UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology

Department of Computer Science

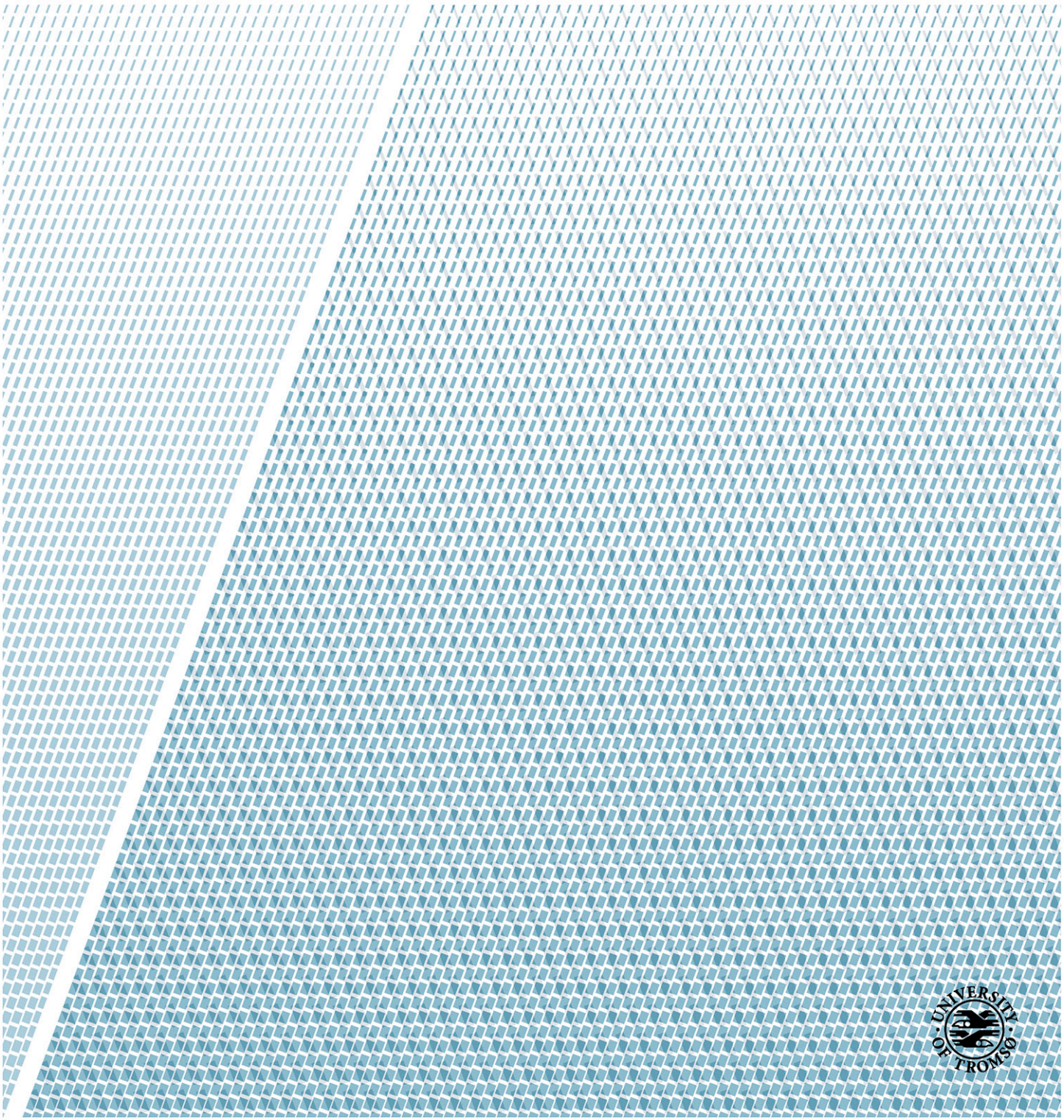
Mario

A System for Iterative and Interactive Processing of Biological Data

—

Martin Ernstsen

INF-3990 Master's Thesis in Computer Science, November 2013



Abstract

This thesis address challenges in metagenomic data processing on clusters of computers; in particular the need for interactive response times during development, debugging and tuning of data processing pipelines. Typical metagenomics pipelines batch process data, and have execution times ranging from hours to months, making configuration and tuning time consuming and impractical.

We have analyzed the data usage of metagenomic pipelines, including a visualization frontend, to develop an approach that use an online, data-parallel processing model, where changes in the pipeline configuration are quickly reflected in updated pipeline output available to the user.

We describe the design and implementation of the Mario system that realizes the approach. Mario is a distributed system built on top of the HBase storage system, that provide data processing using commonly used bioinformatics applications, interactive tuning, automatic parallelization and data provenance support.

We evaluate Mario and its underlying storage system, HBase, using a benchmark developed to simulate I/O loads that are representative for biological data processing. The results show that Mario adds less than 100 milliseconds to the end-to-end latency of processing one item of data. This low latency, combined with Mario's storage of all intermediate data generated by the processing, enables easy parameter tuning. In addition to improved interactivity, Mario also offer integrated data provenance, by storing detailed pipeline configurations associated with the data.

The evaluation of Mario demonstrate that it can be used to achieve more interactivity in the configuration of pipelines for processing biological data. We believe that biology researchers can take advantage of this interactivity to perform better parameter tuning, which may lead to more accurate analyses,

and ultimately to new scientific discoveries.

Acknowledgements

First and foremost I would like to thank my advisor, Associate Professor Lars Ailo Bongo, for providing invaluable guidance throughout this project. I would also like to thank my co-advisor, Professor Nils-Peder Willassen for providing me with insights from the biology side of things.

Jon Ivar Kristiansen has been very helpful with installation of software and troubleshooting the systems I have used. I am also very grateful to Erik Kjærner-Semb, for letting me use results from his Master's thesis as a motivation in my own work.

Finally, special thanks go to Laura Liikanen for supporting my career change, and for her support and encouragement during the last months.

Contents

List of Figures

List of Abbreviations

GFS	Google File System.
GUI	Graphical User Interface.
HDFS	Hadoop Distributed File System.
RDD	Resilient Distributed Dataset.
RPC	Remote Procedure Call.
WAL	Write-Ahead Log.

Chapter 1

Introduction

1.1 Bioinformatics and Metagenomics

Metagenomics is the study of metagenomes - genetic material isolated directly from environmental samples. While traditional genomics (e.g. analysis of structure and function on genomes) rely on being able to isolate and cultivate the organism under study, metagenomics is cultivation independent. With todays cultivation technologies only a small fraction of microorganisms have been successfully cultivated. Advances in sequencing and computing technologies have made metagenomics feasible, and it has now become a preferred technology to study whole bacterial communities, addressing questions like; who is there, what are they doing and how are they doing it[30].

Bioinformatics is an interdisciplinary field comprising algorithms and applications for storing, processing and analyzing biological data. Bioinformatics and computer systems research is becoming more and more important because data generation from sequencing is doubling every nine months - much faster than the increase in processing and storage capacity[16]. According to Sboner et al.[26], in year 2000 the sequencing itself would dominate the overall cost of a sequencing project, while in 2010 the cost of data management and analysis would dominate. As a consequence of this trend, new infrastructure systems are needed for efficient handling and analysis of the data.

1.2 Pipelines in Bioinformatics

A computer system for analyzing biological data typically consist of three main components: the input data, a set of tools "chained" together in a pipeline, and finally an analysis- and/or a visualization system (figure 1.1).

Input data to a typical pipeline are produced by instruments such as sequencing machines in a laboratory. This data consist of sequences of nucleotides of varying length, and the datasets can range in size from megabytes to several terabytes of data.

The input data are typically processed by a series of applications, arranged so that the output of one application is the input to the next application (figure 1.1). Many different applications can be used in the pipeline stages. Some are small user-created scripts, others are large complex applications. Some are open-source and others are proprietary with source code that is not available to the end-user. This setup is referred to as a *pipeline* or sometimes as a *workflow*.

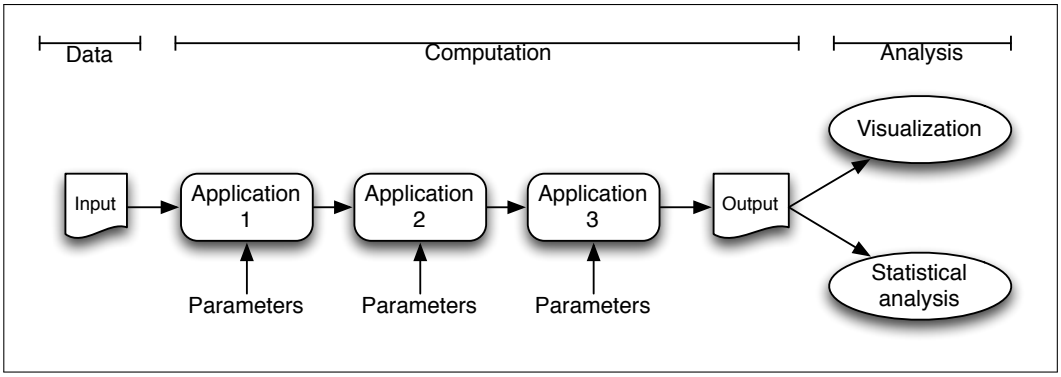


Figure 1.1: Applications arranged in a pipeline

The final output from the pipeline can be imported into applications that perform statistical analysis or visualization of the results.

1.2.1 Observations

As an example of a typical pipeline, the *METApipe* pipeline[15] used for metagenomics at the University of Tromsø, contains the following applications or stages:

1. MGA - Multiple Genome Aligner[11]
2. MGA-Exporter (in house)
3. Filescheduler
4. BLASTP[1]
5. HMMer[6]
6. Annotator
7. Annotator-Exporter

The output is manually imported into METAREP[10] for statistical analysis and visualization. The pipeline batch-processes the data, meaning that one stage is completed before the next stage is started. Common for many of these pipelines is that one or more of the stages are CPU-intensive, resulting in execution times of days, weeks or even months on the compute clusters available to the research groups.

Some other characteristics seen in these pipelines are:

1. The tools used in the pipeline stages take files as input and produce files as output. These files are copied between computers, either manually, or using scripts.
2. Parallelization is performed only on the most resource-intensive pipeline stages, by splitting input files and moving these to other compute nodes and then collecting results.
3. Pipeline configuration is time consuming, since changing a stage requires recomputing downstream stages to see the effect of the change. Returning to the original setting requires another recomputation.
4. Data and the pipeline configuration are separate, in the sense that the configuration is not recorded with the data. The researcher must therefore manually keep track of which configurations were used with which data, thereby increasing the risk of making mistakes.

1.2.2 Issues

An important and time consuming part of bioinformatics analysis is setup and configuration of pipelines. This involves deciding on which tools to use for each stage, and the best parameters for each tool. The parameters used may have a big impact on the quality of the output data from the pipeline, but since the pipeline typically contains long-running batch jobs, it is time consuming and difficult to make an informed decision on the settings.

As an example of the significance of parameter tuning in taxonomic classification of metagenomics samples, figure 1.2 show the number of different taxa remaining for increasing values of a confidence cutoff parameter (unpublished, Kjærner-Semb, Department of Chemistry, University of Tromsø[17]). If the parameter is set to a low value, little statistical confidence is needed to include a data point. For example, if the parameter is set to 0.1, approximately 80% of genera remains. If the parameter is set to a higher value of 0.9, approximately 10% of genera remain.

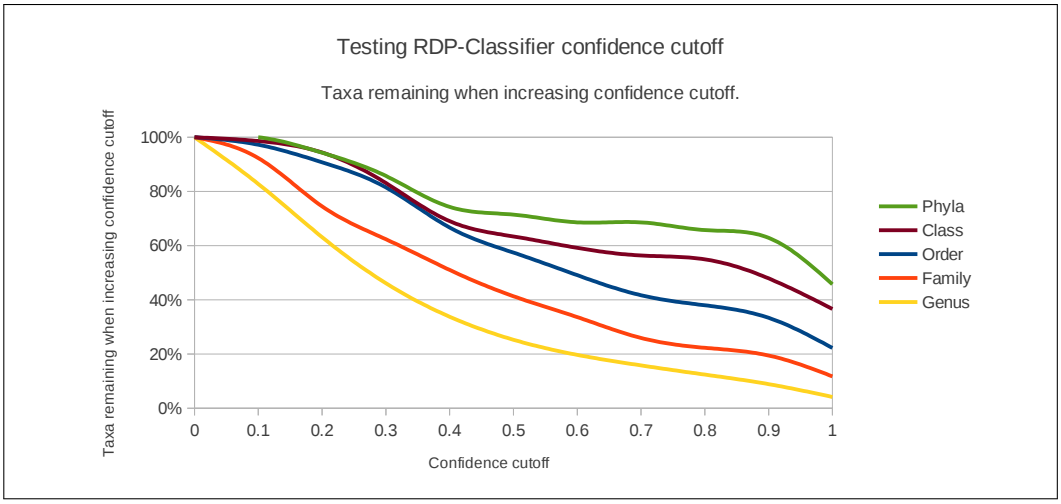


Figure 1.2: Example of parameter tuning: analysis of remaining taxa for varying cutoff parameter to RDP-Classifier application. Figure from [17].

