UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

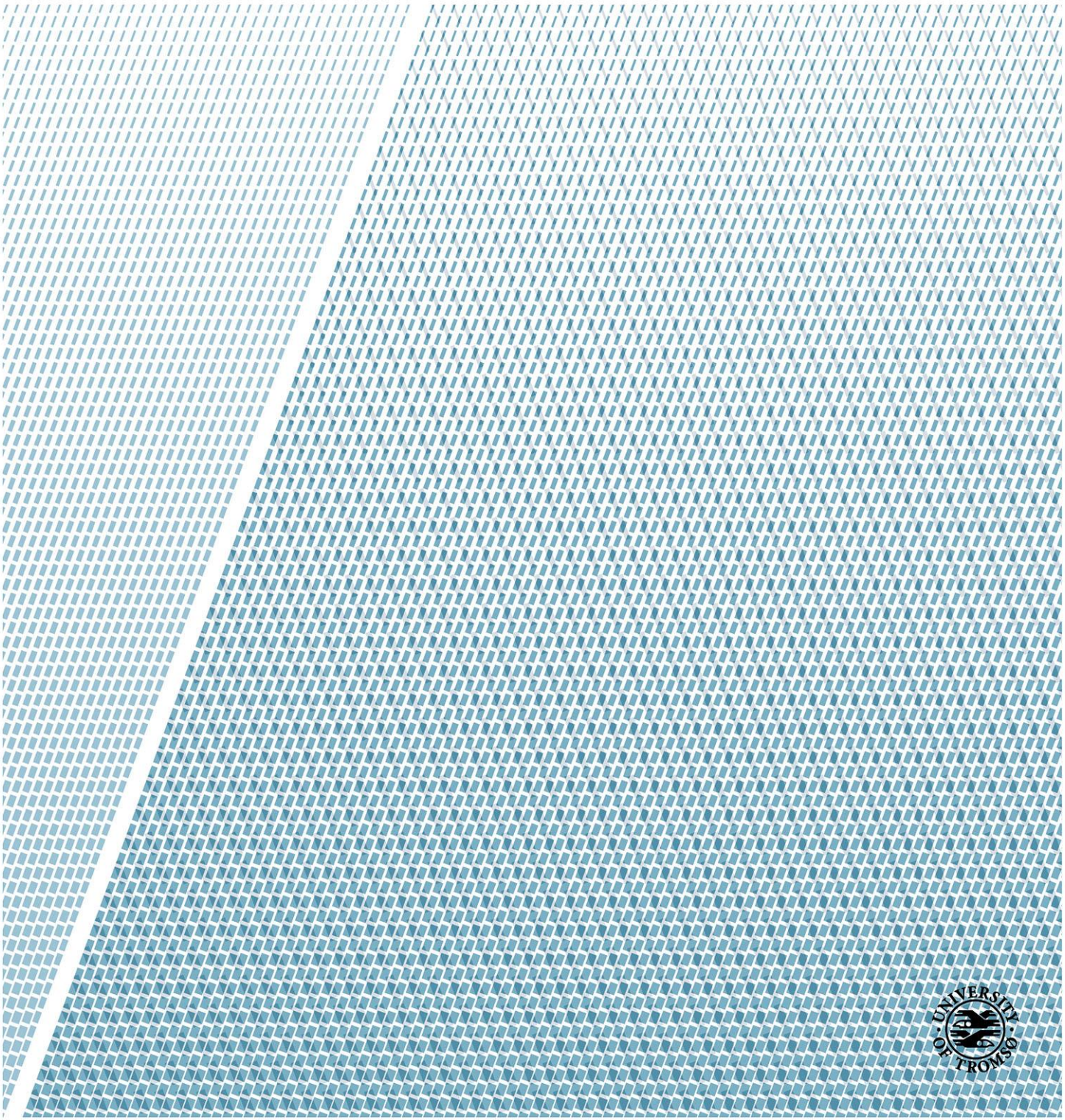
Faculty of Science and Technology, Department of Computer Science

Spark-SPELL: Low-latency Query-based Search for Gene Expression Compendia on Cluster Computers

—

Inge Alexander Raknes

Master thesis in Computer Science INF-3981 – June 2014



1 Abstract

Exploratory analyses are vital to fully realize the potential for scientific discoveries in large-scale biomedical data compendia. Specifically, most biomedical data analyses require a human expert to interactively explore the data to find novel hypotheses or conclusions. However, recent developments in biotechnology instruments are generating Tera-scale datasets. No interactive biomedical data analysis systems scale to such large datasets. We present the design, implementation and optimization of the SPELL biomedical search algorithm on the Spark framework. We demonstrate the scalability and interactive performance of our Spark-SPELL system. In addition, we demonstrate the performance improvements of our optimizations to the SPELL algorithm and the Spark framework.

Acknowledgements

Thanks to Jon Ivar Kristiansen for help with setting up the cluster. We also thank Kai Li, Olga Troyanskaya and Mathew Hibbs for providing the SPELL code. We also thank Wenli Zhang for working with DistSpell.

Contents

List of Algorithms

List of Figures

List of Tables

2 Introduction

Exploratory analysis of large-scale scientific datasets is vital for the advancement of knowledge in many fields. In molecular biology and molecular medicine the recent advances in omics technologies, such as next-generation sequencing machines, has the potential of producing data that provide views of biological processes at different resolutions and conditions, opening a new era in these fields[15]. Specifically, the cost of next-generation sequencing machines and analyses has become low enough that it is practical to purchase and operate these machines in individual labs or clinical care[11]. However, integrated data analysis and exploration is essential to fully realize the potential in the data for understanding the biological context of for example cancer[12], or to enable

personalized[7] and stratified [16] approaches for disease diagnosis and treatment of cancer patients.

Large-scale integrated data analysis and exploration combine data from ten thousands of experiments with millions of samples. The data is generated under diverse conditions, and is noisy. An important challenge when developing a data analysis or exploration method is to ensure that the signal in the aggregated results is not lost in noise. It is therefore necessary to develop algorithms and approaches that take the biological context of the data into account to reduce the number of unknown parameters[12] and hence improve the precision and accuracy of the results. Such analyses methods are often computationally intensive, and typically require a human expert to interpret the results. The analysis must therefore be interactive and hence the results should be computed in a few seconds.

To our knowledge, no existing system for large-scale data analysis enables interactive exploration of heterogeneous and noisy biomedical datasets. This severely limits either the size of the dataset, the methods that can be used for the analysis, or the interactivity and hence usefulness of the analysis. A system for interactive exploration using computationally intensive algorithms will allow development of better analysis approaches, that enable exploratory analyses that can provide researchers novel hypotheses or conclusions.

We describe the design, implementation, and optimization of the SPELL[9] algorithm on the Spark [19, 20, 21] system. SPELL is a search algorithm for functional prediction of large microarray compendia. SPELL is deployed as a web application that provides search and visual exploration of a large yeast (S. cerevisiae) compendium, it is used for search in Wormbase[18] (a large C. Elegans compendium), and it is used to explore data in the HIDRA visualization tool [8]. We believe SPELL is representative for interactive integrated exploration methods for heterogeneous and noisy biomedical data, and that clinical approaches for personalized and stratified diagnosis must perform a similar type of search in a compendia of previous diagnoses. We describe SPELL in detail in section 4.2, and we discuss why scaling and making SPELL searches interactive is difficult.

Spark is a new system for large-scale data analysis that can provide interactive job execution times (less than a second). We believe Spark has multiple advantages for use in interactive exploration of large-scale biomedical compendia. First, compared to for example Hadoop MapReduce jobs [6], Spark jobs can complete in less than a second[19]. Second, Spark is well suited for iterative analysis and therefore well suited for exploratory analysis where a human expert typically refines an analysis by parameter tuning. Third, it enables implementation of application-, and data specific analysis and search algorithms. Spark is well suited for iterative analysis and therefore well suited for exploratory analysis where a human expert typically refines an analysis by parameter tuning.

We describe how we adapted the sequential SPELL for our Spark implementation and how we optimized the SPELL code for efficient parallel execution. We also provide an experimental evaluation of the interactivity, efficiency, and scalability of our implementation. Our results demonstrate that we achieved search latencies of less than X seconds for a large compendia with Y samples.

Our results contribute with an approach for designing interactive search algorithms for distributed execution, performance and memory usage optimizations for biomedical data structures, and lessons learned implementing such

algorithms on Spark.

2.1 Report

This report will be structured as follows:

In section §3 I will explain the architecture of the system. In section §4 I will explain the background for this report which will include descriptions of the involved techologies. In section §5 I will explain the legacy system: the general architecture, some necessary design and implementation details, and some general issues in the implementation. In section §6 I will explain the design of my system and how the different components relate. In section §7 I will explain the implementation of my system, and the general abstractions that were used. In section §8 i will explain different optimizations that were performed on the implemented system in order to make it run faster with larger data sets. In section §9 I will describe the experimental evaluation of the system. In section §11 I will provide the conclusion.

3 Architecture

In this section I will describe a general architecture for implementing a gene search with Spark. The implemented Spell search is an implementation of this architecture. A brief overview can be seen in figure 1.

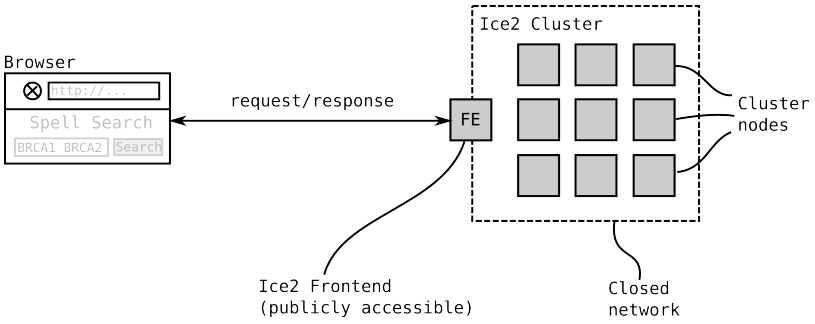


Figure 1: High-level architecture

The general idea is to let a user interact with the search via a web browser and then have a web server that interacts with a search service that runs the search on Spark. This is detailed in figure 2.

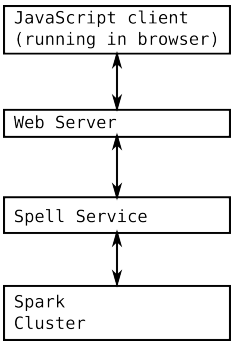


Figure 2: Architecture

JavaScript client The JavaScript web browser client presents the GUI and the search results to the user. It also holds all state related to a user's search session.

Web Server The web server forwards client requests to the Spell Service and returns the results to the client. The web server's state is scoped to a single request.

Spell Service The Spell Service is the driver for the Spark cluster and translates every search query into a Spak job to be run on the cluster. The Spell Service's state is scoped to a single request.

Spark Cluster The Spark cluster does the actual processing of a Search Query in order to derive a search result. The Spark cluster's state include job scheduling, and caching of datasets. This is handled by Spark, and is not a concern for the consuming service other than it may cause additional latency for a request if there is a long job queue.

4 Background

4.1 Compendium

4.1.1 PCL data format

The PCL file format describes a tab delimited file containing gene expression values in a dataset. The first three columns in a PCL file contain a unique ID, a gene name, and the weight of a gene in the dataset. The columns that follows after the three first columns are expression values for the different experiments.

The first row contains column headers and names for the experiments in the dataset. The second row contains weights for the experiments in the dataset.

4.2 The Spell Search algorithm

The Spell search algorithm[9] is an algorithm that can be used to search for genes in a compendium of datasets, given a set of query genes. When the search is performed each dataset will be given a weight and each gene in the

compendium will be given a score. The genes and datasets are then sorted by score and weight respectively and returned to the user in the form of a search result.

4.2.1 Description of the algorithm

The dataset weight is calculated by equation (2), where Q is is the set of query genes and z_{q_i,q_j} is the Fisher z-transformed correlations between the two query genes q_i and q_j in that particular dataset.

$$f(x) = \begin{cases} x^2 & if\ x \geq 1 \\ 0 & otherwise \end{cases} \tag{1}$$

$$w_d = \left(\frac{2}{|Q|\left(|Q|-1\right)} \right) \sum_{i=1}^{|Q|-1} \sum_{j=i+1}^{|Q|} f\left(z_{q_i,q_j}\right) \tag{2}$$

$$s_x = \frac{1}{|Q| \sum_{d \in D} w_d} \sum_{d \in D} \sum_{q \in Q} w_d f\left(z_{x,q}\right) \tag{3}$$

The score for a gene s_x is given by equation (3).

