Faculty of Science and Technology
Department of Computer Science

Data Cleaning for Biological Data

—

Inge Alexander Raknes

INF-3983 Capstone Project in Computer Science - December 2013

Abstract

Data cleaning is important for the analysis of large data sets. There are many tools for cleaning biological and non-biological data, but these have several limitations with respect to handling future biological data. First, tools can often only identify potential issues that a human expert must check in order to determine if it is an error or just expected variance. This requires interactive response times. Second, future biological data sets are predicted to be Peta-scale. Biological data cleaning tools do not currently handle Peta-scale data sets. Third, non-biological data cleaning tools that scale well do not work well with biological data sets.

To address these issues we present OutlierApp and SparkStats. OutlierApp is a prototype for a GUI that simplifies some manual steps in the current data cleaning process, by providing a web interface in place of user written R-scripts. SparkStats is a library of statistical methods for outlier detection. It provides interactive response times for large biological data sets.

The implemented GUI prototype shows that it is possible to support the basic data cleaning work-flow. The experimental evaluation of the SparkStats library demonstrate that large amounts of biological data can be analyzed on Spark with interactive response times.

We believe that by combining OutlierApp and SparkStats it is possible to provide an interactive solution for cleaning future big biological data sets.

Acknowledgements

I would like to thank my adviser Associate Professor Lars Ailo Bongo and my co-advisor Nicolle Mode for providing feedback and support throughout this project.

I would like to thank Professor Eiliv Lund for providing access to the NOWAC data set.

Finally, I would like to thank Jon Ivar Kristiansen, for help setting up Spark.

Contents

List of Figures

List of Tables

1 Introduction

Big datasets have great potential for novel insight in fields such as biology, physics, and social sciences. However, they require data cleaning as they contain a lot of noise that must be removed in order to gain the insight.

Big biological datasets often comprise many samples analyzed using a wide variety of instruments, models, and methods. For these datasets noise may be caused by sample contamination, instrument errors and inaccuracies, analysis methods and models faults, human mistakes, and even by the measured biological processes themselves. There have been developed many statistical methods and tools for removal of such noise[22]. But often these tools can only identify potential issues that a human expert must check in order to determine if it is an error or just expected variance. This can be a tedious and time consuming task, so there is a need to provide better tools support for such data cleaning.

We believe that a system for data cleaning of next-generation biological data should satisfy the following requirements. First, it should handle Peta-scale datasets since it is expected that biological datasets soon will be of that size. Second, it should be interactive since data cleaning requires the involvement of a human expert as discussed above. Third, it should automate data management and provenance to provide traceability for which, when, and why data was removed. To our knowledge no existing system for biological data cleaning provides these requirements.

We believe that a system for data cleaning of next-generation biological data should satisfy the following requirements. First, it should handle Peta-scale datasets since it is expected that biological datasets soon will be of that size[17]. Second, it should be interactive since data cleaning requires the involvement of a human expert as discussed above. Third, it should automate data management and provenance to provide traceability for which, when, and why data was removed. To our knowledge no existing system for biological data cleaning provides these requirements.

Existing approaches for biological data cleaning often use scripts written with statistical libraries provided by frameworks such as R. These work well for small datasets that fit into the DRAM of a single computer, but do not scale to Peta-scale datasets. Also, statistical libraries do not provide automated data provenance, but instead assume that the script ensures that enough provenance data is saved.

Wrangler is a tool for data cleaning and transformation. While Wrangler provides tools to create conversions between formats, it does not provide mechanisms for detecting outlying samples like distance measures[18, 15].

Usher provides a technique for flagging form submissions that are predicted to be outliers, and will make the user re-submit forms based on those predictions[8]. While this is suitable for questionnaires, it does not seem to be applicable to biological data.

DataPlay is a system that lets users use a trial-and-error approach to specify queries[5]. While this is useful when specifying complex queries on relational data, it does not seem likely that this can be used on biological data, as it is typically not relational.

Hadoop is a MapReduce framework for performing parallel computations on massive data sets. It does not however provide interactive response times, and even simple jobs can take minutes to schedule and execute. This poses a problem for cleaning biological data as the work-flow is iterative and requires user interaction on every iteration.

In this report we present a prototype system for interactive data cleaning of Peta-scale datasets. The system is built on Spark that is a relatively new system that provides interactive response times to large distributed computations. In this work I will describe the implementation of some basic operations that are essential to any data cleaning task*[16]* in Spark. We built a library that include distances-from-mean, and methods for selecting samples based on their relative position within the data set.

In addition to the abovementioned, we implemented a GUI for data cleaning for the NOWAC dataset. The GUI displays plots that are generated by user supplied scripts and implements methods for selecting and removing outliers. The GUI also displays which outliers have been removed.

We evaluated the statistical methods on a 78.3 GB generated data set that resembles a large set of biological data. The data set was cached in DRAM on a 10 node Spark cluster. Vector sum on all the data was performed in 430 ms and 384 ms using two different representations of the vectors. Calculating vector distance from mean was done in 6079ms. This demonstrate that large amounts of biological data can be analyzed on Spark with interactive response times.

The data cleaning GUI prototype demonstrates that it is possible to support the basic data cleaning work-flow with user supplied scripts from a simple web interface. It tracks removed outliers in the browser and lets the user undo any action.

The implemented GUI prototype shows that it is possible to support the basic data cleaning work-flow. By merging the GUI and the Spark application we can support data cleaning of future biological data sets.

In this report I will begin by explaining some data cleaning and big data systems in section §2 and section §3. Then I will explain a subset of Scala's syntax in 4. In 5 and 6 I will explain OutlierApp and SparkStats respectively. In the final part of the report I will provide a conclusion (section §7).

2 Data Cleaning

2.1 Wrangler

Wrangler[18] is an interactive system for data cleaning and transformation. Wrangler simplifies "data wrangling" by presenting a user interface, and suggestions about how to transform data. The user can explore the suggestions to see how different transformations transform the data.

A central feature of Wrangler is that its output is: "not just transformed data, but an editable and auditable description of the data transformations applied". This auditable description is in the form of a declarative data transformation language. The intent is to define robust transformations, rather than having the user do manual editing.

Transformations The Wrangler Transformation Language is a declarative data transformation language. It relies on a small set of primitives for transforming data. These primitives include map, lookups and joins (with external tables), approximate joins using string edit distance, reshape (fold, unfold), positional transforms (fill and lag operations), and various sorting and aggregation functions.