Doing exhaustive parameter studies on pipelines with many stages and parameters is computationally expensive and probably not practical. However, we believe having system support that make it easy and fast (more interactive) for the user to try out different values may lead in scientific discoveries that would otherwise have been missed.

A related issue is that of data provenance. Reproducibility of research re-

sults are at the center of the scientific method. Studies have shown that bioinformatics-based research can be difficult to reproduce[13]. To quote from that study:

> *The main reason for failure to reproduce was data unavailability, and discrepancies were mostly due to incomplete data annotation or specification of data processing and analysis.*

The *specification of data processing and analysis* for a metagenomic pipeline can be challenging due to the complexity of the processing:

- The pipelines contain many pipeline stages.
- The application used in each stage can exist in multiple versions, giving different output.
- Each application typically take parameters that can have different values.
- Pipeline stages are likely to contain custom made applications, such as filtering scripts and data transformation scripts that may not be publicly available and may not be under version control.
- The data is stored separate from the processing specification, putting a bookkeeping burden on the researcher.

This gets more challenging when parameter tuning is involved, since the researcher need to keep track of multiple datasets with corresponding configurations.

With reference to the above discussion, there is a need for infrastructure systems that can support easy configuration of pipelines, recording of provenance data and reliable data storage.

1.2.3 Approaches

Typically, data is inspected at the end of the pipeline in the form of a visualization or a statistical analysis. To support easy parameter tuning, it is therefore important that the delay from a parameter change until new data show up at the end of the pipeline, is a short as possible.

By storing intermediate data between the stages, only downstream stages need to be recomputed after a parameter change. This reduces the number of recomputations and improves the response time to a parameter change. One problem with this approach is that for the early stages in the pipeline, execution time will be close to that of re-executing the whole pipeline.

Another problem is that if the user, after trying new parameters, decides to revert back to some previous setting, the computations must be redone. To avoid this, all versions of the intermediate data can be stored. Increasing interactivity using this technique therefore incur storage overhead and data management issues that must be handled.

Another way of improving the pipeline response time is to use a subset of the full input dataset. By sampling the input dataset, and then executing the pipeline on the sample, execution times can be significantly reduced. For example, the commonly used BLAST application does a similarity search against a database for each query sequence in the input data[1]. A BLAST stage operating on a 1% sample of the full input would then require 1% of the execution time of a similar stage operating on the full input.

Sampling is not only useful for parameter tuning: a sample of the dataset can contain enough information for scientific discoveries. Figure 1.3 is another example of current work being done (unpublished, Kjærner-Semb, E., Dept. of Chemistry, University of Tromsø[17]). It shows the number of taxa found versus the amount of data processed in a metagenomic dataset. In this example, about 50% of genera are discovered after processing 4 million reads (one fourth of the dataset). About 75% of orders are discovered after processing 2 million reads (one eighth of the dataset). This shows that samples of a metagenomic dataset can provide biological insight.

A third way of improving response time is to use online processing, where each data point is processed through the whole pipeline before the next data point is processed, as opposed to batch processing, where all data points are processed through one stage before the next stage is started. This technique is particularly well suited to data parallel execution. It can also be combined with the two previous techniques (downstream recomputing, and sampling).

For complex metagenomic pipelines, it is important that the complete configuration of the pipeline is recorded, so that results can be reproduced. Some guidelines for achieving reproducible computational research are discussed in [25].

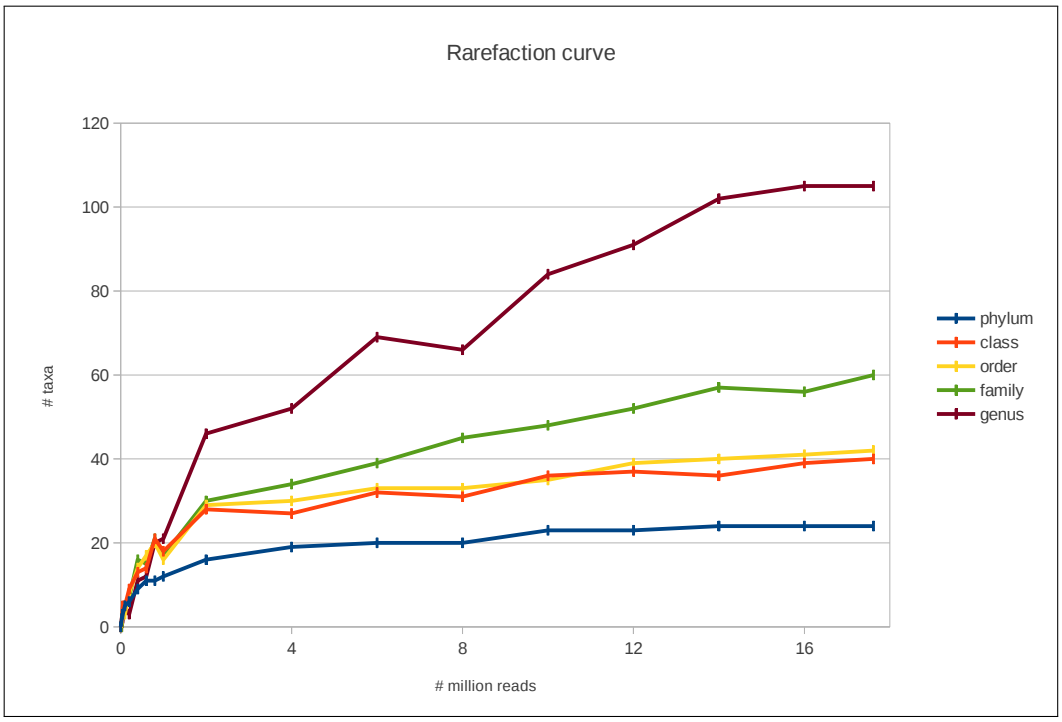


Figure 1.3: Number of taxa found vs. number of reads processed from a metagenomic sample. Figure from [17].

1.3 Big Data Analysis

Mario use the storage and processing capacity of a cluster of computers with Hadoop, Hadoop Distributed File System (HDFS) and HBase installed.

HDFS is a open-source distributed file system that provides reliable storage of petabyte-sized datasets. HDFS is inspired by Google File System (GFS)[8]. Hadoop[1] is a system for high throughput, data-parallel processing of data stored in HDFS. Hadoop is an open-source implementation of Google's MapReduce system[3]. Several systems based on Hadoop and HDFS provide additional capabilities for handling petabyte-scale datasets. One of these is HBase[2], a distributed, random access storage system for structured data, modelled after Google's Bigtable system[2]. The Mario system, presented in this work, use HBase extensively.

Other systems in the Hadoop ecosystem include:

- Hive[3], a data warehousing system.
- Pig[4], a system that provides an easy to use, SQL-like interface to MapReduce.
- Mahout[5], a system providing implementations of machine learning algorithms that can be executed on Hadoop.

Data in bioinformatics are typically multi-dimensional, heterogeneous and noisy, in contrast to the text-based web-page data that Hadoop and HBase was originally designed for. Nevertheless, systems from the Hadoop ecosystem are being increasingly used for bioinformatics due to their scalability to large datasets[29]. There is however, no Hadoop-based system that provides interactivity and iterative computations for biological data.

Mario use HBase to improve on some of the weak points of the piplines discussed in section 1.2.

[1] `http://hadoop.apache.org`
[2] `http://hbase.apache.org/`
[3] `http://hive.apache.org/`
[4] `http://pig.apache.org/`
[5] `http://mahout.apache.org/`

1.4 Mario

We believe that a system for processing and analyzing metagenomic datasets should satisfy the following requirements:

1. Interactivity. A system with response time from user input until results start showing up on screen of less than 100ms, will appear to the user as responding instantaneously. If the response time is longer than 10 seconds, the user's attention may be lost [19]. Since a metagenomics pipeline can contain a variable number of stages containing applications where even the smallest input can take seconds to compute, it is difficult to define general response time requirements. Instead, this requirement will be stated in terms of latency for doing a null operation on input data at the finest granularity (a single nucleotide sequence). This requirement is set at 100ms.
2. Flexibility. The main goal is to make it easy to tune parameters to applications used in pipelines.
3. Generality. The system should make it easy to replace stages with a variety of metagenomic pipelines and tools.
4. Scalability. The system should scale to meet the demands of processing upcoming petabyte-scale datasets.
5. Ease-of-use. The system should be easy to adapt to existing pipelines, since it is not practical to make changes to pipeline tool code. The system should also handle input and output from each stage regardless of the data format the tools use.
6. Provenance. Data provenance support should be an integrated part of the system.

To our knowledge, no existing system fulfill all these requirements. Hadoop/MapReduce[3] does not satisfy the interactivity requirement, since even a null operation can take tens of seconds to complete. Apache Pig is an interface to Hadoop and is therefore not interactive. GeStore focuses on incremental updates of metadata, and does not satisfy the interactivity requirement. Galaxy and Taverna are workflow managers that provides provenance and an easy to use interface to applications, but does not provide interactivity. Spark has, to our knowledge, not been integrated with bioinformatics tools. More detailed descriptions of these systems are given in chapter 5.

Based on these requirements, the Mario system is proposed, which fulfill the requirements as follows:

1. Interactivity is achieved using iterative processing, sampling, and storage of intermediate data.
2. Flexibility is achieved using an interface where pipeline configurations can be changed during processing, by sending messages with updated configuration to the system.
3. Generality is achieved by using existing, unmodified tools in the pipeline stages.
4. Scalability is achieved using a parallel shared-nothing architecture for computations and a highly scalable storage system.
5. Ease-of-use is achieved by using existing, unmodified tools in the pipeline stages, and through the use of a storage model that is agnostic to the data types used by the tools.
6. Data provenance is achieved by storing the complete configuration of the pipeline, including versions of applications used in each stage, and providing a mapping between data and configuration.

1.5 Contributions

The contributions of this work are:

1. An analysis of the METApipe metagenomics pipeline, including the METAREP visualization and analysis frontend, to better understand how to make real-world bioinformatics pipelines more interactive.
2. An approach for, and an implementation, of a bioinformatics pipeline system, Mario, that provide iterative and interactive processing, and has support for data provenance.
3. An experimental evaluation to determine whether HBase provides the required features and performance to be used as a storage system for interactive processing of biological data.
4. An experimental evaluation of the Mario system, demonstrating that it can achieve interactive performance for processing of biological data.

1.6 Conclusion

The evaluation of Mario and HBase indicate that that Mario add considerably less than 100 milliseconds to the latency of processing one item of data. This low latency, combined with Mario's storage of versioned intermediate data enables easy parameter tuning. Mario also have high throughput, making it suitable for processing large datasets. In addition to this, Mario offer integrated data provenance, with detailed pipeline configurations being stored in the system, and associated with the data.

The evaluation of Mario demonstrate that it can be used to achieve more interactivity in the configuration of pipelines for processing biological data. We believe that biology researchers can take advantage of this interactivity to perform better parameter tuning, which may lead to more accurate analyses, and ultimately to new scientific discoveries.

Chapter 2

Mario Architecture

Based on analysis of METApipe and other bioinformatics analysis pipelines, the following assumptions are believed to be valid for many use cases, and form the basis for for the architecture and design of Mario:

1. Input data can be split into parts with fine granularity.

2. No intermediate pipeline stage requires access to the complete input data.

3. There is enough storage to hold the intermediate data.

The first two assumptions allow the dataset to be processed iteratively with inspection of output as the computation proceeds. This is the main key to achieving interactivity in the configuration of the pipeline. The third assumptions allow intermediate data to be stored, thereby reducing recomputation after configuration changes.

