UNIVERSITY OF TROMSØ UIT

FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

GeStore – Incremental Computation for Metagenomic Pipelines

Edvard Pedersen

INF-3990
Master's Thesis in Computer Science
May, 2012

Abstract

Genomics is the study of the genomes of organisms. Metagenomics is the study of environmental genomic samples. For both genomics and metagenomics DNA sequencing, and the analysis of these sequences, is an important tool. This analysis is done through integration of sequence data with existing meta-data collections.

Genomics is the study of the genomes of organisms, and involves cultivating organisms in a lab and analyzing them. Metagenomics is the study of genomic samples collected directly from the environment, allowing researchers to study organisms that are difficult to cultivate in a petri dish. DNA sequencing and the analysis of these sequences is an important tool for both genomics and metagenomics. The integration of the data produced by sequencing with existing meta-data collections is particularly interesting for metagenomics, as a single biological sample can contain thousands of different organisms.

The recent developments in DNA sequencing technology mean that the volume of data that can be produced per dollar is increasing faster than the volume of data that can be analyzed and stored per dollar. This data growth means that the initial analysis of these massive data sets becomes increasingly expensive. In addition, there is a need to periodically update old results using new meta-data from the many knowledge bases (meta-data collections) for biological data. Today, this typically requires rerunning the experimental analysis. Such incremental analysis is interesting for metagenomics since environmental samples potentially contain thousands of organisms.

In metagenomic analysis, different sets of tools are used depending on the type of information required. These tools are generally arranged in a pipeline, where the output files of one tool acts as the input for the next. The analysis done by some steps is dependent on different meta-data collections. When meta-data is updated, these steps and all subsequent steps typically need to be executed again. Incremental updates can save significant computation time by running these pipelines against the updated segments, rather than the full meta-data collections.

We believe that systems for incremental updates for metagenomic analysis pipelines have the following requirements; (i) reduce the computational resource requirements by using incremental update techniques (ii) the meta-data collections should be accessible without the use of proprietary or computationally expensive techniques (iii) do the incremental updates on demand, due to different needs of experiments, through handling meta-data updates and generating arbitrary delta meta-data collections (iv) support most genomic analysis tools and run on most job management systems (v) no changes should be made to the tools that the pipeline is comprised of, since modifying the many available tools is impractical (vi) the changes to the job management and resource allocation system should be minimal, to save implementation time for the pipeline system maintainer (vii) maintain a view of previous meta-data collections, so old experiments can be repeated with the correct meta-data collection version.

To our knowledge no existing incremental update systems satisfy all seven requirements. Often they do not support on-demand processing or maintaining views of old data, in addition many systems require computations to be done within a specific framework or programming language.

In this thesis we describe the GeStore incremental update system which satisfies all seven requirements. GeStore reduces the size of the meta-data collections, and thus the computational requirements for the

pipeline, by leveraging incremental update techniques, satisfying requirements (i) and (iii). In addition it reduces the storage requirements of the meta-data collections, while still maintaining a complete view of the meta-data collection in a plain-text format, fulfilling requirement (ii) and (vii). It also presents a simple interface to the application programmer, so that integrating the system with existing pipeline solutions does not require large changes to the pipeline system or tools, in accordance with requirements (vi), (iv) and (v).

GeStore has been implemented using the MapReduce framework, along with HBase, to provide scalable meta-data processing. We demonstrate the system by generating subsets of meta-data collections for use by the widely used genomic tool BLAST.

In our evaluation, we have integrated GeStore with an existing pipelining system, GePan; a metagenomic pipeline system developed for a local biotech company in Tromsø, Norway, and used real-world data to evaluate the performance and benefits of GeStore.

Our experimental results show that GeStore is able to reduce the runtime of the incremental updates by up to 65% when compared to unmodified GePan, while introducing a low storage overhead and requiring minimal changes to GePan.

We beleive that efficient on-demand updates of metagenomic data, as provided by GeStore, will be useful to our biology collaborators.

Acknowledgments

I extend my thanks to my supervisor Lars Ailo Bongo for his thoughtful comments throughout this project, my co-adviser Professor Nils P. Willassen for his help and critique of the biological sections, Espen Robertsen and Tim Kahlke for their help with the pipeline and Jon Ivar Kristiansen for maintaining the cluster. I would also like to thank my fellow students for insightful discourse and motivation, and my family for their support.

Contents

Word list

Genome	The complete DNA of an organism, which contains both genes and non-protein coding sequences (ncRNA).
Gene	A hereditary part of a genome. Each gene codes for either a protein or an RNA chain that has a function in the organism.
DNA	The hereditary material in organisms.
Read	Short sequence of DNA produced by sequencing machines.
Contig	Long sequence of DNA made up of overlapping reads.
Meta-data collection	A collection of meta-data, such as the Uniprot knowledge base. Often referred to as databases.
Pipeline run	A single execution of a pipeline.
Pipeline	A collection of tools that process input data and produce output arranged in a sequence.
UniProtKB/Swiss-Prot	The manually annotated and reviewed portion of the Uniprot Knowledge Base meta-data collection.
UniProtKB/TrEMBL	The automatically annotated and unreviewed portion of the Uniprot Knowledge Base meta-data collection.

1 Introduction

Genomics [1] is the study of the genomes of organisms. For microorganisms this is often done by cultivation of the organism, isolation of the DNA, reading the DNA sequence using a sequencing machine, and then doing statistical analysis on this sequence data, often by integrating it with already known sequences.

The primary motivation for metagenomics [2] is that more than 99% of the microorganisms in nature are uncultivable in laboratory and therefore their genomic content is not accessible through the usual genomic methods as described. Metagenomics [3] avoids this problem by isolating and sequencing of DNA from environmental samples which contain whole communities of microorganisms, instead of growing the samples in the lab, these samples may come from a variety of sources, e.g. sea-floor sediment. Such environmental samples may have as many as 10 000 different species per gram of sediment, in contrast to conventional genomics, where only a few genomes are in focus. This makes analysis difficult, as there are a myriad of short DNA sequences[1] from hundreds to thousands of heterogeneous genomes, but it also gives a more complete view of the genomic diversity around us [4].

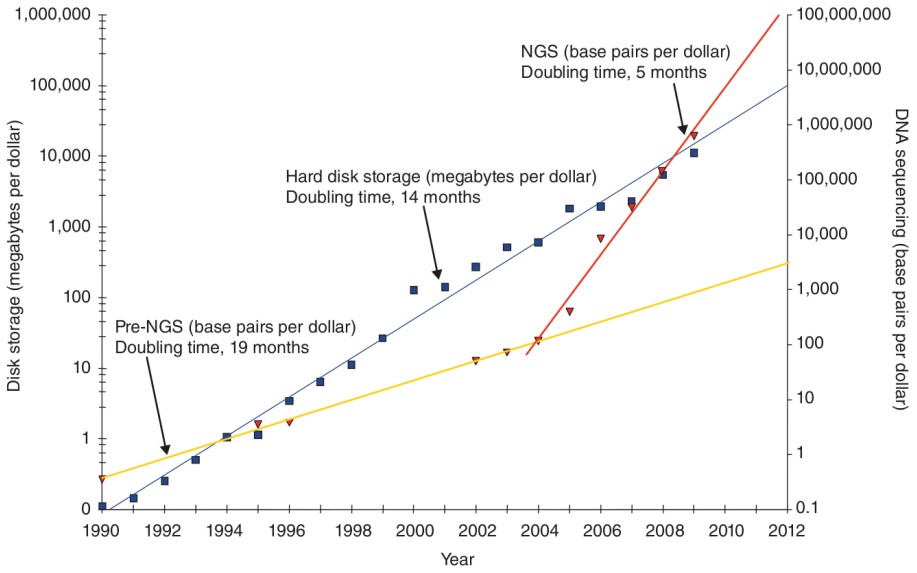

Figure 1: Growth of sequencing volume per dollar, compared to storage (reprinted from [5])

These environmental samples are often, like in genomics, analyzed using DNA sequencing. Recent and ongoing advances in sequencing technology lead to a decrease in sequencing cost that is accelerating faster than the decrease in storage and processing cost [6] (figure 1). This decrease in cost leads to an increase in data produced, which creates computer science challenges when it comes to analyzing the data. The result of this is that new techniques for analyzing and storing data are required to cope with the ever-increasing data deluge, as current systems are not good enough at handling these data volumes.

[1] DNA reads

The processing of metagenomic sequence data is typically done by a variety of tools arranged into a pipeline, where the output of one tool serves as the input for another. Due to the large amount of CPU time a single pipeline run takes, these tools are often run in parallel on clusters of computers, either in the cloud, on supercomputers or smaller clusters. Several tools exist to manage this pipelining and parallelization [7, 8], but some properties are common when it comes to these pipelines.

A typical use case for such pipeline systems for metagenomic analysis is a researcher, having gathered an environmental sample, sequences the sample. The researcher then decides which tools to use to annotate these sequences, and runs the pipeline using the data from the sequencer and possibly other metadata. When the pipeline is finished, the researcher then explores the data to identify the probable genes in the sample, and what the functions of these genes are predicted to be.

The steps below outline the basics of a typical metagenomic pipeline. The tools used in the different parts, or indeed if all steps are followed, depend on the pipeline used. Which pipeline is used depends on the support for tools in the pipeline manager and the needs of the researcher.

- Assembly. Assembly is the process of combining short reads into larger contigs by finding overlaps in the reads. This is done because the reads, which are the output from sequencing machines, may be very short, and to get usable data, longer coherent sequences must be assembled. Many pipelines do not handle this step, as the methodology for combining reads into contigs is frequently different between sequencing machines, and the software that does the assembly is usually provided by the sequencing hardware vendor.

- Filtering. Filtering refers to the removal of unwanted data from different stages of the pipeline. This is typically done to reduce the number of low-quality reads, or to limit the data that enters a particular branch or stage of the pipeline to data that is relevant. Examples include non-coding RNA, low-quality reads or short contigs. Filtering is done by examining the data and moving or removing the unwanted data from the working set, which reduces the runtime of the pipeline and improves the quality of results.

- Clustering. Clustering is the process of grouping similar contigs together, so that the contigs within a group are predicted as a single contig. This is done to reduce the number of contigs that need to be processed in the pipeline. It is done by examining the similarity between the contigs, and grouping very similar ones together. A representative contig for that group is then chosen, and is used for further processing.

- Gene prediction. This is the process of locating probable genes within the sequence. This is done to locate the biologically significant parts of the sequence for the annotation step. It is often done by using tools such as Glimmer [9] and the Basic Local Alignment Search Tool (BLAST) [10], and techniques ranging from sequence comparison to probabilistic models. BLAST compares input sequences to a collection of sequences,usually

taken from a knowledge base of mapped sequences (a meta-data collection), like the Uniprot Knowledge Base [11]. For our pipelines this is the most computationally expensive step.

- Annotation. Biological meta-data is attached to the predicted genes found in the prediction step. This is done to find the specific function or structure of the sequences predicted, and it is done by looking up the sequence IDs found in the prediction step in a meta-data collection.

- Visualization. Finally, the results are exported to some format that can be interpreted visually, so that the results can be interpreted. The tools (review [12] provides a comprehensive list of visualization tools) used in this step depend on the type of analysis being done.

In addition to these things, the pipeline system will usually coordinate these steps in collaboration with the job management system so that they can be run in parallel on several compute nodes at once.

As meta-data collections are updated with new or updated information about existing sequences, new discoveries may be found by analyzing old samples against the new meta-data collections. This is a costly process, as the tools have to be run against the new meta-data collection in its entirety, including data that hasn't changed since the last iteration of the meta-data collection. Current systems like CloVR [7] and the JCVI metagenomics analysis pipeline [8] do not support incremental updates to the results when new data is added to the meta-data collections. However, such updates may give new insight into old data, and they are therefore required by our biology collaborators.

A system for incremental updates is needed in order to reduce running times when updating previous experiments with new meta-data collections. As a typical pipeline run for metagenomic data can take days to complete, updating these results as meta-data collections are updated is very expensive. In this thesis we describe a system for incremental updates of metagenomics data.

1.1 Requirements and rationale

Our primary goal is to provide a system that can aid genomic pipelines doing incremental updates, while maintaining compatibility with existing tools.

The main use case we envision is where a data center is running metagenomic experiments for a researcher who does an analysis of a metegenomic sample, and later wishes to run this experiment on the same sample again, using updated meta-data collections. We optimize for the latter part of this use case.

For such systems we believe the requirements are as follows:

1. The runtime overhead of the system when generating incremental meta-data collections should be less than the reduction in runtime when using these incremental meta-data collections for computations, resulting in a net reduction in pipeline runtime.

2. The system should provide a storage service for the meta-data collections and intermediate data that is possible to access without the use of proprietary tools or computationally expensive techniques. This is so that further analysis and computations does not require a large amount of processing power or lisencing to access the data.

3. When the meta-data collections are updated, the system should be able to integrate these updates into the internal storage system. This is a requirement due to the large size of the meta-data collections, so that the storage requirements for the meta-data collections in the system do not grow faster than necessary.

4. The system should provide a minimal user interface, so that it is easy to use, and does not require a large effort to integrate with existing pipeline management systems. This is required to minimize the changes necessary in the existing pipeline management systems.

5. It should be easy to extend the support for meta-data collections and tools within the system, by providing a framework for developing support for new file formats, both as input and output. This is required so that it will be easy to expand the use of the system beyond the tools and meta-data collections we have implemented support for.

6. The system should not require modifications of the tools used in the pipelines; rather, the work to support the tools should be done within the system. Because the tools used are complex systems that still receive updates, we do not want to burden the application developer with the maintenance of the tools as well as the pipeline creation system.

7. The system should maintain a view of old meta-data collections, so that any old experiments can be repeated with the same meta-data as they were originally run with. This is so that experiments can be repeated reliably, even when new meta-data collections are entered into the system. This is a fundamental requirement in scientific work.

There exist systems [13, 14, 15, 16] that satisfy some of these requirements (are described in section 2), but none that satisfy all seven.

We have built a system, called GeStore, that satisfies these requirements. This system is described below.

1.1.1 GeStore

The main goal of our GeStore is that pipeline systems should not require large modifications to perform incremental updates. As such, the interface point between the pipeline and GeStore lies at the file storage and retrieval level. We have chosen this interface since many genomic tools use a relatively small collection of simple file formats.

GeStore lies between the metagenomic pipeline and the storage system, as seen in figure 2, providing services for management and storage of files, and for generating incremental meta-data collections for use by the tools in the pipeline. The meta-data collections are updated and added to the system outside the pipeline execution.

The actual interface between the pipeline and GeStore is provided by the **move** module of GeStore. This module handles the movement of files to and from GeStore, incremental or complete. The generation of incremental meta-data collections is transparent to the pipeline.

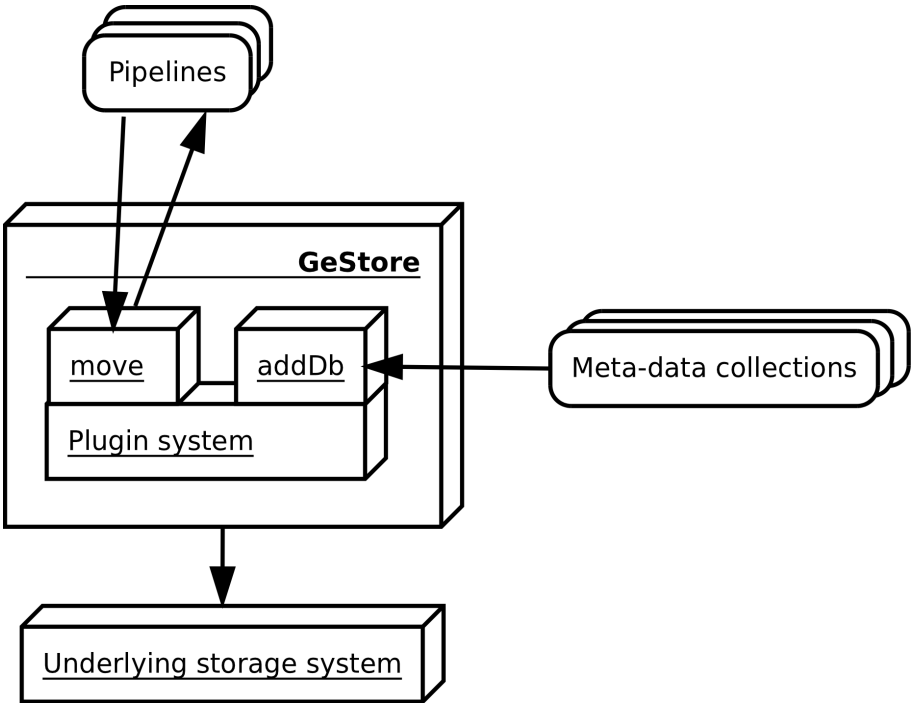

Figure 2: Overview of GeStore architecture

Adding and updating meta-data collections is done through the use of the **addDb** module, which handles the work needed to maintain a coherent view of the meta-data collections.