Visual transformation previews An important feature of Wrangler is that it visualizes every transformation immediately after it has been selected. It follows the concept of programming by demonstration (BPD), where the system automatically suggest transformations and the user can explore these transformations by clicking on the suggestions and immediately see how they affect the data.

Wrangler has an inference engine that will use a corpus of usage statistics to predict what transformations the user might be interested in, and rank them according to complexity. They are ranked according to complexity in order to make it easier for the user to read through the top suggestions.

Data types and semantic roles Wrangler has a set of data types, that it uses to discover data quality issues. Wrangler will try to infer the data-type of a column by seeing which data types validate for more than half of the non-missing values. This, in turn, is used to create the data quality meter, which is: "a divided bar chart that indicates the proportion of values in the column that verify completely". The user can click the bar chart to get suggestions for transformations.

History viewer Wrangler includes a history viewer where the user can see a list of the transformations that has been performed on the data set. The history list is in human readable language, and the user can use it to modify the steps that leads to the resulting output.

2.2 Proactive Wrangling

In Wrangler[15], users are often unsure which sequence of operation is needed to convert data into the desired format. The authors behind this paper have observed that: “both novice and expert users may consequently resort to blind exploration of the available transforms[15]”.

The authors behind this paper address these issues by providing proactive suggestions. This paper describes a method to proactively suggest data transforms that map input data to relational formats. The key idea behind the technique is to calculate a suitability score for a table, then suggest transforms that will increase the suitability of a table. Empirical data has been used to limit the search space for solutions, such that not all possible transforms are evaluated, but instead, only a small subset. The transformations are chosen such that they cover “the majority of use cases, as indicated by our data corpus[15]”.

2.3 NOWAC Data cleaning

The Norwegian Woman and Cancer (NOWAC) is a study that consists of more than 170,000 Norwegian women[12]. The data that has been used in this work is from the NOWAC postgenome cohort study, which is explained by the quotation below:

> The Norwegian Women and Cancer (NOWAC) postgenome cohort study consists of approximately 50,000 women born between 1943 and 1957 who gave blood samples between 2003 and 2006 and filled out a two-page questionnaire. Blood was collected in such a way that RNA is preserved and can be used for gene expression analyses. (Dumeaux 2008 [12])

At time of writing, data cleaning of NOWAC data sets is done using custom R scripts. This imposes certain limitations, as it requires the whole dataset to fit in RAM. As future biological datasets are predicted to be in the PB-scale, a new approach is needed.

3 Big Data Systems

In this section I will explain some systems that allows a user or a system to interact with large amounts of data.

3.1 Spark

Spark[30] 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. The main abstractions in Spark is RDD (Resilient Distributed Dataset), and Parallel Operations, which is an operation on an RDD.

An RDD is a distributed, immutable dataset that can be constructed from another RDD by applying an operation on it, or it can be constructed from a (HDFS) file.

3.2 Brainwash

Brainwash[6] is a system that reduces the difficulties in creating trained systems. Trained systems are systems that allows to run queries on data that are less structured than traditional relational data.

According to the authors these systems are challenging to build and that one of the critical pain points in building trained systems is feature engineering. From the paper:

> Features, sometimes called signals, encode information from raw data that allows machine learning algorithms to classify an unknown object or estimate an unknown value.

A benefit of using feature engineering is that it lets the system creators combine different techniques from different sources. However, according to the paper writing features can be extremely difficult.

Simplify feature engineering According to the authors, these tasks makes feature engineering painful:

1. Statistical "grunt work": The dataset's characteristics are often completely different from anything the developer has seen before.

2. Unknowable Specs: Interesting datasets are often large and noisy. This makes: "the actual feature code "spec" nearly unknowable without repeated testing against the data itself[6]". The authors demonstrate this by showing an example where after collecting usernames from social media to determine users age, the developer discover that usernames often have numbers and adjectives appended to the username.

3. Unexpected Failure: It is possible that a feature, when implemented, does not capture any useful information, or it's information is already captured by a previously implemented feature.

Design The authors envision a centralized system to give programmer hints, derived from data as well as code written by other users.

Brainwash attempts to place few assumptions on the input format, as it needs to be able to process a diverse variety of data. The authors treat feature development as a work-flow of developer-written functions (udf).

3.3 DataPlay

DataPlay[5] is a system that lets users use a trial-and-error approach to specify queries. Central to DataPlay's approach is to display non-answers as well as correct answers to a user's query.

Non-answers The authors behind the paper argues that when only looking at different correct query results, it is impossible to determine which results come from which query. By displaying non-answers it becomes possible to tell queries apart.

An example that is used in the paper, is the task of selecting straight-A students from a student database. It first presents the following query to select all students that got some A (existential):

```
SELECT * FROM students s, takes t,
WHERE t.grade = 'A' AND t.student_id = s.id;
```

In this example the 'takes' table relates students to classes, and contains their grades.

The result from the query above will display users and grades, where all grades are A.

Next it presents a query to select students that only got A (universal):

```
SELECT * FROM students s, JOIN takes t WHERE NOT
EXISTS
(SELECT grade FROM takes WHERE grade != 'A'
AND student_id = s.id);
```

The result from this query, as with the above, are tuples containing students and grades, where all grades are A.

Only by looking at the non-answers can we tell these tables apart: in the existential examples the non-answers will not contain any 'A's. In the universal example, however, there are students with 'A's.

Data and query model DataPlay uses the nested universal relation (nested UR). This model combines the properties of universal relations with nested data models such as JSON, XML and nested relations. The purpose is to have a single representation of the relations in the database, and to have an hierarchical structure of the relations.

4 Scala

Scala is a multi-paradigm programming language for the JVM. 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[10].

Scala is used for most of the implementation in this project (in addition to JavaScript). 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. Its, however, been assumed after this section that the reader has a basic understanding of Scala.

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.

4.0.1 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.

4.1 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.

4.2 Operator syntax

In Scala operators are method calls. That is, the following are equivalent:

Algorithm 2 Operator syntax

```
val ten = 5 + 5
val ten = 5.+(5)

val h = "hello".charAt(0)
val h = "hello" charAt 0
```

Using this property it is easy to define our own types with custom operators:

Algorithm 3 Vector operators

```
class Vector {
  def +(other: Vector): Vector = { ... }
  def −(other: Vector): Vector = { ... }

  // More methods ...
}

// Usage
val v1: Vector = ...
val v2: Vector = ...
val result = v1 + v2
```

This technique has been used to create a small vector class in the SparkStats application (section §6), which is frequently referred to in this report.

