trinder P.; Loidl H. (2011), *"Comparing High Level MapReduce Query Languages",*. Springer Berlin Heidelberg, Advanced Parallel Processing Technologies, pages 58-72.

[18] Transaction Processing Performance Council (TPC) (2013), *"TPC Benchmark H",*. Standard Specification, Revision 2.15.0, Transaction Processing Performance Council (TPC), Presidio of San Francisco

[19] Rutherglen J.; Wampler D.; Capriolo E. (2012), *"Programming Hive",*. O'Reilly, ISBN: 978-1-449-31933-5

[20] Stackoverflow. (2013), *"Performance: Pig vs Hive",*. http://stackoverflow.com/questions/17422005/perforr pig-vs-hive, Visited 03/07/2013

[21] Stackoverflow. (2013), *"Why to use StringBuffer in Java instead of the string concatenation operator",*. http://stackoverflow.com/questions/65668/why-to-use-stringbuffer-in-java-instead-of-the-string-concatenation-operator2005/performance-pig-vs-hive, Visited 17/07/2013

[22] PMD. (2013), *"Java Optimization",*. http://pmd.sourceforge.net/pmd-5.0.4/rules/java/optimizations.html Visited 17/07/2013

[23] Java Documentation, Oracle. (2013), *"Class Arrays - Javadoc",*. http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html#asList%28T...%29 Visited 17/07/2013

[24] Java Documentation, Oracle. (2013), *"Writing Final Classes and Methods",*. http://docs.oracle.com/javase/tutorial/java/IandI/final.html Visited 21/07/2013

[25] Java Documentation, Oracle. (2013), *"Java^{TM} Platform, Standard Edition 6 API Specification",*. http://docs.oracle.com/javase/6/docs/api/ Visited 24/07/2013

[26] Java Documentation, Oracle. (2013), *"Defining Methods",*. http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html Visited 24/07/2013

[27] Java Documentation, Oracle. (2013), *"Variables",*. http://docs.oracle.com/javase/tutorial/java/nutsandbolts/ Visited 24/07/2013

[28] McCabe J. T. (1976), *"A Complexity Measure",*. IEEE Transactions on Software Engineering, Vol. SE-2, No. 4.

[29] The Apache Software Foundation (2013), *"Apache Pig",*. https://blogs.apache.org/pig/entry/apache_pig_it_goes_to Visited 25/08/2013