An overview of the Mario architecture is given in figure 2.1. It consists of four tiers: storage, logic/computation, the web server and the client/UI. The system will normally be installed on a cluster of computers, with the master process at the cluster frontend, and the workers at the compute nodes of the cluster. Mario will normally be colocated with an HBase installation that has the HBase master at the frontend and the HBase region servers at the compute nodes. The web server and the MySQL server can be located on the cluster frontend, or on separate computers.

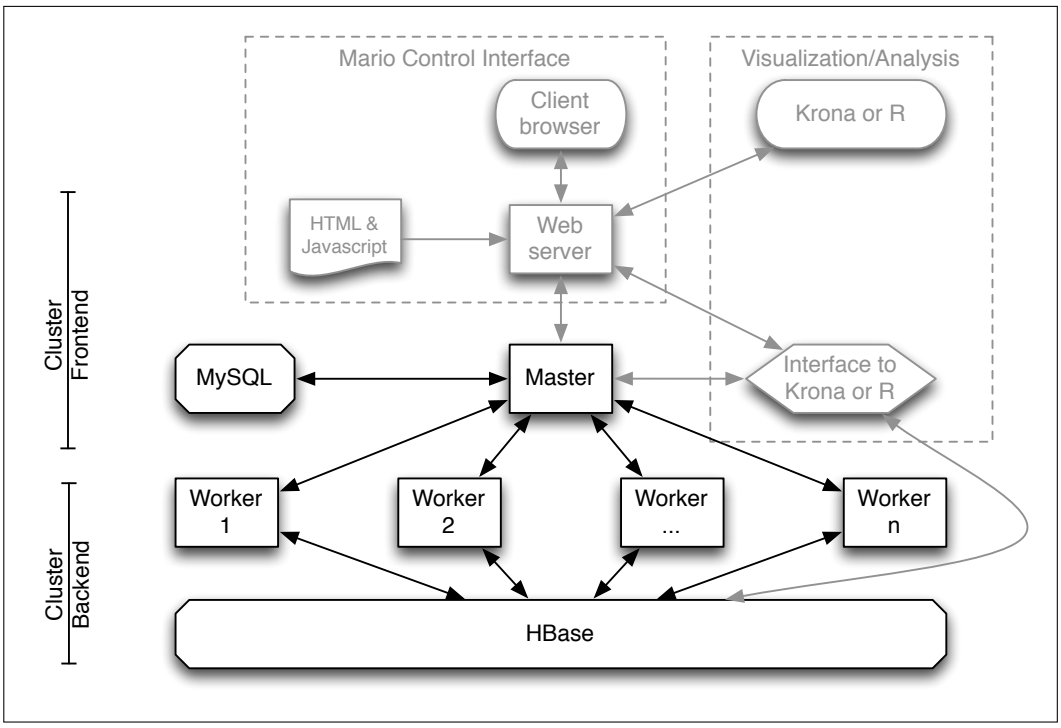


Figure 2.1: Architecture of Mario. Greyed out parts are not implemented in prototype.

The user control Mario via a web interface, where the pipline is configured, dataset selected, and the computations can be started, paused and stopped. The configurations and control messages are sent as JSON messages from the client browser to the web server, which forward them to the Mario master server. The master server transmit task messages to the Mario worker processes and receive notifications when work is completed. The workers retrieve input data from HBase, run it through the pipeline, and write the results to HBase.

A visualization/analysis interface retrieve results from HBase, either periodically, or when notified of the presence of new results by the Mario master, and formats the results for the visualization or analysis system being used. For example, if Krona[23] is used for visualization, the interface will create an XML-file with hierachies of organisms that can be visualized. The web server transfer the XML-file to the web browser running Krona (the same browser that are used for controlling Mario).

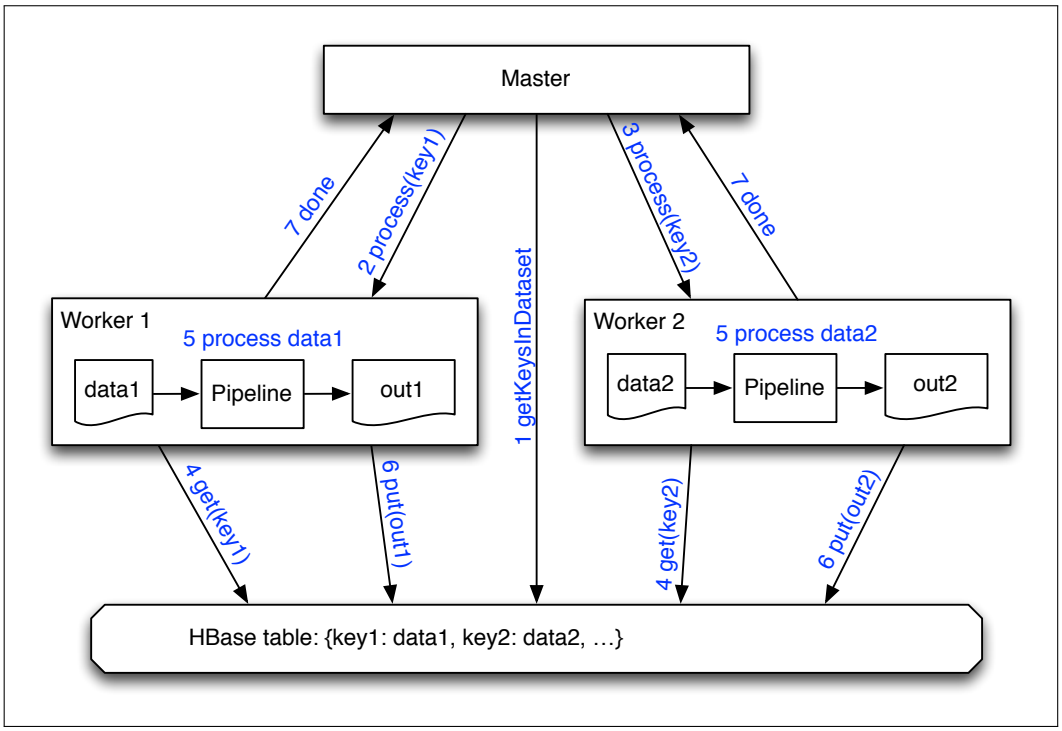


Figure 2.2: Independent parallel processing of data by two workers. Blue labels show sequence of events after a user starts Mario.

All workers operate independent of each other, processing separate parts of the dataset in parallel. Figure 2.2 show the sequence of events when the Mario

master schedule tasks to workers, who then process the data independently in parllel: The master first request a key iterator from HBase. Iterating over the keys, the master send task messages to each worker, containing the key. Each worker then retrieve the data associated with the key, and process this data. After processing, the worker put the data to HBase and send a notification to the master that processing is completed.

2.1 Use Case

To analyze a dataset using Mario, the user would first insert the input data into HBase. She would then define the pipeline operations that will operate on the data. This is done by, for each pipeline stage, specifying the application to execute, the version of the application, and the parameters to pass to the application. This pipeline configuration can be entered via a web interface such as Galaxy[9] or Taverna[21] or from a script that sends the configuration to the Mario master. As part of the configuration, the option to sample the dataset can be selected together with the sample size. The user then start the initial computation. As the computation proceeds, she might want to change the parameters or application used in a stage. This is done by sending an updated configuration message to the master, which will start scheduling work with the new configuration. If the new configuration is not satisfactory, the previous configuration can be restarted, and old data is restored without recomputing.

2.2 Storage Layer

The primary component of the storage layer is an HBase installation. HBase is used to store input data, intermediate data and output data. Intermediate and output data can be stored in multiple versions resulting from the use of different settings to pipeline stages. HBase was selected as storage backend due to its low-latency random read and write capability (chapter 4), its ability to efficiently store sparse data, and its ability to easily scale to store current and future large biological datasets on clusters used by bioinformatics research groups. When a metagenomics project is completed the intermediate data can be deleted and a major compaction performed(ref. section 3.1) to reduce the long time storage requirements.

Also part of the storage layer is a MySQL database. This database has three uses in Mario: First, it is used to store the different settings used in each of the pipeline stages. This provides access to the different pipeline configurations, including parameters to each stage, used for computing the intermediate and output data stored in HBase. This represents a history of configurations, so that a Mario user can revert to previous configurations and benefit from previously computed results. Second, the database is used to store metadata about datasets stored in Mario's HBase tables. Third, the database is used to store information about available tools that can be used in each pipeline stage, such as version and allowed parameters.

2.3 Logic and Computation Layer

This layer contains a single master server, and multiple worker processes. The master is controlled by the user through the web client. When starting a job, the master will distribute work to the workers. It does this by providing each worker with the current configuration of the pipeline and the HBase row key of the data to be processed. If desired, the master can also query HBase for the location of the HBase region server responsible for the key, and assign the key to a Mario worker located on the same server. This will improve data locality and potentially reduce network traffic.

The master retrieves the row keys from HBase, but does not retrieve the data stored under each key, or perform any processing. It is therefore lightly loaded. If sampling is selected, the sample is stored in memory as a list of row keys. This sample is the main source of memory usage in the master. Assuming 20 byte key length, a large sample of four million keys will only consume approximately 80MB of memory.

The worker processes wait for messages from the master server. When such a message is received, the worker retrieves the relevant data from HBase. This data is then processed through all the stages of the pipeline, with intermediate and final output inserted back into HBase. When a worker has completed its work, a message is sent to the master. This enables the master to adjust work distribution to the capacity of the workers. This also makes it easy to notify the METAREP/Krona interface that work has been completed. The worker processes can be expected to be CPU and memory intensive, due to the applications used in the pipeline stages.

Communications between the master server and workers, web server and visualization/analysis interface is performed using the ZeroMQ library[1], which is a low-latency, high-performance asynchronous messaging library. ZeroMQ provides a brokerless communication architecture with automatic handling of transfer and buffering of messages.

2.4 Web Server

The web server serves the Mario control application to the users web browser, and forwards requests from this application to the master server. The web server is also used to serve data generated by the visualization and analysis interface to the visualization or analysis application being used.

2.5 Visualization and Analysis

To integrate Mario with a visualization system such as Krona, an interface must be implemented. This interface simply generates the data required by the visualization system based on the data available. For Krona, this involves generating an XML-file of the organism hierarchies found in the data. This interface can be implemented in any programming language that has ZeroMQ bindings and can access data from HBase, such as Python or Perl, both of which are popular in the bioinformatics community.

To perform analysis, the interface could be implemented as part of an analysis script in R, since R has ZeroMQ bindings and can access HBase.

[1] `http://zeromq.org/`

Chapter 3

Mario Design and Implementation

This chapter begins with a detailed description of the HBase storage system used by Mario. This is necessary to understand the performance characteristics of Mario. The design of the Mario system is then presented with some detail.

3.1 HBase

HBase is an open-source, distributed storage system for structured data, based on Google's Bigtable[2]. It has a single HBase master server, and multiple region servers. These servers are located on a cluster of computers and are often co-located with other systems. In addition to the master and the region servers, HBase use the Apache ZooKeeper[12] system for tasks such as bootstrapping, server discovery and server failure detection. Data is primarily stored in HDFS (similar to GFS), but can HBase can also be configured to use Amazon S3.

Mario store data in HBase tables. A table consists of rows that are identified by row keys. Each row has cells containing data. The cells are identified uniquely by a row key consisting of column family, a column and a cell version(a timestamp by default). Thus, for a given table, a cell in a specific row is identified by the following vector: (row key, column family, column qualifier, cell version). The first three components are strings, and the cell

version/timestamp is a long. The key is stored together with each cell in a byte array known as a KeyValue(figure 3.1). The KeyValues are stored in immutable HFiles, lexicographically ordered by row key. This design makes HBase ideal for efficient storage of sparse data, which is data with many columns, most of which are empty. This is precisely the storage characteristics of Mario.

Another consequence of this design is that columns can be added dynamically at runtime. Mario use dynamically generated column names to provide a mapping between the data in the column and the pipeline configuration used to generate that data.