4.2.2 The implementation of the algorithm

One of the challenges with this algorithm is that the provided implementation is more complicated than the algorithm that is described in the paper (summarized in section 4.2.1). In this section I will explain how the provided algorithm can be interpreted.

In addition to the steps given in section 4.2.1 the algorithm collects some extra statistics while calculating the dataset weights and the gene scores. These statistics are then used to clean the final search result: some gene scores and dataset weights are set to either 0 or 1 if certain conditions are met.

Dataset weights Let the sum of the transformed correlations in a dataset d be defined as $u_d = \sum_{i=1}^{|Q|-1} \sum_{j=i+1}^{|Q|} f\left(z_{q_i,q_j}\right)$. Let p_d be the number of query gene pairs that are present in the dataset. If $p_d > 0$ and $s_d > 0$, then the dataset is said to be *weighted*. If less than 3 datasets are *weighted* or $|Q| = 1$ then all dataset weights are set to 1.0. Otherwise the dataset weight is be given by

$$w_d = \begin{cases} u_d/p_d & if\ datase\ is\ weighted \\ 0 & otherwise \end{cases} \tag{4}$$

Note that a dataset weight is not only determined by its own properties in relation with the query genes, but it also depends on how many of the other datasets are being *weighted*.

A relevant fact is that the provided implementation of the Spell algorithm supports more than one function to compare two genes, that is, the parts in the equations that read $f\left(z_{q_i,q_j}\right)$ may be replaced by another function. This functionality is not used by DistSpell, nor is it described in the paper, however in this case equation (4) becomes slightly more complicated: instead of letting w_d being equal to 0 when the dataset is *not weighted*, it will instead be equal

to u_d. See equation (5) for the generalized version, where M is an arbitrary function that relates two genes in a dataset, such that "similar" genes has an higher value.

$$w_d = \begin{cases} u_d/p_d & if\ datase\ is\ weighted \\ u_d & otherwise \end{cases} \tag{5}$$

$$u_d = \sum_{i=1}^{|Q|-1} \sum_{j=i+1}^{|Q|} M(q_i, q_j) \tag{6}$$

This represents how the dataset weights are calculated by the implementation. However when an alternative function is *not* being used, then f as defined in equation (1) guarantees that $u_d \geq 0$ and as such a dataset can only be *not weighted* in the case when $p_d = 0$ in which case u_d will be 0. Therefore equation (4) represents an accurate simplification of the algorithm.

Gene scores Gene scores are calculated by the formula in equation (3), but with similar data cleaning modifications as Dataset weights.

4.3 The Scala programming language

Scala is used for most of the implementation in this project. Scala is a multi-paradigm programming language for the JVM[13]. It supports both functional- and object-oriented programming, and has become a trending language on the JVM. Companies that use Scala include Twitter, LinkedIn, Novell, The Guardian, Xerox, FourSquare, Sony, Siemens, and many others[**?**].

In this section I will explain a small subset of Scala's syntax that is used in the examples in this report, and that might be unfamiliar to programmers coming from other languages. Because Scala is a large programming language, in terms of features, I will only explain the minimum syntax that is required to understand the examples in this report.

As some of the concepts in this section will be explained in the context of Java, it is assumed that the reader has a prior basic understanding of the Java programming language.

4.3.1 Types and generics

Some examples in this report will use Scala's syntax for types and generics. In this section I will explain the minimum amount of syntax surrounding types that is used in this report, and describe some relevant semantics.

Scala is a statically typed programming language. While Scala has type inference, it is sometimes necessary to explicitly annotate types (method arguments is an example). Sometimes type annotations are added for clarity. In this section I will explain some of the syntax concerning types in Scala. Scala's type system is quite complicated and a detailed discussion would not be within the scope of this report.

In Scala type annotations are added after the declaration of a value (or a variable):

```
val x: Int = 5
val x = 5  // type is infered by the compiler
```

Generics are described using the following notation (for a List of Integers):

```
val list: List[Int] = List(1,2,3)
val list = List(1,2,3) // compiler inferes List[Int]
```

Values vs. variables (mutable vs. immutable) In Scala a *val* is an immutable reference (similar to *final* in Java). A *var* is a mutable variable. For example:

```
val list: List[Int] = ... // Immutable reference to a List
var list: List[Int] = ... // Mutable reference to a List
```

In Scala it is idiomatic to use *val* whenever possible. In addition, all collections are immutable by default. Mutable collections are found in a separate package. Thus both Lists in the example above are immutable. Transformations on an immutable collection will result in a new immutable collection.

4.3.2 Closures

Some of the examples that are given later in this report will rely on Scala's support for closures, and anonymous functions. Below, I will explain the different closure syntaxes that I use in the examples in this report.

In Scala anonymous functions are known as *function literals*. I will present a few examples that are equivalent to each other, but use different syntaxes. If we assume that we have a method called *reduce*, on an object named *list* that takes a function called *reducer* as a parameter, and that *reducer* is a function that takes two arguments of type Double, and returns a Double. The type signature for *reduce* could be defined as follows.

```
def reduce(reducer: (Double, Double) => Double): Double
```

The following ways to call *reduce* with a reducer are equivalent:

Algorithm 1 Different ways to declare a function

```
// 1: Explicitly defining a function called 'reducer'
def reducer(lhs: Double, rhs: Double): Double = {
  // Last expression is always returned: no need
  // for an explicit return statement.
  lhs + rhs
}
val result = reduce(reducer)

// 2: Using a function literal
val result = reduce( (lhs, rhs) => lhs + rhs )

// 3: A function literal may span multiple lines if
//    it's enclosed in brackets
val result = reduce { (lhs, rhs) =>
  lhs + rhs
}

// 4: Shorthand for the above: the first '_' means the first
//    argument, the second '_' means the second argument (and so on).
val result = reduce(_+_)
```

In the examples in this report, all four ways of declaring a function will be used.

4.3.3 Functions

In Scala any object that defines a method named apply can have its apply-method invoked with function-syntax on the object. An example can be seen in alg. 2.

Algorithm 2 Functions

```
class StringLength {
  def apply(str: String) = str.length
}

val length = new StringLength
length.apply("Hello") // Standard method invocation.
                      // Result equals 5
length("Hello") // Same as above, but with function syntax
```

4.3.4 Java integration

One of the main benefits of Scala is that it integrates with Java code and libraries. For example, Java classes can be instantiated and extended from Scala and Java interfaces can be implemented with Scala classes. This makes it possible reuse legacy Java code in a Scala project.

4.4 Libraries used

In this section I will give a short description of the libraries that were used to create the implementation.

Twitter Finagle[1] Twitter Finagle is a general purposes RPC library developed at Twitter. It supports multiple protocols including HTTP and Thrift. Finagle is a Scala library, but it can also be used from Java. Finagle is used to implement RPC between the web server and the back-end.

Apache Thrift[2] Apache Thrift is an interface definition language and communications protocol. Objects serialized with Thrift are given a compact representation and can be parsed in multiple programming languages. A property of thrift is that changes to the schema may be backwards compatible. Thrift is used for RPC with Finagle. It is also used to store expression datasets in HDFS.

Scrooge[3] Scrooge is a Thrift parser/generator for Scala. It generates Scala classes from Thrift IDL and it can also automatically generate service interfaces for Finagle. Scrooge is used to generate all the Thrift-related objects in this project. Scrooge integrates with SBT (Simple Build Tool) and runs automatically every time the project is built.

Twitter Bijection[4] Library to manage conversions between related objects/-types. It uses type classes[5] to provide a simple, type safe syntax for conversions. This library is used to integrate the legacy code with the new code. It is also used for serialization of objects and to create multiple similar Finagle services while honoring the DRY principle.

Scallop[6] Command line argument parser for Scala. It s used to parse command line arguments for the various CLI tools that were made during the development of this project.

Apache Commons Math[7] Math library from Apache. Its Pearson correlation feature is used when implementing the custom Spell implementation (section 7.5).

ScalaTest[8] Testing framework for Scala. All new tests were written with this framework.

[1] `http://twitter.github.io/finagle/`
[2] `http://thrift.apache.org/`
[3] `http://twitter.github.io/scrooge/`
[4] `https://github.com/twitter/bijection`
[5] Type classes is an advanced concept in Scala and is outside the scope of this report. For an introduction to type classes and how they are implemented in Scala see [14].
[6] `https://github.com/scallop/scallop`
[7] `http://commons.apache.org/proper/commons-math/`
[8] `http://www.scalatest.org/`

ScalaMock[9] A mocking library for Scala. Is used for testing.

JUnit[10] A unit testing framework for Java. Is used to test the legacy code.

4.5 Spark

Spark is used extensively in this assignment, and the backend implementation runs on top of Spark. In this section I will explain the most important aspects of Spark that is related to this assignment.

Spark[20] is a distributed system that is designed for parallel operations where parts of the data set can be reused in between computations. Spark is distributed and offers an alternative computation model to MapReduce. Spark builds on top of Mesos which is "a platform for fine-grained resource sharing in the data center (Hindman [10])".

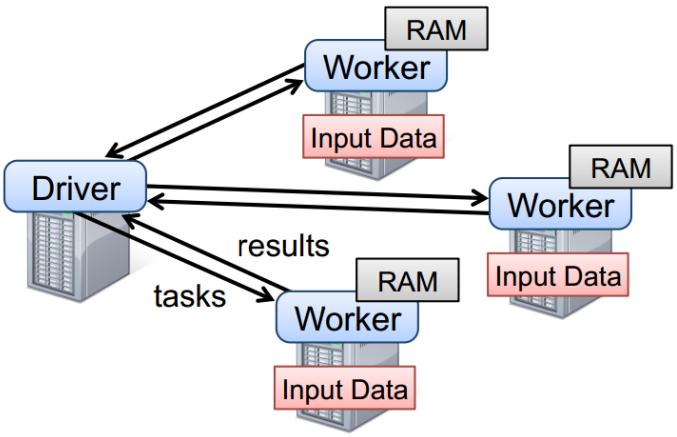


Figure 3: Spark runtime. Illustration is taken from [19].

4.5.1 Resilient Distributed Dataset

The most important abstraction in Spark is the RDD (Resilient Distributed Dataset). An RDD is an immutable fault tolerant distributed shared memory abstraction that is based on coarse-grained transformations rather than fine-grained updates[19]. An RDD can be explicitly cached in-memory or persisted to disk (or both).

An RDD is deterministically created by transforming another RDD or by reading data from stable storage. These operations are called *transformations*. Examples of transformations include *map* and *filter*. All transformations are lazy operations that define a new RDD. There is a second type of operation on an RDD that is called an *action*. An action is an operation that launch a computation in order to return a value to the user (program) or that writes data to external storage. Transformations and actions are further described in section 4.5.2.

[9] `http://scalamock.org/`
[10] `http://junit.org/`

An RDD is partitioned into several partitions. These partitions may be persisted (cached) on worker nodes. When an RDD is transformed the transformations that are used to build a dataset (its lineage) are logged rather than the actual data. This means that if some of the partitions of an RDD is lost due to a failure they may be recomputed quickly rather than using costly replication.

For a deeper understanding of RDD i suggest reading [19].

4.5.2 Computational Abstractions

As mentioned in 4.5.1 Spark has a set of computational abstractions. These have the capability to define transformations or actions on an RDD. In this section I will describe some of the most important computational abstractions:

map Maps every element in an RDD by applying a user supplied function to each element.

filter Filters elements in an RDD when given a predicate

reduce Combines all elements in an RDD when given a user supplied operator

fold Similar to reduce, except that it requires a user to supply a "zero" or "neutral" value.

collect Retrieves every element of an RDD and returns it as an array on the driver node.

A complete list of methods supported on an RDD can be found in the Spark API documentation[3].

4.5.3 Tasks

In Spark a task is the basic unit of computation. When a transformation is performed the scheduler creates a task to process each partition. Tasks can be scheduled based on data locality. Because RDDs are immutable the system can run backup tasks in order to mitigate slow nodes (stragglers). Sending a task to a worker requires sending closures to them. To achieve this Spark relies on the fact that Scala closures are Java objects and can be serialized using Java serialization.

4.5.4 Broadcast Variable

A broadcast variable is an immutable shared variable that is cached on all the worker nodes[20]. This is useful for lookup-tables or meta-data. In practice this means that broadcast variables can be referenced from within a closure instead of the actual variables themselves. This means that tasks can be made smaller before they are serialized and sent to the workers. In *Tuning Spark*[5], under the headline *Broadcasting Large Variables*, there is a statement that "in general tasks larger than about 20 KB are probably worth optimizing (Apache [5])". In comparison, for the largest synthetic dataset evaluated in this thesis, the lookup tables for IDs and statistics are in total 12.5MB.