5 OutlierApp

OutlierApp is a web GUI application that was developed to assist in data cleaning of NOWAC data. This application is designed to both assist in the current data cleaning process, and to assist in a future process which involves much larger data sets (see SparkStats: section 6 on page 18).

5.1 Current process

The current process, consists of a statistician using R-scripts for outlier detection. The statistician will write an R-script that produces different plots where the outliers can easily be found [22]. The R-script is then modified by the user so that it removes the outliers. A new plot is then generated from the resulting dataset, and the process is repeated.

Manual labor in this process includes:

- Modifying the R-script in order to remove outliers
- Running the R-script from the command line

Despite of the drawbacks of the current process, it also has a benefit: The fact that the user is creating their own script to generate their plots, gives the user a lot of power and flexibility. For example, if requirements were to change, and a new type of plot- or a different outlier detection algorithm were required, then the user could easily accommodate this by using one of the many features that are already present in R. The users are already using different outlier detection algorithms in their current process.

The goal of OutlierApp is to automate the tedious parts of this process, so that the user is relieved of having to manually modify and run a custom script, while at the same time preserve the user's flexibility. An important point w.r.t. to automating the process, is that the users don't actually *want* to automate the entire process; they want to make sure that a human expert makes the final decision about every outlier, as they don't trust statistical models to do this job.

5.2 Challenges

Flexibility The current process is very flexible, since the user supplies her own script. OutlierApp needs a similar amount of flexibility. In particular the algorithm that detects outliers must be easy to replace.

Security As the data is of commercial interest, it is a requirement that the data never leaves Stallo, which is the supercomputer at UiT[2] .

5.3 Architecture

The architecture of OutlierApp is as follows: It is divided in two parts: A web GUI that is available from the user's web browser, and a backend that interacts with the data. The web GUI lets the user interact with the data via the backend. Only aggregated results are sent back to the client (e.g. plots), and no actual data. This is in line with the security requirements mentioned in section 5.2. A general principle in this architecture is that the app itself doesn't make any assumptions about the data formats, or how the plots are generated, or how outliers are detected. These concerns are left to user supplied scripts. OutlierApp will only run those scripts when the results are required by the user.

The architecture is illustrated in figure 1.

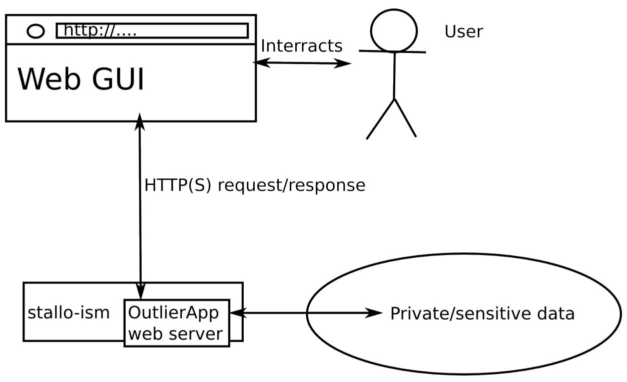

Figure 1: OutlierApp architecture

5.4 Design

The user has to supply three scripts. The separation into three scripts makes it possible for the statistician to change the detection algorithms without having to duplicate the removal- and plotting code.

The execution of the processing the data is broken down into three steps, which will be executed by scripts that are provided by the user (figure 2).

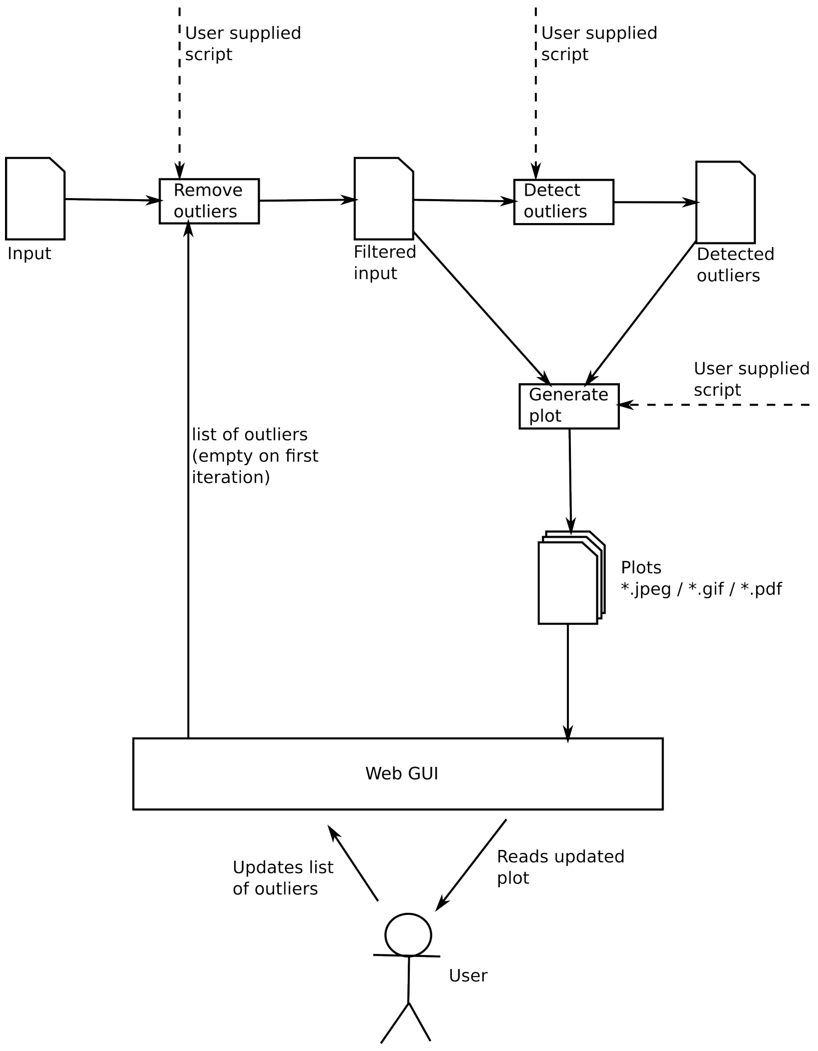

Figure 2: OutlierApp design

5.4.1 User supplied scripts

The scripts are parametrized via environment variables, as these are easily accessible from R, as well as most other programming languages and platforms. The scripts are parametrized and run by the backend server, whenever the user demands an updated plot in the browser.