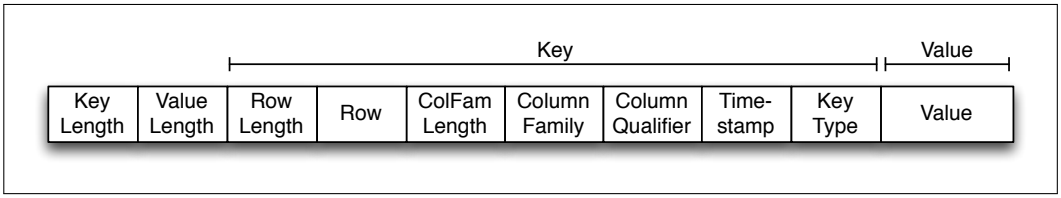


Figure 3.1: KeyValue format. Figure based on figure 8-7 in [7].

A disadvantage of storing the key with every data cell is that for very small cells, the key can represents a large part of the total data. Whether this is the case for Mario depends on the applications used in each stage of the pipeline. Some of this disadvantage is mitigated with compression, but it is still important to keep the row key, column family name and column qualifier as short as possible to reduce overhead on small cells.

The HBase master server decides which region server shall handle which region, handles creation and deletion of tables, load balancing, and also handle region server failures. HBase clients does not communicate with the master.

The HBase region servers are responsible for reading and writing data. Each region server is responsible for zero or more regions, each containing a given range of keys. HBase metadata is stored in two special HBase tables: the -ROOT- table contain the locations of the .META. table, which contain the locations of the different regions. When a client send a (get) request to HBase for the first time, the following events take place (figure 3.2): 1. the client sends a request to ZooKeeper for the location of the -ROOT- table. 2. the client sends a request to the region server holding the -ROOT- table for the location of the .META. table. 3. the client sends a request to the region server holding the relevant part of the .META. table for the location of the KeyValue. 4. the client sends a request to the region server holding the

KeyValue. All results of these metadata requests are cached by the client, to minimise subsequent lookups.

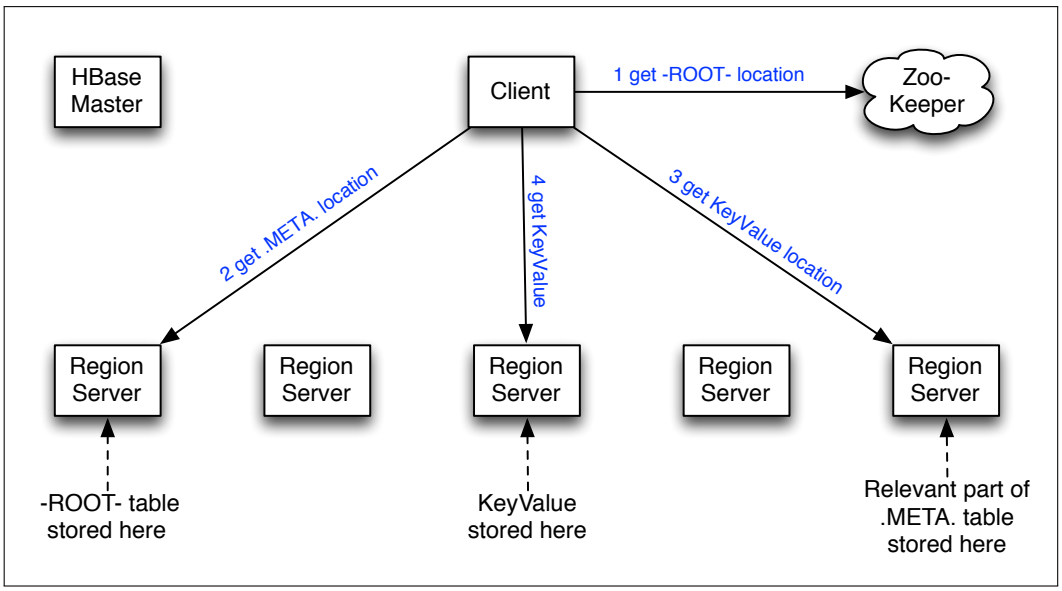


Figure 3.2: HBase client request with empty client cache.

A region server can contain multiple regions. Each of these are represented as HRegion instances (figure 3.3) containing one Store instance for each column family and HBase table. Each store has a MemStore, and one or more StoreFiles, which are wrappers around an HFile. Each column family can be configured to use compression, which will generally increase performance, due to reduced disk access. Each column family can also be configured to use Bloom filters, which can be used to exclude files from searching for a given row key, thereby increasing read performance.

When a region server receives a put request, it passes the request to the relevant HRegion object responsible for the key range the put belongs to. The HRegion object first write the data to the Write-Ahead Log (WAL). The WAL store the request in HLog files in case the server fails. To ensure consistency in case of disk failures the log entry is synchronously written to a configurable number (default 3) of replicas on different servers. If a slight reduction in reliability is acceptable, the log replication can be performed asynchronously for better performance. Although not recommended, the WAL can also be disabled completely, resulting in data loss if the server fails before or during a write.

After updating the WAL, the HRegion instance then it inserts the data into

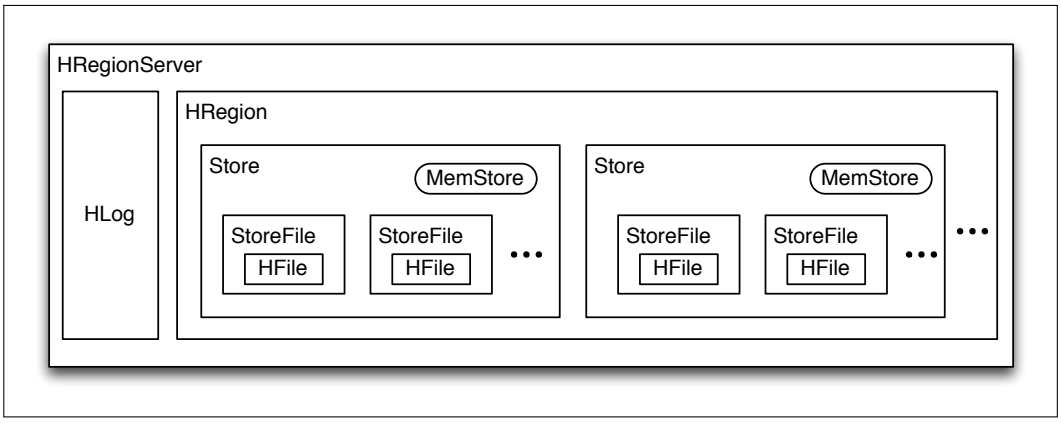


Figure 3.3: HBase region server design. Figure based on figure 8-3 in [7].

the in-memory MemStore. If the MemStore is full, it is flushed to disk, creating a new StoreFile. Since data is generally not inserted in lexicographic order, multiple StoreFiles is created, which internally are in lexicographic order. These are periodically cleaned up by merging the latest StoreFiles together into a larger StoreFile. This process is called a minor compaction. Minor compactions ignore StoreFiles larger than configurable maximum size. Periodically (default: every 24 hours) a major compaction is performed. These merge all the StoreFiles into one large StoreFile, at the same time removing data that have been marked for deletion.

When the size of the largest StoreFile exceeds some configurable limit, a region split is triggered. This splits the region key range in the middle, thereby creating two new regions. After updating the .META. tables these new regions are served like other regions. If the load on the region server is high, the master server can move some regions to other region servers for load balancing.

When a region server receives a get request, each associated cell of data can be located in several HFiles or in the memstore. When a get is performed, an exclusion check is first performed to exclude HFiles from search. If the get includes a timestamp, all HFiles that were written earlier than the timestamp can be ignored. If the optional Bloom filter is used, it is also queried to exclude files not containing the key. All included files are then scanned for the requested key. By using an index at the end of the StoreFiles, this scanning can be performed fast.

HDFS can store data reliably by taking advantage of HDFS replication. HDFS is by default set to replicate data to 3 other nodes. HBase can also

be configured to replicate the whole HBase cluster to other slave cluster at geographically distant locations.

3.2 Mario Storage

3.2.1 HBase

The HBase storage system form the backbone of Mario. It handles the following tasks:

1. Store the input data to Mario. This data is loaded using an external loader script.
2. Store intermediate and output data from the pipeline, in a way such that the pipeline configuration used to process the data can be inferred from the data.
3. Provide random access to the data with latencies that make it possible to meet the requirements outlined in chapter 1.

The HBase schema used is shown in figure 3.4. Before starting execution, input data is loaded and stored in the *in* column family. Input data is stored as a key-value pair, where the key can be any identifier that uniquely identifies the value. For example, the key can be the line number in the input file that contains the data value, or it can be a sequence ID if the input is a FASTA file.

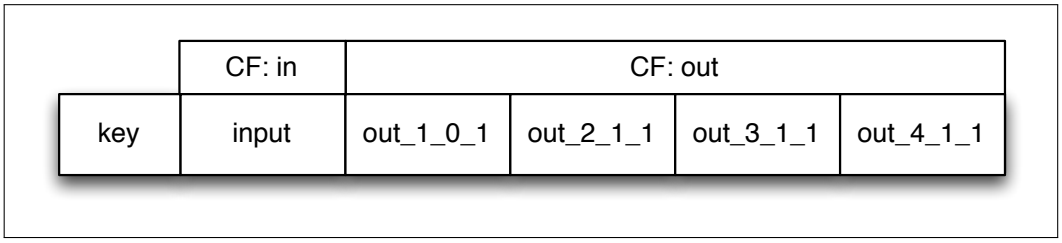


Figure 3.4: HBase schema

Output from pipeline stages are stored in the *out* column family. When a pipeline is configured, each stage is given a version number of 1. The version numbers are used in the column names of the HBase columns that store

the data. The configuration used to process the data in a given column is identified in the following way: a column name of *out_3_2_1* means that the data contained in the column is the output from the 3rd stage of the pipeline, using version 1 of the stage and based on input from version 2 of the parent stage.

Figure 3.5 show an example of a three stage pipeline where the stages have been modified by the user three times. The top row show the column names for the initial versions of each stage of the pipeline. When a stage is changed, by setting a different parameter or using a different application, the result can be a version tree as shown. The second branch in the version tree in figure 3.5 is the result of changing the first stage of the pipeline, but leaving the other two stages unchanged. Even if only the first stage is changed, the version numbers of the downstream stages must be incremented to create columns for storing the data based on the output from the new first stage. In the same way, the lower branch in figure 3.5 is the result of changing the second stage of the pipeline.

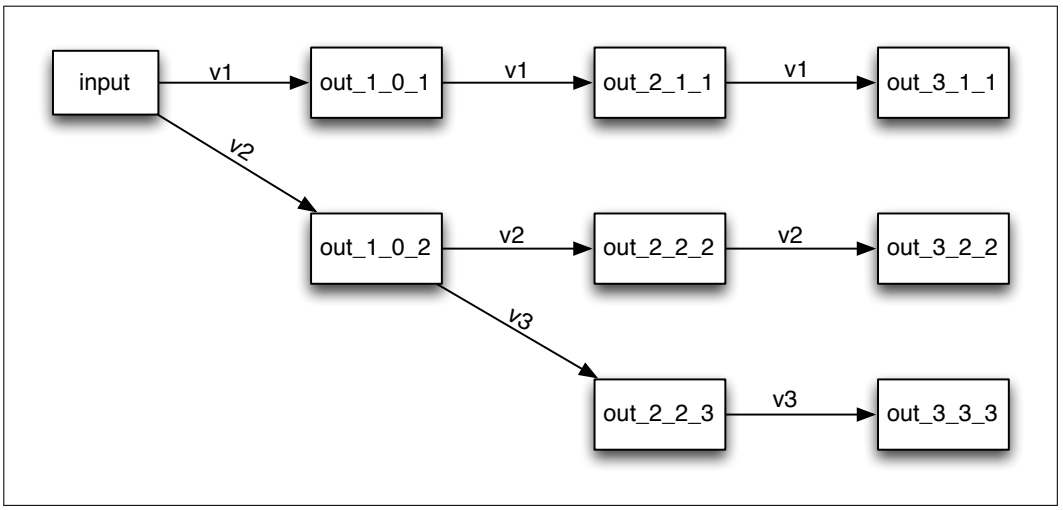


Figure 3.5: Data versions with HBase column names

The HBase table resulting from the previous example will have eight columns in the *out* column family. For a long running computation, it is reasonable to believe that most parameter tuning will occur during the first minutes of the computation. This implies that after running the whole dataset through the pipeline, most columns will be empty for most of the rows. HBase is ideally suited for storage of sparse data such as this ref. section 3.1.