4.5.5 Accumulators

Accumulators "are variables that workers can only "add" to using an associative operation, and that only the driver can read (...) Accumulators can be defined for any type that has an "add" operation and a "zero" value (Zaharia [20])". Accumulators are not used for the implementation of the Spell Search algorithm.

4.5.6 Streams

Spark has support for processing data that is arriving in real time. It does this by "performing a series of batch computations on small time intervals (Zaharia [21])". This functionality is not used in my implementation.

4.5.7 Shark SQL

A system that is related to Spark is Shark SQL[17]. Shark provides an SQL interface to Spark RDDs.

4.6 Profilers

Profilers are an invaluable tool when debugging performance issues on the JVM. During the development of Spark-SPELL a few profilers were evaluated. JProfiler an YourKit were the two profilers that were mostly used, although VisualVM was also tested.

VisualVM VisualVM is a free profiler that comes with the Oracle JDK. It supports common functionality like locating hot-spots and monitoring memory consumption.

JProfiler JProfiler is a commercial profiler for the JVM. It integrates with IntelliJ (IDE) and it provides a user-friendly interface for a programmer. An interesting feature of JProfiler is that it has the ability to view aggregated incoming references to objects. This makes it easy to identify objects that are referencing large amounts of data.

YourKit YourKit, like JProfiler, integrates with IntelliJ.

5 Legacy Spell

In this section I will give a brief overview of the legacy Spell code and its context. Much of this code has been reused in my project and some of the decisions that were made in the design and architecture of my project are based on the details of this inherited code. I will explain in brief the architecture and the design of the legacy application, some of the fundamental abstractions, and some issues that motivates some of the decisions that were made in my project which are described later in this report.

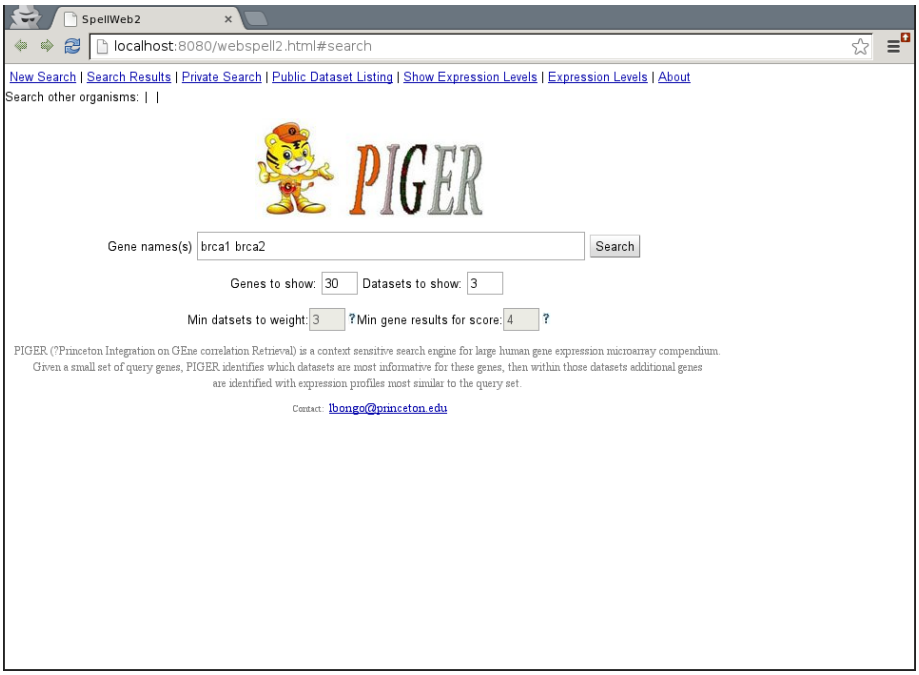


Figure 4: SpellWeb

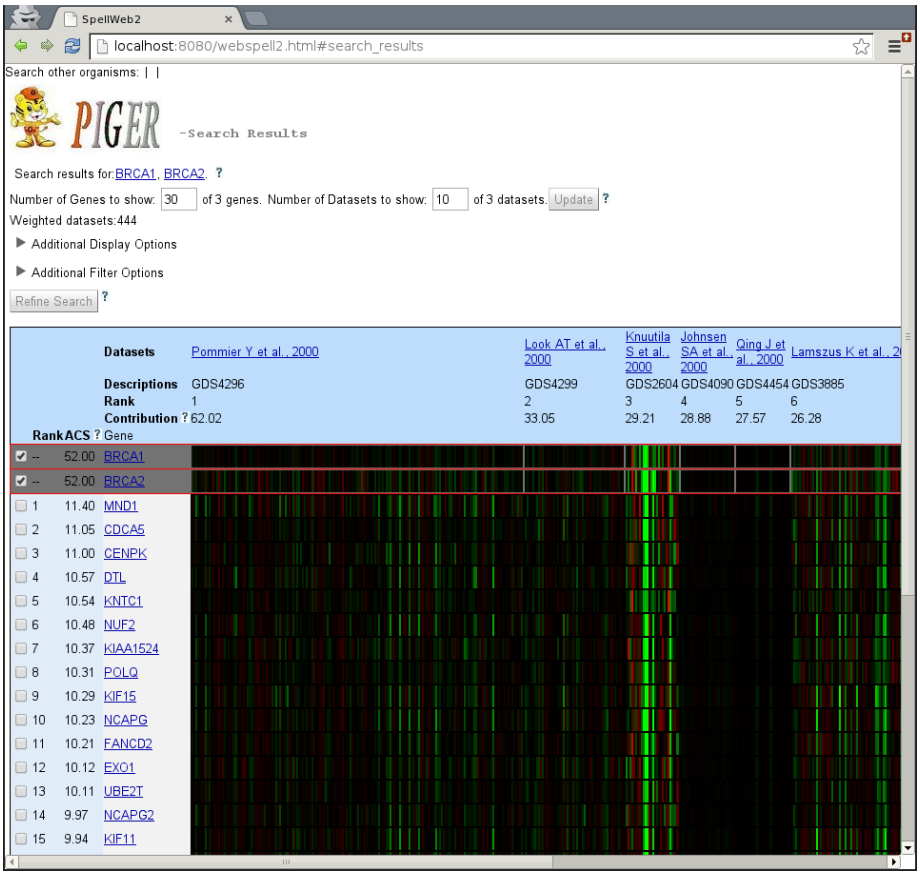


Figure 5: SpellWeb search result, showing genes, datasets and heat maps. Note that the search result is provided by the new back-end that was developed as part of this thesis.

5.1 Legacy architecture and design

In this section I will describe the legacy architecture and give a high-level overview of its design and runtime-behavior. This is important as it provides the context of the borrowed code and some abstractions that were key to my design are based on this architecture.

At the highest level (figure 6). There is a client browser application that communicates with a web front-end which in turn communicates with a distributed spell search service (aka. DistSpell) (figure 7) . The web front-end also communicates with a GoTerm finder that is used to do meta-data search on Go Terms. This chapter will focus on the web front-end and DistSpell.

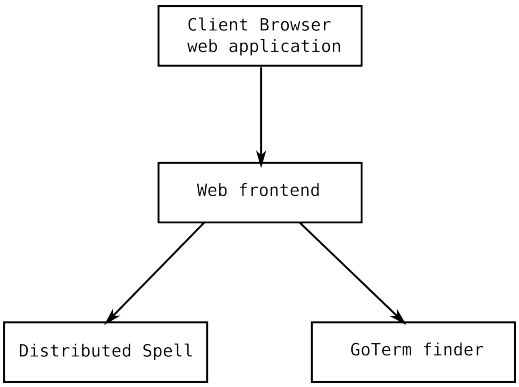


Figure 6: Legacy architecture

The distributed Spell service is implemented as a master/worker architecture, where the master partitions the datasets for the workers, schedules searches and talks to the client. The master communicates with the workers and the client via a custom binary message passing protocol.

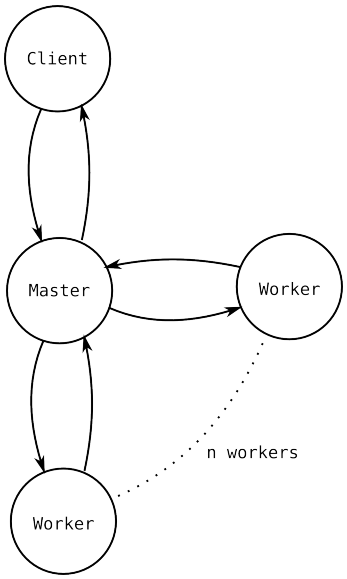


Figure 7: Legacy DistSpell master/worker architecture

DistSpell is parallelized on two levels; first datasets are distributed among the worker nodes. Second, the worker nodes are multi-threaded.

5.1.1 The messaging protocol

While this report will not go into detail about the message passing protocol, there are a few observations that are important for explaining the reasoning behind my own design. The messaging protocol is organized into two main parts: client/master communication and master/worker communication. In total there

were 24 messages, including heart-beat messages and other administrative messages (i.e. for loading a dataset on a worker node). While I will not go into detail about every message, there were a few that were inspirational to the design of the Spark-SPELL implementation. These have been summarized below

Client/master

Search request Request to initiate a search. Contains dataset IDs (int array), query gene IDs (int array), organism ID (string)

Search response Response to a search request. Contains number of *weighted* datasets (int), dataset weights (array of tuples of dataset ID: Int, weight: Float), gene scores (tuples of gene ID: Int, score: Float).

Master/worker

Partial search request Request to initiate a partial search. Contains dataset IDs (int array), query gene IDs (int array).

Partial search response Response to a partial search request. Contains dataset weights (int array), partial scores (an ordered array of tuples (partial score: Float, gene weight: Float, dataset count: Int). The tuples are ordered after a global list of gene IDs that are known to both the master and the workers)

5.2 Legacy code overview

In this section I will explain some of the basic structure of the legacy code. This is relevant in order to explain how I reused some of the old code, and why I choose to use certain abstractions. In sections where I explain my abstractions and how I were able to optimize my implementation based on these I will refer back to this section.

5.2.1 Legacy code dependencies

The reasoning behind some of the abstractions I describe later in this report is influenced by the structure of the legacy system. Reusable components were recognized, factored out and re-used in the new implementation. The separations of the reusable components were heavily influenced by the dependencies in the legacy code. A coarse overview of the code dependencies can be seen in figure 8. The finer grained structure for the different components is generally more dense.

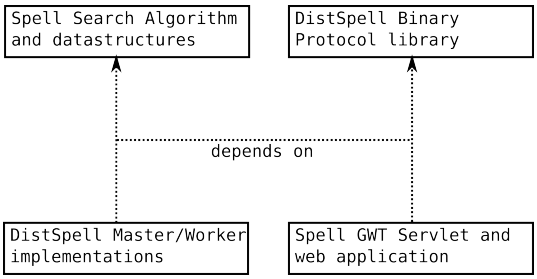


Figure 8: Legacy code dependencies

5.2.2 Web Frontend

The web front-end is divided into three parts:

GWT browser client This is the code that implements the GUI of the application. It communicates with the GWT Servlet in order to perform a search on the back-end. This is Java code that is compiled to JavaScript by the GWT compiler.

GWT Servlet This is the server code of the application. In the original implementation it contains code to communicate with DistSpell.

Shared code Interfaces and classes that are shared the client and the Servlet are defined here.

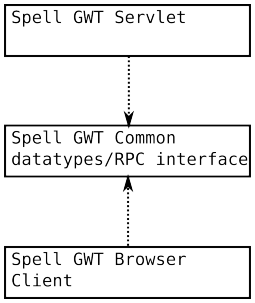


Figure 9: GWT code dependencies

As can be seen from 9, both the Servlet and the client depend on the common code, but no code depends directly on the Servlet or the client. This means that the Servlet can be completely reimplemented in order to integrate with SparkSPELL. The Spell client does not need to be modified.

5.3 Issues

In this section I will explain some issues with the legacy code. This section will motivate some of the decisions made in my own project, including the creation of a complete wrapper for the Spell Searcher algorithm. Most of the focus is

going to be on the Spell Searcher algorithm where most of the issues were. These issues were carefully avoided in the new implementation.

5.3.1 Hard to understand

There were many issues in the legacy code that made the code difficult to understand.