All file paths in the environment variables are defined to be absolute file system paths. The format of the actual data files and the outlier identifiers have been left undefined on purpose to allow for maximum flexibility on behalf of the user. This means that it's up to the user to decide on the input- and output formats, and to make sure that they align.

Remove outliers This script takes a raw dataset that contains all the data points, and removes the outliers that are specified by the user.

Name	Description
INPUT_FILE	Path to the file that contains the raw data
OUTPUT_FILE	Path to the file that should contain the data with the outliers removed
OUTLIERS	Comma-separated list of outliers

Table 1: Environment variables for *remove outliers*

Detect outliers This script detects possible outliers within a dataset.

Name	Description
INPUT_FILE	Path to the file that contains the data
OUTPUT_FILE	Path to the file that should contain the detected outliers

Table 2: Environment variables for *detect outliers*

Generate plot This script generates different plots from the generated outliers, and the filtered input, and puts them in a directory.

Name	Description
DATA_FILE	Path to the file that contains the data
OUTLIERS_FILE	Path to the file that contains the detected outliers
OUTPUT_DIR	Directory that should contain the generated plots.

Table 3: Environment variables for *generate plot*

5.5 Implementation

The implementation is separated into a front end / JavaScript application that runs in the browser, and a back-end web server that serves the web app and acts as an interface between the web app and the data cleaning process.

The front-end is implemented in JavaScript. It uses Angular.js for MVC, and Twitter Bootstrap for style and layout, whereas the back-end is implemented in Scala and the Play Framework.

5.6 Future work

GUI A key component of the application are the user supplied scripts. Currently there is no way for the user to upload- or manage these scripts from the browser. This functionality would be simple to implement.

Plot file formats The current prototype only supports **.jpeg*, but the actual R-scripts are capable of producing a much wider selection of formats. It would be easy to implement support for all browser supported image formats, as they only need to be served and referenced in an *img*-tag.

Support for plotting in the browser Currently OutlierApp depends on a user-supplied script to put plots into a directory on the file system. A more flexible approach would be to allow the user generated scripts to generate data points that can be plotted in the browser. This would make it possible for the user to interact with the plots.

Live updates of plots in the browser Currently the browser waits for all the plots to be completed before it start drawing, which can lead to long perceived response times if the plot generator takes a significant amount of time to complete. An alternative solution would be to send the plots to the browser using Web Sockets immediately after they become available on the file system.

Integration with Spark It should be possible to use Spark on the back-end in addition to user supplied scripts.

5.7 Evaluation

All the data must fit in DRAM. This can be a challenge for biological datasets, as these are predicted to be Peta-scale. To solve this issue, I introduce SparkStats, which is an application that can run simple outlier detection algorithms on a Spark cluster.

6 SparkStats

Currently data cleaning is done in R, however this will not scale as future methods will generate data that does not fit into memory of a singe computer. For these cases something new is required, and Spark could be a suitable platform.

A key observation that can be made from OutlierApp is that the back end that currently executes user supplied scripts, might also execute interactive jobs on a large cluster. However, it is a requirement that the system on the cluster allows for interactive response times, as observed by the user. Spark is a system that provides interactive response times. This makes Spark different from Hadoop in which even a simple job can take minutes to schedule and execute.

SparkStats is an application that was developed to demonstrate different data cleaning techniques on a Spark cluster.

6.1 Vectors

6.1.1 Array representation

In this section I will discuss vectors that are stored in sequence in the RDD, such that the RDD has type *RDD[Vector]*. A small vector class which supports both destructive- as well as non-destructive operations was written for this purpose. Because the name *Vector was* already taken in the Scala standard library, I named the class *Vec*. A simplified description of the class is shown in (table 4 on the next page). Each object of type *Vec* is backed by an

Array[Double], and all operations are implemented in terms of the underlying array.

Method signature	Description
def +(other: Vec): Vec	Produce a new vector which value is *this + other*.
def -(other: Vec): Vec	Analogous to +
def +=(other: Vec): Vec	Add *other* to *this* (updates *this* in-place). Returns *this*.
def -=(other: Vec): Vec	Analogous to +=
def negate: Vec	Produce a new vector which value is the negative of *this*.
def norm: Double	The euclidean vector norm of *this*.
def copy(): Unit	Produce a new vector which is identical to *this*.
def hasChanged: Boolean	*true* if the vector has been mutated after it's creation. Otherwise *false*.

Table 4: Class Vec.

6.1.2 Vector sum on Spark using array representation

Naive implementation Given an RDD[Vec], named rdd, the most straightforward way to sum all the vectors in rdd would be by reducing it using the '+' operator (as defined on Vec):

```
val sum: Vec = rdd.reduce(_+_)
```

While this implementation is correct, it's performance is far from optimal. Since the '+' operator allocates a new vector (and thus a new Array), $n-1$ new vectors will have to be allocated for a partition containing n vectors. This is illustrated in (figure 3). As shown in table 5, allocating these arrays is very expensive.

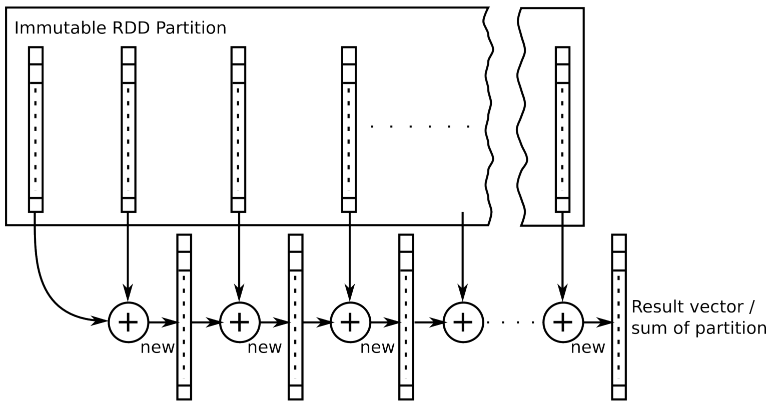

Figure 3: Naive vector sum, using rdd.reduce(_+_)

Optimization In order to describe how this implementation can be optimized, we must first take a closer look at how Spark implements *reduce* (fig. 4).