3.2.2 MySQL

The pipeline configuration is stored in a MySQL database. It consists of two tables. The *stageversion* table contains the configuration of each stage in the pipeline. The definition of this table is shown in table 3.1. The `stage` field is a foreign key into the *stage* table, and indicates which stage the stageversion belongs to. The `sequence_number` field hold the sequence number of the stage, and the `parent` field is a foreign key into the *stageversion* table itself, pointing to the parent stageversion. Together, these are used to store the data version tree show in figure 3.5. The `data` field hold a string serialization of the complete stage configuration, including the command to execute, version of the application, and values of parameters. Since stages need to be compared to determine if a stage has changed, and the `data` string can be long, a `hash` field store a 32 bit hashcode of the `data` string. If two stages have different hash, the stages are different. If two stages have similar hash, the `data` field is compared for similarity.

Table 3.1: Schema of *stageversion* table

Field	Type	Null	Key	Defaul	Extra
id	int(10) unsigned	NO	PRI	NULL	auto_increment
stage	int(10) unsigned	NO		NULL	
parent	int(10) unsigned	NO		NULL	
sequence_number	int(10) unsigned)	NO		NULL	
hash	int(11)	NO		NULL	
data	text	NO		NULL	

The *stage* table contains name, description and the current version number of each stage, as shown in table 3.2. By querying this table the system can determine the latest configuration set by the user, and thereby know which HBase columns contain valid data.

Table 3.2: Schema of *stage* table

Field	Type	Null	Key	Default	Extra
id	int(10 unsigned	NO	PRI	NULL	auto_increment
name	varchar(255)	NO			
current_value	int(10) unsigned	NO		1	
description	text	NO			

3.3 Mario Master Server

Mario has a single master server. The master server handle the following tasks:

- Communication with the frontend control interface.
- Storage of pipeline configurations.
- Pushing notifications of completed work to the visualization or analysis interface.
- Retrieving dataset keys from HBase, and schedule work by transmitting task messages containing keys to the Mario workers.

The master server is implemented using two threads; a *master* thread listen for control messages and take appropriate actions, and a task *ventilator* thread handle distribution of tasks to the workers.

In the Mario prototype, the web interface is not completed, and the web server has only been used for proof-of-concept testing. For the evaluation of Mario, messages from the web server are simulated using a Python-script. These `ClientMasterMessage` messages contain a JSON object with an optional command and an optional pipeline configuration. The following Python function show the structure of the JSON objects, for one of the dummy pipelines used in the evaluation of Mario:

```
def experiment():
  mongrel = context.socket(zmq.PUSH)
  mongrel.bind('tcp://*:20003')
  time.sleep(1)
  stage1 = {'command': 'cat', 'version': '1.0', 'parameterList': []}
  stage2 = {'command': 'cat', 'version': '0.2', 'parameterList': []}
  stage3 = {'command': 'cat', 'version': '3.1', 'parameterList': []}
  stage4 = {'command': 'cat', 'version': '3.1', 'parameterList': []}
  desc = {'command': 'START',
          'pipelineDescription': {'stages': [stage1, stage2,
                                             stage3, stage4]}}
  msg = json.dumps(desc)
  mongrel.send(msg)
```

This message contain a four-stage pipeline configuration, having the Linux *cat* application in each stage. Each stage contain a command, a version number (random in the example), and a parameter list (empty in the example). Together, the stages represent a complete pipeline. The example also include a *START* command, that tell the Mario master to start processing when this message is received.

The master thread contain a version manager that is responsible for storing pipeline configurations in the MySQL database and maintaining the pipeline version numbers as pipeline configurations are updated. The version manager contain all logic related to the construction of the version tree, which is used to decide which version numbers to change when an updated pipeline configuration is received. Furthermore, the version manager maintain the Master server's list of currently valid version numbers. This list is distributed with each task message to a worker. If a worker find that its own version numbers are outdated, meaning that the worker has an outdated pipeline configuration, the worker will request a new pipeline configuration description from the master server. This configuration description is created by the version manager.

The other thread in the master server is a task ventilator that distribute task messages to the Mario workers. When starting up the master server, the ventilator will first wait for synchronization messages from the expected number of workers. When the workers are connected, the ventilator opens a connection the HBase table containing the input data. It then perform a *scan* over the keys. For each key, it construct a `TaskMessage` object containing the key (a string), and an array of version numbers (of type Long). This object is serialized and transmitted to the worker.

The ventilator throttle the distribution of tasks to the workers. This is done by sending a batch of task messages to each worker, and then waiting for the workers to complete processing, before sending the next batch. The batch number is configurable at compile time, but for most of the development and evaluation a number of two has been used. This mean that each worker receive two task messages that must be processed before before being assigned more tasks.

The ventilator thread is also responsible performing the optional sampling. The sampling is performed using the algorithm described in section 3.5. The sample is stored in memory as an array of keys. After the sampling, the array is used as the source for generating task messages to the workers.

3.4 Mario Worker Server

Mario use one or more worker servers, ideally one for each available compute node in the cluster. The task of the workers is to listen for incoming task messages and, when one is received, retrieve the specified data from HBase and process it through the pipeline, finally writing the intermediate and output data to HBase.

The worker use two threads. The *worker* thread listen for incoming control and task messages, and take appropriate actions when one is received. The other thread is a *TaskRunner* thread that perform the processing.

When a worker thread receive a task message, it checks if it has a valid pipeline configuration, by comparing the version numbers in the task message with the version numbers of its own pipeline configurations. If the pipeline configuration is outdated, the worker send a request to the Mario master server for an updated configuration. This configuration is then stored in memory. After checking the versions numbers, the task message is passed to the *TaskRunner* thread for processing.

To execute a single stage in a pipeline, the TaskRunner thread retrieve the data from the relevant HBase row and write it to a temporary file. Executing the stage will result in an output file which is then put into HBase (figure 3.6). For pipeline stages where the application can *stdin* and *stdout* for reading and writing data, the relevant temporary files will be automatically piped to/from the application. If the application need the filenames of input and output files, the position of these names must be specified in the command that the stage will execute. This is done by inserting special tokens in the command at the right position, for example a GCC command would look like this: "gcc -o {out} {in}". Here the {out} and {in} tokens will be automatically replaced by the relevant temporary files before the command is executed.

Before processing each stage in the pipeline, the TaskRunner generate the column names for the input and output data. It then checks which columns exist in the row retrieved from HBase. If the column already exist in the row, meaning that the result has been computed previously, the pipeline stage is skipped.

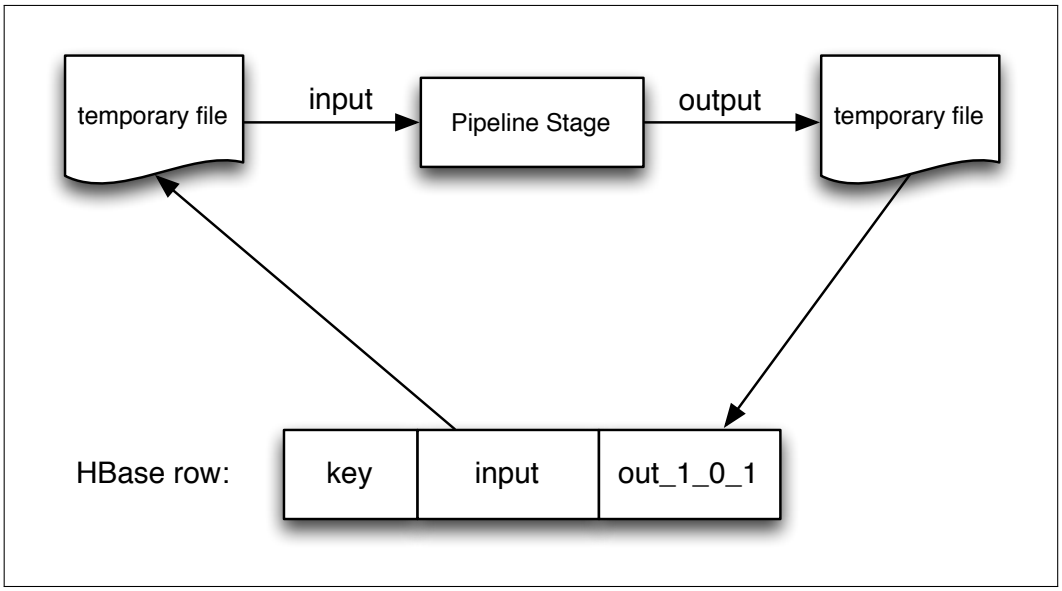


Figure 3.6: Use of temporary files

3.5 Reservoir Sampling

Reservoir sampling is a collection of algorithms that produce a random sample of elements from a stream, without knowing the number of elements in the stream beforehand. Mario perform the sampling using algorithm 1. It can be shown that using this algorithm, each element in the stream has equal probability of being in the sample. The algorithm does one pass through the stream, and requires the generation of one random number per element. The *single-pass property* of reservoir sampling make this technique well suited for sampling large datasets where performance is I/O limited. The algorithm can be improved to require less random number generation. Algorithm 1 and improvements are detailed in [31].

In Mario, the sampling is done by the master when processing the dataset for the first time. The sample is stored as an array of IDs in memory. It is not persisted to disk.

Algorithm 1 samples uniformly from the input stream. If the sample is used for analysis, for example because analysing the whole dataset in intractable, there is a risk of weak signals in the data being lost in the sampling process. Research is being done into weighted sampling methods that increase the chance of sampling data points that are deemed important[27]. Weighted

Algorithm 1 Uniform reservoir sampling

```
Require: |A| ≤ |s|
  function SAMPLE(s, A)                    ▷ Sample from stream s into array A
    i ← 0
    while s.hasNext() do
      e ← s.next()
      if i < |A| then
        A[i] ← e
      else
        r ← random(0, i − 1)                              ▷ Inclusive range
        if r < |A| then
          A[r] ← e
        end if
      end if
      i ← i + 1
    end while
  end function
```

sampling can also be implemented using single pass reservoir methods[5].

3.6 Scheduling

In the Mario prototype, the master process schedules work using a round robin scheduler. This is done for simplicity, but will result in reduced performance on realistic clusters where the performance can be expected to vary between nodes. Also, round robin scheduling precludes the possibility of scheduling work close to the data (data locality).

To synchronize the transmission of work messages from the master with the work done by the worker, the master will transmit a certain number of messages to each worker, before pausing and waiting for completion messages. When enough completion messages are received, the master send another batch. The number of messages sent in each batch is configurable at compile time as a multiple of the number of workers. Sending multiple messages to each worker can improve performance, since a worker can start a new task immediately after finishing the previous task without having to wait for the master to perform a new scheduling.

3.7 Visualization and Analysis Interface

The visualization and analysis interface is a component of Mario that must be custom made to support the visualization and analysis frontend used by the researcher. In MapReduce terminology, this interface is a *Reducer*. Its task is to aggregate the available results, and present the data in a format that the frontend can use. For a Krona[23] visualization, this would involve generating an XML-file containing hierarchical data. Due to time constraints, no such interface is yet implemented in Mario.

3.8 Technologies

The backend system is implemented using the Java programming language. This choice is largely pragmatic: the language is easy to use because of the native interfaces offered by the Hadoop ecosystem.

For communications, the *ZeroMQ* library is used[1]. ZeroMQ is a low-latency, high-performance asynchronous messaging library with origins in the finance industry. It provides a socket-like interface, and has features that make it easy to implement common communications pattern such as *publish - subscribe*, *push - pull* and *request - response*. ZeroMQ is used for communications because it is easier to use than regular sockets, and also because it makes it easy to implement components of the system in any language that has bindings to ZeroMQ.

The web server used for Mario is *Mongrel2*[2]. Mongrel2 is used because it is designed for easy communication with ZeroMQ backends. The *AngularJS*[3] framework is used for Mario's web control interface. AngularJS is a JavaScript based MVC framework for web applications. One of the principle features is a binding between model and view, so that an update of a model is immediately reflected in an updated view, and vice versa. In the Mario prototype, the web interface is not completed, and the web server has only been used for proof-of-concept testing. For the evaluation of Mario, messages from the web server are simulated using a Python-script.

[1] `http://zeromq.org/`
[2] `http://mongrel2.org/`
[3] `http://angularjs.org/`

Chapter 4

Evaluation

The goal of the experimental evaluation is to 1. validate the suitability of HBase as a storage backend for a iterative, interactive system, and 2. validate the architecture and the design choices made for the Mario prototype.