These user-facing modules use a **plugin** system to provide the support for different meta-data collections and file formats. The **plugin** system separates the logic connected to the individual meta-data collection formats from the core operation of GeStore. Adding a new meta-data collection format to GeStore is done through the development of a **plugin**.

In addition to these modules, GeStore consists of several modules that perform tasks internally like generating MapReduce [17] jobs for scalable data processing, splitting files and maintaining the database, which are not used by the pipeline directly.

GeStore fulfills the requirements set forth in section 1.1 as follows:

1. Generating incremental meta-data collections in a timely manner is done by parallelizing the work of extracting and inserting meta-data collection through the use of the MapReduce paradigm in the data processing in GeStore.

2. By storing the meta-data in a plain-text format that mirrors the format in the meta-data collections before processing, it is possible to examine and process the meta-data collection without the use of proprietary tools. Data maintenance is provided by HBase [18] and HDFS [19].

3. Data maintenance is handled by the **addDb** design. As **addDb** adds a new meta-data collection to GeStore, previous and later entries are in-

spected and modified using delta encoding techniques [20] to maintain a correct and complete view for every version of the meta-data collections.

4. The user interface is kept small through the **move** and **addDb** interfaces. These modules require as little user interaction as possible, through the use of sensible and automatically generated default parameters.

5. Extensibility is ensured through the **plugin** system, which provides the ability to add new file formats and meta-data collections without changes to the core of GeStore, and requiring only a relatively small development effort.

6. The separation between the tools used and the incremental meta-data collection processing is handled through the file-level interface between GeStore and the tools. The collection of files created are the same if the meta-data collection is incremental or complete, so the processing done on them is the same, even if the contents is not.

7. Maintaining a view of old meta-data collections is done through the use of HBase timestamps and the partial cell updates, thus a complete view of the meta-data collection exists for every version of the collection.

1.2 Methodology

In this thesis, we have followed a systems approach to reach our findings.

- We have identified common requirements for incremental update systems for metagenomic pipelines and created an architecture and design describing a system for meeting these requirements.
- We have built GeStore to implement our design, using the Hadoop stack to provide parallelization, storage and database services.
- We have evaluated the performance of GeStore using realistic metagenomic data taken from a published metagenomic project, and prevalent meta-data collections.

1.3 Contributions

Our scientific contributions are the following:

- We have designed, implemented and evaluated the GeStore system for incremental updates of metagenomic data.
- We have demonstrated the viability of incremental updates for metagenomic work, showing that computational resource requirements can be reduced when using incremental update techniques. We provide an experimental evaluation that shows that GeStore can reduce the runtime of incremental pipeline runs done in the GePan metagenomic analysis pipeline system.
- We have demonstrated that extending and integrating GeStore with existing pipeline systems is easy to do, by integrating GeStore with the GePan pipeline system and providing support for the Uniprot meta-data collections with few lines of code.

1.4 Summary of results and conclusions

We have done an experimental evaluation of GeStore. These are our findings:

- We have shown that incremental meta-data collection reduce the runtime of BLAST since there are few changes in the meta-data collections, and BLAST execution time depends on the size of the meta-data collection.
- We have shown that incremental updates using GeStore save up to 65% runtime when compared to full pipeline runs, since the runtime overhead of GeStore is lower than the time saved by doing incremental updates.
- We have shown that storage requirements are lowered by using GeStore when compared to storing the meta-data collections individually. Our experiments show up to 80% reduction in storage requirements.
- We have shown that integrating GeStore with an existing pipeline system is easy, since the integration with GePan takes less than 100 lines of code in our case study.
- We have shown that extending GeStore to support new formats is easy, since the support for the Uniprot and FASTA formats takes less than 300 lines of code each.
- We have shown that on-demand updates saves runtime when compared to doing continous updates.

This evaluation shows that incremental computations for metagenomic pipelines have the potential to reduce the computational time required for updating results of metagenomic pipelines significantly for our use case. We beleive that efficient on-demand updates of metagenomic data, as provided by GeStore, will be useful to our biology collaborators.

1.5 Organization

The remainder of this thesis is organized as follows: In section 2 we discuss the related work, and reason about their viability for our needs. In section 3 we describe the design of our system. In section 4 we describe the implementation details of GeStore. In section 5 we describe a case study, where we integrate GeStore with GePan. In section 6 we describe our findings, and in section 7 we discuss if the system fulfills our requirements, describe our future work and present our conclusions.

2 Related work

In this chapter, we describe one system that represents one approach that is alternative to the systems we have chosen to use, or GeStore itself. The SciDB and Nectar sections are based on the descriptions in my special curriculum [21], and additional related systems are described there. In section 2.1 we describe the underlying systems we have chosen to use for data storage, processing and our experimental integration, in section 2.2 we describe the most important relevant systems for doing incremental computations, in section 2.3 we describe storage systems, and in section 2.4 we describe some alternative metagenomic pipeline software.

2.1 Software stack

In this section we describe the software stack we have used to implement GeStore.

2.1.1 HDFS

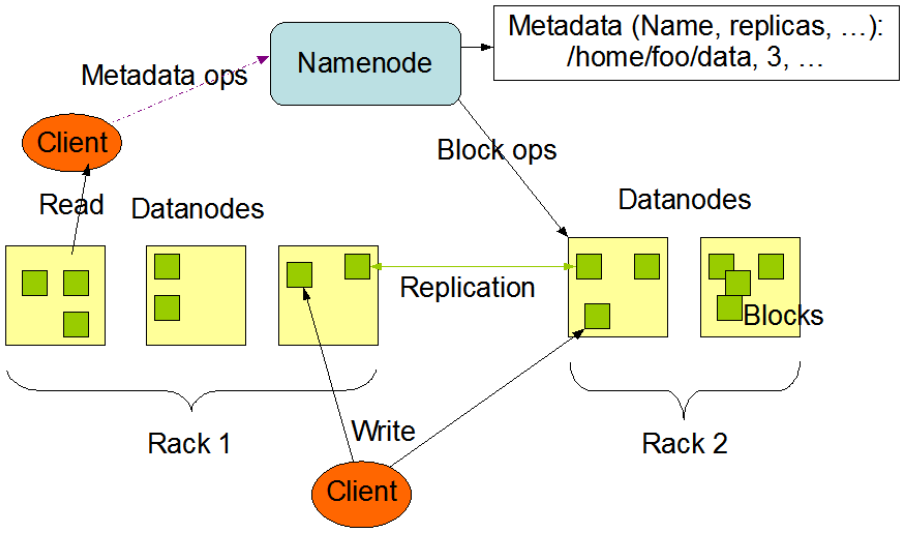

Figure 3: HDFS architecture, reprinted from [22]

The Hadoop Distributed File System (HDFS) [19] is a distributed file system based on the Google File System (GFS) [23] which is designed to handle very large data sets reliably, and stream these data sets at high bandwidth to users.

HDFS is designed for an environment where reliable, distributed storage of files, and bandwidth is important.

The interface to HDFS is modeled after the UNIX file system, but strict adherence to the POSIX standard has been sacrificed in favor of performance. The system is divided among several virtual nodes with different areas of responsibility (figure 3). NameNodes contain the meta-data for the data stored in the system; this includes data about the path to the files, which blocks are

part of a file, permissions, access times and so on. DataNodes contain the application data, as well as meta-data like checksums for a block. Replication in the system is done across DataNodes, and is done in such a way that data is spread across different racks, as HDFS is rack-aware. This is an important property for large data centers, as the failure of a single rack would not mean data loss, as it may mean in a system where data is replicated through a RAID system. HDFS exposes the physical location of the data to the client through the API, this is important for MapReduce-based applications, as this means the computations can be done close to the data, increasing bandwidth and reducing network strain.

HDFS is in relatively widespread use, users like Yahoo! and Facebook being perhaps the most prominent.

We have chosen to use HDFS primarily because it is a part of the Hadoop stack, and it is already in use by other systems on our cluster. Other distributed file systems may provide similar services, but we have not investigated this in detail as we have not found the distributed file system to be a bottleneck in our implementation.

2.1.2 HBase

HBase [18] is the Hadoop stack implementation of the BigTable [24] design, both are distributed database systems designed to handle big data. HBase provides random, real-time access to big data.

The design of HBase is similar to that of traditional distributed relational database management systems, but differs in that it doesn't offer a full relational model. Data is structured in cells, and is uniquely referred to by the triplet $(row, column, timestamp)$. Rows are grouped as the units of computation, and columns are grouped as the units of access control and resource management, columns are further grouped into column families.

HBase consists of several virtual nodes, similar to HDFS. The Master typically runs on a HDFS NameNode, and is responsible for managing the RegionServers in the cluster, along with meta-data associated with the database. The RegionServers typically run on a HDFS DataNode, and are responsible for doing the work in HBase, clients interact directly with the RegionServer. The Master and RegionServer are analogous to the Master and TabletServer in BigTable.

HBase is in use several places, along with the rest of the Hadoop Stack.

We use HBase to store meta-data about files, pipeline runs and meta-data collections, as well as the meta-data collections themselves.

We have chosen to use HBase due to the compelling feature set, in particular the versioning support based on timestamps, the simple API, the ability to add columns as needed, as well as the distributed nature of the system and the tight integration with MapReduce.

2.1.3 MapReduce

MapReduce [17] is a programming model for parallelizing work, popularized by Google. The primary use case for MapReduce is embarrassingly parallel data-intensive work.

MapReduce is designed with a very simple interface. At the heart of a MapReduce application are two functions implemented by the application pro-

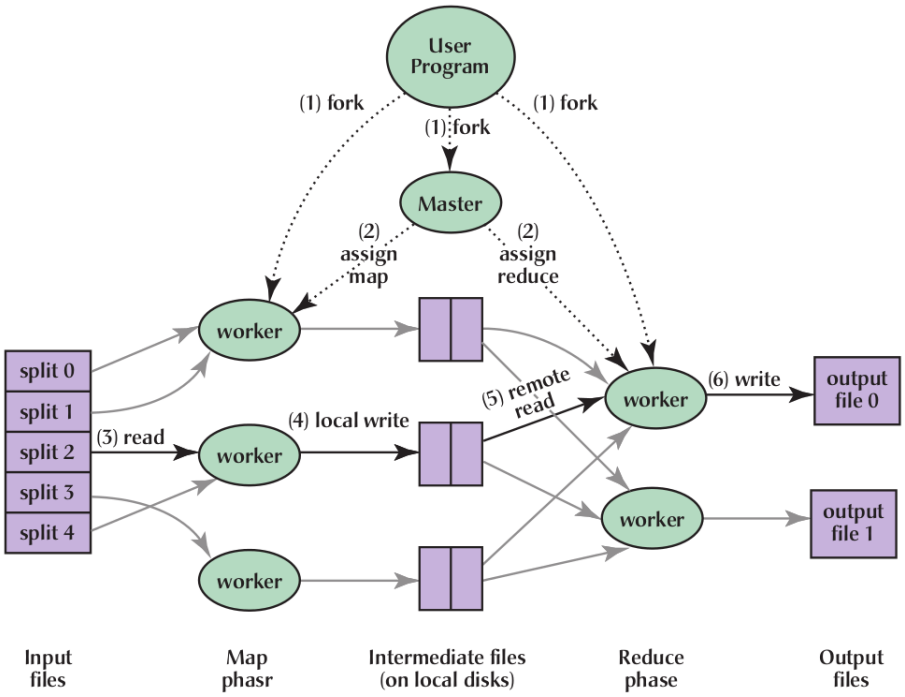

Figure 4: MapReduce execution (reprinted from fig. 1 in [17])

grammer, the map function, and the reduce function. The map function takes an input key-value pair, and produces an intermediate key-value pair, the reducer gets a list of all values associated with a single intermediate key, and produces some output based on this.

In MapReduce the distribution of mappers and reducers across nodes in a cluster is done by the system (figure 4), rather than manually by the application programmer. The system also keeps track of locality, which is supported by HDFS and HBase in the Hadoop stack, to send the computation to the data, rather than the other way around. Fault tolerance is another important feature, when jobs fail, they are simply restarted, perhaps on another node, without it affecting the job. This ability to run tasks independently is also used to do speculative execution, where several copies of a single task is started, and the results from the first task that finishes is used as output. This is done to avoid straggler nodes holding up the job execution.

We use MapReduce to parse, process and retrieve data in GeStore.

We have chosen to use MapReduce because it fits in well with the rest of the Hadoop stack, and the ease of use. The data-intensive tasks we do fit in well with the MapReduce model, the fault tolerance and automatic load balancing capabilities are also important reasons for our choice.

2.1.4 GePan

GePan[2], the metagenomics pipeline system we use to test GeStore. GePan is designed to be modular in nature, such that new tools can be plugged in with ease. It is also designed to be a generic pipeline that can be modified to suit the needs of different metagenomic analysis.

End users specify which tools and meta-data collections to be used by giving them in a list to GePan. GePan then arranges the tools in the correct order, and generates shell scripts that execute the tools in the correct order, as well as handling the parsing of input and output data for the tools.

Jobs in the pipeline are run using the Oracle Grid Engine [25] (previously known as the Sun Grid Engine), which handles the load balancing and scheduling of tasks. The final output of the pipeline is a file containing predicted genes and their annotation, which can be used to locate and identify genes in the input sample.

Adding tools and meta-data collections to the pipeline is a relatively simple task, XML files provide the information the workflow needs to use the tools and meta-data collections in pipelines, and abstract classes provide a framework for implementing tool-specific parsing and execution. For our pipeline we use a subset of the tools available today, the tools are chosen to be representative of a typical pipeline used in research at the time of writing.

We have chosen to use GePan because it is the pipeline creation system used by our collaborators, meaning we have direct access to the source code, developer and end-users. As well as the reassuring fact that it is in production use.

A more detailed description of GePan can be found in section 3.2.

2.2 Incremental computation

There has been much work done in the field of incremental computation, stemming from a need to analyze large volumes of data that are incrementally updated. Here we describe some systems that are closely related to GeStore in terms of incremental computations.

2.2.1 Incoop and Percolator

Incoop [13] and Percolator [14] are two closely related systems for doing incremental updates. They are developed to keep results up to date when faced with updated input data, in particular, Percolator was designed to aid Google in their web search index updated when crawling new sites, while Incoop was developed as a generic extension to the Hadoop framework for incremental computations.

The use cases for these systems are where you have a stream of data into the system, where the data is related to the data already in the system, and you want to integrate the new data into the overall view of the complete database.