1. Spark will run reduceLeft on each partition.
2. Spark will send the result of each partition to the master node, and reduce them to an accumulated value.

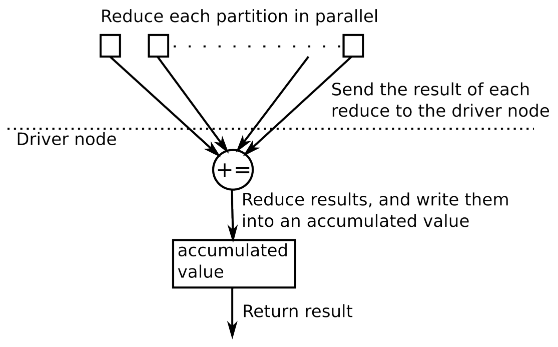
Reduce each partition in parallel
Send the result of each reduce to the driver node
Driver node
+=
Reduce results, and write them into an accumulated value
accumulated value
Return result

Figure 4: Reduce, as implemented by Spark

The essential part is that Spark runs reduceLeft on each partition. The reduceLeft method is defined in the trait TraversableOnce in the Scala standard library (see algorithm 4).

Algorithm 4 Implementation of *reduceLeft*, as defined in the Scala standard library (*scala/collection/TraversableOnce.scala*).

```
1  def reduceLeft[B >: A](op: (B, A) => B): B = {
2    if (isEmpty)
3      throw new UnsupportedOperationException("empty.reduceLeft")
4
5    var first = true
6    var acc: B = 0.asInstanceOf[B]
7
8    for (x <- self) {
9      if (first) {
10       acc = x
11       first = false
12     }
13     else acc = op(acc, x)
14   }
15   acc
16 }
```

If we study the implementation in Algorithm 4, we see that it keeps the accumulated partial result in the variable *acc* on line 6, and that this variable is consistently passed into the supplied function *op* as the first argument (line 13). This implies that whenever we return the first argument in *op*, the same object will be passed in as the first argument in the next iteration. This means that we can accumulate the result ourselves (in the *op* function) by mutating the first argument and return it. When the results of each partition is reduced on the driver node, they are reduced in a similar manner as in the standard library, using the first argument as the accumulator. This means that we will benefit from such an optimization twice: once when reducing the partitions, and once again when reducing the results from the partitions.

A simple way to implement an optimized vector sum, using this technique, would be the following code:

```
val rdd: RDD[Vec] = ...
val sum = rdd.reduce(_ += _) // Wrong: breaks the RDD's
                             // immutable property
```

While this code would produce the correct result, it would also break the immutable property of the RDD (see figure 5). This would corrupt future computations on the RDD if the RDD is persistent (i.e. cached).

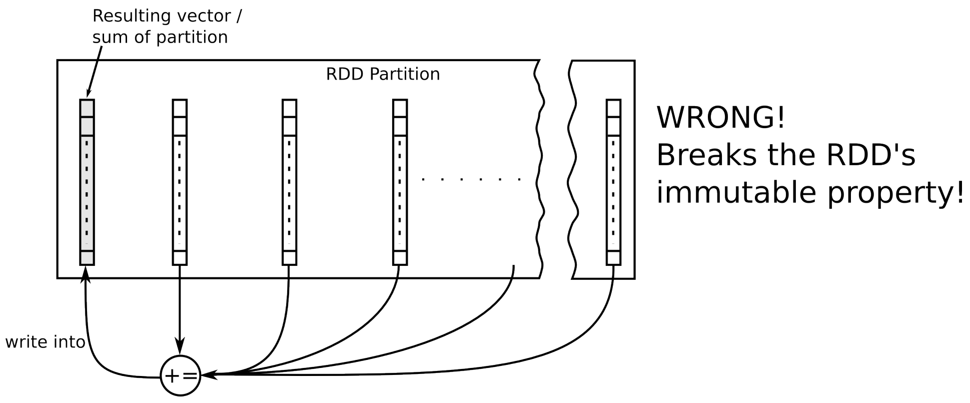
Resulting vector / sum of partition
RDD Partition
WRONG! Breaks the RDD's immutable property!
write into
+=

Figure 5: Incorrectly optimized vector sum. The color of the vector indicates it's '*hasChanged*' property at the end of the computation.

An obvious solution to this problem would be to copy the first vector in the partition, and then write the accumulated results into the copy (fig. 6 on the next page).

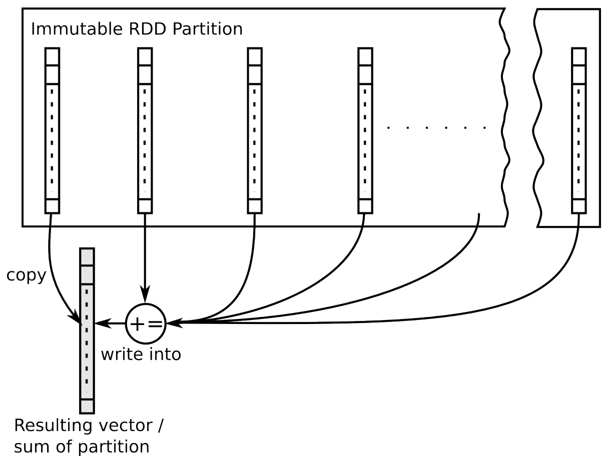

Figure 6: Optimized vector sum

In order to implement this solution we must somehow identify the first vector within a partition from within the reduce function *(op*), copy it, and return the copy for use in future iterations: Because the RDD is immutable none of the contained vectors may ever have been mutated. This implies that the *hasChanged* property of each vector within the partition is *false*. If we were to return a mutated copy from *op* then its *hasChanged* property would be *true* and the copy would be passed into *op* as the first argument on the next iteration (ref. implementation of *reduceLeft*). This gives us a simple way to determine if a vector is a copy that can be safely mutated: it's a copy if (and only if) its *hasChanged* property is *true*. If we consistently return the copy when we encounter it in the reduce function, then the only occasions where the reduce function will encounter a vector with *hasChanged=false* as it's first argument will be when the first argument is the first vector in a partition, and the second argument is the second vector in a partition. In this case we can return a copy. This yields the algorithm listed in (Algorithm 5).