The model described in the legacy search algorithm was more complex than the model that was needed to perform a search. For example, the code had classes representing homologies, different organisms and gene families in addition to the required concepts like genes and data sets.

The legacy search algorithm had several mutable public fields that were used by other parts of the code. This made it difficult to determine when state would be consistent and when it would be safe to access fields.

There was also a large amount of duplication. Parts of the code had been copied to multiple places with only small differences that had been made to the code at each place. A consequence of the duplicated code was methods that were spanning several hundred lines. This made the code much larger than it needed to be and it made it more difficult to understand the implementation.

5.3.2 Hard to use

Instantiating and performing a search using the old code is complicated and non-obvious. The required objects had several file system dependencies as well as dependencies on each other that needed to be resolved upon instantiation. When the search result were ready, different parts of the result was represented as sequences which had significant ordering based on other fields or sequences.

In addition to these issues, parts of the parallel algorithm was embedded in DistSpell in ways which made them impossible to reuse without reimplementing.

The collection of these issues made it a difficult and error-prone process to reuse the old code in a new project.

5.3.3 Hard to change

The legacy search algorithm did not come with a good test suite. This made it difficult to safely change or refactor. Because of this, no large changes were made to the legacy search algorithm. Instead a wrapper was built around it and only minor changes, like changing the visibility of fields, were done in order to accommodate the wrapper. The abstractions chosen for the wrapper was inspired by the DistSpell message passing protocol, as well as the input file formats and are described in section 7.2.

5.3.4 Hard to test

In order to make it practical to refactor it was considered to write unit tests for the legacy search algorithm. These plans were discarded at an early stage for the following reasons:

No proper specification While there was a paper describing the SPELL algorithm the actual implementation was different from the description in the paper; certain data cleaning steps had been added and these were intimately embedded in the rest of the code. Individual methods were large, had many branches and accessed multiple public fields. This made it difficult to determine what was the "correct behavior" of the algorithm. The only practical way to evaluate whether the algorithm was being used correctly was to compare example queries with example results that had been generated with DistSpell.

No clean separation of components As mentioned in 5.3.1 the legacy search algorithm had many classes with public fields which were accessed from other classes. A consequence of this is that it would be very difficult to test components in isolation.

Instantiation relies on outside world As mentioned in 5.3.2 instantiating the legacy searcher algorithm required file system dependencies. While this is a problem that had to be worked around in order to run on Spark by finding alternative ways to initialize the objects, I mention them here because it also makes the code more difficult to test.

While no unit-tests were written for the legacy code, tests were made for the code that wraps it. The tests compares output of the algorithm with examples that are known to be correct.

5.3.5 Performance

In the legacy search implementation, there are objects that are referencing several hash maps that are used to store different views of the same data. The most significant consequence of this is wasted memory. Later in this report I will show that memory consumption is the most significant bottleneck of Spark-SPELL (section §9) and that the objects needed to represent a search can be stored more than 4 times as efficiently with a custom implementation of the search algorithm (section 9.2.3).

5.4 Lessons learned

5.4.1 Don't keep the abstractions. Write a wrapper.

In order to abstract away the problems with the legacy search algorithm it was decided to write a wrapper for it and to keep all direct references to the legacy code out of the new code base. Provided a good abstraction, this would enable simple client code without having to refactor the legacy code. It would also ensure that all code that interacts with the legacy algorithm is kept in a single place, thereby making it easier to debug and understand. A working abstraction was inspired by the message passing protocol of DistSpell and the file formats (section 7.2).

Abstracting away the legacy Spell algorithm turned out to be one of the most important decisions in this project. It enabled the re-implementation of the algorithm without requiring any changes to the Spark implementation. Since tests were written against the interface of the wrapper, the same test code could also be used to test the new implementation.

5.4.2 Learn from DistSpell

The messaging protocol define a functioning abstraction for using the underlying search algorithm As stated above, one of the first challenges that i encountered was to find a good abstraction for defining a search. The fact that the message passing protocol had already been successfully used in DistSpell meant that the abstraction that it defines had already been demonstrated to work.

Most of the messages are organized in a request/reply pattern This suggests that most of the Spell functionality can be invoked using simple method invocation where the request-messages define the arguments and the reply-messages define the return types.

Performing a search does not modify any persistent state While this is a more subtle point, it greatly simplifies the reasoning about how to use the algorithm. For example a Spark RDD is assumed to be immutable, and as such it is required that any data structure that is stored inside an RDD is also immutable. State-less code naturally maps to those constraints.

6 Design

In this section I will explain the overall design of Spark-SPELL.

6.1 Server Design

In this section I will describe the high-level design of the services that implements the back-end system. A detailed overview of the architecture can be seen in figure 10. The front-end server and the Spark-SPELL service will run on the cluster front-end machine, whereas the spark workers will run on the cluster compute nodes.

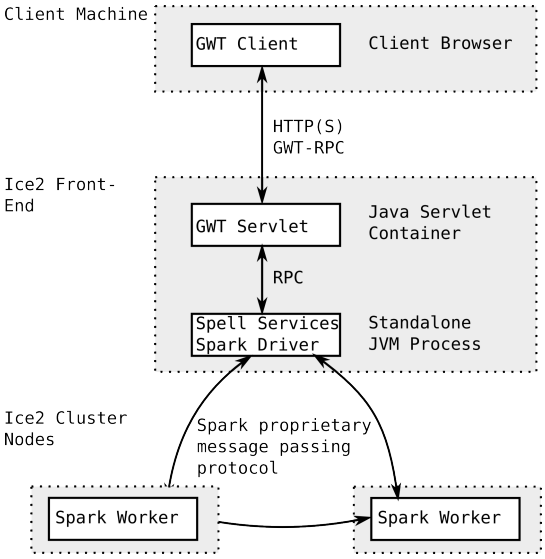


Figure 10: Detailed architecture

The server process that runs the Spell services can be seen in figure 11. A detailed description is given below.

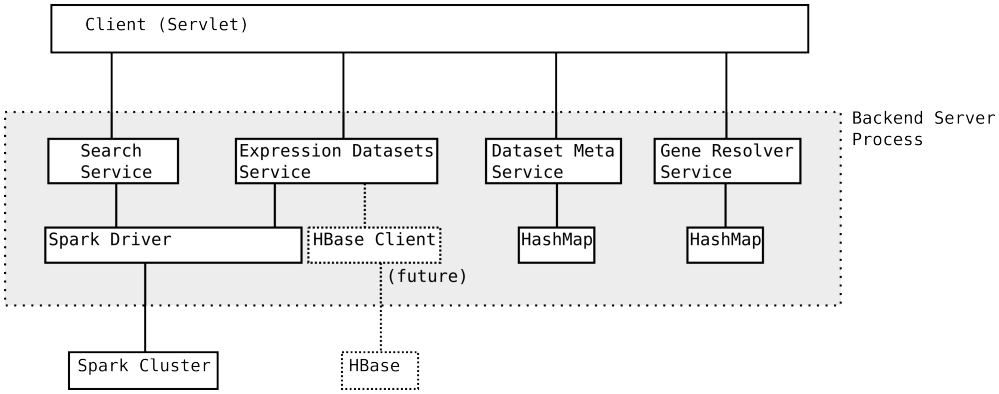


Figure 11: Backend server

Search Service This service provides the core search functionality and is the most important part of this project. Specifically it takes a set of gene IDs, performs a search, and returns a set of ranked gene IDs and ranked dataset IDs. When the result is returned it is up to the client to resolve the IDs into human-readable names that can be displayed to the user. The GWT Servlet will do this by calling the Gene Resolver Service. Other clients, such as the automated Search Service benchmark, do not care about the human-readable names and will not resolve them after obtaining the search result.

Expression Datasets Service The Expression Datasets Service lets the client obtain the expression values for specific genes in specific datasets. This is

used by the GWT Servlet in order to let the GWT browser client render- and display the expression values. Values returned by this service will never change, and can be safely cached by the client.

Dataset Meta Service Locates meta data about a dataset so that it can be displayed by the client. Meta data includes publication name, year, authors, description, etc.. Values will never change, and can be safely cached by the client.

Gene Resolver Service Translates between gene IDs and human readable gene names. This service may be called twice during a search; first to resolve the gene names specified in the query into gene IDs, then a second time when the search return in order to translate the gene IDs returned in the search result to human-readable gene names. Results will never change, and can be safely cached by the client.

6.2 Spark Searcher Design

The Spell algorithm can be parallelized by partitioning the the input data by individual datasets. A partial search is then performed within each dataset, in parallel, and the partial search results are then cleaned and merged into a final result.

The datasets are stored in a sequence file on HDFS where the key is a dataset identifier and the value is a textual PCL representation of the data (later optimized to use Thrift). The sequence file is represented as s Spark RDD. Upon a search request, the driver node will order the workers to map each partition of the RDD to a partial search result and finally, it will merge all the partial search results into a single result and return it to the client.

When the datasets are read from HDFS and have been parsed, they will be cached locally on the worker nodes in order to speed up subsequent searches. The caching is an essential part of the design because the time it takes to read data from HDFS and parse it is too long for interactive analysis.

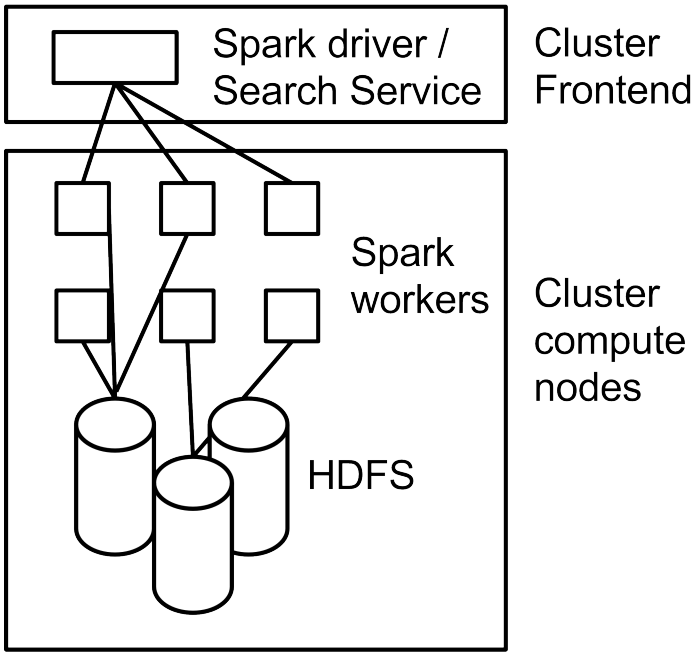


Figure 12: Spark Searcher

7 Implementation

In this section I will explain the implementation challenges that were encountered during the development of this project. I Will also discuss some key decisions that enabled the optimizations that are explained in section §8.

A wrapper was built for the old SPELL algorithm. The purpose of this wrapper was to hide the complexities of the original implementation behind a simple, testable interface. The wrapper was tested against examples of known query/result pairs in order to evaluate its correctness. This wrapper was also convenient for generating examples when I later on created my own optimized implementation of the SPELL algorithm.

7.1 Code organization

In this section I will explain the overall organization of the source code.

7.1.1 Spell back-end server and client library

The main components of the back-end server and their dependencies are shown in figure 13.

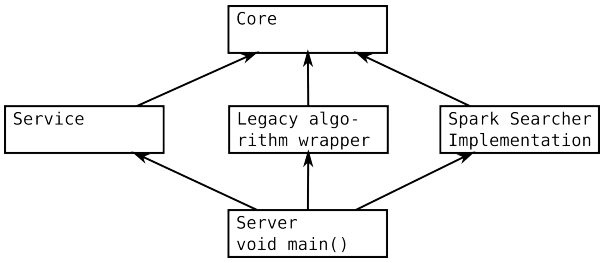


Figure 13: Source code components and their dependencies

Core The core contains an interface for the search algorithm (section 7.2). It contains all the classes that are necessary to describe the data dependencies of a Spell search, and it also contains an alternative implementation of the Spell search algorithm.

Service The service component contains all the parts that are necessary in order to make the spell search available to clients. It contains RPC client libraries as well as the services that are necessary to perform a search (as shown in fig. 11 on page 24). Note that the service component itself does not directly depend on a Spark implementation nor any specific searcher algorithm, instead these concrete implementations are provided upon instantiation. The service component also contains code to simplify the integration with the Spell GWT front-end. The service component is shared between the front-end server and the back-end.

Legacy algorithm wrapper This is a wrapper for the legacy Spell algorithm. It implements the interface that is described in section 7.2 and thereby simplifies how the original algorithm can be used from within the Spark implementation.