Design-wise, these systems are highly distributed and are designed to be working on massive volumes of data. The updates are done by identifying modified chunks of data, and processing only the chunks that are updated. In Incoop, this monitoring is done through the use of IncHDFS, an extension to HDFS that supports content-based chunking, which makes the partitioning

[2]Unpublished system developed by Tim Kahlke at the University of Tromsø

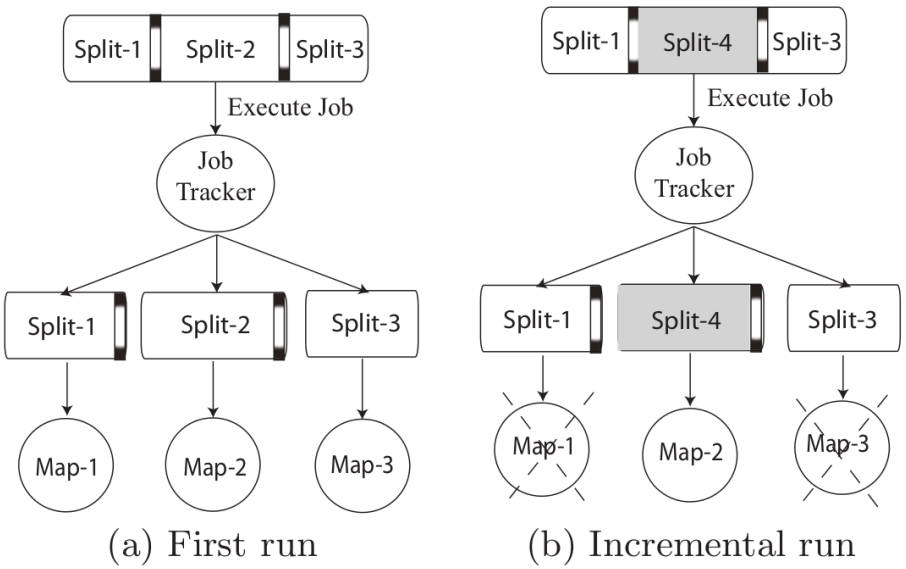

Figure 5: Incoop MapReduce job scheduling (reprinted from fig. 3 in [13])

of data stable when insertions are made. Thus a chunk receiving an update will have a new checksum, while a chunk not receiving an update will not. The checksum is used to detect updates in the data, so that Incoop does not need to re-run map tasks that operate on chunks that have not changed, as illustrated in figure 5. This is handled in a similar way in the reduce phase, to avoid redundant reducers. Percolator handles the change detection within the underlying storage system differently, by using BigTable [26] to execute code when data within BigTable columns changes.

Both systems show great potential for increasing throughput when dealing with incremental data streams. Percolator's evaluation focuses on the decrease in time between when a document enters the system and when it is available for use in their web search index, showing that the median document moves from crawled document to availability 100x faster than the previous system. Incoop was evaluated with synthetic tests, which also showed significant performance gains.

Percolator is in use at Google, while we do now know of any users of Incoop, and the source code is not available for either system.

These systems are incompatible with our requirements for three reasons: (i) the updates to our meta-data collections can effect as much as 50% of entries at different areas of the meta-data collections, this makes the use of IncHDFS in Incoop a performance bottleneck, as we would see updates in all chunks, (ii) we need to export the incremental meta-data collections from the system to do the computations, and be able to store the incremental meta-data collections to increase performance on subsequent runs on the same meta-data collection using different input data, (iii) the computations need to support the MapReduce framework, we do not want to modify the tools used in the computations.

2.2.2 Nectar

Nectar [15] is a system from Microsoft Research for handling data and computations to reduce storage requirements. Nectar stores provenance data and the computations needed to produce the final data for data that isn't accessed often.

Nectar is useful in a storage-intensive field such as metagenomics, as storing the data from every step of computation is not always practical, and discarding the data without preserving the provenance violates the principles of reproducibility.

This system is realized by associating data with the programs that produced them, or in other words, storing the provenance of the data, meaning both the data, and the computation that produced it. This allows the system to discard storage-intensive data that isn't accessed frequently, and recompute it if needed. As a consequence, intermediate computations can be re-used, and throw-away results (i.e. results that are only useful for a short time) can be discarded automatically.

Nectar relies on Dryad/DryadLINQ [27] and TidyFS [28] to supply the underlying framework. This is an alternative to the Hadoop framework, where Dryad replaces MapReduce as the computational component, DryadLINQ has a Hadoop equivalent in Pig [29] or Hive [30], and TidyFS is very similar to HDFS.

The evaluation in the paper shows the savings for applications analyzing a 1-terabyte document collection. By saving intermediate computations instead of recalculating them, they are able to run 4 different experiments using the same preliminary computations, saving over 90% of the computation time per job. There are also some analysis done of 25 production clusters, estimating the computational time saved by using cached results from previous calculations instead of redoing them, showing that most of the clusters can save 20% to 40% of computation time.

At the time of writing, we do not know if Nectar is in use on production clusters, since the experiments in the paper were done on experimental clusters.

Nectar handles append-only increments of the data, which does not conform to our requirements, in addition, it requires computations to be done using DryadLINQ, making it unsuitable for our use, since our requirements mandate that we can not modify the tools performing the computations.

2.2.3 Data warehousing and BLAST

Turcu, Nestorov and Foster [16] describe a system with similar goals to ours, that uses data warehousing methods to generate incremental meta-data collections for BLAST, which is the same application as we are using for our experiments.

The use case for this system is very similar to what we envision for GeStore; the focus is on doing incremental updates using BLAST, without modifying the BLAST source code, through generating an incremental meta-data collection.

The meta-data collections are parsed into a common format for entering into the data warehouse. A BLAST-compatible meta-data collection is generated, BLAST is run against that collection, and the results are put into the data warehouse, as shown in figure 6. Updates are tracked inside the data warehouse to find deleted, updated and new entries.

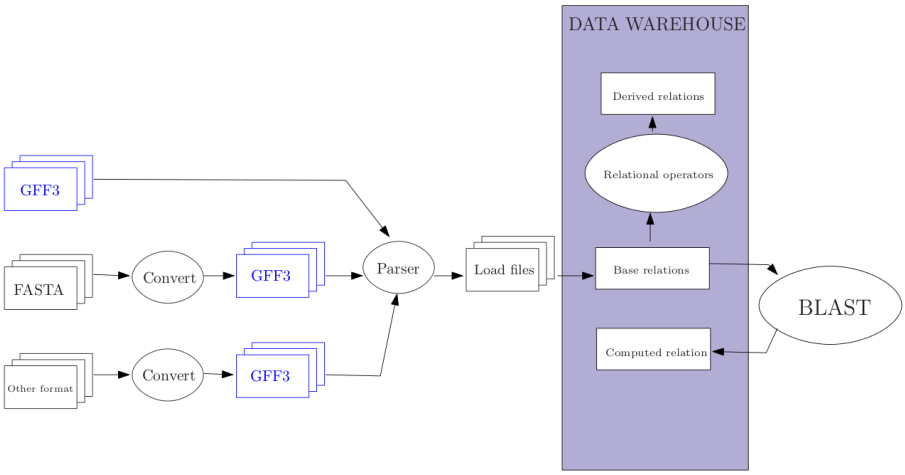

Figure 6: Data warehouse/BLAST illustration (reprinted from fig. 1 in [16])

The evaluation done shows up to 90% reduction in compute time for BLAST using a realistic update percentage (5%), similar numbers to what our experiments show when doing the same work. It is not mentioned if this system uses the BLAST toolkits meta-data collection formatting tool, a custom tool to do this work, or if the meta-data collections are stored in the BLAST format within the system.

It is unclear if this system is in use, or is only experimental.

This system is similar to ours, but the interface and lacking pipeline integration makes it difficult to say how well it would integrate into GePan. The paper does not describe how this system fits into a pipeline workflow system, or how it can be expanded to use other tools than BLAST.

2.3 Storage systems

At the core of GeStore is the storage system duo of HDFS and HBase. Other storage systems offer other benefits and challenges, here we explore some of them.

2.3.1 Data Warehouses/OLAP

Data warehousing and on-line analytical processing [31] are models for analyzing data, enabling users to gain a historical, summarized and consolidated view of the data in the system. The data is arranged in multiple dimensions, allowing users to inspect the data from different angles.

Data warehouses and OLAP are used to analyze data quickly, by inspecting the data along several different dimensions.

The design of the data warehouse is centered around analyzing the data in the most practical way, and as such, operations like rollup (increasing the level of aggregation), drill-down (increasing detail), pivot (rotating the multidimensional view) and slice and dice (projection). These are operations that are meant to aid in the decision support capabilities of the data warehouse solution, and

represent a way to quickly get an overview of the data, as well as being able to investigate the more detailed view when needed.

Data warehousing technology is well-studied, and its widespread use indicates that is useful for many applications.

In terms of suitability to our use, data warehousing offers some features that are useful to us, in particular the ability to look at data points at a specific time is an important requirement to us. However, concepts like drill-down, rollup and pivot aren't obviously useful to our needs. A data warehouse solution could well be used as a replacement for HBase, as it is done in the preceding section, however, the integration of HBase into Hadoop, and our relatively basic requirements for our storage system, makes HBase a more attractive choice, a possible alternative system within the Hadoop framework is Hive [30], but due to issues including immutability of data and lack of row-level updates we found it to be no better than HBase for our use.

2.3.2 SciDB

SciDB [7] is a distributed data management system in the same vein as traditional distributed database management systems, with a focus on scientific data. The biggest difference from traditional relational DBMS, is the array focus, where data is stored not according to a schema, but in a multidimensional array. This fits well with scientific data like the output from DNA sequencing machines.

The architecture of SciDB is a central server that manages the distribution of data on the nodes in the system. Each node stores the data it contains in columnar fashion. The data is further split into overlapping chunks between servers. It should also be noted that data is immutable in this system.

SciDB can be viewed as a possible alternative to the entire Hadoop framework, as it supports very similar operations, from storage to computations, through an interface familiar to those who have worked with relational DBMS in the past. It is also designed to scale to thousands of nodes. One of the biggest differences between the Hadoop framework and SciDB is the modular approach within the Hadoop framework, versus the package supplied by SciDB. This means that if the requirements align closely with what SciDB offers, it will probably be a more suited tool to the task than the Hadoop stack, where custom components are required to mix and match pieces to best suit the requirements.

Unfortunately, as SciDB is still in the early stages of development and deployment, there aren't at the time of writing any papers published evaluating SciDB against the Google framework, so it's difficult to say anything about the performance of the system, or the suitability for different problems.

SciDB was not chosen as the storage backend, since it does not have the same tight coupling with the Hadoop stack, and the lack of widespread use and relative immaturity of the project may make it difficult to use for development.

2.3.3 Ibis

Ibis [32] is a provenance manager for pipelined systems, which tracks provenance in systems composed of several independent sub-systems.

The use case for Ibis is multi-stage systems that do not track provenance in a sufficiently detailed manner, where meta-data produced by the independent

steps is not aggregated into a consistent view of the provenance of data.

Ibis is designed as a database with a query language designed around metadata and provenance queries. It is able to infer provenance relationships across multiple levels of granularity by storing provenance at the highest level of granularity, and associating the different levels with each other.

A prototype implementation was done using SQLite, utilizing query rewriting from the Ibis Query Language to SQL to perform queries. This implementation was not concerned with performance or scalability, only with feasibility of the system.

Ibis, or Ibis-like systems, are a possible future supplement to GeStore, as an important part of our future work is concerned with provenance. Ibis may be useful for maintaining more complex provenance relations than GeStore is currently designed to handle.

2.3.4 Dynamo

Dynamo [33] is Amazon's distributed key-value store. It is a system designed to handle many requests for data, with a very simple interface.

The primary use case for Dynamo is Amazon's web store, where many simple transactions are done, requiring high availability as well as performance.

The design of Dynamo is based around distributed computing principles, maintaining availability in the face of node failures, network partitions, and other typical distributed system faults. It is also designed to be "always writeable", meaning that every effort is spent to maintain writability, as well as keeping the number of hops between the request and the appropriate data node minimal.

The evaluation done on Dynamo shows that it is able to maintain a low response time on a live system at Amazon.

Due to our requirement to inspect data closely, key-value stores are not ideal as a replacement for HBase for us. Although the support for versioning and the simple interface are attractive, our requirements would require a substantial amount of work to leverage the system for our use. Key-value stores could be a replacement for HDFS for us, however, as our usage of HDFS resembles that of key-value stores, where the key is the file name, and the value is the file. One difference here is that Dynamo is designed to handle relatively small files, while we generally operate on large files, so bandwidth is more important than response time.

2.4 Metagenomic pipelines

GePan is one of many genomic pipeline systems, offering different services and approaches to managing pipelines for genomic work.

2.4.1 CloVR

CloVR [7] is a system for doing sequence analysis. It uses local resources, and can provide scalability through the cloud when needed.

The use case is doing sequence analysis on the local computer or the cloud, where small projects can be done locally, and larger projects can be transferred to the cloud as needed.

It is designed as a virtual machine image, which is capable of doing computations on the cloud seamlessly by requesting resources on-the-fly as they are needed. The actual pipeline part is similar to that of GePan, with a similar (but different) set of tools included. What makes the cloud-centric design appealing is the elasticity in resource allocation.

Their evaluation measures scaling in cluster using automatic resource acquisition. The results are mixed, with seemingly sub-linear scaling, although it is hard to interpret the results as different hardware is being used when scaling the system.

CloVR is one possible alternative to GePan, and can be integrated with GeStore, although our hardware is arranged in a small grid, rather than a cloud. In addition, substantial work has been done to support a suite of tools in GePan that are not directly supported by CloVR. CloVR, like GePan, does not support incremental computations.

2.4.2 Magellan

Magellan [34] is a cloud service for scientific applications, in the same vein as Amazon EC2 or other cloud services.

It provides virtual machines to do computations on based on resource requirements. The Magellan project consists of the Eucalyptus [35] cloud package running on a cluster of computers, and using the Hadoop framework for job management.

For this project, Magellan is a possible companion to CloVR, however, we have not investigated the viability of running GePan on a cloud-based infrastructure.

2.4.3 JCVI Pipeline

The JCVI metagenomics analysis pipeline [8] is another metagenomic pipeline system, in use at the J. Craig Venter Institute.

The use case for the JCVI pipeline is similar to that of GePan, involving the analysis of metagenomic samples. The main difference between the use cases is that GePan has the ability to generate more or less arbitrary pipelines, while the JCVI pipeline is more rigid.

The JCVI pipeline is split into two distinct components, structural and functional annotation, as shown in figure 7. Where the structural annotation handles the filtering of non-coding and transport RNA sequences, and attempts to find the best Open Reading Frames for the metagenomic data. The functional annotation component attempts to find the most probably biological roles for the ORFs found in the structural annotation component by using an array of tools. The results are produced by looking at the results for each tool, and selecting the best result based on a ranking of the confidence in each tool. There are plans to integrate more tools into the pipeline.

The JCVI metagenomic pipeline could potentially use GeStore for incremental updates, but it is less attractive for our collaborators due to the rigid nature of the system.

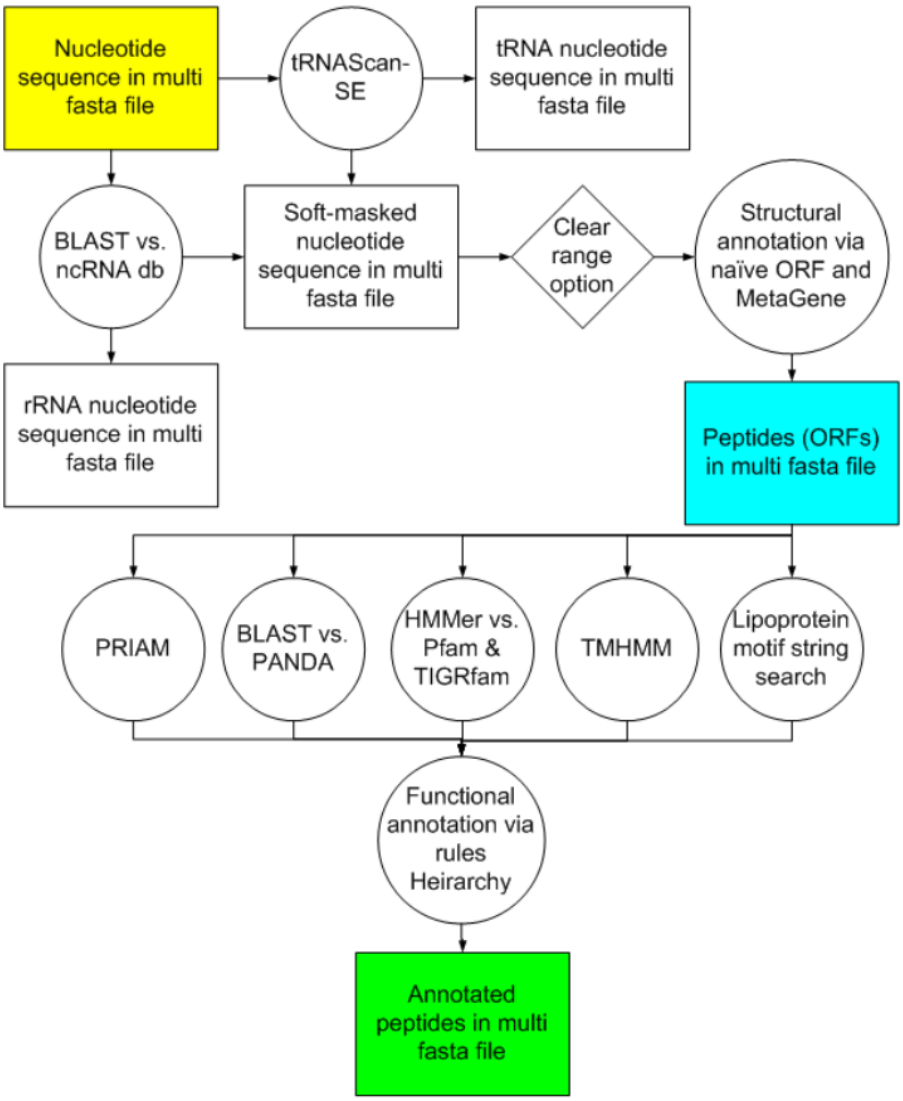

Figure 7: The JCVI metagenomics analysis pipeline, yellow through blue is structural annotation, blue through green is functional annotation (reprinted from fig. 1 in [8])

3 Design

This section describes the design and architecture of GeStore. In section 3.1 we describe the architecture of GeStore by describing how it fits in with a standard pipeline system. In section 3.2 we describe the pipeline execution. In section 3.3 we describe how we store meta-data collections. In section 3.4 we descibe the file storage system in GeStore. In section 3.5 we describe the parallel data processing within GeStore and in section 3.6 we describe the interface to GeStore.