To do this, latency, throughput and resource usage is measured and discussed.

Since interactivity is the primary goal of Mario, *latency* is the most important metric to evaluate. Latency can be defined as the time from an input is made to the system, until some result or consequence of that input is visible to the user. For Mario, latency can be defined as the time from a computation is started or a pipeline configuration change is made, until the first results are ready to be visualized.

Interactivity, and therefore latency, is most important during pipeline development, debugging and tuning. After that point, throughput becomes more important, especially if the user want to process the complete dataset (as opposed to using sampling). Throughput is defined as the amount of data that can be processed by the system per unit time. Throughput is therefore the second most important metric to evaluate.

Mario's latency and throughput is dependent on several factors:

- The number of stages in the pipeline. More stages in the pipeline will result in more intermediate input and output files being created by the workers, and will therefore reduce throughput and increase latency.
- The computations being done in each stage of the pipeline, which is

related to the applications used in the stages, will directly affect both latency and throughput.

- The granularity of input data to the pipeline. Does the input data consist of a few large items, or many small items? The former will result in higher latencies, but might improve throughput. The latter will result in lower latencies, but might reduce throughput. The total amount of input data is irrelevant to both latency and throughput, but will naturally affect total execution time.
- Scheduling. By scheduling a worker to use data from a HBase region server on the same node, data transfer over the network can be reduced, and both throughput and latency improved. The Mario prototype does not support location aware scheduling.

Since Mario essentially is an orchestrator that provide unmodified bioinformatics applications with data and take care of the output, it is important that Mario leave as much as possible of the hardware resources available for use by the applications. These resources include CPU, network, memory and disk usage.

The experiments consist of two parts: the first part is an evaluation of the latencies that can be expected when using HBase to store and retrieve representative biological data. Mario can only be made interactive if these latencies are within an acceptable level. The second part is the evaluation of Mario itself, with focus on throughput and end-to-end latencies, but also including an evaluation of CPU, memory, storage and network requirements.

A cluster of nine computers was used for the experiments. All computers in the cluster had the following hardware:

- CPU: 8 core Intel Xeon E5-1620 3.6GHz
- Memory: 32GB
- Disk: 2 x 2TB
- Network configuration: all servers have 1Gbps network cards and are connected via a single 1Gbps switch.

The operating system used on the cluster is CentOS 6.3[1], distributed as part

[1] `http://www.centos.org/`

of the Rocks Cluster Distribution[2]. The HBase and Hadoop stack used is from the Cloudera cdh4.3.0[3] distribution. This includes HBase v0.94.6 and Hadoop v2.0.0. ZeroMQ v3.2.4 is used for communication.

The HBase master server was configured with 4GB of memory. HBase regionservers were configured with 12GB of memory. These settings are recommended in [7, pp. 37].

The most important HBase and HDFS settings are summarized here:

- HDFS block size: 128MB
- HDFS replication factor: 3
- HDFS datanode Java heap size: 1GB
- HDFS namenode Java heap size: 1GB
- HBase master Java heap size: 4GB
- HBase region server Java heap size: 12GB
- HBase client write buffer: 2MB
- HBase maximum size of all memstores in region server: 40% of heap size
- HBase region server memstore flush size: 128MB
- HBase region server maximum file size: 1GB
- HBase region server HFile block cache size: 25% of heap size

4.1 Evaluation of HBase as Storage Backend

It is likely that the storage and retrieval requirements of Mario will involve jobs accessing both large and small amounts of data. The performance of HBase and Google's Bigtable has been evaluated by many researchers, for example [2]. These experiments focus on throughput by testing with millions of rows. This evaluation therefore focus on the latencies involved in insertion and retrieval of small amounts of data to and from HBase.

[2] `http://www.rocksclusters.org/wordpress/`
[3] `http://www.cloudera.com/`

4.1.1 Test Data Generator

Test data is needed to evaluate the read and write performance of HBase. The test data should be representative of data that is expected to be used by Mario.

The evaluation started out using a dataset with real biological data, but the amount of data was not sufficient for all experiments. A data generator was therefore implemented, that can generate specified amounts of two kinds of representative output:

1. FASTA files with random nucleotide sequences, ranging in length between 100 and 5000 bases. IDs were random 15 character strings. These were used to test retrieval of data from HBase.
2. Emulated BLAST[1] tabular output, with random values in all fields (similar to -m 8 option).

The following is an example of the contents of a FASTA file with two very short sequences. A FASTA entry begin an angle bracket followed by the ID of the entry. On the following lines follow the sequence data. An entry end when a new angle bracket, or end of file, is encountered.

```
>72T70EOKK2ZZB1S
GGGTTGTATTCGACGCCAAGTCAGCTGAAGCACCATTACCCGATCAAAACATATCAGAAA
TGATTGACGTATCACAAGCCGGATTTTGTTTACAGCCTGTCTTA
>QXCEYEJ50XUPCZP
CCGCCTATTCGAACGGGCGAATCTACCTAGGTCGCTCAGAACCGGCACCCTTAACCATCC
ATATCCTTCAGTTCCATAGGCCTCTGTGCGGGATTTGTGAACGTTC
```

An example of emulated BLAST tabular output can be as follows:

```
KVBU00MPH538IQJ KXR7KCKPTO7GPTC 46.383774280548096 81 3 50 35 83\
102 69 0.7433696 91.58855080604553
YRYLF3AWRME2UPQ NP3O8X6P3FOU7LA 71.27521634101868 43 13 38 86 66\
13 6 0.31663144 86.0497236251831
```

This data consist of rows, with some columns containing characters, and others containing numerical data.

The test data output is representative of real biological data used by researchers in the group.

4.1.2 Experiment Design

The measurements consists of timing the following two operations:

1. Retrieve 200 000 nucleotide sequences in FASTA format from HBase. The size of this dataset is approximately 500MB.

2. Insert 200 000 rows of BLAST output data into HBase. The size of this dataset is 22MB.

The experiment is designed to emulate the I/O operations of a stage in the *METApipe* pipeline[15] used at the University of Tromsø. The stage does a BLAST similarity search with a FASTA file as input and a BLAST output file as output. Executing this stage with HBase as a storage backend would involve retrieving the data from HBase, writing it to a file, and then loading the output file into HBase. This workflow is illustrated in figure 4.1.

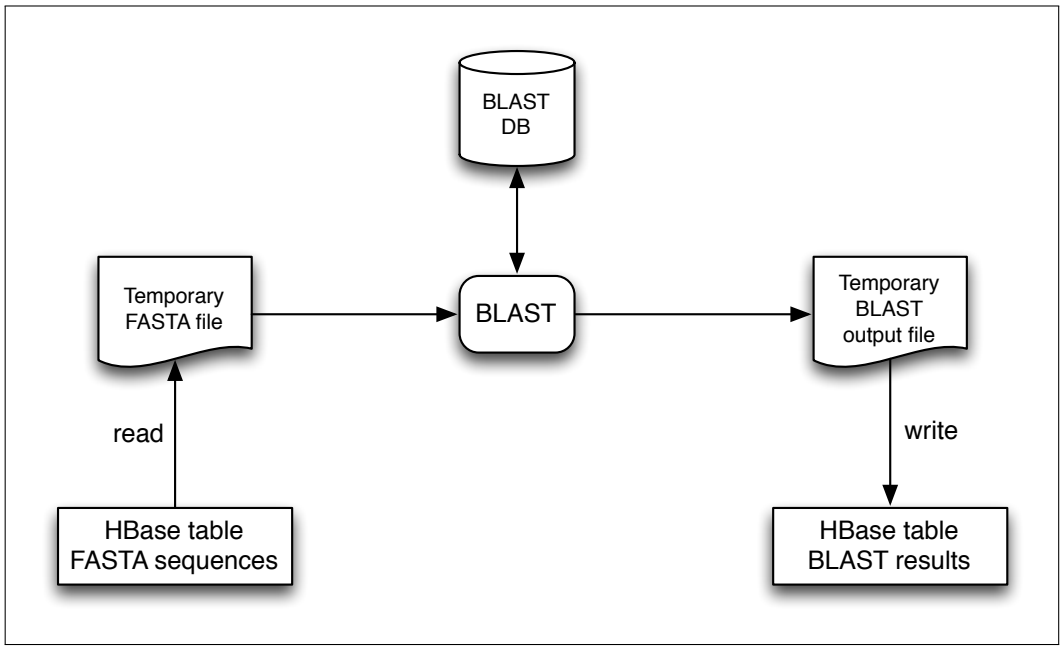


Figure 4.1: HBase evaluation: workflow in a single stage

When data are inserted in an empty HBase table, all are written to one region, handled by one region server (a number of other region servers will store replicas as well). As the amount of data in the table grows, the region will be split into two ranges of keys, one of which can be moved to a different region server. This behaviour makes it reasonable to believe that read and write performance on almost empty tables served by one region server can be different from larger tables handled by multiple region servers. The measurements are therefore done both on an almost empty table and on a larger table with approximately 500GB of data. 500GB is chosen deliberately to be larger than the available memory of the cluster (which is 288GB).

Since the BLAST search is CPU intensive and does not give information of interest to the HBase evaluation, this step of the workflow is not performed.

For the retrival experiment, the following HBase configurations are tested:

1. No compression, Bloom filter off.
2. No compression, Bloom filter on.
3. Snappy compression, Bloom filter off.
4. Snappy compression, Bloom filter on.

As discussed in section 3.1, compression can reduce storage requirements and increase read performance by reducing network traffic. HBase support several compression algorithms: GZIP, LZO and Snappy, of which Snappy is the one offering the highest encoding and decoding rate[7, pp. 424]. Snappy is available on our test cluster. Unless storing already compressed data, it is generally recommended to always use compression with HBase.

Bloom filters are data structures that can can be used to check if a key has been registered/stored. It is a hash-based probabilistic data structure that is usually very compact, so it can fit in memory. A Bloom filter will never return a false negative, but can return false positives. HBase can use Bloom filters to avoid having to scan files for keys that are not in the file. This can reduce disk accesses, thereby improving read performance at the cost of a slight storage overhead.

Scanner caching is an HBase feature that reduce the number of Remote Procedure Calls (RPCs) by transferring multiple rows per RPC, during a scan operation. Initial I/O operations were performed with scanner caching off,

but due to a significant decrease in scanning performance this configuration was dropped to reduce the time required for the experiments.

For the insertion experiment, the following HBase configurations are evaluated:

1. No compression.
2. Snappy compression.

Ideally, the Bloom filter configuration should have been a part of this evaluation, since updating the Bloom filter might have an impact on write performance. This was, unfortunately, not done.

The initial evaluation was performed with the default *write-ahead logger*(WAL) setting, which is to log each write to disk before the write itself is performed. The performance using this configuration was an order of magnitude worse then what was achieved using deferred log flushing, which will collect log entries in memory and flush to disk periodically. As a result, all inserts were performed with deferred log flushing. This configuration has the potential of losing some data if a server fails, but this is not critical since data is handled at fine granularity and can be easily recomputed.

The experiments measure the elapsed wall time. Each experiment is performed five times, and then average time and sample standard deviation is calculated. The HBase cache is flushed between each measurement to get the worst case performance. Informal experiments show that not flushing the cache result in an order of magnitude better performance.

4.1.3 Results and Discussion

The results of the read evaluation are shown in table 4.1.

These results show that reading 200 000 rows of data from an HBase table containing only that data, takes approximately 11 seconds, regardless of compression and Bloom filter settings. Reading the same number of rows from the larger table containing other data, takes approximately 13 seconds. This slightly worse performance on the larger table is probably caused by the single client having to access different HBase region servers during the scan, and these servers are cold on first contact. Due to caching performed

Table 4.1: Time to retrieve 200k rows from HBase

	Small table[1]		Large table[2]	
Configuration	Avg(s)	SD	Avg(s)	SD
Compression: off, Bloom filter: off	11.1	0.4	13.2	1.2
Compression: off, Bloom filter: row	11.1	3.2	13.1	1.7
Compression: Snappy, Bloom filter: off	10.9	1.6	13.8	1.1
Compression: Snappy, Bloom filter: row	11.4	1.4	13.7	1.2

[1] A table containing only the 200k rows (approximately 500MB)
[2] A table containing approximately 500GB of data

by HBase, these results represent worst case performance. The results are considered sufficient to use HBase as a storage backend for Mario. For a more detailed discussion on read performance, see the discussion on Mario performance in section 4.2.