Algorithm 5 Optimized vector sum on Spark

```
def vectorSum(rdd: RDD[Vec]): Vec = rdd.reduce { (lhs, rhs) =>
  if(lhs.hasChanged) {
    lhs += rhs
    lhs
  } else {
    val copy = lhs.copy()
    copy += rhs

    assert(copy.hasChanged)

    copy
  }
}
```

6.1.3 Vector distances

Given a tuple of vectors, $V = (\mathbf{v}_0, \ldots, \mathbf{v}_n)$, and an average vector $\bar{\mathbf{v}}$ defined as $\bar{\mathbf{v}} = \frac{1}{n-1} \sum_{i=0}^{n} \mathbf{v}_i$, we want the tuple of vector distances defined as $D = (\|\mathbf{d}_0\|, \ldots, \|\mathbf{d}_n\|)$, where $\mathbf{d}_i = \mathbf{v}_i - \bar{\mathbf{v}}$.

Let j represent the indices into the vectors $\mathbf{v}_i$ and $\bar{\mathbf{v}}$. A way to calculate $\|\mathbf{d}_i\|^2$ would be $\|\mathbf{d}_i\|^2 = \sum_j (v_{i,j} - \bar{v_j})^2$.

Calculating distances An efficient representation for this problem on Spark, is to lay the vectors in the same direction as the RDD, as can be seen in figure 7.

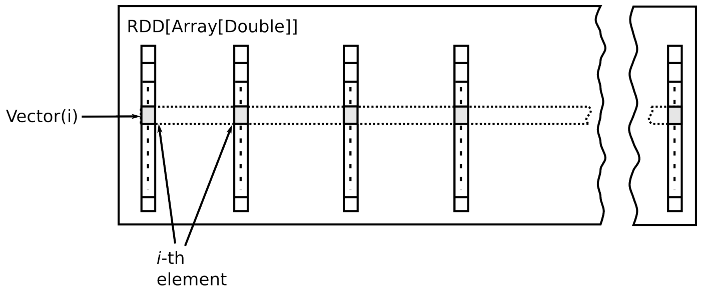

Figure 7: Vector repressentation

We can now calculate all $(v_{i,j} - \bar{v_j})^2$ by performing a single *map* on the RDD (Algorithm 6).

Algorithm 6 Calculating all $(v_{i,j} - \bar{v_j})^2$

```
val r: RDD[Vec] = rdd.map { array =>
  val average = array.sum / numVectors
  array.map (x => (x - average) * (x - average))
  Vec(array)
}
```

This results in an RDD consisting of vectors that are represented using *array representation*. In order to get $\|\mathbf{d}_i\|^2$ we sum the resulting RDD of vectors using the technique described in Algorithm 5.

Comparison to array representation The benefit of using this representation is that calculating $(v_{i,j} - \bar{v_j})^2$ can be done in a single map operation, whereas if we use the array representation we would need a separate reduce operation just to calculate the average. Because of this and the fact that arrays have fast lookups and writes, we could say that we trade away the ability to do fast operations between a single vector's dimensions, for the ability to do fast operations between multiple vectors, but on a single dimension.

6.1.4 Conclusion

Array representation The timings for the array representations are in table 5. These are acceptable latencies for interactive work-flows.

Experiment	Average	Median	Min	Max	Std.dev
Wrongly optimized vector sum	522	523	492	553	13
Optimized vector sum	430	428	406	605	21
Naive vector sum	3323	3309	3240	4338	110

Table 5: Vector sum run times. 2M vectors, 5K dimensions. All measurements in milliseconds. Each experiment has been run 100 times.

Vector distances The timings for calculating vector sums and distances from average using the representation explained in section 6.1.3 are shown in table 6. We can see a slight improvement on vector sum. Calculating vector distances takes longer, as the algorithm has to do several passes over the data. The timings are still tolerable for interactive work-flows, although some improvements would be beneficial.

Experiment	Average	Median	Min	Max	Std.dev
Vector distance from average.	6079	6070	5829	6453	118
Vector sums.	384	371	320	677	66

Table 6: Vector distance run times. 5K vectors, 2M dimensions. All measurements in milliseconds. Each experiment has been run 100 times.

6.2 Associating elements of an RDD with their relative position

There are use cases in which it is necessary to associate each element of a sequence to its relative position (index) within that sequence. Use cases include slicing operations and element selection, which are both important in data cleaning. To accommodate this the Scala standard collection library comes with a method called *zipWithIndex* that associates each element in a sequence to its (relative) index. The result is a sequence of tuples of the form $(element, index)$. An example can be seen in fig. 8.

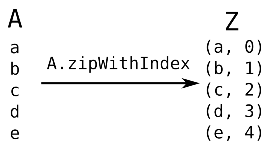

Figure 8: zipWithIndex

Many operations, including slicing and element selection, can easily be implemented from *zipWithIndex*, *filter*, and *map*. An example of a slicing operation can be seen in figure 9 on the facing page.

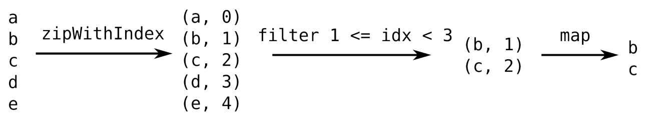

Figure 9: Slicing elements from index 1 until 3.

The Scala standard library also includes a method named *slice(from,to)* that lets the user slice any sequence.

A Spark RDD implements the methods *filter* and *map*, but it does not come with a method like *zipWithIndex* that lets the user associate each element to its relative index. Nor does it come with methods for slicing, or arbitrary element selection based on an element's relative position. These operations are important in data cleaning, and therefore motivate an implementation.

6.2.1 Implementing *zipWithIndex*

In the standard Scala collection library *zipWithIndex* is implemented using a regular (sequential) for-loop. Unfortunately, this is not a practical way to implement *zipWithIndex* on a large multi-GB dataset distributed across a large cluster. However, there are two methods that lets us implement *zipWithIndex* in a concurrent fashion: *zip* and *Range*. Zip is a method defined on any sequence in Scala that lets the user combine one sequence with another into tuples (like a zipper on a jacket). An example can be seen in figure 10.