Spark Searcher Implementation This is the component that defines the Spark algorithm for the Spell search. The concrete Spell searcher algorithm implementation is provided upon instantiation. This algorithm will in turn be applied to the data sets that are stored on the cluster upon subsequent search requests (section 7.3).

Server void main() The main entry point for the server will instantiate and wire all the components together upon server startup.

The components Core, Service, Legacy Wrapper, and Spark Searcher are organized both as individual libraries and as sub-projects. The Server process is a client of these libraries. The sub-project dependencies are managed explicitly by SBT(Simple Build Tool)[11].

[11] In the project source tree, the components are named slightly differently. *Core* is known as *spell-search* and *Server* (named SparkSearchService) is a part of *cli-tools* which contains a collection of tools that are run from the command line and uses the Spell libraries.

7.2 Searcher Algorithm Interface

As described in section 5.3 the given abstractions for the search algorithm were difficult to use and test. In this section I will explain how I designed a different interface for using the search algorithm and how I created a wrapper for the legacy implementation. The main goal was to hide the most important problems with the legacy code and to create a simple and safe usage pattern for the algorithm that could be used from within a Spark implementation. I also wanted to make it possible to change the implementation of the algorithm in order to both facilitate testing and my own optimized implementation.

As described in 5.4.2 it was initially a challenge to find a suitable abstraction for using the algorithm. I also outline some options for defining such an abstraction. In this section I will describe how this abstraction is defined and materialized in the code.

As suggested in 5.4.2 I use the messaging protocol as inspiration.

Two implementations of the search algorithm interface were developed: a wrapper that wraps the search algorithm from DistSpell and a from-scratch complete re-implementation that is more efficient in terms of memory.

7.2.1 Functional

The observations described in 5.1.1 suggests an interface that is modeled as a function that maps a search query to a search result, and that is free of side-effects.

7.2.2 Abstract data types

In the legacy implementation queries and results were represented by explicit concrete data types. In my implementation more abstract representations were chosen in order to ease programming and facilitate optimizations.

7.2.3 High-level Searcher

The desired level of abstraction is a simple function that maps a query to a search result, as seen in alg. 3.

Algorithm 3 Basic Searcher Usage

```
val search: Searcher = ...
val result: SearchResult = search(query)
```

A definition for a searcher that cover all the client use-cases was inspired by the DistSpell master/client message passing protocol and can be seen in alg. 4. GeneID and DatasetID are (at runtime) represented by standard Java integers.

Algorithm 4 Basic Searcher Definition

```
type Searcher = SearchQuery => SearchResult

case class SearchQuery(
  genes: Seq[GeneId],
  organism: OrganismId,
  overrideDatasets: Option[Seq[DatasetId]]
)

case class SearchResult(
  geneScores: Map[GeneId, Float],
  datasetWeights: Map[DatasetId, Float]
)
```

The abstraction level is slightly higher than the legacy implementation of the sequential Spell algorithm. Abstract maps are chosen instead of relying on the implicit ordering of arrays, and a SearchQuery class is chosen instead of a long list of method parameters. This has the following benefits:

- Maps are easy and safe to use as opposed to the implicitly ordered arrays that were used in the legacy code.
- A Map in Scala is abstract and therefore a lot of flexibility is retained with respect to its concrete representation. In addition to allow for the same optimizations that were achieved with ordered arrays in the legacy code it also allows for alternative optimizations without requiring a change to the code that is using the maps. This is discussed in section 8.2.
- A Map in Scala is immutable. That means that we avoid the uncertainties that were present in the legacy code about which parts of the program modifies the data. It also means that we can safely pass them by reference to different parts of the program without having to make defensive copies or worry about synchronization or data races. This is a big benefit in a multi-threaded program.

7.2.4 Low-level Partial Searcher

A drawback of the Searcher definition described in section 7.2.3 is that it leaves concurrency as an implementation detail. This means that by itself it does not solve the stated goal of providing a simple and safe usage pattern that can be used from within a Spark implementation. In order to solve this an interface at a lower abstraction level is required.

DistSpell shows how the algorithm can be parallelized by partitioning the datasets onto different workers and the DistSpell master/worker message passing protocol shows which data is needed for each step of the computation. An interface modeled after the message passing protocol can be seen in alg. 5. The abstraction level was raised slightly from the level of the protocol for the same reasons that were stated in 7.2.3.

Algorithm 5 Partial Searcher Definition

```
type PartialSearcher = SearchQuery => PartialSearchResult

case class PartialSearchResult(
  geneScores: Map[GeneId, PartialScore],
  datasetWeights: Map[DatasetId, Float]
)

case class PartialScore(
  score: Float,
  weight: Float,
  numDatasets: Int
)
```

Given the abstraction in alg. 5 we can map a query to a result by performing the steps in alg. 6. Note that each step is a transformation of the data produced in the previous step and that no state is modified in any part of the algorithm. The definition of merge, clean and convert is left to the user of the interface.

Algorithm 6 Partial Searcher Usage

```
val query = ...

val partialSearchers: List[PartialSearcher] = ...

val partialResults: List[PartialSearchResult] =
  partialSearchers.map(searcher => searcher(query))

val merged: PartialSearchResult = merge(partialResults)
val cleaned: PartialSearchResult = clean(merged)

val result: SearchResult = convert(cleaned)
```

How the Searcher and the Partial Searcher abstractions relates is visible in fig. 14 on page 33.

7.2.5 Instantiation

As described in 5.3 one of the issues with the legacy implementation was that it was complicated to correctly instantiate all the objects that were required to perform a search. In the Spark implementation the searcher would have to be instantiated from elements in an RDD. In unit tests the searcher would have to be instantiated from in-memory examples or from files. This suggests that the instantiation of the searcher should be abstracted in a way that allows for all these use-cases in a simple manner. Ideally we want an abstraction that is not only agnostic about the purpose of the searcher, but one that is also agnostic about the searcher implementation itself. If we can achieve the latter then we

can also provide our own (optimized) searcher implementation without changing the other parts of the code.

The chosen strategy for for abstracting the instantiation of a searcher was to recognize all its data dependencies and then define a function that maps the data dependencies to a searcher. A type signature for such a function can be seen in alg. 7.

Algorithm 7 Partial Searcher Factory

```
type PartialSearcherFactory =
  SearcherDependencies => PartialSearcher
```

The legacy implementation was analyzed and the following dependencies were recognized: organism id, set of genes (with ID's), expression datasets and expression dataset statistics (alg. 8). The in-memory representation of these dependencies were inspired by the data files that were used by the legacy system. The data files were chosen as a reference because they define an abstraction for instantiating a searcher that had already been demonstrated to work.

Algorithm 8 Searcher dependencies

```
case class SearcherDependencies(
  id: OrganismId,
  genes: Set[Gene],
  datasets: Map[DatasetId, ExpressionDataset],
  stats: Map[DatasetId, ExpressionDatasetStats]
)
```

The PCL datasets are represented by the ExpressionDataset class which is designed to be close to the PCL format. That is essentially just a sequence of records containing basic information about a gene and its expression values. This can be seen in alg. 9.

Algorithm 9 Expression dataset

```
case class ExpressionDataset(records: Seq[ExpressionRecord])
case class ExpressionRecord(
  yorf: String,
  name: String,
  weight: Float,
  experimentData: Array[Float]
)
```

The process of instantiating a partial searcher with explanation of each step can be seen in alg. 10.

Algorithm 10 Partial Searcher Instantiation

```
val partialSearcherFactory = {
  // A concrete implemetation
  // of PartialSearcherFactory.
  // This can be a wrapper for the legacy
  // code or it can be a custom implementation.
  // The concrete implementation is selected
  // on program startup and is then
  // injected from a higher scope.
  ...
}

val searcherDependencies = {
  // The data dependencies for a searcher. This can
  // be instantiated from:
  //    - elements in an RDD
  //    - files on hard drive
  //    - examples from unit test
  // Since SearcherDependencies is pure data / serializable
  // it can be safely persisted in an RDD.
  ...
}

// The searcher can be instantiated when the user performs a search.
// If the Searcher implementation is serializeable then it can also
// be persisted in an RDD between invocations.
val search = partialSearcherFactory(searcherDependencies)

val partialResult = search(query)
```

7.3 Spark Searcher Algorithm

In this section I will explain the basic workings of the Spark implementation of the algorithm. The explanation in this section builds on the abstractions described in 7.2.

As mentioned in section 4.5.4 Spark has an abstraction called a *broadcast variable* that can be used to broadcast data to all the worker nodes in a cluster. In the Spark Searcher implementation the expression datasets are read from HDFS and stored in an RDD, while all the meta-data (gene names, dataset IDs, etc.) are broadcast using a *broadcast variable*.

7.3.1 Preparing a search

As I described in section 7.2.5 a SearcherDependencies object is passed to a PartialSearcherFactory in order to create a new PartialSearcher. In order to instantiate a SearcherDependencies object, and thereby a PartialSearcher, a

map operation is performed on the RDD that contains the expression datasets. This *map* operation combines the expression data sets that are stored in the RDD with the meta data that is stored in the *broadcast variable* in order to instantiate a SearcherDependencies object. From this object a PartialSearcher is instantiated by the method described in 7.2.5.

By looking at how a PartialSearcher is instantiated we can immediately see a possible trade-off with regards to caching: Should the expression data sets be cached, or is it better to cache the fully instantiated PartialSearchers? As mentioned in 7.2.5 both are possible. The performance implications of these two approaches are described in section §8 and section §9.

7.3.2 Performing a search

As described in 7.2.4 a PartialSearcher is simply a function that maps a Query to a PartialSearchResult. As described in 7.3.1 the PartialSearcher objects are stored in an RDD. When a search is performed the query is broadcast to the worker nodes using a *broadcast variable*. We obtain an RDD of PartialSearchResult by mapping the RDD with the partial searchers and apply every partial searcher to the given query. The resulting RDD is then reduced to a single PartialSearchResult which is cleaned and converted to a SearchResult that can be consumed by the user. This process is analogous to the algorithm described in alg. 6 and the end result is equivalent to directly applying a Searcher to a Query. This is illustrated in 14.

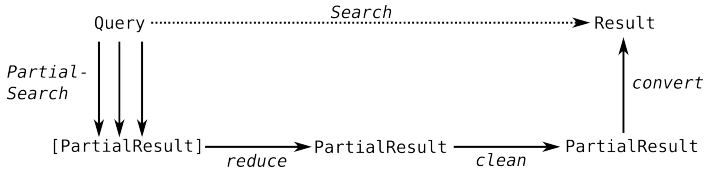


Figure 14: Distributed searcher algorithm

The *map* and *reduce* operations happens on Spark, whereas the *clean* and *convert* happens in the client library. An interesting property of the chosen abstractions is that all the steps that are performed on Spark amounts to mapping a Query to a PartialSearchResult. That is functionally the same as what a PartialSearcher does. This makes it possible to merge the partial results from multiple Spark clusters in the same way as they are merged in a single Spark job. An actual implementation of a system that uses multiple clusters to perform a search is left for future work.

7.4 Interaction with legacy code

7.4.1 Spell searcher algorithm

As mentioned in section 5.3, the legacy Spell searcher algorithm had a some usability issues: both instantiating the required objects and performing a search required the programmer to set up some complicated instantiation code. The legacy implementation assumed file system dependencies (that wouldn't be available within a Spark context) and search queries had to be built by navigating a complicated object graph. As these issues were assumed to be difficult

to debug in a distributed setting a wrapper was created that implements the interface that is described in section 7.2. This made it easy to use the search classes from within the Spark algorithm, and it also made it easy to test my assumptions about the Spell algorithm without involving a distributed system.

7.4.2 Spell front-end GWT application

The Spark Spell implementation was integrated with the existing front-end GWT application. The Servlet part of the existing GWT application was mostly removed and replaced with code that would use the new Spell RPC client library (explained in section 7.1) to interact with the backend. Helper functions were written in Scala in order to help transform between the data structures that were used by the Servlet and the Spell service. The client part of the GWT application was only slightly modified by adding a few log-statements in order to understand how it communicates with the Servlet.

7.5 Custom SPELL implementation

During the implementation phase of the project I made my own implementation of the Spell algorithm that is simplified and optimized towards low memory usage.

The custom Spell implementation implements what is referred to as PartialSearcher in section 7.2.4. The custom PartialSearcher is designed to be cached in an RDD (as described in section 7.2.5) with a relatively low memory overhead compared to the original Spell implementation. It achieves a reduced memory footprint by only referencing data that is required in order to perform a search. In this implementation an expression dataset is represented by the class given in alg. 11. In addition to the fields shown in alg. 11 it also contains methods for calculating the dataset weight when given a set of query genes.