3.1 GeStore architecture

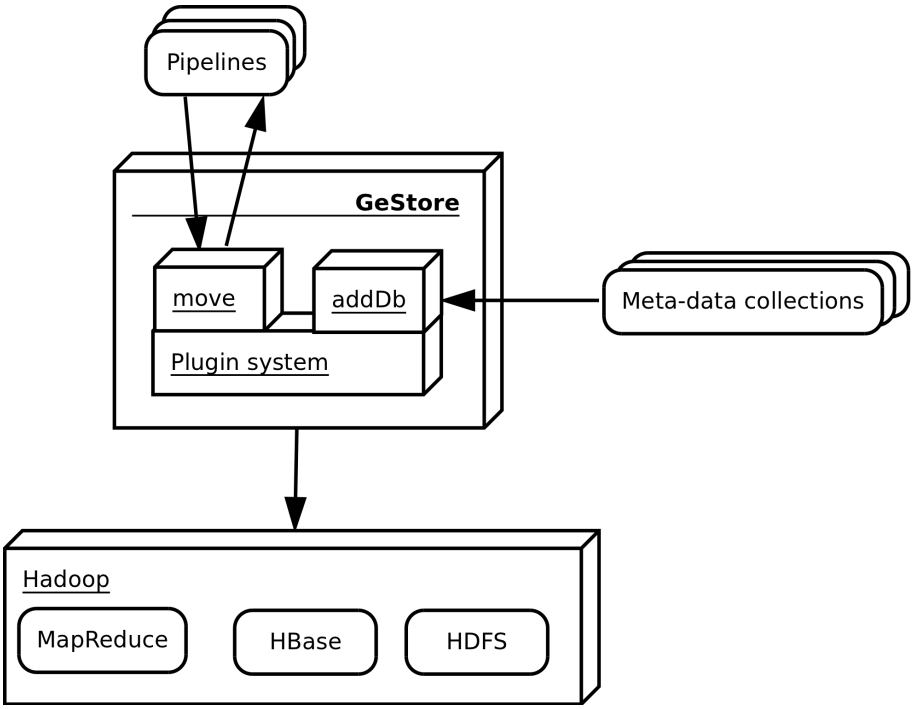

Figure 8: Overview of the architecture of GeStore

GeStore is designed for incremental computations for metagenomic pipelines. To facilitate this, our design reflects the requirements for a minimal interface and transparency for tools in a pipeline.

GeStore lies between the metagenomic pipeline and the storage system (illustrated in figure 8), and provides services for management and storage of files, and for generating meta-data collections for use by the pipeline.

It is designed as a collection of modules, organized into five categories: user-facing, utility, database, plugin and plugin base. The user-facing modules are used to interact with GeStore, and they use the other modules to generate the data requested in the action.

Interactions with GeStore are done through the user-facing modules. The interface between the pipeline and GeStore is provided by the **move** module,

while adding or updating meta-data collections is done through the **addDb** module. The user-facing modules use the **utility** and **database** modules to generate data processing jobs.

Extending the system to support more meta-data collection formats is done through the development of **plugins**. These plugins are based on the **plugin base** modules, which provide the format-independent data processing.

3.1.1 GeStore actions

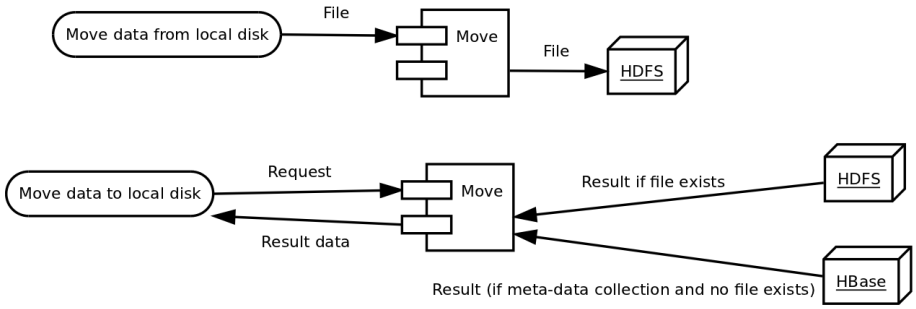

Figure 9: Move to and from GeStore

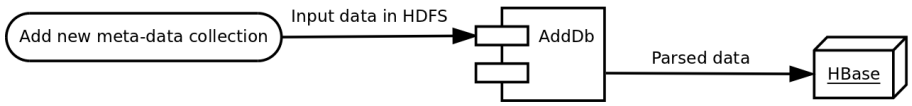

Figure 10: AddDb operation

GeStore supports three basic operations, moving a file to GeStore, retrieving a file from GeStore, and adding meta-data collections to GeStore. The first two operations are illustrated in figure 9, and are handled by the **move** module. The third operation (illustrated in figure 10) is handled by the **addDb** module.

The moving of files (done with the **move** module) is done every step in the pipeline, for every pipeline run.

The **addDb** module is only used when adding new or updated meta-data collections to the system. This is done by the system administrator when meta-data collections are updated.

3.2 Pipeline execution

To illustrate the role of GeStore in a pipeline, it helps to first have a good concept of how pipelines are typically executed, and where GeStore fits into this execution. We use GePan as an example of a typical pipeline system here, but other pipeline systems are similar in their functionality.

Figure 11 and 12 illustrate how GePan generates the pipeline and submits it to the Sun Grid Engine for parallel execution. Our goal here is to not change this execution flow when interfacing with GePan.

A typical step in the pipeline can be described with three operations (fig. 13, getting data from the network file system, operating on the data, moving

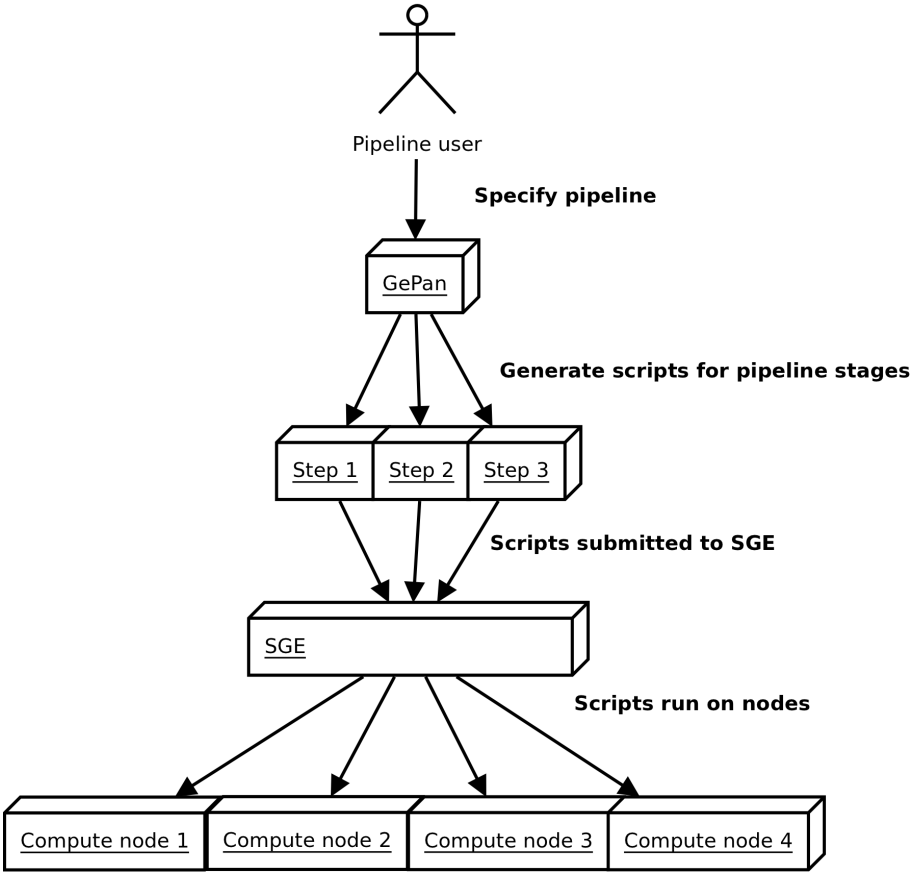

Figure 11: Overview of pipeline execution on our experimental platform. First GePan generates a script for every step in the pipeline, and then it submits these scripts to the Sun Grid Engine, which runs them in parallel on the cluster.

the data to the network file system. Each step in the pipeline follows this basic framework. GeStore interfaces with the pipeline at this level, as can be seen in figure 14 and 15. Movement of files to and from the network file system is replaced with requests to move files to and from GeStore, as described in the next section.

1. GePan parses the arguments and generates a pipeline instance based on the chosen tools.
2. GePan generates scripts to execute the pipeline.
3. The scripts are submitted to the Sun Grid Engine for execution.
   (a) The Sun Grid Engine executes a copy of the script on the nodes.
   (b) When the previous step of the pipeline finishes, SGE submits the next step to the cluster in the same manner.

Figure 12: GePan execution of a pipeline, steps 1-3 are executed once per pipeline, 3(a) and 3(b) are executed once for every step in the pipeline

1. Copy file from NFS to local file system.
2. Run tool with the file as input.
3. Copy results to NFS.

Figure 13: Typical step in the GePan pipeline executed on a compute node

1. Copy file from **GeStore** to local file system.
2. Run tool with the file as input.
3. Copy results to **GeStore**.

Figure 14: Typical step in the GePan pipeline executed on a compute node, when using GeStore

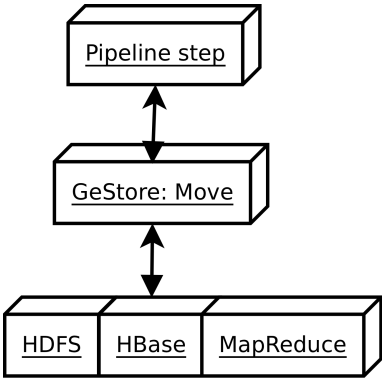

Figure 15: GeStore is used by each pipeline step

3.2.1 Moving files

When moving files to and from GeStore, there are three execution paths the **move** module can take, depending on which request is made and which files have been produced.

In the first execution path, when moving a file to GeStore (step 3 in figure 14), the system will generate a filename, update the internal meta-data, and copy the file to HDFS under the new file name.

In the second execution path, when moving a file from GeStore that exists in the system as a file (step 1 in figure 14), GeStore translates the request to a filename, and copies the file from HDFS to the local disk.

In the third execution path, when moving a file from GeStore that does not exist in the system as a file, but exists as a meta-data collection (step 1 in figure 14), the **move** module determines the appropriate source for the file, and calls on the corresponding **plugin** module to process the data. The **plugin** module then uses the **getfasta** and **getdeleted** modules to generate the files; these are given a file name, and copied to the local disk. Any further requests for this file will then invoke the second execution path.

The distinction between an incremental and a complete execution is transparent to the end user, and semi-transparent to the pipeline (more detailed discussion can be found in section 4). GeStore produces the same set of files when doing an incremental and complete update, with different content.

3.2.2 Meta-data collection updates

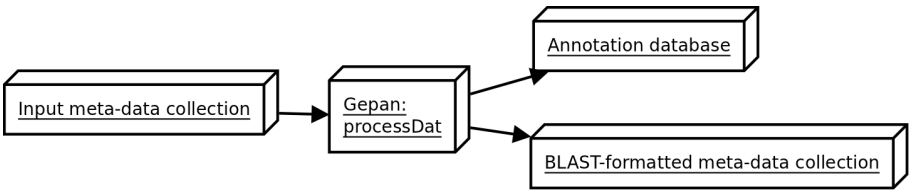

Figure 16: Adding or updating a supported meta-data collection to GePan without GeStore.

Figure 17: Adding or updating a supported meta-data collection to GeStore

The other major action in GeStore is preparing meta-data collections for use by the system. This is done by the system maintainer when a meta-data collection is updated. In GePan, this is currently handled by an application that parses the database files, and generates two data collections for use in the pipeline, an annotation file, and a BLAST-formatted meta-data collection file, illustrated in figure 16.

Adding meta-data collections to GePan (fig. 17) does not produce any files, as GeStore does not create these files at the time of entry into the system, but

generates them on demand when the pipeline is run. As such, the generation of the BLAST-formatted meta-data collection is moved from when the meta-data collections enter the system, to the pipeline run.

3.2.3 Adding meta-data collection support

Adding support for new meta-data collections to GeStore is done through the development of plugins. These plugins are responsible for the processing that is dependent of the format of the meta-data collection, while the plugin base modules handle the data processing that is independent of the internal format of the meta-data collection.

The support for different file formats and meta-data collections is handled by the plugin system, which consists of two parts, the **Entry** and **Source** modules. The **Entry** plugin determines how data is processed when entering and leaving the database, and is as such used by the database category of modules. The **Source** plugin determines which modules are used when extracting data, and any extra processing that needs to be done. This is used by the **move** module to facilitate integration between GeStore and the pipeline being used.

3.2.4 GeStore modules

Module	Type	Function
move	User-facing	Move files
addDb	User-facing	Add meta-data collections
dbUtil	Utility	Handle database operations
DatInputFormat	Utility	Input format parser
LongRecordReader	Utility	Input format parser
getfasta	Database	Extract data from HBase
getdeleted	Database	Extract deleted entries from HBase
genericEntry	Base plugin	Base module for **Entry** plugins
sourceType	Base plugin	Base module for **Source** plugins
Entry	Plugin	Handling a specific file format
Source	Plugin	generating output data for a specific source type

Table 1: List of GeStore modules

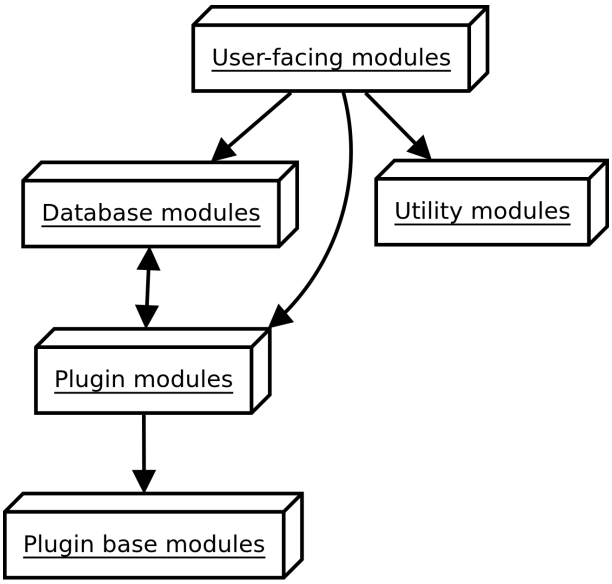

Figure 18: Dependencies of modules

GeStore consists of several different modules, a listing can be seen in table 1, they can be grouped into five categories. The dependencies between these categories is illustrated in figure 18.