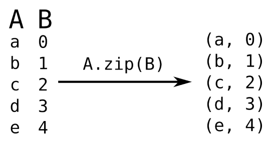

Figure 10: Combining the sequences A and B, using zip

Range is an object that represents a range of numbers (say from 0 to 10). It implements the trait Seq and features lazy evaluation. An obvious way to implement *zipWithIndex* using the standard library would be to *zip* a given sequence with a *Range* that represents the sequence's indices (Algorithm 7).

Algorithm 7 Implementing *zipWithIndex* using *Range* and *zip.*

```
def zipWithIndex[T](seq: Seq[T]) = {
  seq.zip(Range(0, seq.length))
}
```

Spark has specific optimizations in place to generate an RDD from a *Range*. An RDD does come with a method named *zip*, but it's subtly different from the *zip* that is defined in the standard library. Below I have quoted the documentation for the method "zip" from the Spark API documentation:

> Zips this RDD with another one, returning key-value pairs with the first element in each RDD, second element in each RDD, etc.

> **Assumes that the two RDDs have the *same number of partitions* and the *same number of elements in each partition* (e.g. one was made through a map on the other)**. [My emphasis] [3]

Studying the source code in the official Spark repository, reveals why these assumptions need to hold: it appears that *zip* will simply align all the partitions of the two RDDs[1] and then perform *zip* on each pair of partitions[2][4].

This insight gives a clue as to what could go wrong if we naively attempt to implement *zipWithIndex* on an RDD in a similar manner as implemented in Algorithm 7 on the previous page: It could be that the partitions of the two RDDs, the one that we want to index and the one with the indices, line up differently and we get an incorrect result! This is illustrated in figure 11.

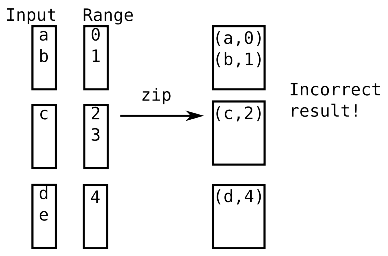

Figure 11: Naive *zipWithIndex* implemented on an RDD using *Range*.

It turns out that correctly implementing *zipWithIndex* on an RDD requires a more elaborate solution (as shown in 6.2.2).

6.2.2 Implementing *zipWithIndex* on an RDD

We need to generate the indices such that they line up with each partition. For transforming partitions an RDD[T] has two methods, *mapPartitions* and *mapPartitionsWithIndex*. Their documentation is quoted below (some arguments have been omitted for brevity and clarity):

def mapPartitions[U](f: Iterator[T] => Iterator[U]): RDD[U] Return a new RDD by applying a function to each partition of this RDD.[3]

def mapPartitionsWithIndex[U](f: (Int, Iterator[T] => Iterator[U]): RDD[U] Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.[3]

By using *mapPartitions* we can count the number of elements in a single partition, as seen in Algorithm 8 on the next page.

[1] *core/src/main/scala/org/apache/spark/rdd/ZippedRDD.scala*, lines 30-34 [4]

[2] *core/src/main/scala/org/apache/spark/rdd/ZippedRDD.scala*, line 64. Class instantiated by method *zip* in *core/src/main/scala/org/apache/spark/rdd/RDD.scala*, line 537 [4]

Algorithm 8 Counting the elements of each partition in an RDD

```
val counts = rdd.mapPartitions(_.length)
```

If we *collect()* these counts (send them to the driver node), we can compute the index of the first element of each partition. When we have an array of first indices we can use *mapPartitionsWithIndex* to map every partition of the input, into a partition of tuples $(element, index)$, where $index$ is the global index of the first element plus the relative position of $element$ within the partition. The whole process is illustrated in figure 12.

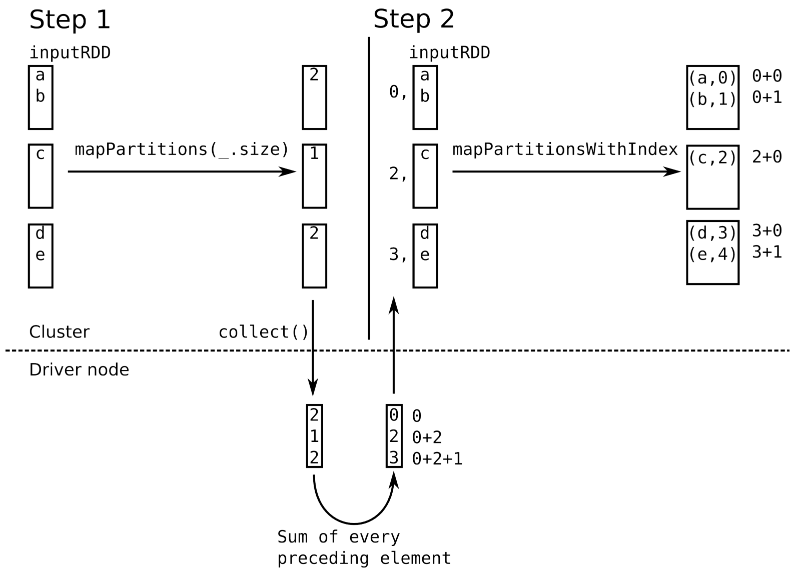

Figure 12: zipWithIndex implemented on an RDD

6.2.3 Discussion and future work

This implementation of *zipWithIndex* is both correct, and preferment. However, it still has a few drawbacks that I will discuss below.

Evaluation is not lazy Because *collect()* on an RDD is defined as an *action*, it follows that the entire input RDD has to be materialized in order to compute the offsets. In the delivered implementation this is done eagerly as soon as *zipWithIndex* is called. This may lead to wasted (re-)computations in cases where the input RDD has a deep dependency graph. A workaround would be to cache the input RDD until the result of *zipWithIndex* is ready (this assumes that the input will benefit from caching, i.e. it fits mostly in ram of the cluster).

There might exist a better solution where the array of counts is stored in an RDD, and the computation of offsets is a mapping on this RDD between step 1 and step 2. This approach was not explored, as it was less obvious that it would work, but if it works it would retain the lazy property of the transformation.