Algorithm 11 Dataset

```
class Dataset(
  val id: DatasetId,
  val expressionValues: Map[GeneId, Array[Double]],
  val stats: ExpressionDatasetStats
) extends Serializable {
  // Methods to calculate dataset weights and
  // partial scores for genes goes here
}
```

Algorithm 12 PrototypePartialSearcher

```
class PrototypePartialSearcher(datasets: Seq[Dataset])
    extends PartialSearcher with Serializable {
  override def apply(query: SearchQuery) = ...
}
```

A PartialSearcherFactory, as described in section 7.2.5, instantiates the PrototypePartialSearcher and the Dataset objects when given an instance of SearcherDependencies. This makes it easy to swap the implementations of the algorithm such that the new and the old implementations can be compared.

The legacy code is still used to compute the dataset statistics, but the actual search is performed by the new implementation. The new implementation also uses Apache Commons Math for calculating the Pearson's correlations instead of re-implementing the algorithm like the old code does.

The correctness of the custom implementation was evaluated by running example queries in both the new implementation and the legacy implementation and then compare the results. Performance comparisons of the implementations can be seen in 9.2.3 on page 42.

8 Optimization

8.1 Memory overhead

The most important point to optimize was the memory usage. The most important memory issues are space and allocations.

8.1.1 Memory footprint

The most important optimization is the caching of the data sets. In order for as much data as possible to fit into the cache it needs to be represented in an efficient manner. The basic techniques were used in order to achieve this:

Use arrays of primitives where possible The largest single category of data are the expression values. These can be expressed efficiently as arrays of floats.

Use interning of strings The PCL format contains string identifiers for every gene. While there are many records containing gene identifiers, only about 70,000 of them are unique. Therefore interning gene IDs saves a large amount of memory.

8.1.2 Garbage collection

Some of the most challenging performance issues encountered when implementing were related to garbage collector pauses. The most significant kind of pauses were caused by full GC collections. This in turn would cause the jobs to fail as an OutOfMemoryException is throw when to much time is spent collecting garbage[1]. As stated in *Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning*:

> The parallel collector will throw an OutOfMemoryError if too much time is being spent in garbage collection: if more than 98% of the total time is spent in garbage collection and less than 2% of the heap is recovered, an OutOfMemoryError will be thrown. This feature is designed to prevent applications from running for an

> extended period of time while making little or no progress because the heap is too small. (Oracle [1])

There are a few techniques to solve this problem, many of which are outlined in *Tuning Spark* [5] and *Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning* [1].

In order to diagnose GC issues on Spark there are a few tools that are particularly useful: the Spark web console has a *Stage* view that lets the operator inspect the executed stages and their tasks. This view includes information about individual task execution time as well as how much of the time was spent on garbage collection. Another useful tool is the JVM built in reporting of GC details. These can be enabled by starting the JVM with the following flags −verbose:gc −XX:+PrintGCDetails −XX:+PrintGCTimeStamps. Spark can pass flags to the JVM via the environment variable named SPARK_JAVA_OPTS as described in [5]. Output from the JVM GC details is available on *stdout* on the different nodes which is accessible via the Spark web console.

The most efficient solutions to GC issues for this implementation were to increase the number of Spark partitions and to reduce the total amount of allocated objects (by storing data in arrays). Changing GC parameters were also found to be efficient at first, but as other optimizations were made the benefit of GC tuning became less apparent: the time it took to perform the first search, which includes the time it takes to read and parse the datasets and fill the cache, were found to be 25% faster when survivor spaces were increased, but the time it took to perform subsequent searches were not notably different.

8.2 Custom data structures

8.2.1 PartialScoresMap

When a partial search result (see alg. 5) is instantiated from a search within a subset of the datasets, then the largest portion of the partial result is going to be the map between gene IDs and their partial scores. The type of this map is Map[GeneId, PartialScore]. The map implementations that are found in the standard library are general-purpose and not optimized for the particular data types and usage patterns of this map. Since this particular map is used quite intensively it makes sense to make some optimizations. In order to understand how it can be optimized we need to start by looking at its usage patterns. The map is used in the following ways:

- It is constructed by a large number of items for each dataset (thousands).
- A large number of maps are merged when merging the partial results of a search. When the same gene id is found in two maps their partial scores are added.
- The map is serialized before it is sent to the driver node.
- It is iterated a final time when the PartialSearchResult is converted to a SearchResult.

The following *non*-uses can also be noted:

- There are no random look-ups

- The map is fully constructed before any values are read from it (it is immutable)

These requirements suggest that we can implement a map that is optimized for fast merge and low memory overhead without having to optimize for random look-ups or inserts after the map has been instantiated.

As explained in section 8.1.1 it is more space efficient to store arrays of primitives when possible. As explained in section 7.2.4 GeneID is represented by an Integer and PartialScore is an object with 3 fields: two floats and one integer.

The problem of implementing the optimized map is broken down into the following parts:

- Field-to-array optimization for the partial scores. Each PartialScore object is represented by an index into three arrays that contain its fields.

- Mapping each gene id to the correct index in the arrays.

Constructing a PartialScore object from the arrays when given an index is trivial. Therefore I will instead focus on how the IDs are mapped to array indices. A relevant observation is that since gene IDs are represented by integers they have a defined ordering and can be sorted. Because of this we can represent contiguous intervals of gene IDs as tuples of min/max values. We can then calculate an index based on an ID by calculating the number of elements represented by all the intervals with a lower ID, and then add the offset into the interval that contains the given ID. An example of this approach is illustrated in figure 15.

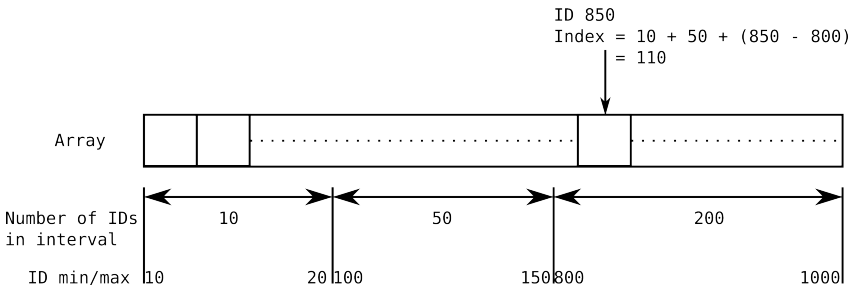


Figure 15: Calculating an array index for a given ID

The minimum and maximum values can be stored in two integer arrays. The time to locate an index given an ID would be proportional to the number of contiguous ID intervals (i.e. relatively slow compared to a standard Map), however it is very fast to iterate the values in sorted order and the use of arrays of primitives makes it very efficient as opposed to trees or hash tables pointing to a large amount of objects. This is used to implement fast merge of multiple maps: by always performing the next iteration on the map that will yield the lowest ID it becomes possible to iterate multiple maps in sorted order and thus build a new map.

8.2.2 Evaluation

The optimized map was evaluated with the following experiments (seen in figure 16 and table 1):

Unoptimized In the unoptimized version the default map from Scala's standard library is used to hold the data. In order to merge two maps, a and b, the key sets of a and b are merged and then every key is looked up in a and b. If a key exists in both maps they are merged using a resolver function, otherwise if the key exists in only one of the maps then the corresponding value is kept unmodified.

Using this method, maps are merged two-by-two as part of a *fold* operation on an RDD.

Optimized The custom data structure is used to hold the map. For each dataset a partial search result is produced. These are then merged two-by-two as part of a *fold*, using the custom merge algorithm described in section 8.2.1.

Builder-1 The Spark implementation of the algorithm is modified such that a single partial searcher can hold a whole RDD partition of datasets (instead of just one). The custom prototype implementation of the Spell algorithm is modified such that instead of returning a partial scores map for each dataset the algorithm is set to accept a builder. This results in fewer maps that needs to be merged (1 pr. partition).

A builder class is created in order to instantiate the optimized map. Merge in this case happens automatically upon *insert*. Since lookups (as described in section 8.2.1) are relatively slow we cache the ID-to-array-index lookups during the build phase by using the Scala standard library's IntMap.

Builder-2 In this experiment the builder class has been slightly modified. Instead of using an IntMap to cache the array indices the builder will allocate temporary arrays for a larger sequence of gene IDs, some of which will not be contained in the final map. The temporary arrays are defined such that the index into the arrays can be calculated simply by adding an offset to the ID. Upon *build* all the unused values are discarded together with the temporary arrays.

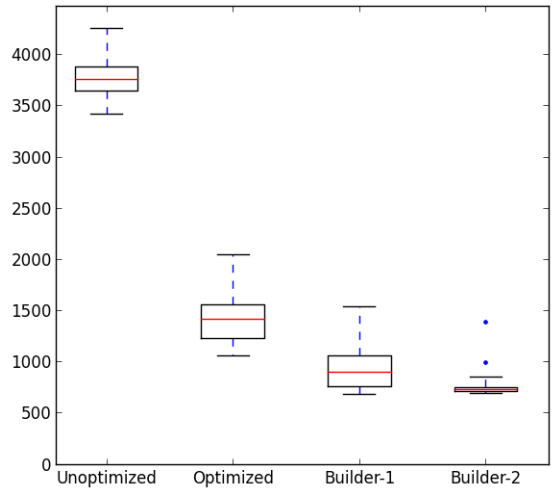


Figure 16: Different merge strategies

Experiment	Mean	Median	Std.dev.
Unoptimized	3762	3756	153
Optimized	1404	1415	231
Builder-1	931	901	205
Builder-2	737	732	44

Table 1: Different merge strategies

8.3 Tuning Spark

In this section I will explain some tuning parameters that are available to a Spark job. These parameters are explained in the *Tuning Spark[5]* part of the Spark documentation. What I will focus on in this section is how these parameters relates to the Spell implementation.

8.3.1 Cache size tuning

When persisting data in an RDD it is possible to select the *cache size*, or the amount of data that should be cached in memory of the JVM. As explained in *Tuning Spark[5]* it is possible to reduce GC pressure by reducing the amount of data that is cached in memory.

8.3.2 Number of partitions

In Spark the number of partitions of an RDD can be set by the programmer. As explained in section 4.5.3 each partition is mapped to a task, and as a consequence the number of partitions specifies the maximum amount of parallelism for a job. The cost of using many partitions is that they need to be

merged at the master node[12]. This increases the sequential part of the algorithm. A consequence of having to few partitions is that a task may fail with an OutOfMemoryException as a result of not having enough heap space.

8.3.3 RDD persistence level

As explained in section 4.5.1 and [19, 1], Spark lets the programmer decide explicitly when to cache data that is stored in an RDD in between computations. In Spark a generalization of this concept is described as *persistence*, for which there are several *persistence levels*. Here I will explain the persistence levels that are supported by Spark:

MEMORY_ONLY This is the default persistence level when caching data in spark. It will store the data in-memory as deserialized Java objects. If the amount of data in the RDD is larger than the amount of available memory then some partitions will not be cached; instead they will be recomputed. For the Spell algorithm it is quite expensive to recompute data that has been dropped from an RDD: PCL datasets will have to be read from HDFS and parsed before any search can be performed. For the GEO dataset filling the cache takes about 50 seconds, whereas subsequent searches take about 500ms.

MEMORY_AND_DISK This persistence level is similar to *MEMORY_ONLY* in that it stores deserialized Java objects in memory. However instead of dropping partitions when it runs out of memory it will spill data to disk instead. For the Spell algorithm spilling to disk is much cheaper than recomputing lost partitions.

MEMORY_ONLY_SER, MEMORY_AND_DISK_SER Same as *MEMORY_ONLY* and *MEMORY_AND_DISK* except that the Java objects that are stored in-memory are serialized. This leads to a smaller memory-footprint at the cost of CPU cycles. It is also possible to specify that the objects should be serialized with the *Kryo* serializer, and that the serialized objects should be compressed[2, 5]. For the Spell algorithm serializing the data in-memory did not seem to provide a performance benefit.

DISK_ONLY All of the RDD's data is written to disk.

MEMORY_ONLY_2, etc. Same as the levels above, but the data is replicated across multiple cluster nodes. This can be used for fast fault recovery.

9 Evaluation

In this section I will explain the evaluation of my implementation. I will explain how the correctness was evaluated and I will explain how the implementation performs under different conditions.