- The user-facing modules are the modules that are used to interface with GeStore by the pipeline or directly by the user.
  - **move** moves files to and from GeStore, and creates meta-data collections that are compatible with the tools used.
  - **addDb** adds meta-data collections to GeStore.
- The utility modules provide services to the other modules.
  - **dbUtil** manages the namespace of GeStore within HBase, as well as creating tables if they don't exist.
  - **DatInputFormat** and **LongRecordReader** provide an input format using a regular expression to detect the borders of an entry for the **addDb** module.
- The database modules operate against the database, by running MapReduce jobs to extract or insert data, these use the plugin modules to determine the processing.
  - **getfasta** extracts data from GeStore based on the format specified.
  - **getdeleted** gets a list of entries that do not exist in the current meta-data collection, but have been present in earlier versions of the meta-data collection.
- The base plugin modules do generic processing which is the same for all input types, and provides an interface that the specific plugin modules must implement.

- **genericEntry** provides basic processing capabilities, and specifies the **Entry** interface.
- **sourceType** specifies the **Source** interface.

- The plugin modules implement the interface specified in the base plugin modules, and do the specific processing required for that specific source format.

It should be noted that in our design, the **addDb** module fulfills both a user-facing and a database role, as it starts a MapReduce job to insert data into the database without using a separate database module.

3.3 Meta-data collection storage

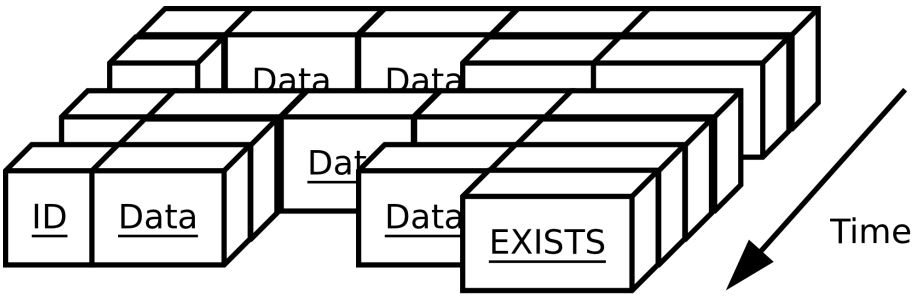

Figure 19: How a single row evolves over time in our HBase storage system

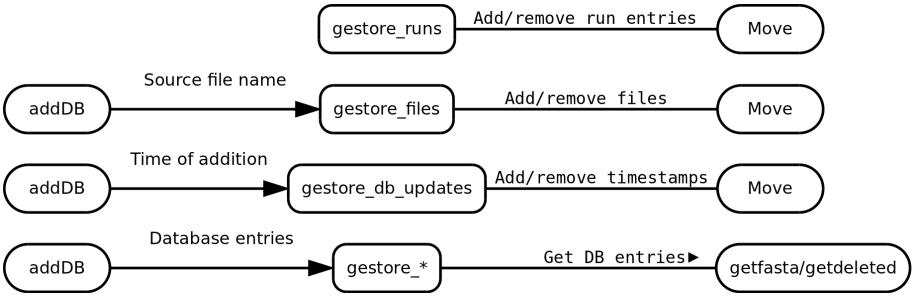

Figure 20: Database use of GeStore modules

In terms of meta-data collection storage, GeStore uses three tables in HBase for internal meta-data, plus one table per meta-data collection, as illustrated in table 2 and their contents and relations can be seen in figure 21. The tables store information about each run, each file and meta-data collection updates. How the different modules use HBase is illustrated in figure 20.

Table name	Table description
gestore_runs	Table that contains information per run of all pipelines.
gestore_files	Table that contains a mapping from file IDs to filenames as well as their source.
gestore_db_updates	Table that contains a listing of all meta-data collection updates.
gestore_*	One table for every meta-data collection, containing meta-data.

Table 2: Description of tables in use by GeStore

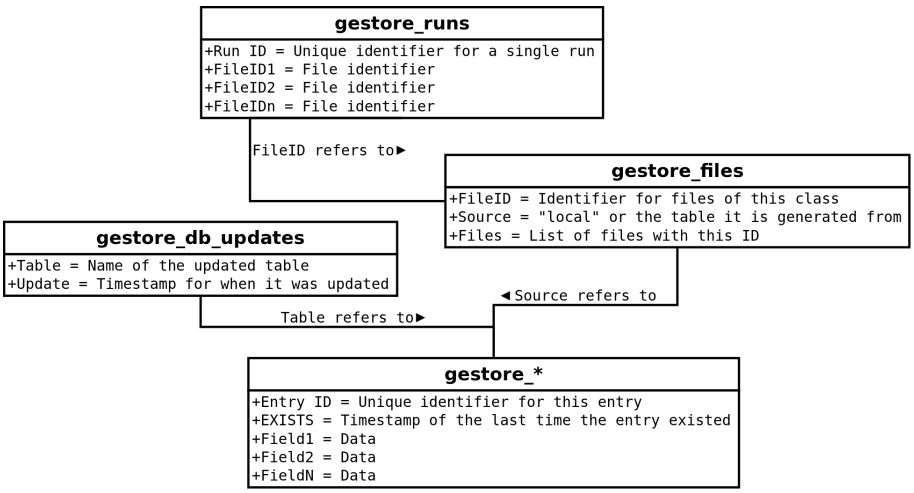

Figure 21: Database schema

1. The meta-data collection update table (gestore_db_updates) contains one row per meta-data collection, containing the timestamp of the meta-data collection in that table, as well as a timestamp for when that update was added to the system.

2. The file table (gestore_files) contains one row per file, with two columns, one for the source of the file (either local, or one of the meta-data collection tables), and one for a list of filenames associated with that file ID. Each file can have many different versions, e.g. different pipelines can use the same file ID, but get different filenames assigned.

3. The run table (gestore_runs) contains one row per run ID (the unique identifier given to a pipeline instance), and one column per file associated with that run, with the value indicating if it was moved to or from the GeStore system. The column name refers to a file ID in the file table.

4. The meta-data collections tables (gestore_*) contain one row per entry in the meta-data collection, a field to indicate if the entry exists in the meta-data collection at the given timestamp, as well as columns for all the fields in the meta-data collection. In the meta-data collection tables

a delta encoding technique is used, only updated fields, as well as the EXISTS field are stored in HBase, this is done to save storage space. GeStore adds a dummy entry in the file table when a meta-data collection is added to the system, this entry has no actual files associated with it, but indicates that the meta-data collection exists, and which table to find it in.

The storage method for the rows in a meta-data collection is illustrated in figure 19. Note that that not every column in each row is stored in HBase, only updated columns and the EXISTS column are stored in every iteration of the meta-data collection. This means that for every entry in the meta-data collection, HBase will return the most recent entry available, if it has not changed, it will be an entry from an old meta-data collection. When doing incremental updates, partial entries are completed if needed, by querying HBase for a complete entry, as not all fields are necessarily returned when looking at a specific time interval.

3.4 File storage

GeStore uses HDFS to store all files, generated by GeStore or input from the pipeline. GeStore generates a unique file name for each file used, containing the information needed to identify it. This filename includes a timestamp interval for which the file was generated, a hash of the regex used to limit the contents, a run ID to associate it with a run (if applicable), a taxon and a file ID. These files are all stored in a single directory in HDFS, and the filenames are associated with their ID in the file table in HBase.

3.5 Parallel data processing

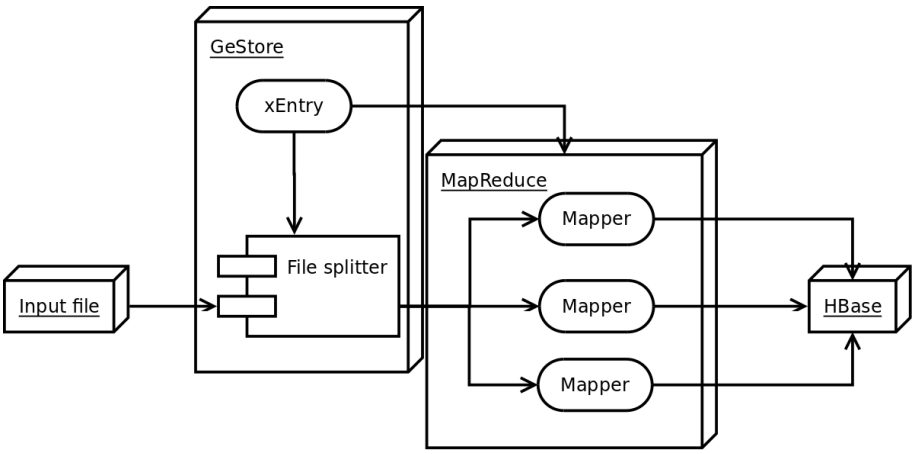

Figure 22: Illustration of how new meta-data collections are processed

GeStore uses MapReduce to provide scalable and fault tolerant data processing, this includes parsing and entering data into the meta-data collection tables, and retrieving data from these tables in the correct format.

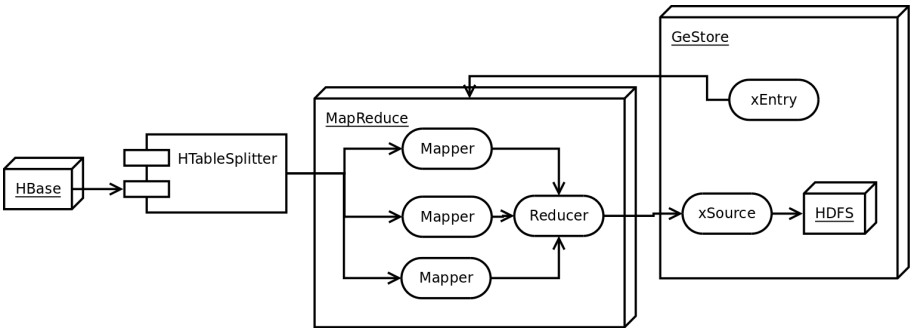

Figure 23: Illustration of how output is generated from meta-data collections

Entering data into the meta-data collection tables is done by the **addDb** application, and is illustrated in figure 22. **addDb** splits the input file into individual entries, each mapper then processes each entry, it is compared to former and future entries with the same ID removing cells where needed, and the modified entry is put into the database.

We have developed a custom file splitter to generate complete input splits for the MapReduce jobs. This file splitter takes two regexes to determine the start and end of a single entry, and is based on the LineRecordReader implementation in Hadoop MapReduce. The file splitter ensures that the mappers receive complete records of the entry.

Each mapper then processes each entry by using a user-specified **Entry** class, which is inherited from the **genericEntry** class in GeStore. This class handles the parsing of that format, and is able to generate Put statements for use with HBase, these Put statements are then executed within the individual mappers, so no reducers are used.

When retrieving data from HBase, another user-specified class is used, based on the **genericSource** class, this class defines how the data is processed, in our implementation, this class calls the **getfasta** component, which uses the same **Entry** class as in the addition phase. Any additional processing is also done here, e.g. the **uniprotSource** class runs the **formatdb** command on the data supplied by the **getfasta** component to generate a blast-formatted meta-data collection, the MapReduce execution of the **getfasta**/**getdeleted** components is illustrated in figure 23.

3.6 Interface

The interface to GeStore consists of the two applications **addDb** and **move**.

The **addDb** application takes a list of arguments consisting of a filename, a name for the database, a timestamp and a format. The filename is the name of the file that contains the meta-data collection in the format specified by the format argument. The format argument is the name of the module used to parse the data in the file. The timestamp is the timestamp of the meta-data collection, the format of the timestamp is a 6-digit user-defined timestamp, which has to be strictly increasing, and the same timestamp format has to be used when using the **move** application.

The **move** application takes one mandatory argument, the file ID, and the

optional arguments are: A run ID, which is the ID for the pipeline run, must be unique to the current run, only used for files, not meta-data collections. A type, which is either "l2r" or "r2l", depending on if you're moving files from the local computer to the remote system, or the other way around. A local path, which describes where the file you're moving can be found if you're moving from local disk, or where to put the file if you're moving to the local disk. Two timestamps, a start and a stop, to determine the interval for which this file should be created, when doing incremental updates, this is the interval between the last meta-data collection update and the target meta-data collection, GeStore will generate these automatically if they are not defined. A delimiter, which is a column name and a regex to limit that column by, e.g. setting this to "OC=.*bacteria.*" will make GeStore only return entries where the column "OC" contains the word "bacteria". A taxon, the function of which is determined by the **Source** type used, and a parameter to decide if the run should be incremental, or if GeStore should ignore previous runs and do a full meta-data collection generation.

4 Implementation

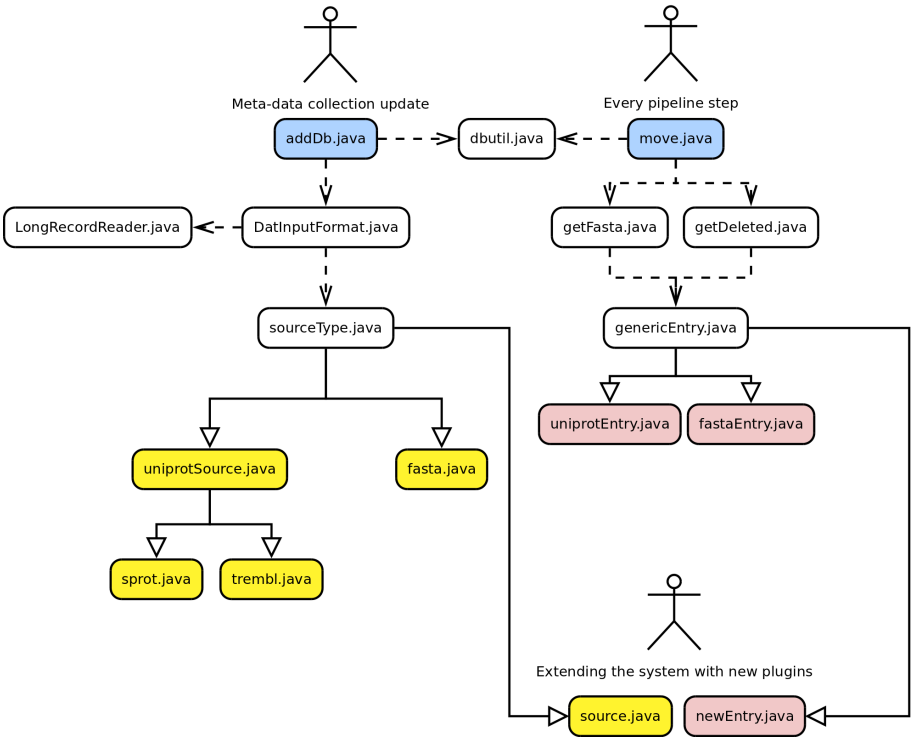

Figure 24: Overview of source files, blue files are used by the end user, red and yellow files are implemented per meta-data collection format, white files make up the core of GeStore

GeStore has been implemented in Java, figure 24 gives an overview of the different modules we have implemented that make up GeStore.

We have also created a helper script (*flatFileToXML*) to parse the BLAST flat-file format to an XML format that the pipeline annotator can handle.

In section 4.5 we describe how we use the Hadoop stack to implement GeStore, and in section 4.4 we describe the implementation details of the plugin system.

4.1 Experimental implementation limitations

Due to the large scale of this problem space, we restrict our system implementation by some limitations.

- We do not attempt to support all the meta-data collections and tools supported by the pipeline system we have chosen in the experimental evaluation. Tools and meta-data collections that are not currently supported will be run non-incrementally. We only provide support for the BLAST tool and meta-data collection for our experiments.

- We do not integrate GeStore with other workflow tools than GePan in our experimental evaluation. Although the integration process is relatively simple, we have chosen to only focus on GeStore, as this is the workflow system used by our collaborators, and as such is the one we are familiar with.
- We do not replace any of the custom steps in GePan, including annotation, file splitting and exporting results. This is an important limitation because some of these steps could benefit from using GeStore, but we have this as future work.

4.2 Moving files

When moving files to or from GeStore, the **move** module is used, it does most of the processing that is independent of the Hadoop stack.

- The application starts by parsing the arguments.
- Tables are generated if they don't exist, using the **dbUtil** utility module.
- The source is determined, if the file is moved from local disk, the source is local, if the file is moved to local disk the source may be local, or the name of a table to get the file from.
- The stop timestamp(t_{stop}) is determined, if the source isn't local, this is determined by min(last update to database, t_{stop} passed in).
- The start timestamp(t_{start}) is determined, if the source isn't local and no t_{start} was passed in, and full_run is false, this is determined by looking at the previous execution of the current run in the run table.
- The file path is determined by using the previously determined parameters.
- If the file doesn't exist, an incremental file is generated by calling the process() method of the **sourceType** for the source. The generated file is put into the file db.
- Finally, the file is moved, depending on where it is moved to and from, and what type it is, entries are added to the run table and the file table.