The results of the write evaluation are shown in table4.2.

Table 4.2: Time to insert 200k rows into HBase

	Empty table		Large table[1]	
Configuration	Avg(s)	SD	Avg(s)	SD
Compression: off	10.3	0.3	8.5	0.4
Compression: Snappy	10.1	0.3	6.1	0.2

[1] A table containing approximately 500GB of data

These results show that inserting 200 000 rows into an empty HBase take approximately 10 seconds, regardless of compression setting. Inserting the same data into an already populated table takes approximately 7 seconds. This increased performance on a populated table is expected, since the writes are then performed on different region servers. These results are also considered sufficient for Mario. For a more detailed discussion on write performance, see the discussion on Mario performance in section 4.2.

It is interesting to note that the learning curve for a new HBase user is quite steep. First of all, installation proved troublesome, even for our very experienced system administrator. This was possibly caused by an old HBase version that had been used previously on the cluster. After installation, the performance and stability proved to be poor. It turned out that the default

Java heap size for both the HBase master server and the region servers was set at approximately 750GB, well short of the recommended minimum 4GB and 12GB, respectively. Solved the stability issues, and also improved the performance. Nevertheless, in [28], the authors argue that getting the best performance from a Hadoop system requires considerable tuning, even when done by experts. In light of this, it is likely that the performance of HBase on our test cluster can be improved if enough effort and/or expert support is used.

4.2 Mario Evaluation

To evaluate the performance of Mario itself, with the minimum influence from external applications, a dummy pipeline is used. This pipeline is a four-stage pipeline using the Linux *cat* application in each stage. With test data preloaded into Mario, the experiments measure execution time from the master start handing out tasks to workers, until the last worker is done processing. Worker processing, in this case, involve writing the data to a local file, piping this file to *cat*, and piping the output to another local file. This mean that all data is read, but not modified, by each stage in the pipeline. This represent a pipeline with null operations in each stage, but where all data is accessed for each stage. The worker write intermediate and final output to HBase, before notifying the master that the task is completed. This is the I/O pattern that the applications in the pipeline stages have.

To emulate a user doing parameter tuning, each experiment consists of three steps, where the time is measured for each individual step:

1. The data is processed by the dummy pipeline.
2. The same data is processed by a pipeline where the last two stages have been modified (still using the *cat* application in each stage, but with modified version string to trigger recomputation). In this step, the worker detect that data exists for the first two stages, and only pipe data through *cat* for the last two stages. At this step, HBase will have cached the data, providing faster access.
3. The same data is processed by the same pipeline as in step 1.

Together, these steps emulate a user that start a computation, then try a different parameter, but decides to revert back to the initial parameter.

The test data used is FASTA-type data generated by the test data generator described in section 4.1.1.

4.2.1 Latency

The latency is measured by processing one item of data using the dummy pipeline described above. The time is measured from the Mario master get the *start* message, until the master receives notification from the worker that processing is completed. Each experiment is performed five times, and the average time (in milliseconds) and standard deviation is calculated for each step in the experiment. A summary of the results are listed in table 4.3.

Table 4.3: Time to process one row

	Avg(ms)[1]	SD
Step 1	68	6.4
Step 2	7	1.4
Step 3	6	1.1

[1] Average of 5 runs

The results show that the latency of the first step in the experiment is 68ms. For step two and three the latencies drop to 7ms and 6ms, respectively. The minimal difference between step two and step three indicate that piping the data through two stages in step two, and skipping all reading of data in step three have negligible performance differences. The significant drop in latency from step one to step two show the effectiveness of caching in HBase. To conclude, for processing a single data element, Mario add latencies that are well within the desired goal of 100ms.

4.2.2 Throughput

The throughput is measured by processing a 500MB dataset consisting of 200 000 FASTA sequences, through the same dummy pipeline described in section 4.2.1. This is done using the same three steps as in section 4.2.1.

The rationale for using a dataset of this size is that it is believed to be sufficiently large to provide biological insights. Referring to figure 1.3, it can

be seen that a large fraction of organisms are discovered after processing two million reads. The sequences used in that particular experiment are unassembled reads with a length of approximately 250 base pairs, which translate to approximately 250 bytes in FASTA format. The two million reads then correspond to approximately 500MB of data. The sequence data available for the HBase evaluation contained assembled sequences of much larger size. So, to simulate assembled data, 200 000 sequences with an average length of approximately 2500 base pairs were used. This resulted in a dataset of approximately 500MB.

The experiment is performed using two, four and eight Mario workers. Each experiment is performed five times, and the average execution time (in milliseconds) and standard deviation is calculated for each step in the experiment. A summary of the results are listed in table 4.4.

Table 4.4: Time to process 200 000 rows

	2 Workers		4 Workers		8 Workers	
	Avg(ms)[1]	SD	Avg(ms)[1]	SD	Avg(ms)[1]	SD
Step 1	11699	766	11874	1871	10817	1224
Step 2	7432	288	8279	2497	8302	1377
Step 3	7689	483	8406	2334	8563	2096

[1] Average of 5 runs

The results show little difference as the number of Mario workers are varied. This result is expected and desired, as it show that the processing is limited by the Mario master's ability to distribute tasks to the workers. Since the dummy pipeline consists of null operations, the worker tasks should complete with low latency.

The results also show that the reduction in processing time from step one to step two and three, are less pronounced than that found in section 4.2.1. Referring to the HBase settings in the beginning of this chapter, it can be seen that the 500MB dataset will fit on a single region server. Since the HFile block cache size of the region servers are 3GB, the dataset should fit completely in the cache. The numbers are then likely explained by the Mario master gaining faster access to the data keys due to caching, but still being limited by its ability to iterate the sequence of keys and transmit task messages to workers.

To conclude, the results essentially show the overhead incurred by using

Mario for processing larger datasets. This overhead is low, showing that Mario can be used for processing such datasets.

4.2.3 Sampling

The purpose of the sampling evaluation is to measure the overhead incurred by the sampling procedure.

Sampling was tested with eight Mario workers, using a similar dataset, pipeline and sequence of steps as used in section 4.2.2. A sample of size 10 000 was created by the Mario master, and then processed by the workers. Each experiment is performed five times, and the average execution time (in milliseconds) and standard deviation is calculated for each step in the experiment. A summary of the results are listen in table 4.5.

Table 4.5: Time to process 10 000 row sample of 200 000 rows using eight workers

	Avg(ms)[1]	SD
Step 1	10777	242
Step 2	55	6
Step 3	38	13

[1] Average of 5 runs

The results show that the execution time for the first step, where the sampling is performed, is approximately equal to the non-sampling case. This is slightly surprising, since the sampling result in a two-step procedure: first do the sampling, putting the sampled keys in a local in-memory array, and then transmit the keys in the array to the workers. The conclusion from this is that the sampling procedure has extremely low overhead. Looking at the numbers for step two and tree, it is clear that transmitting the samples to the workers is also cheap. To conclude, Mario can perform efficient sampling of datasets.

4.2.4 CPU Usage

Mario's resource usage is important, since Mario is running on the same compute nodes that the resource intensive bioinformatics applications are executed on.

Since Mario is tightly integrated with HBase and HDFS which is running on the same cluster, CPU load is best measured as an aggregate for each node in the cluster. To test this, 1 000 000 rows or FASTA data (approximately 2.5GB) was uploaded to Mario. Then, the same four stage pipeline with null operations, as used for the latency tests, was executed twice using eight Mario workers. The one minute CPU load, as reported by Ganglia[4] , is shown in figure 4.2. The first experiment was started at 21:11 and lasted 56.2 seconds. The second experiment was started at 21:12 and lasted for 42.9 seconds. The Mario master server is located on the *compute-0-0* node, together with the HBase Master. Figure 4.2 show that there is very little change in one minute load when processing. The Mario workers and the HBase region servers are running on all the other nodes. On these nodes, figure 4.2 show in increase in one minute load up to approximately 1.0, which is not much on an eight core processor.

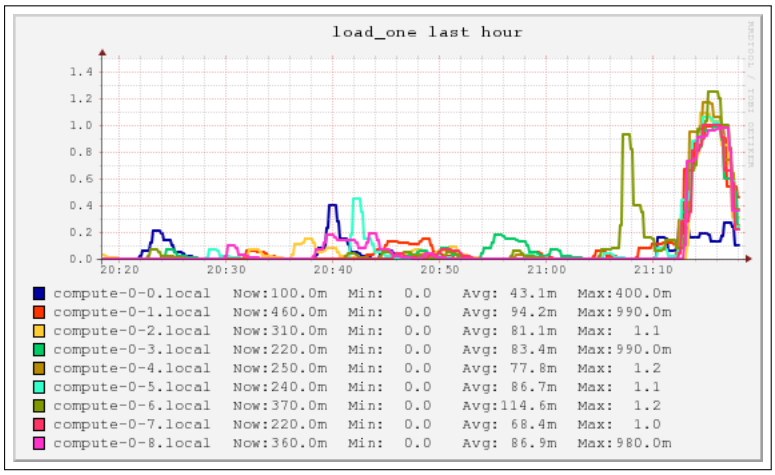


Figure 4.2: One minute CPU load as reported by Ganglia. The experiments were performed between 21:11 and 21:13.

[4] `http://ganglia.sourceforge.net/`

4.2.5 Network Usage

During the same experiment that is described in section 4.2.4, the network traffic was also monitored using the *bytes_out* Ganglia metric. These results are seen in figure 4.3. The *compute-0-0* node with the master servers have a low network out traffic of approximately 3MB/s, equivalent to 24Mbps, which is expected since HBase master server does little communications and the Mario master server only transmit keys to the workers. The other nodes show different amount of network traffic, up to a maximum of approximately 30MB/s, equivalent to 240Mbps. These values are higher than the one for *compute-0-0*, which is expected since there is data transfer from the HBase region servers to the Mario workers. Adding the network traffic for all the nodes, a peak network traffic of approximately 900Mbps is found. This traffic is well below the capacity of the individual network cards and the interconnect switch. Nevertheless, it would be interesting to investigate the use of a data-location aware scheduler in Mario, to see how much the network traffic could be reduced.

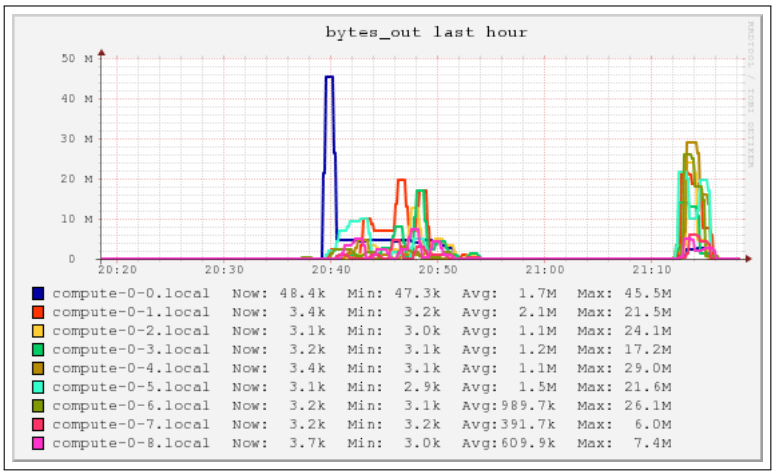


Figure 4.3: Network bytes out as reported by Ganglia. The experiments were performed between 21:11 and 21:13.

4.2.6 Memory

Referring to section 2.3, the memory consumption of Mario itself is minimal since data is only passed through the servers. An exception to this is the array used for sampling, but a large sample of four million keys will still

only consume approximately 80MB of memory. More significant is the fact that Mario require an HBase installation which has considerable memory requirements. For example, the HBase region servers on the test cluster is set up with a 12GB Java heap size. Since servers with more that 32GB of memory are getting more and more common, these memory requirements are considered acceptable.

4.2.7 Storage

The storage requirements of Mario can be easily modelled as follows: Assume that the input data size is 1TB, and this data is processed by a 5 stage pipeline. Normally, the amount of data that is passed from one stage of the pipeline to the next decreases as the data nears the end of the pipeline, but to be conservative we can assume that there is no decrease. Then, a 5 stage pipeline requires storage of 4TB of intermediate data and 1TB of output data, for a total of 6TB. If we further assume that parameter tuning is performed after processing 1GB of data, and 100 parameter changes are made on the first stage of the pipeline (thereby triggering recomputation of all stages), an additional $100 \times 5GB = 500GB$ is require, for a total of 6.5TB. Using a replication factor of three for HDFS then result in a total storage requirement of $6.5 \times 3 = 19.5TB$.