The full input RDD has to be materialized This is a consequence of the fact that we need to count each element in every partition – there is no way to know the number of elements in advance. However, we can't count an element that hasn't been materialized, as the underlying iterator on each partition returns a materialized element (the iterator is used for counting). This can be wasteful if we're only interested in a small subset of the output (i.e. when doing a *slice* on a small interval, or selecting a single element). A solution could be to box every element in a lazy container, meaning that a container will only materialize its content upon demand (and then cache it), but the container still being present in the RDD, such that it can be counted. This was not done, although this would significantly reduce overhead, if we assume that iterating the elements is cheap relative to computing their contents. A lazy container is easy to construct in Scala, as Scala has language support for lazy evaluation (lazy val). An example of a lazy container can be seen in Algorithm 9.

Algorithm 9 A lazy container

```
class LazyContainer[T](_element: => T) { // The '=>' means
                                         // call-by-name
  lazy val element = _element // Only evaluated once
}

// Usage

def expensiveComputation: Int = { println("EVALUATED"); 5}
val container = new LazyContainer(expensiveComputation)

// In the Scala REPL
scala> container.element
EVALUATED
res9: Int = 5

scala> container.element
res10: Int = 5
```

6.2.4 Conclusion

The timings from running *zipWithIndex* and some use cases can be seen in table 7. In the experiment the data set was already cached in RAM on the cluster. Because of this, run times were relatively short, as the algorithm only needed to iterate each partition's iterator once, thus performing only 5000 iterations in total (1 for every vector).

Experiment	Average	Median	Min	Max	Std.dev
zipWithIndex	95	67	44	358	80
slice(1000, 1000)	67	68	49	154	12
slice(1000, 1400)	64	64	51	76	5
slice(1000, 2000)	75	68	47	191	30

Table 7: Use-cases for zip-with-index. 5K vectors, 2M dimensions. All measurements in milliseconds. Each experiment has been run 1000 times.

6.3 Future work

Statisticians use more sophisticated outlier detection algorithms. They use inter-array correlation (IAC) methods and more thorough statistical analysis. This was not implemented in SparkStats due to time constraints, and thus it is left for future work. However, implementing this on top of SpakStats should not pose any major challenges.

7 Conclusion

The experimental evaluation of the SparkStats library demonstrate that large amounts of biological data can be analyzed on Spark with interactive response times. One of the most interesting surprises was how small optimizations can yield massive runtime improvements. Avoiding unnecessary memory allocations and using for-loops instead of iterators when iterating arrays provided large speedups.

The implemented GUI prototype shows that it is possible to support the basic data cleaning work-flow. By merging the GUI and the Spark application we can support data cleaning of future biological data sets.

References

[1] Apache spark - lightning-fast cluster computing. http://spark.incubator.apache.org/, 2013.

[2] Hardware | notur. https://www.notur.no/hardware/stallo, 2013.

[3] Spark api docs (scaladoc). http://spark.incubator.apache.org/docs/latest/api/core/index.html, 2013.

[4] Spark source code (branch-0.8, github). https://github.com/apache/incubator-spark/tree/branch-0.8, 2013.

[5] Azza Abouzied, Joseph Hellerstein, and Avi Silberschatz. Dataplay: interactive tweaking and example-driven correction of graphical database queries. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 207–218. ACM, 2012.

[6] Michael Anderson, Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael J Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher Ré, and Ce Zhang. Brainwash: A data system for feature engineering.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[8] Kuang Chen, Harr Chen, Neil Conway, Joseph M Hellerstein, and Tapan S Parikh. Usher: Improving data quality with dynamic forms. *Knowledge and Data Engineering, IEEE Transactions on*, 23(8):1138–1153, 2011.

[9] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M Hellerstein, and Caleb Welton. Mad skills: new analysis practices for big data. *Proceedings of the VLDB Endowment*, 2(2):1481–1492, 2009.

[10] Ecole Polytechnique Federale de Lausanne (EPFL). Scala in the enterprise. `http://www.scala-lang.org/old/node/1658`, 2012.

[11] Pan Du, Warren A Kibbe, Simon Lin, and Gang Feng. Using lumi, a package processing illumina microarray, 2013.

[12] Vanessa Dumeaux, Anne-Lise Borresen-Dale, Jan-Ole Frantzen, Merethe Kumle, Vessela N Kristensen, and Eiliv Lund. Gene expression analyses in breast cancer epidemiology: the norwegian women and cancer postgenome cohort study. *Breast Cancer Res*, 10(1):R13, 2008.

[13] Vanessa Dumeaux, Karina S Olsen, Gregory Nuel, Ruth H Paulssen, Anne-Lise Børresen-Dale, and Eiliv Lund. Deciphering normal blood gene expression variation - the nowac postgenome study. *PLoS genetics*, 6(3):e1000873, 2010.

[14] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[15] Philip J Guo, Sean Kandel, Joseph M Hellerstein, and Jeffrey Heer. Proactive wrangling: mixed-initiative end-user programming of data transformation scripts. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 65–74. ACM, 2011.

[16] Joseph M Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

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

[18] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 3363–3372, New York, NY, USA, 2011. ACM.

[19] Fumiaki Katagiri and Jane Glazebrook. Overview of mrna expression profiling using dna microarrays. *Current Protocols in Molecular Biology*, pages 22–4, 2009.

[20] Michael L Metzker. Sequencing technologies–the next generation. *Nature Reviews Genetics*, 11(1):31–46, 2009.

[21] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 15–14, Berkeley, CA, USA, 2010. USENIX Association.

[22] Michael C Oldham, Steve Horvath, Genevieve Konopka, Kazuya Iwamoto, Peter Langfelder, Tadafumi Kato, and Daniel H Geschwind. Identification and removal of outlier samples supplement for:" functional organization of the transcriptome in human brain. *dim (dat1)*, 1(18631):105.

[23] Fatma Özcan, David Hoa, Kevin S Beyer, Andrey Balmin, Chuan Jie Liu, and Yu Li. Emerging trends in the enterprise data analytics: connecting hadoop and db2 warehouse. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1161–1164. ACM, 2011.

[24] Nicole Rusk. Focus on next-generation sequencing data analysis. *Nature Methods*, 6:S1–S1, 2009.

[25] Paul A Schulte and Frederica P Perera. *Molecular epidemiology: principles and practices*. Academic Press, 1998.

[26] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[27] Eric P Widmaier, Hershel Raff, and Kevin T Strang. *Vander's human physiology*. McGraw-Hill Higher Education, 2006.

[28] 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 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[29] 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[30] 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.

[31] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming

computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.