4.3 Meta-data collection updates

Meta-data collection updates are handled by the **addDb** module, which is relatively simple, and relies on the custom plugin to do most of the processing.

- The application starts by parsing arguments.
- Tables are generated if they don't exist, using the **dbUtil** helper class.
- MapReduce is set up and jobs are started, using the input split done by our file splitter, based on the regex of the custom plugin.
- Each mapper does this;
  - Add the entry to an **Entry** object by using the user-specified plugin.

- Get an entry for the same row with timestamp interval $[t_{min}, t_{entry})$.
- Get an entry for the same row with timestamp interval $(t_{entry}, t_{max}]$.
- Generate **Entry** objects for the newer and older entries obtained in the previous steps (if they exist).
- Compare the current entry to the older entry and generate a partial put containing the new fields, using the timestamp from the arguments.
- Compare the current entry to the newer entry, and deleted entries in the newer entry that are equal to the current entry.
- An EXISTS field is added to the Put.
- Finally, they execute the Put on the table specified in the arguments.

4.4 Plugin system

Application programmers who wish to extend the support of meta-data collection formats for GeStore, need to implement two classes which are loaded at runtime as needed by GeStore.

- The **Entry** class (**fastaEntry**, **uniprotEntry**) is based on the **genericEntry** class, and describes the format of each entry in the meta-data collection, how to parse it and represent it in the database, as well as handling the different output schemes associated with the format.
- The **Source** class (**uniprotSource**) is based on the **sourceType** class, and does the processing on the data, in the Uniprot case it calls the **getfasta** module to get a file in a pre-determined format, and does post-processing required to use the file in a system. For the uniprot-class of formats, this involves calling the formatdb application from the BLAST toolkit, and calling the **getdeleted** module to get a file containing the list of all deleted entries for the given timestamp.

Adding a new meta-data collection format type to the system is done by implementing 6 methods:

boolean addEntry(String entry)
Parses a string to into a format usable by HBase.

Put getPartialPut(Vector< String > fields, Long timestamp)
Returns a Put over all the fields passed in.

int sanityCheck(String type)
Check if the entry is well-formed.

byte [] getRowID()
Returns the ID of the entry.

String [] get(String type, String options)
Returns a list of strings that represent the entry in a given format.

Vector<String> compare(genericEntry entry)
Returns a list of different fields between two entries.

The **genericEntry** class provides the following methods:

boolean addEntry(Result entry)
Parses a HBase result to add all cells.

String [] getRegexes()
Returns the default start and stop regex for parsing input data.

boolean addEntries(String [] entries)
Calls addEntry() for every string in the entries array.

boolean exists()
Returns true if the entry has at least one cell.

String getTableEntry(String key)
Returns the value stored in the given column.

Long getTimestamp(String field)
Returns the timestamp of the given column.

Vector<String> equalFields(genericEntry entry)
Returns a list of fields that are equal between the two entries.

Vector<String> getDeleted(genericEntry entry)
Returns a list of fields that do not exist in the given entry, but do exist in the current entry.

The **sourceType** class is an abstract class for the **Source** classes, and exposes one abstract method that has to be overwritten;
FileStatus[] process(Hashtable<String, String> params, FileSystem fs) . This method takes some parameters describing the state of the system (e.g. the **uniprotSource** process method requires a file ID, timestamps, a taxon, a source and a delimiter), and a file system handle, and returns a list of files which represent the results of the processing to be used by GeStore.

In the case of **uniprotSource**, the process method calls the **getfasta** method, does a formatdb on the results, calls **getdeleted**, and returns a list of files on the local file system composed of the BLAST-formatted meta-data collection and the list of deleted files.

Our support for the Uniprot Knowledge Base (**uniprotEntry**) supports DAT as an input format, and DAT and FASTA [36] as output formats.

4.5 Hadoop stack

We use Cloudera's Distribution Including Apache Hadoop (CDH) 3 update 3 to supply the Hadoop stack. GeStore uses the Hadoop framework in several ways: Files are stored in HDFS, meta-data collections and file meta-data is stored in HBase, and processing is done using MapReduce.

4.5.1 Meta-data collection storage

We store meta-data collections in HBase. In our usage of HBase, we leverage the timestamp feature of the database to implement our storage schema. We

use a timestamp of the format YYYYMM in the Uniprot meta-data collections for our implementation, this timestamp is associated with the data in HBase.

When HBase is given a time range to do queries over, it will select the newest data with the timestamp that matches $timestamp_{stop} > timestamp > timestamp_{start}$. This means that with no $timestamp_{start}$, the latest cell in every row is selected, up to the $timestamp_{stop}$. When a $timestamp_{start}$ is chosen, HBase will not return cells that have not changed between $timestamp_{start}$ and $timestamp_{stop}$. This can cause problems where a partial entry is returned, which does not contain all the cells required to produce a complete output. We have solved this by the **sanityCheck** method in the plugin class, which checks if the fields required to produce output are in the result given by HBase, if it is not, then a full entry will be requested from the database, and returned.

4.5.2 File storage

We use HDFS for file storage services. We only use a small subset of the features of HDFS directly, MapReduce and HBase handle the more advanced features that HDFS is designed to support. We use the default replication factor of 3 on the cluster.

What is interesting to us here is the collaboration between HDFS and MapReduce to create input splits, as well as the high throughput of HDFS. This allows us to move large files quickly between the local disk and the distributed file system. The files we store in HDFS are all the user-supplied files given to the **move** application to transfer to GePan, and the results from generating meta-data collections. The **move** application itself only adds and retrieves files from HDFS, and does not do deletions, but detect them when they occur and produce new files as needed.

4.5.3 Parallel data processing

We use MapReduce to facilitate our data processing. Our use of MapReduce consists of three separate modules, which generate MapReduce jobs; **addDb**, **getfasta** and **getdeleted**. These modules use the plugin system to parse the data to/from the system.

- The **addDb** module takes a file (located on HDFS) as input, and does the splitting based on a custom regular expression defined in the format-specific plugin defined as an argument. These splits are then given to the mappers, and each mapper generates one row per entry in the meta-data collection. The mapper attempts to get the previous and next entries in the timestamp range (so for timestamp 10, it will attempt to get 0-9 and 11-max-int). If there exists a previous entry, it will generate a Put for HBase containing only the updated columns, and the EXISTS column. If there exists a later entry, it will generate a Delete for HBase containing the equal fields, excepting EXISTS. This maintains the data in HBase correctly.

- The **getfasta** module uses the HBase-supplied input splitter, which gives one row to each mapper. This row is checked against the supplied regular expression, and if it matches, the entry is sanity-checked, and a full entry

is retrieved if necessary. The format-specific plugin then generates output based on the output format given as an argument to **getfasta**.

- The **getdeleted** module also uses the format-specific plugin to parse the data from HBase, but only checks if the EXISTS field matches the timestamp passed in. If the timestamp of the EXISTS field is lower than the passed-in timestamp, then the row ID is emitted, otherwise the entry exists in the current version of the meta-data collection, and nothing is emitted.

5 Integration case study

In this section we describe how GeStore has been integrated with GePan, in section 5.1 we describe the details of the pipeline execution, and in section 5.2 we describe the changes we have made to GePan to integrate GeStore.

5.1 GePan workflow

The GePan-generated pipeline consists of 6 steps, executed in sequence. The following step does not start before the preceding step is completely finished.

1. The pipeline starts out with Glimmer3, which is a gene prediction tool.
2. This is followed by the Glimmer3 exporter, which converts the output from Glimmer3 to FASTA format. The Glimmer3 exporter is an integrated part of GePan.
3. The file partitioner (FileScheduler) splits up the input file into N parts, where N is the number of threads to be run. The FileScheduler is an integrated part of GePan.
4. After this, BLASTp is run against the Uniprot meta-data collection, for transferred annotation.
5. The annotator gets the hits from the output of the annotation tools, adds the annotation data to these hits, and combines them into result files. The annotator is an integrated part of GePan.
6. The exporter then combines the annotation data from the annotators into a single result file of a format specified by the user. The exporter is an integrated part of GePan.

The input to the pipeline is assembled contigs stored on a NFS network drive, and each step of the pipeline copies the file(s) from the network drive to local disk, does the computations, and copies the results back to the network drive. This input consists of genomic data in a FASTA file, which may be output from a sequencing machine, already-sequenced genomes, or subsets of genomes.

5.2 GeStore integration

As GePan uses Perl scripts to generate shell scripts that are submitted to the SGE job queue, the changed required are made in the script generation portion of the system. "mv" and "cp" commands are changed to appropriate **move** commands, and two new steps are created.

The first step is an additional step in the pipeline, done before any of the processing in the pipeline is done. This generates or copies a meta-data collection by doing a **move** operation which is identical to the ones done in the BLAST step. This is done before the pipeline starts to avoid starting up more jobs than is needed, and is required due to the combination of the Sun Grid Engine and the Hadoop stack.

The second is an expansion of the BLAST step, and is performed in the script where BLAST is executed. This combines the result files if there is a previous

version, and removes hits that have been removed in the meta-data collection. The combination of files is a simple append operation (using the cat Linux application), and the removal of deleted entries is done in the flatFileToXML application, which is a small python script designed to convert the output data from BLAST from the tab-delimited format to the XML format required by the annotator.

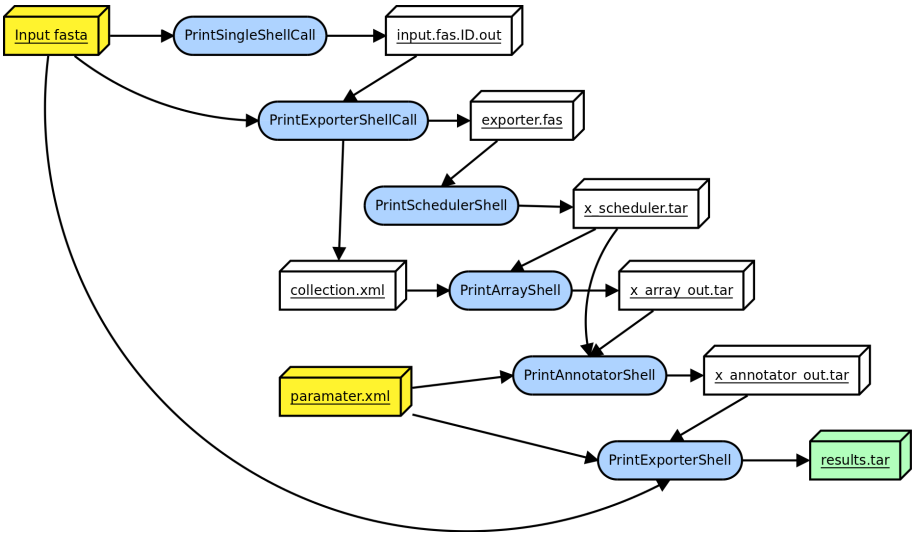

Figure 25: Data flow within the pipeline scripts in GePan. Yellow are input files, blue are script generation methods, green is pipeline output, and white are files generated throughout the pipeline.

For the steps that do not use the full capabilities of GeStore, simply changing "mv" and "cp" to **move** commands and packaging files appropriately is enough to integrate GeStore. These changes were made in the six methods of GePan that generate script files (the blue methods in figure 25).

In addition, a new meta-data collection definition file was created for GePan, to add support for the meta-data collections generated by GeStore.

6 Evaluation

In our evaluation, the important questions we want to answer are how much overhead does GeStore add, how much time can be saved by using GeStore for incremental updates, how complex is GeStore and how difficult is it to integrate with existing pipeline systems.

6.1 Hardware

Our experiments were done on a small cluster of 5 nodes, with one computer working as a master, with four compute nodes. This cluster size is realistic for a small data center for local metagenomic analysis. Each node is equipped with two Intel Xeon E5620 CPUs running at 2.4 GHz and 24 gigabytes of RAM. They have a total of 4.5 TB of local HDD storage distributed between 3 disks. GePan only uses one 1.5 TB disk. They also have 2.6 TB of NFS storage shared between them. The machines are networked using gigabit ethernet.

The experiment described in section 6.3.2 was done on a desktop computer, equipped with a dual-core Intel Pentium 4 CPU running at 3.2 GHz and 3 gigabytes of RAM.

6.2 Software

The cluster was running CentOS 5.6 (Linux 2.6.18) on all machines. We used Cloudera's Distribution Including Hadoop (CDH) version 3 update 3, which includes HBase 0.90.1 and Hadoop 0.20.2. The MapReduce setup was configured to use 32 simultaneous map tasks. HDFS was configured with the replication factor of 3. The version of GePan we used was a modified version of a branch taken in the fall of 2011, and we used the Sun Grid Engine version 6.2 update 5. SGE was configured to use all 4 compute nodes. The pipeline we used included Glimmer 3 [9] and BLAST version 2.2.21. BLAST was run against varying versions of the Uniprot Knowledge Base, starting at the one released in January 2010, and ending in December 2011. We used both the Swiss-Prot and TrEMBL parts of the Uniprot Knowledge Base. The input data used is a subset of metagenomic data collected for a project in Yellowstone national park [37]. We have used around 15 MBP of the total set of around 30 MBP, this was done to reduce the time required for the experiments, as our cluster is shared between multiple projects.

These tools and data were verified as relevant choices by our biology collaborators.

The computer used in the experiment described in section 6.3.2 was running Ubuntu 11.04 (Linux 2.6.32), and used BLAST version 2.2.21.

6.3 Preliminary experiments

In our preliminary experiments, we evaluate how BLAST runtime scales with the size of the meta-data collection. This is important to find out if doing incremental meta-data collection generation will have an effect on the total runtime of the pipelines. We also examine the updates in the UniProtKB meta-data collection to estimate the size of the incremental meta-data collections, as

well as examining the scalability of BLAST with regards to the number of CPUs used.