[12] This can be seen in the Spark Source code on GitHub[4] *(core/src/main/scala/org/apache/spark/rdd/RDD.scala* method *fold*, line 693-701)

9.1 Correctness

The correctness of the algorithm was evaluated by comparing search results against a set of examples that were known to be correct. The examples consists of search queries and their correct result. A challenge with comparing search results is that values are represented by floating point numbers. The rounding of these numbers are not deterministic in a distributed system, thus it becomes necessary to compare

When Spark-SPELL was run with the legacy search algorithm the results came out to be equal to the examples.

9.2 Performance and scalability

In this section I will explain the performance evaluation of the implementation. Human datasets from GEO Omnibus is used as a reference for the performance benchmarks. The raw data set consists of 9.0 GB of gzipped SOFT files, which was converted to PCL files and stored in an HDFS sequence file. This amounts to 1335 datasets. The number of genes that are known to the system is 24410.

The sequence file uses zlib compression and has a final size of 2.91GB.

The evaluation was run on a 9 node cluster with the following hardware:

- 8 core Intel Xeon E5-1620 3.6GHz
- 32 GB RAM
- 2 hard drives, 2 TB each.
- A single 1 Gbps Ethernet card connected to a shared switch

9.2.1 Construction of synthetic datasets

The scalability of the system was tested with synthetic datasets. These were generated by creating multiple copies of the original data set. Noise was then applied to the expression values of the synthetic data. The synthetic data was then shuffled and written into a new sequence file.

Synthetic datasets were generated with different sizes, rangeng from 20 to 100 times the size of the original GEO dataset. For the rest of this chapter I will refer to the synthetic datasets by their multiplication factor; that is, a *20x dataset* will refer to 20 copies of the GEO Omnibus dataset, *40x* will refer to 40 copies and so on.

9.2.2 Presentation of results

All timings in the evaluation are measured in milliseconds. Box plots have been used. The box extends from the upper to the lower quartiles of the measured values. The red line shows the median. The length of the whiskers extend to the most extreme data point within the data range[13].

For tables containing mean values of measured timings, the first search (i.e. the time it takes to fill the cache) has not been included. The time it takes to

[13]For a full description of the box-plot, see `http://matplotlib.org/1.3.1/api/pyplot_api.html#matplotlib.pyplot.boxplot`

fill the cache is mentioned separately when it is relevant. The time it takes to fill the cache varies mostly with the size of the dataset.

For tables containing memory and disk consumption the values represent the aggregated sums on all nodes.

9.2.3 Comparison of partial searcher implementations

In this subsection I will compare the legacy partial searcher implementation with my own prototype, using the GEO dataset. I will compare both the time it takes to perform a search, as well as the total memory consumed by the different implementations. I will also compare different caching strategies, like caching the datasets vs. caching the fully instantiated partial searchers (section 7.2.5). The times to perform a search can be seen in figure 17 and table 2. The memory consumption for the different caching strategies can be seen in table 3.

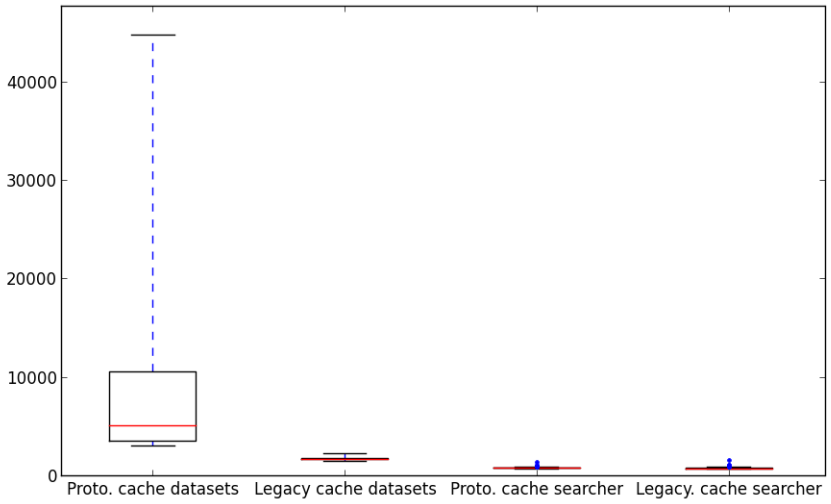


Figure 17: Comparison of partial searcher implementations. The number of samples for each configuration is 1000. The number of RDD partitions is 72.

We can see from the first plot in figure 17 that when we are instantiating a prototype searcher from an expression dataset there is a long tail in the performance measurements. This is likely due to garbage collection, as indicated by GC debugging output. Fast instantiation was not considered an important feature for the prototype searcher as it consumes less memory than the PCL data objects (see table 3), and as such, this was not optimized for.

From the plot in figure 17 we can make the following observations:

- Instantiating a searcher is relatively expensive compared to performing a search. This means that it's more performant to cache the fully instantiated searchers rather than the raw expression data sets.
- The legacy implementation and the prototype perform similarly when instantiated, with the legacy implementation being marginally faster.

- The prototype searcher is slow to instantiate. The prototype searcher was designed to be cached in an RDD and thus fast instantiation was not optimized for.

Experiment	Mean	Median	Std.dev.
Proto. cache datasets	8633	5054	8570
Legacy cache datasets	1697	1685	94
Proto. cache searcher	789	784	31
Legacy. cache searcher	728	715	50

Table 2: Comparison of partial searcher implementations. The number of samples for each configuration is 1000.

While the prototype implementation is not much faster than the legacy implementation it consumes much less memory. Comparison of the memory consumptions under different configurations can be seen in table 3. As we shall see later in this evaluation, memory consumption becomes the main performance issue for larger datasets, which in turn gives the prototype searcher a big advantage over the legacy implementation. *Because of this, all experiments described later in this report was performed with the prototype searcher.*

Configuration	Size in memory	Size on disk
Cached datasets	10.3 GB	0 B
Cached prototype searcher	4.8 GB	0 B
Cached legacy searcher	20.7 GB	0 B

Table 3: Memory consumption for the GEO dataset as reported by the Spark web console. All data sizes represents the aggregated sum across all nodes.

9.2.4 Node scalability

In this section I will evaluate how well the system scales with respect to the number of worker nodes. This evaluation is needed in order to show whether it's possible to reduce search latency by adding more nodes.

A 20x dataset was chosen, as this is the largest of the generated dataset that fits in memory on the 9 node cluster without serializing the searcher objects. This means that it is the largest dataset for which we can expect fast, interactive queries. The results can be seen in figure 19.

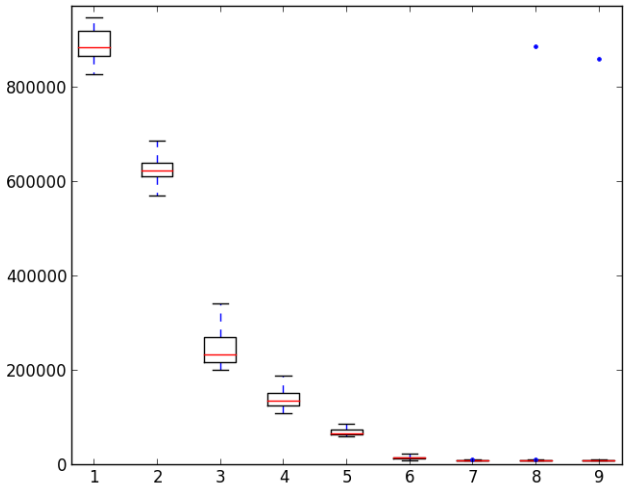


Figure 18: Node scalability using a 20x dataset. Number of nodes along the x-axis, time along the y-axis. Uses 360 partitions and RDD persistence level is *memory-and-disk*.

Number of nodes	Size in memory	Size on disk
1	- GB	- GB
2	28.3 GB	45.5 GB
3	42.4 GB	35.7 GB
4	56.3 GB	26.0 GB
5	70.4 GB	16.2 GB
6	84.0 GB	6.7 GB
7	93.3 GB	0.27 GB
8	93.7 GB	0 B
9	93.7 GB	0 B

Table 4: Node scalability using a 20x dataset. All reported sizes represents the aggregated sum across all active nodes.

As we can see the computation scales well up until 6-7 nodes where it stops scaling. As we can see in table 4 the running times is closely related to the amount of data that is stored in memory vs. disk. While this seems like a plausible explanation for why a 20x dataset stops scaling, we were also interested in seeing how well the Spark-SPELL scales with a larger dataset that does not fit in memory on the cluster. A 40x dataset does not fit in memory of the cluster. However, we still run into scalability issues at around 8 nodes (figure 19).

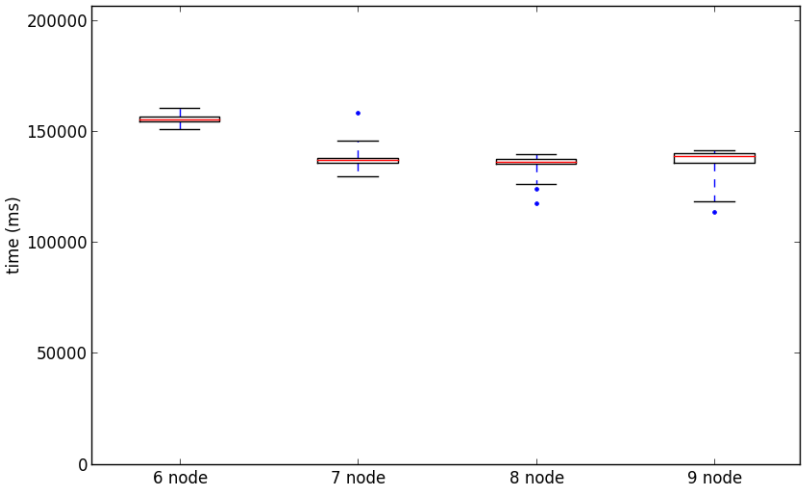


Figure 19: Node scalability using a 40x dataset. Number of nodes along the x-axis, time along the y-axis. Uses 360 partitions and RDD persistence level is *memory-and-disk*.

There are two different explanations for why this might be the case

Stragglers When benchmarking large datasets it was observed via the Spark console that some tasks suffered from long GC pauses. These would typically occur at 1 node at a time, for 1 individual search at a time. As a long GC pause would likely increase the sequential part of the algorithm it is believed that this is the main cause of the observed effect. As we shall see in section 9.2.5 performance is lost as the JVM heap fills up, which is consistent with the observation that scalability suffers, even with larger datasets.

Bottleneck at the driver This is the most obvious possibility. There are two things that could be causing such a bottleneck: communications overhead (all aggregated partial results are sent to the driver) and inefficient merging of results at the driver node. While the communications overhead would increase with the number of partitions, the amount of transferred data is nearly constant with respect to the number of datasets as whole partitions are merged at a time. Therefore it seems reasonable that by adding more datasets one would observe better node scalability as more computation would be required at the worker nodes. This, however, was not the case, thus making it less likely that the bottleneck is at the driver.

9.2.5 Dataset scalability

In this section I will explain how well the algorithm scales with respect to the number of datasets. This is relevant in order to show how performance characteristics change as more data is added to the system.

The different parameters, like number of RDD partition and persistence levels are tuned to make each dataset as fast as possible. For example, the 20x dataset is using 360 partitions and *memory+disk* persistence, whereas the 100x dataset is using 7200 partitions and *memory-and-disk-serialized* persistence. Evaluation of different persistence levels can be seen in section 9.2.6.

One of the most difficult aspects with scaling the algorithm is that different settings are better suited for different dataset size. For example, if we try to run the 100x dataset with 360 partitions, the computation will fail with an OutOfMemoryException. On the other hand, if we try to run the 20x dataset with 7200 partitions we will slow down the final *merge* of the datasets. In general, it seems that for the Spark-SPELL algorithm fewer partitions are better, and avoiding serialization is better in the case where the deserialized objects fit in memory (this last point is discussed in 9.2.6). The best achieved performance for the different dataset sizes can be seen in 20. The amount of data cached in memory and on disk can be seen in 5.

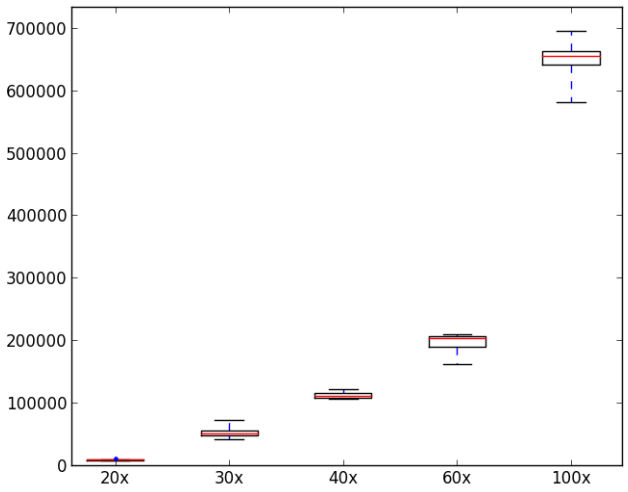


Figure 20: Dataset scalability

As we can see from figure 20 performance decrease rapidly as we increase the dataset size. We can see that performing a search on the 30x dataset is almost 7 times slower than performing the same search on the 20x dataset. This is most likely due to a change in persistence level parameter, where the 30x dataset needs to be serialized in order to fit into DRAM [14].