If later use of the intermediate data is unlikely, the user can delete it to reduce the storage requirements. This reduces the data size to 6TB (input data, output data, both stored with a replication factor of three). To conclude, Mario's storage requirements are acceptable.

4.2.8 Reliability

Mario is tested on a small cluster of computers, where failures are expected to be infrequent. HBase data storage is reliable due to the data replication in the underlying HDFS. Therefore, data that has already been processed in Mario is reliably stored. Failures during processing will result in reprocessing of the data being in use at the time of the failure. The cost of this depend on the level of granularity of the input data. For fine grained input data, the cost will be small. For coarse grained input data, the cost might be high.

Chapter 5

Related Work

5.1 Hadoop/MapReduce

Hadoop and MapReduce[3] is probably the most popular system for processing of large datasets. The input dataset is split into smaller parts that are first processed in parallel using user-defined *map* functions. The results are then collected and processed by user-defined *reduce* functions. Parallelization, scheduling (including load balancing and data locality) and fault tolerance are all handled automatically by the system.

MapReduce and Hadoop work by splitting data into parts, that are processed independently by workers processed and combined by combiner processes. This is similar to the way Mario process data. MapReduce and Hadoop process the data completely before results are available to the user. Also, job startup times are long, normally tens of seconds in Hadoop (the problem of long startup latencies are reportedly solve in Google's MapReduce implementation by keeping workers alive instead of starting them for each job[4]). This makes Hadoop unsuitable for interactive processing of the kind that Mario does.

5.2 HBase

HBase, which is based on Google's Bigtable[2] is designed for real time random access to data, and is often used for interactive analytics. A detailed

description of HBase is given in section 3.1.

5.3 Apache Pig

Pig Latin is a high-level language with SQL-like syntax that executes on the Apache Pig[22] runtime. Pig compile Pig Latin into MapReduce code that is executed on Hadoop. The primary advantage of this is that it is easy to implement an analysis using Pig Latin, even for non-programmers, end especially for data analysts with knowledge of SQL.

Pig include *Pig Pen*; a debugging environment for Pig Latin programs. Pig Pen will create a small dataset that can quickly show the result of the program statements, therefore freeing the user from having to wait for a long-running computation to see if the program is correct. The small dataset (called a sandbox dataset) is created partly by sampling from the real dataset, and partly by generating data that look like the real data.

Mario uses a similar idea, in that the user should be presented with results of the computation as quickly as possible, so that the pipeline configuration (which is Mario's equivalent of a Pig Latin program) can be adjusted or "debugged" easily. Mario does this by quickly presenting the user with results of computing only parts of the complete dataset, obtained either through sampling (like Pig Pen) or by processing from the start of the dataset. Mario does not generate artificial data.

Pig Pen make it easy to create correct Pig Latin programs. However, when compiled and executed on real data, the program performs like a regular MapReduce program on Hadoop, which has little or no possibilities for interactivity.

5.4 GeStore

Several common bioinformatics applications use external metadata collections for data processing. When these collections are updated, new knowledge can be gained by executing a pipeline on an old dataset. GeStore[24] reduce the cost of this recomputing, by allowing incremental updates of results when metadata collections are updated. GeStore is essentially a layer

between a pipeline and the storage layer, that is able to perform the incremental updates. In contrast to Mario, GeStore does not implement the pipeline itself. Also GeStore perform the updates using MapReduce, without requiring inspection by the user. This is different from the Mario use case, where interactivity is required to efficiently configure the pipeline.

5.5 Galaxy and Taverna

Galaxy[9] is a platform for accessible and reproducible analysis of genomic data. Galaxy provides a toolbox of applications from which the user can compose workflows or pipelines using a user-friendly Graphical User Interface (GUI). The workflow is shown in a web page that can be shared between users together with the data, improving the reproducibility of the analysis. Galaxy is similar to Mario in that the user can compose a pipeline from existing applications, but Galaxy support the creation of more general execution graphs, as opposed to Mario's linear pipeline model. The cost of this generality is that Galaxy does not provide interactive analysis. Mario's limitation to linear pipelines is caused by the interactivity requirement. Linear pipelines also make it easy for Mario to provide automatic parallelization, which is not provided by Galaxy.

Taverna[21] is a platform that is very similar to Galaxy, and the same comments above apply to Taverna vs. Mario.

5.6 Spark

Spark[32] is a system for efficient parallel execution of computations that reuse the same dateset, that is iterative processing. Such computations can be performed on MapReduce/Hadoop, but then require loading the dataset from disk for each use. Spark keep the data in in-memory read-only data structures called Resilient Distributed Datasets (RDDs), so that subsequent operations on the same data avoids disk operations. This makes iterative computation over the same dataset efficient, if the dataset fits in memory on the cluster running Spark. Spark is also efficient for the same type of analytics typically performed from Apache Pig, since the dataset can be held in memory after the first query, but this also assume that there is enough memory available.

Discretized streams is a computing model implemented on Streaming Spark [33], where a stream of input data is divided into short intervals of data that are batch-processed, using RDDs for temporary storage to increase performance. Discretized streams focus on low latency operations and fault recovery. Fault recovery is handled by recording lineage data for a stream window of sufficient length.

In contrast to Discretized streams on Spark, Mario provides permanent lineage/provenance, by implicitly tagging data (through column naming), and storing metadata related to the computations.

It is, however, likely that Mario could be implemented with Spark as a replacement for HBase.

5.7 Dryad

Dryad[14] is an execution engine for coarse-grain data-parallel applications. It lets users write sequential programs, which are then automatically scheduled in parallel on systems ranging from a single multi-core computer up to clusters of thousands of computers. In contrast to MapReduce, Dryad allow the user to specify arbitrary acyclic execution graphs. This means that Dryad can execute programs that would require the composition of multiple MapReduce programs. Similar to Mario, dryad support the execution of legacy executables, through wrappers that work with arbitrary data types. Dryad operates on coarse-grained data, making it unsuitable for low latency interactive computations.

5.8 Naiad

Naiad[20] is a system for data parallel processing on streams of data. Naiad uses a computational model called *timely dataflow*, that support cycles in the dataflow and stateful processing vertices that do not require global coordination. This enables Naiad to provide high throughput, low latency, iterative computations, where subcomputations can be nested and composed.

To our knowledge, no metagenomics pipelines have been integrated with Naiad. It is likely that Mario could have been designed to have a similar ex-

ecution model as Naiad, but that would require a redesign of Mario's storage model.

5.9 Dremel

Google's Dremel[18] system is designed for ad-hoc, low latency analytics on large, nested, read-only datasets. By using a columnar storage model, Dremel can keep disk accesses to a minimum and therefore access data quickly. Dremel provide a SQL-like language for writing queries.

Dremel provide low latency reading of data and efficient queries on that data, but is not optimized for writing data. It is therefore not suitable for Mario's use case.

Chapter 6

Conclusion

This work has outlined Mario - a system for iterative and interactive processing of biological data. Mario provides a solution to some of the issues that are typically associated with the batch processing pipelines used in metagenomics. These issues include the difficulty of tuning pipelines, and the maintaining of data provenance.

The Mario system offer an online, data-parallel processing model where changes in the pipeline configuration are quickly reflected in update of pipeline output available to the user, and where provenance data is stored in the system as a "first-class citizen".

Mario and its underlying storage system, HBase, were evaluated using a benchmark developed to simulate I/O loads that are representative for biological data processing. The results showed that Mario adds less than 100 milliseconds to the end-to-end latency of processing one item of data. This low latency, combined with Mario's storage of all intermediate data generated by the processing, enables easy parameter tuning. In addition to the improved interactivity, Mario also offer integrated data provenance, by storing detailed pipeline configurations associated with the data.

The evaluation of Mario demonstrated that it can be used to achieve more interactivity in the configuration of pipelines for processing biological data. We believe that biology researchers can take advantage of this interactivity to perform better parameter tuning, which may lead to more accurate analyses, and ultimately to new scientific discoveries.

Chapter 7

Future Work

Although Mario solve the problems outlined in section 1.2.2 there are still opportunities for future work. The following is a list of some improvements that would make Mario more useful. These features were not implemented for this project due to time contstraints.

1. Implement a graphical user interface, in the form of a web application.
2. Integration with Krona or other tools for visualization of output data. This integration is done in the interface shown in figure 2.1. It involves aggregating data from the pipeline output, to present it in a format suitable for Krona or METAREP. This is the one component of Mario that require custom code depending on the tool used.
3. Improved scheduler to handle variations in worker performance (stragglers) and take advantage of data locality. This will improve the performance of Mario by improving the utilization of the workers, and by providing the workers with faster access to data.
4. Support for dataset management. The prototype only support processing of a single dataset. Since Mario is also a long term storage system, functionality should be added to enable upload and storage of multiple datasets. The user should be able to select one or more datasets to be used for analysis.
5. Improve source code quality. The prototype contain very little error handling. In addition to improvements in code quality, support should be added to forward error messages from the pipeline stage tools to

the users. This is important since it is likely that the user can input application parameters that will lead to errors.

6. Mario should contain a database of bioinformatics applications that can be used in the pipeline stages. This database should contain the different parameters available for tuning, and if possible, their allowed values. This approach is used by Galaxy[9] and Taverna[21].

References

[1] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.

[2] 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.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

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

[5] Pavlos S Efraimidis and Paul G Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.

[6] Robert D Finn, Jody Clements, and Sean R Eddy. HMMer web server: interactive sequence similarity searching. *Nucleic acids research*, 39(suppl 2):W29–W37, 2011.

[7] Lars George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[9] Jeremy Goecks, Anton Nekrutenko, James Taylor, T Galaxy Team, et al. Galaxy: a comprehensive approach for supporting accessible, re-

producible, and transparent computational research in the life sciences. *Genome Biol*, 11(8):R86, 2010.

[10] Johannes Goll, Douglas B Rusch, David M Tanenbaum, Mathangi Thiagarajan, Kelvin Li, Barbara A Methé, and Shibu Yooseph. Metarep: Jcvi metagenomics reports—an open source tool for high-performance comparative metagenomics. *Bioinformatics*, 26(20):2631–2632, 2010.

[11] Michael Höhl, Stefan Kurtz, and Enno Ohlebusch. Efficient multiple genome alignment. *Bioinformatics*, 18(suppl 1):S312–S320, 2002.

[12] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[13] John PA Ioannidis, David B Allison, Catherine A Ball, Issa Coulibaly, Xiangqin Cui, Aedín C Culhane, Mario Falchi, Cesare Furlanello, Laurence Game, Giuseppe Jurman, et al. Repeatability of published microarray gene expression analyses. *Nature genetics*, 41(2):149–155, 2008.

[14] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[15] Tim Kahlke. METApipe pipeline. Unpublished work at the University of Tromsø.

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

[17] Erik Kjærner-Semb. Master's thesis in chemistry. Master's thesis, University of Tromsø, 2013. To be submitted December 2013.

[18] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[19] Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277. ACM, 1968.

[20] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[21] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[22] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[23] Brian Ondov, Nicholas Bergman, and Adam Phillippy. Interactive metagenomic visualization in a web browser. *BMC bioinformatics*, 12(1):385, 2011.

[24] Edvard Pedersen. GeStore - incremental computation for metagenomic pipelines. Master's thesis, University of Tromsø, 2012.

[25] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten simple rules for reproducible computational research. *PLoS Computational Biology*, 9(10):e1003285, 2013.

[26] Andrea Sboner, Xinmeng Jasmine Mu, Dov Greenbaum, Raymond K Auerbach, Mark B Gerstein, et al. The real cost of sequencing: higher than you think. *Genome Biol*, 12(8):125, 2011.

[27] Lefteris Sidirourgos, Martin Kersten, and Peter Boncz. Scientific discovery through weighted sampling.

[28] Michael Stonebraker, Daniel Abadi, David J DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[29] Ronald C Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. *BMC bioinformatics*, 11(Suppl 12):S1, 2010.

[30] Susannah Green Tringe and Edward M Rubin. Metagenomics: Dna sequencing of environmental samples. *Nature reviews genetics*, 6(11):805–814, 2005.

[31] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[32] 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.

[33] 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*, pages 423–438. ACM, 2013.