6.3.1 BLAST parallelism

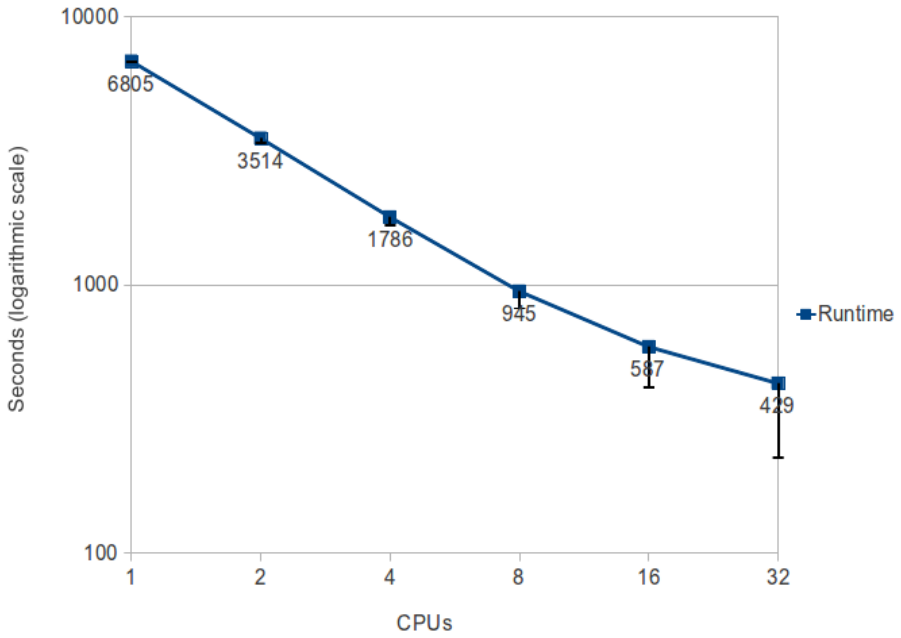

Figure 26: BLAST scaling based on number of CPUs used, error bars show potential time lost due to load-balancing (reprinted from fig. 14 in [21])

In previous work [21] we have shown that BLAST scales well with regards to the number of CPUs used, as seen in figure 26. This experiment was done on the same cluster as we have used for the rest of the experiments, using GePan to manage the concurrent execution of BLAST over the input. The results show that increasing the number of CPUs used leads to an approximately linear decrease in runtime,

6.3.2 BLAST scaling with regards to meta-data collection size

We examine the effect the meta-data collection size has on BLAST runtime when analyzing the same input data, in order to find out how BLAST scales with varying meta-data collection size. This is accomplished by taking the first N entries, starting at 1000 and increasing by 1000 up to 20000 for each BLAST execution, of the UniProtKB/Swiss-Prot meta-data collection from December, 2010, and doing a BLAST search over this partial meta-data collection. We used the same input data as in the other experiments. This experiment was not done on the cluster, but on the local workstation, since we are only interested in the isolated scaling of BLAST, and not the relationship with the rest of the pipeline. As shown in the preceding section, BLAST shows good scaling when

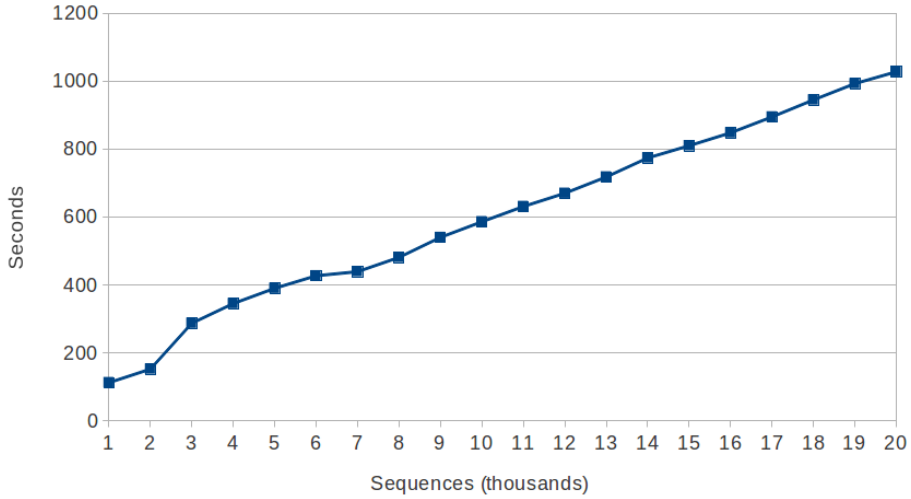

Figure 27: BLAST scaling based on meta-data collection size

increasing the number of CPUs, so the results when doing this on a cluster should be similar. We use the same version of BLAST as on the cluster.

Figure 27 shows that the BLAST runtime scales linearly with the number of sequences in the meta-data collection. Thus, reducing the volume of meta-data supplied to BLAST should result in a corresponding reduced runtime (e.g. reducing the sequences by 90% should reduce the BLAST runtime by 90%). This is comparable to the results obtained by Turcu, Nestorov and Foster [16].

These results show that reducing the size of the meta-data collection will result in a correspondingly reduced runtime of BLAST calculations.

6.3.3 Updates to meta-data collections

We have examined at how large the monthly updates to the meta-data collections are and what types of updates these are, in order to estimate how large the input data to BLAST can be expected to be when doing incremental updates. When discussing the types of updates to the meta-data collections, we make a distinction between relevant and irrelevant updates to the BLAST computations on the meta-data collection. We have chosen the TrEMBL and Swiss-Prot meta-data collections from October, 2011 as a representative example of our findings, and the results can be seen in figure 28.

These numbers are similar for other periods of similar length in the meta-data collections, and when evaluated in relation with the numbers in figure 27 they show that the potential for decreased runtime is large, potentially up to 95%. We can see this from the BLAST-relevant updates column in figure 28, as this is the percentage of entries in the meta-data collection which lead to differences in the BLAST runs, and as such is the percentage of the complete meta-data collection we use for our incremental meta-data collections.

Collection	BLAST-relevant updates	Annotation updates
TrEMBL	3.71%	42.65%
Swiss-Prot	0.28%	82.34%

Collection	New entries	Deleted entries	Total entries
TrEMBL	625 848	468 139	16 869 266
Swiss-Prot	1 294	621	532 146

Figure 28: Changes in the TrEMBL and Swiss-Prot meta-data collections between September and October in 2011, BLAST-relevant updates represents the percentage of entries which has updates the data used by BLAST, annotation updates is the percentage of entries which have updates to the data which is not used by BLAST.

Conclusion

From these results, we conclude that doing incremental updates of BLAST results using incremental meta-data collections should reduce execution time when compared to using complete meta-data collections, since (i) the small amount of BLAST-relevant updates to the meta-data collections per month means that the size of the incremental meta-data collections will be small, and (ii) the scaling of BLAST based on meta-data collection size shows that the size of the meta-data collection is an important factor in the runtime of BLAST.

6.4 Overhead

To evaluate the overhead imposed by GeStore, we measure the time it takes to generate a BLAST-formatted meta-data collection from scratch. This represents the worst-case overhead that is added by using GeStore.

We also measure the overhead of GeStore without meta-data collection generation, by using a pre-generated meta-data collection, which is comparable to the current system, where generation of the meta-data collections is not done at runtime.

We also evaluate the storage requirements of GeStore, compared to storing the meta-data collections individually.

6.4.1 GeStore overhead

We have chosen to use a subset of metagenomic data collected for a project in Yellowstone national park. We have used around 15 MBP of the total set of around 30 MBP, this was done to reduce the time required for the experiments from around two weeks to around one week, as our cluster is shared between multiple projects. Figure 29 shows that BLAST runtime scales linearly with input size, and as such, doubling the input size will approximately double the runtime for BLAST.

The first question we wish to answer, is how much overhead does GeStore add to pipeline runs. Looking at figure 30, we see that GeStore imposes a significant overhead when generating a new BLAST-formatted meta-data collection, and a much smaller overhead when a BLAST-formatted meta-data collection

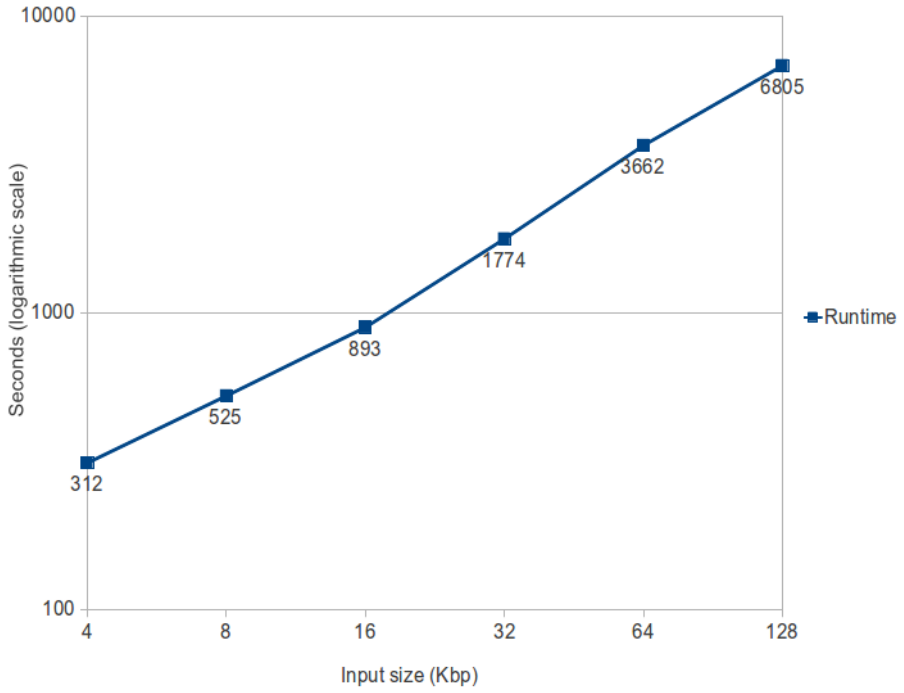

Figure 29: BLAST scaling based on input size (reprinted from fig. 12 in our previous work [21])

already exists. The total runtime for the pipeline is 23336 seconds when using unmodified GePan, using GeStore when a meta-data collection has already been generated takes 10% (2345 seconds) less time for the pipeline, and using GeStore to generate a collection makes the pipeline take 15% (3406 seconds) longer to run. Note that the decrease in time spent in the annotator when using GeStore is due to the much smaller XML files that we generate when compared to the unmodified GePan.

We consider this overhead of 15% acceptable in this case, since (i) the pipeline execution we have chosen is a short one, and ordinary pipelines are expected to take longer, while the overhead of GeStore remains constant, and (ii) several pipelines are expected to share precomputed meta-data collections, so most pipelines should be able to use existing meta-data collections.

6.4.2 Storage requirements

Secondly, we want to know what effect GeStore has on storage requirements for the meta-data collections. Figure 31 shows that although GeStore imposes a larger initial storage requirement for the meta-data collections, the storage requirements grow much slower than the meta-data collections if stored individually, 2 years of meta-data collections requires 806 GB of storage space when stored individually, but only 165 GB of storage space in GeStore, a reduction of almost 80%. This is because most of the data in the meta-data collections is not modified, and since we can find column-level differences, we only need to store the columns that are updated when a collection is updated, and re-use the

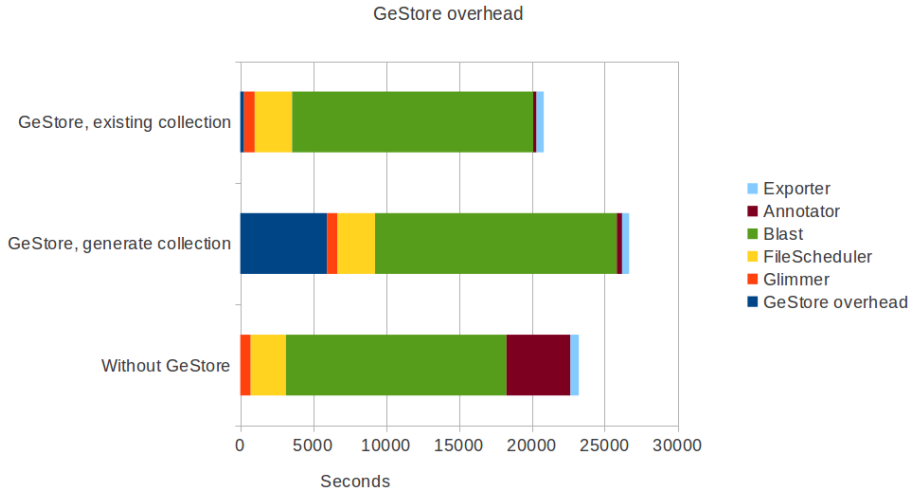

Figure 30: GeStore runtime overhead measurements

old columns that were not updated.

To put these numbers into perspective, the storage requirements for these 24 meta-data collections in GeStore is less than four times the storage requirements on disk for the meta-data collection from December of 2011 (the most recent meta-data collection used in this experiment). We consider this to be more than acceptable. We expect that the size of the meta-data collections in the live system will exceed several terabytes.

6.4.3 Conclusion

We find that GeStore imposes a significant runtime overhead when generating an incremental meta-data collection. However, this overhead is constant with the size of the input data, and as such, when running larger experiments, the fraction of the pipeline runtime spent on the GeStore overhead is smaller.

We also find that when compared to storing the meta-data collections individually, GeStore is able to reduce storage requirements significantly.

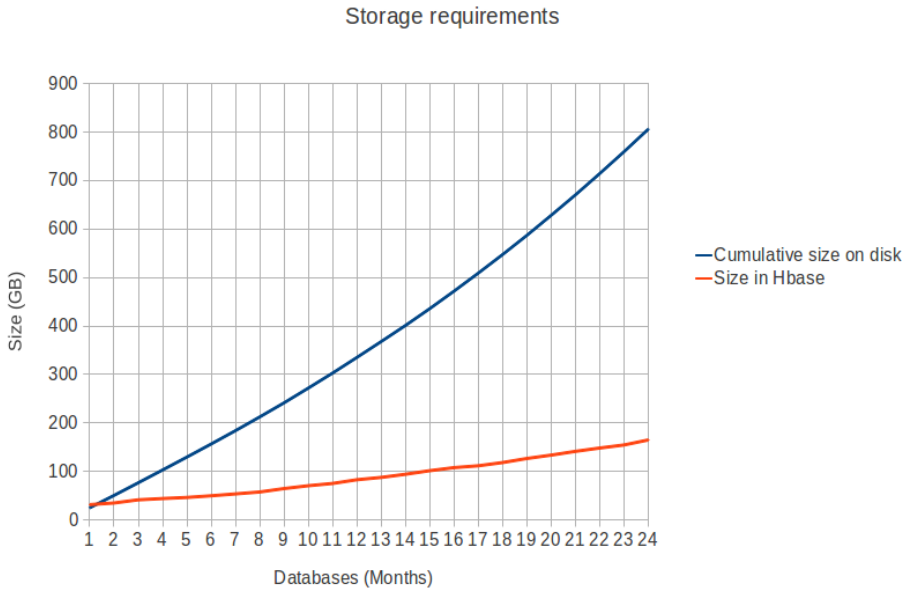

Figure 31: Non-replicated storage requirements

6.5 Incremental updates

We measure how much time is saved by BLAST when using an incremental meta-data collection, compared to using the full one, when running a realistic experiment.

Figure 32 shows that the runtime of BLAST is reduced significantly. In the case of a 1-month update, the runtime is reduced from 11508 seconds, down to 828 seconds, a reduction of 93%. The runtime of the entire pipeline is reduced from 20462 seconds to 7114 seconds, a reduction of 65%. The difference between the reduced BLAST runtime and the reduced pipeline runtime is in part due to the overhead imposed by GeStore (2816 seconds for the 1-month update), and partly due to inefficiencies in other parts of the pipeline. Our results also show that doing a 5-month update takes less processing time than doing 5 1-month updates. It is therefore better to do these updates on demand, rather than continuously.

The GeStore overhead is smaller when doing a 1-month incremental update than when generating a full meta-data collection or doing a 5-month incremental update, this is due to the formatdb command, which is run to format the output from GeStore into a BLAST-compatible database. With a smaller meta-data collection, the formatdb command takes less time to complete, leading to a smaller GeStore-imposed overhead.

Also note that, as seen in the previous section, when meta-data collections are already processed, the GeStore overhead is significantly reduced. This also holds true for incremental meta-data collections.

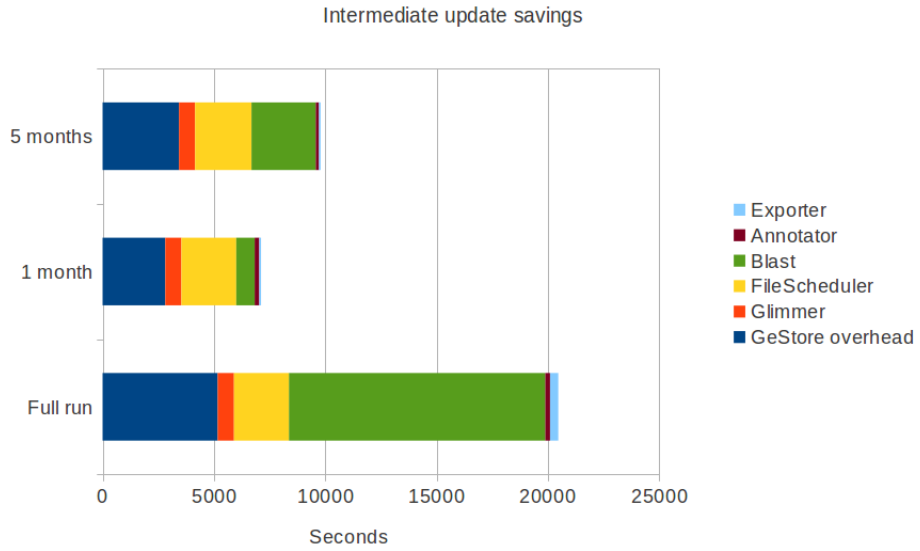

Figure 32: GeStore incremental update measurements

6.5.1 Conclusion

These results show that GeStore can save a significant amount of time when doing incremental updates compared to doing full runs of the pipeline, and that doing these updates on demand rather than continuously means the total time spent doing updates is reduced.

6.6 Complexity of GeStore

To evaluate the complexity of the GeStore implementation, we count lines of code. This does not directly correlate with complexity, but gives an indication.