If we compare the 30x and the 40x dataset, we see that running a search on the 40x dataset is almost twice as slow as on the 30x dataset. If we study 5 we can see two things that are different from when running the 30x dataset:

- Some of the data is now cached on local disk instead of in DRAM.

[14] For a discussion about what happens if we don't change the persistence level, see section 9.2.6

- We are now using 124.9 GB of total JVM heap (as opposed to 97.8)

As we shall see in section 9.2.6 storing such small amounts of data on disk does not make a big difference, as compared to deserializing the data. Another observation is that GC overhead becomes significant when the JVM heap is running close to full. The *Tuning Spark[5]* guide suggests to reduce the cache size as a possible solution, as it may speed up task execution times. However, due to time constraints, this possibility was not further investigated.

Experiment	Size in memory	Size on disk
20x	93.7 GB	0 B
30x	97.8 GB	0 B
40x	124.9 GB	5.4 GB
60x	127.1 GB	69.3 GB
100x	127.0 GB	200.9 GB

Table 5: Dataset scalability. All data sizes represents the aggregated sum across all nodes.

To search the 100x dataset takes about three times as much as the time to search a 60x dataset. This is explained by the fact that the dataset is almost twice as big, and that a larger portion of the dataset is stored on disk (ref. table 5).

9.2.6 RDD Persistence levels and serialization

As explained in section 8.3.3 one of the tunable parameters of Spark is the RDD's persistence level. In this section I will explain how different persistence levels affects performance of the Spark-SPELL algorithm. The goal of this evaluation is to discover which persistence level is best for running Spark-SPELL.

In the first series of experiments the dataset size was chosen such that the serialized objects would fit in memory on the cluster nodes whereas the deserialized objects would spill to disk. Since memory access is much faster that disk access one might intuitively believe that it is always preferable to serialize data in memory rather than storing some of the data on disk. As we can see in figure 21, this is not always the case.

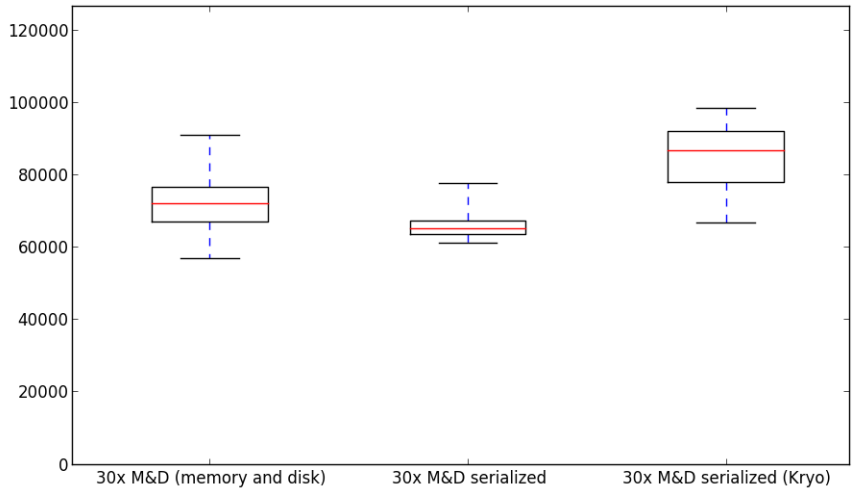


Figure 21: RDD persistence levels when using a 30x dataset

While determining the exact reason for this behavior is difficult, it seems reasonable that since the data that is stored in memory is much larger than the data that is stored on disk the deserialization overhead becomes larger than the extra IO overhead. To reduce the CPU overhead of Java serialization Spark also supports serialization with Kryo. We did not, however, manage to get better performance with Kryo serialization. The disk vs memory consumption can be seen in table 7. Another factor is that the OS block cache may be a source of error for the reported memory and disk usage; while Spark is reporting to store a certain amount of data on disk it might be the case that the OS is caching some of that data in memory. In order to investigate this further a second series of experiments were run with twice the amount of data (figure 22). An experiment where all data was stored on disk was also performed.

Experiment	Mean	Median	Std.dev.
Reference	742	735	33
30x M&D (memory and disk)	71802	71780	6734
30x M&D serialized	65589	65162	2901
30x M&D serialized (Kryo)	84785	86568	8676

Table 6: Timings – RDD persistence levels when using a 30x dataset

Experiment	Size in memory	Size on disk
30x M&D (memory and disk)	125.9 GB	9.9 GB
30x M&D serialized	97.8 GB	0 B
30x M&D serialized (Kryo)	91.3 GB	0 B

Table 7: Cache size – RDD persistence levels when using a 30x dataset. All data sizes represents the aggregated sum across all nodes.

As can be seen in figure 22 and table 8, even with twice the amount of data the difference between caching serialized and deserialized objects in memory is not significant. The only significant change is in the standard deviations of the measurements, where the deserialized objects come out as favorable with a smaller standard deviation. Another interesting observation is that storing the data on disk is only 1/3 more expensive than storing the same data in memory. The small difference may in part be caused by the OS block cache, or it may also in part be caused by reduced GC pressure as a result of not caching any data[15].

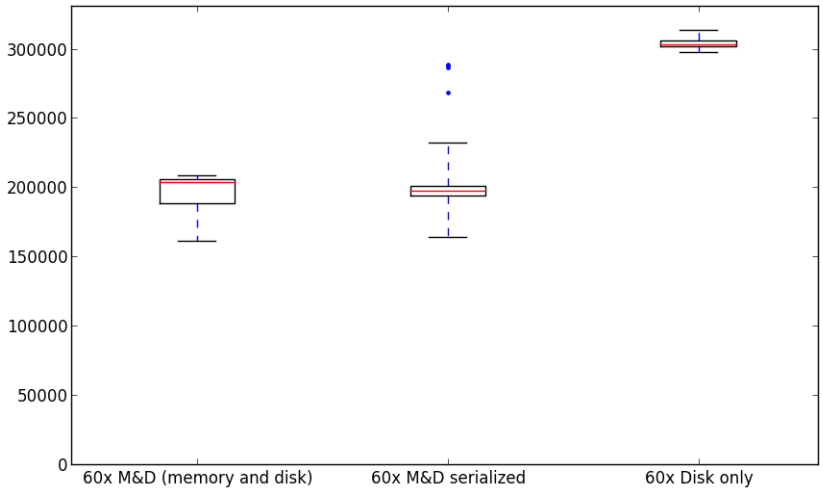


Figure 22: RDD persistence levels when using a 60x dataset

Experiment	Size in memory	Size on disk
60x M&D (memory and disk)	127.1 GB	109.2 GB
60x M&D serialized	127.1 GB	69.3 GB
60x Disk only	0 B	196.5 GB

Table 8: Cache size – RDD persistence levels when using a 60x dataset. All data sizes represents the aggregated sum across all nodes.

Another possible explanation for the similar results is that scalability issues in the implementation trumps some of the performance benefits that would otherwise be expected from serializing the data, thereby making the performance difference seem relatively smaller. This could seem consistent with the scalability-issues that are discovered in section 9.2.4. However, since the 40GB difference between the serialized and the deserialized (60x) dataset is comparable- and larger than some of the data size differences that are discussed

[15]Ref. the *Spark Tuning[5]* guide advice on reducing cache size in order to speed up task execution times.

in section 9.2.5 this seems unlikely, as the algorithm is demonstrated to scale with respect to dataset size.

For datasets that don't fit in memory when deserialized, *Memory and disk serialized* is the better choice for Spark-SPELL. For datasets that do fit in memory, *Memory and disk* is faster.

10 Future work

10.1 Distributed resources

As described in section 7.3 it is possible to use the created abstractions to perform a search on multiple Spark clusters. More generally, it is possible to use the same abstractions to search multiple resources that can be either local or remote. This would make it possible for Spell to search both public and private data within a single search.

10.2 Parallelization with respect to genes

The current implementation is parallelized with respect to datasets. Since there are many genes in each dataset the result is that every worker needs to send partial scores about every gene to the driver node. We believe that it is possible to parallelize the algorithm with respect to genes in each dataset. In this case each workers would be responsible for a set of genes instead of a set of datasets. It is likely that this would reduce communications overhead.

11 Conclusion

Spark seems to be a well suited for implementing a biological search infrastructure. Its ability to load large datasets from HDFS and cache them in memory for future computations makes it a fitting technology for implementing a biological search infrastructure. Its simple programming model also makes it a useful tool for rapid prototyping of such a service.

The scalability of Spark-SPELL depends on the size of the dataset. For small compendia of only a few GB there is no benefit from running on multiple nodes. However, for compendia exceeding 100GB in size massive improvements can be seen from using multiple nodes in a cluster.

The main performance issues when implementing Spark-SPELL were caused by the garbage collector. Long GC runs would cause computations to be delayed and increase the latency experienced by the user. Numerous optimizations were applied in order to mitigate this problem. While some optimizations, like field-to-array optimizations, provided a benefit on all dataset sizes, the benefit of other optimizations, like increasing the number of RDD partitions, were highly dependent on the dataset size.

Legacy code written for biological analysis may be poorly tested and implemented. This motivates building custom abstractions for interacting with the code, as well as reimplementing certain algorithms.

Spark-SPELL enables low-latency query-based search for large-scale gene expression compendia on cluster computers

References

[1] Java se 6 hotspot[tm] virtual machine garbage collection tuning. `http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html`, 2012.

[2] Spark configuration. `http://spark.apache.org/docs/0.9.0/configuration.html`, 2014.

[3] Spark scaladoc. `http://spark.apache.org/docs/0.9.0/api/core/index.html`, 2014.

[4] Spark source code (tag v0.9.1, github). `https://github.com/apache/spark/tree/v0.9.1`, 2014.

[5] Tuning spark. `http://spark.apache.org/docs/0.9.0/tuning.html`, 2014.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[7] Eric D Green, Mark S Guyer, National Human Genome Research Institute, et al. Charting a course for genomic medicine from base pairs to bedside. *Nature*, 470(7333):204–213, 2011.

[8] Matthew Hibbs, Grant Wallace, Maitreya Dunham, Kai Li, and Olga Troyanskaya. Viewing the larger context of genomic data through horizontal integration. In *Information Visualization, 2007. IV'07. 11th International Conference*, pages 326–334. IEEE, 2007.

[9] Matthew A. Hibbs, David C. Hess, Chad L. Myers, Curtis Huttenhower, Kai Li, and Olga G. Troyanskaya. Exploring the functional landscape of gene expression: directed search of large microarray compendia. *Bioinformatics*, 23(20):2692–2699, 2007.

[10] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22, 2011.

[11] Scott D Kahn. On the future of genomic data. *Science*, 331(6018):728–729, 2011.

[12] Vessela N Kristensen, Ole Christian Lingjærde, Hege G Russnes, Hans Kristian M Vollan, Arnoldo Frigessi, and Anne-Lise Børresen-Dale. Principles and methods of integrative genomic analyses in cancer. *Nature Reviews Cancer*, 14(5):299–313, 2014.

[13] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[14] Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *ACM Sigplan Notices*, volume 45, pages 341–360. ACM, 2010.

[15] Stephan C Schuster. Next-generation sequencing transforms today's biology. *Nature*, 200(8), 2007.

[16] Mark R Trusheim, Ernst R Berndt, and Frank L Douglas. Stratified medicine: strategic and economic implications of combining drugs and clinical biomarkers. *Nature Reviews Drug Discovery*, 6(4):287–293, 2007.

[17] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 international conference on Management of data*, pages 13–24. ACM, 2013.

[18] Karen Yook, Todd W Harris, Tamberlyn Bieri, Abigail Cabunoc, Juancarlos Chan, Wen J Chen, Paul Davis, Norie De La Cruz, Adrian Duong, Ruihua Fang, et al. Wormbase 2012: more genomes, more data, new website. *Nucleic acids research*, 40(D1):D735–D741, 2012.

[19] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[20] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[21] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.