The core of GeStore is around 1300 lines of code, these are the components in white and blue in figure 33, and does not include the flatFileToXML script.

The support for the different formats (uniprot and FASTA), come in at 390 lines of code, with 250 of them coming from the uniprot support, and 140 from the partial FASTA support. Note that the FASTA support is incomplete, as only adding FASTA files to the database is currently supported, and the retrieval of FASTA files is not implemented in the current version of GeStore. The implementation is trivial compared to the Uniprot implementation. This is worth noting because there are relatively few and simple file formats in use by the different tools, so only a handful of formats need to be supported by GePan to support a large range of tools. In comparison, the code to parse the XML output from BLAST into the format used internally in GeStore is 180 lines of code.

6.6.1 Conclusion

We conclude that most of the complexity in GeStore comes from the core modules, and that developing plugins is not a large undertaking when compared to similar components in GePan.

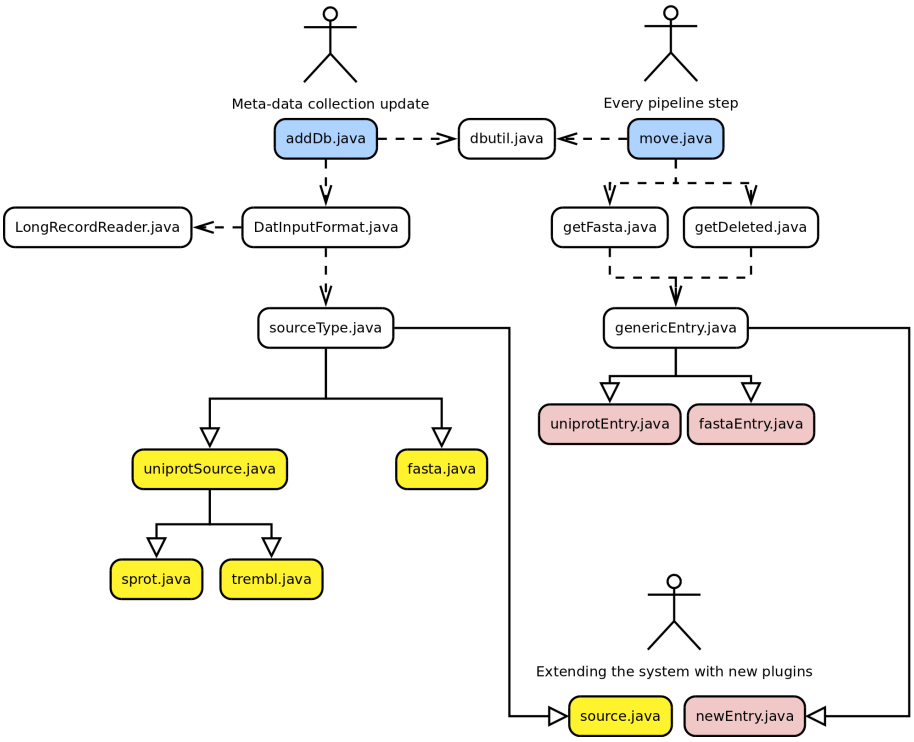

Figure 33: Overview of source files, blue files are used by the end user, red and yellow files are implemented per meta-data collection format, white files make up the core of GeStore

6.7 Ease of integration

When evaluating the effort required to integrate GeStore into GePan, we have examined the changes required in GePan to use GeStore. We have not included the parts of GePan that can be replaced, or are no longer necessary, when using GeStore into these calculations.

We found that 70 lines of code were changed to use GeStore, not including the code required to maintain backwards compatibility with the old system. For comparison, the XML database definitions in GePan which support the Uniprot Knowledge Base are 91 lines of code, and are all made redundant by using GeStore.

6.7.1 Conclusion

We conclude that integrating GeStore into an existing pipeline system is not a large undertaking. Our results show that the overall complexity of GePan when measured in lines of code is reduced by using GeStore.

6.8 Correctness

The integration was tested for correctness by examining the final output from the pipeline, and was found to identify the same genes when using GeStore as

when not using GeStore. This was done for incremental and full updates when using GeStore, and compared with using unmodified GePan.

We found that while using GeStore and not using GeStore both result in identifying the same genes, there are some differences in the final output. Formatting differences between the XML generated by BLAST and the XML generated by the flatFileToXML script are propagated to the end result. More severe is the difference in the E-values generated when doing incremental updates, this is discussed in further detail the next section.

6.9 Discussion

The results obtained in the evaluation highlight some important issues. Perhaps the most obvious is the reduced time spent in the annotator when using GeStore. This is due to two things; BLAST only gives us the top X results for each contig, but X varies depending on the output format used. When using GeStore, we use the tab-delimited format (-m 8), while GePan uses the XML format (-m 7). This was done due to the need to combine the result files when doing incremental updates, and remove deleted entries and duplicates. We have solved that problem by using the tab-delimited format, and generating an XML file for the annotator using the data from the tab-delimited formatted file. This results in a XML file that contains fewer entries, and as we only include the data required by the annotator, much smaller entries.

Another interesting side-effect when using GeStore for incremental updates, is that the resulting BLAST E-values differ from the ones produced by a full run. The E-value of a hit represents the statistical chance that this hit would be found by chance in a randomly generated meta-data collection. As the meta-data collection is much smaller, and the E-value depends on the size of the meta-data collection, the E-value computed by BLAST when using an incremental meta-data collection is different from the E-value computed when using a full meta-data collection. These can be corrected by re-computation, which would best be done in the annotator. We plan to replace the annotator in future work. This problem was also discussed by Turcu et al. [16].

What also becomes apparent in our analysis, is that other parts of the pipeline take over as the most time-consuming elements when doing incremental updates. BLAST is reduced from 56% to 12% of the total runtime for our 1-month incremental update, so the other parts of the pipeline that take time become more apparent.

My experiences with the Hadoop stack have been largely positive, although there were two recurring problems I encountered. The first problem was that log files were filling up the log partition on the compute nodes, making us unable to run jobs without manually deleting these logs. This problem was compounded by a bug in our Hadoop version that made it impossible to change the logging level through configuration files. The second issue was nodes crashing in the middle of jobs, which made the experiments take longer than expected. Development within the Hadoop framework has been a great help in reducing the complexity of the GeStore implementation, and the problems I experienced were not serious enough to deem Hadoop unusable.

The experience of cooperation with Tim Kahlke and Espen Robertsen, who are both working on the biological side of things, was a positive one. Knowledge exchange through meetings, e-mails and chats has been invaluable for learning

about GePan and answering the many questions I had about the pipelines and genomics.

GePan itself is a well-written piece of software, although inflexibility in the design has led to some hurdles. The other tools we have used have generally had good documentation, and online resources have been helpful.

Obtaining usable metagenomic data for our experiments was done with the help of Espen Robertsen, and using online resources. We were able to find usable data using the MG-RAST online resource [38].

7 Conclusion

In this thesis, we have described and evaluated the GeStore system, which enables incremental computations for metagenomic pipelines.

We have designed, implemented, integrated and evaluated the GeStore system, which solves the seven requirements described in the introduction as follows:

1. The overhead of generating incremental meta-data collections in GeStore is much lower than the reduction in runtime when using these incremental meta-data collections, resulting in a net reduction in runtime of up to 65% in our experimental evaluation.

2. By storing the meta-data collections in HBase, we are able to access the data without using proprietary tools or computationally expensive techniques.

3. By using delta encoding in our HBase meta-data collection storage system, we are able to acheive storage savings of up to 80% in our experimental evaluation when compared to storing the meta-data collections individually.

4. The interface to GeStore is small, as corroborated by the requiring changes to less than 100 lines of code to modify GePan to utilize GeStore.

5. Extending the support for meta-data collections and tools is easily done through a small plugin system. Our implementation supports the Uniprot Knowledge Base DAT format in around 250 lines of code.

6. GeStore does not require modifications to the tools used in the pipeline, in order to minimize maintenance required when tools are added or updated. This has been demonstrated by using BLAST without modification in our experimental evaluation.

7. GeStore is able to retrieve meta-data collections for an arbitrary point in time, and hence old experiments are repeatable with the same data to verify results.

We beleive that efficient on-demand updates of metagenomic data, as provided by GeStore, will be useful to our biology collaborators.

7.1 Future work

Due to time constraints, we were not able to achieve the full benefits of the GeStore design. Some future work is outlined here.

1. We plan to provide support for additional file formats and meta-data collections in GeStore, including complete support for the FASTA file format, and the Pfam [39] meta-data collections. This also includes support for data collections that are not meta-data collections, but intermediate data produced by the pipeline.

2. We plan to leverage the provenance collected by GeStore to provide more detailed information about each result that the pipeline delivers, such that researchers can follow a sequence from the start of the pipeline to the end. One example is when producing data for end-users, a pipeline can generate reports for how a sequence hit has been processed, providing all the meta-data for every step in the pipeline for the given sequence. Such a feature has been requested by our biology collaborators.

3. An evaluation of the overhead and compression ratio of the compression support in HBase was not done, but is one possible avenue of further improvement. This involves a simple option when creating the tables in **dbUtil**, but would require experiments to verify that runtime is not increased.

4. Another area of potential improvement, would be to check which stages in GePan need to be re-run for an incremental update, Glimmer and FileScheduler do not need to be re-run in our experiments for example, but doing this would require larger changed in GePan than just the incremental updates.

5. **getdeleted** and **getfasta** can be combined, since these two modules operate on the same data, and perform very similar operations. This could potentially cut the GeStore overhead in half for incremental updates.

6. Integrating GeStore with other pipeline systems than GePan is important to gain a better understanding of how the integration process works in a different software ecosystem, such as cloud-based pipelines like CloVR.

7. In a production environment, we expect that old meta-data collections, both incremental and complete, will only very rarely be used when a new version of the meta-data collection enters the system, and since we can reproduce these files on demand from the meta-data collections stored in the database, these files can be deleted automatically without violating the principle of reproducability [15].

8. Closer integration with GePan:

   (a) Replace the FileScheduler to reduce the runtime of incremental updates. The evaluation shows that FileScheduler takes up a large percentage of the runtime for our experiments with incremental meta-data collections.

   (b) We are planning on replacing the Annotation step in GePan with one that can leverage the benefits of GeStore to reduce runtime and memory footprint. Giving the annotator direct access to the meta-data collections stored in GeStore would also enable us to solve the E-value issue discussed in section 6.9 quickly and efficiently.

References

[1] Brent, R. Genomic biology. *Cell*, 100(1):169–183, 2000. ISSN 00928674. doi:10.1016/S0092-8674(00)81693-1.

[2] Wooley, J. C., Godzik, A., and Friedberg, I. A primer on metagenomics. *PLoS computational biology*, 6(2):e1000667, January 2010. ISSN 1553-7358.

[3] Shi, Y., Tyson, G. W., and DeLong, E. F. Metatranscriptomics reveals unique microbial small RNAs in the ocean's water column. *Nature*, 459(7244):266–9, May 2009. ISSN 1476-4687.

[4] Bohannon, J. Metagenomics. Ocean study yields a tidal wave of microbial DNA. *Science (New York, N.Y.)*, 315(5818):1486–7, March 2007. ISSN 1095-9203.

[5] Stein, L. D. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[6] Baker, M. Next-generation sequencing: adjusting to data overload. *Nature Methods*, 7(7):495–499, July 2010. ISSN 1548-7091.

[7] Angiuoli, S. V., et al. CloVR: A virtual machine for automated and portable sequence analysis from the desktop using cloud computing. *BMC Bioinformatics*, 12(1):356, 2011. ISSN 14712105. doi:10.1186/1471-2105-12-356.

[8] Tanenbaum, D. M., et al. The JCVI standard operating procedure for annotating prokaryotic metagenomic shotgun sequencing data. *Standards in genomic sciences*, 2(2):229–237, 2010.

[9] Delcher, A. L., et al. Identifying bacterial genes and endosymbiont DNA with Glimmer. *Bioinformatics*, 23(6):673–679, 2007. ISSN 13674811. doi: 10.1093/bioinformatics/btm009.

[10] Altschul, S. F., et al. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990. ISSN 00222836. doi:10.1006/jmbi.1990.9999.

[11] Magrane, M. and Consortium, U. UniProt Knowledgebase: a hub of integrated protein data. *Database the journal of biological databases and curation*, 2011(0):bar009, 2011.

[12] Gehlenborg, N., et al. Visualization of omics data for systems biology. *Methods*, 7(3):S56–68, 2010. ISSN 15487105. doi:10.1038/NMETH.1436.

[13] Bhatotia, P., et al. Incoop : MapReduce for Incremental Computations. *System*, page 7, 2011. doi:10.1145/2038916.2038923.

[14] Peng, D. and Dabek, F. Large-scale Incremental Processing Using Distributed Transactions and Notifications. *DBMS*, 2006:1–15, 2010.

[15] Gunda, P. K., et al. Nectar: automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8. USENIX Association, Berkeley, 2010.

[16] Turcu, G., Nestorov, S., and Foster, I. Efficient Incremental Maintenance of Derived Relations and BLAST Computations in Bioinformatics Data Warehouses. In I.-Y. Song, J. Eder, and T. Nguyen, editors, *Data Warehousing and Knowledge Discovery*, volume 5182 of *Lecture Notes in Computer Science*, pages 135–145. Springer, 2008.

[17] Dean, J. and Ghemawat, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. ISSN 00010782. doi:10.1145/1327452.1327492.

[18] Apache HBase. `http://hbase.apache.org/`. Retreived May 2nd 2012.

[19] Shvachko, K., et al. The Hadoop Distributed File System. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies MSST*, 0(5):1–10, 2010. doi:10.1109/MSST.2010.5496972.

[20] Hunt, J. W., Laboratories, B., and Hill, M. An Algorithm for Differential File Comparison. *Bell Telephone Laboratories CSTR 41*, (41):1–9, 1976.

[21] Pedersen, E. *Big data analysis for metagenomics*. Special corriculum, University of Tromsø, Tromsø, 2011.

[22] HDFS Architecture Guide. `http://hadoop.apache.org/common/docs/current/hdfs_design.html`. Retreived May 2nd 2012.

[23] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29, December 2003. ISSN 01635980. doi:10.1145/1165389.945450.

[24] Chang, F., et al. Bigtable : A Distributed Storage System for Structured Data. *Sports Illustrated*, 26(2):1–26, 2008. ISSN 07342071. doi:10.1145/1365815.1365816.

[25] Oracle grid engine. `http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html`. Retreived May 2nd 2012.

[26] Chang, F., et al. BigTable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):1–26, June 2008. ISSN 07342071. doi:10.1145/1365815.1365816.

[27] Yu, Y., et al. DryadLINQ : A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages:1–14, 2008.

[28] Fetterly, D. and Isard, M. TidyFS : A Simple and Small Distributed File System. *researchmicrosoftcom*, 2010.

[29] Hadoop Pig. `http://pig.apache.org/`. Retreived May 2nd 2012.

[30] Thusoo, A., et al. Hive – A Petabyte Scale Data Warehouse Using Hadoop. *Architecture*, pages 996–1005, 2010. doi:10.1109/ICDE.2010.5447738.

[31] Chaudhuri, S. and Dayal, U. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997. ISSN 01635808. doi:10.1145/248603.248616.

[32] Olston, C. and Sarma, A. D. Ibis : A Provenance Manager for Multi-Layer Systems. In *5th Biennial Conference on Innovative Data Systems Research*, pages 152–159. 2011.

[33] DeCandia, G., et al. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007. ISSN 01635980. doi:10.1145/1294261.1294281.

[34] Ramakrishnan, L., et al. Magellan : Experiences from a Science Cloud. *Star*, pages 49–58, 2011.

[35] Eucalyptus. `http://www.eucalyptus.com/`. Retreived May 2nd 2012.

[36] FASTA format description. `http://www.ncbi.nlm.nih.gov/blast/fasta.shtml`. Retreived May 4th 2012.

[37] Bhaya, D., et al. Population level functional diversity in a microbial community revealed by comparative genomic and metagenomic analyses. *The ISME journal*, 1(8):703–713, 2007.

[38] MG-RAST. `http://metagenomics.anl.gov/metagenomics.cgi?page=MetagenomeSelect`. Retreived May 4th 2012.

[39] Finn, R. D., et al. The Pfam protein families database. *Nucleic Acids Research*, 38(Database issue):D211–D222, 2010.