# Adresser les défis de passage à l'échelle en génomique comparée

Natalia Golenetskaya

$N°$d'ordre : 4840

# THÈSE

PRÉSENTÉE À

# L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Natalia Golenetskaya**

POUR OBTENIR LE GRADE DE

**DOCTEUR**

SPÉCIALITÉ : **INFORMATIQUE**

---

## Addressing scaling challenges in comparative genomics

## *Adresser les défis de passage à l'échelle en génomique comparée*

---

**Soutenue le :** 9 Septembre 2013

**Après avis des rapporteurs :**

| | |
|---|---|
| Amedeo NAPOLI ..... | DR CNRS |
| Jean-Stéphane VARRÉ | Professeur des Universités |

**Devant la commission d'examen composée de :**

| | | |
|---|---|---|
| Amedeo NAPOLI ..... | DR CNRS ..... | Rapporteur |
| Jean-Stéphane VARRÉ | Professeur des Universités ..... | Rapporteur |
| Pascal DURRENS ..... | CR CNRS ..... | Examinateur |
| Rodolphe THIÉBAUT | DR INSERM ..... | Examinateur |
| Alexandre ZVONKINE | Professeur des Universités ..... | Examinateur |
| David SHERMAN ..... | DR Inria ..... | Directeur de thèse |

**– 2013 –**

# Abstract

Comparative genomics is essentially a form of data mining in large collections of $n$-ary relations between genomic elements. Increases in the number of sequenced genomes create a stress on comparative genomics that grows, at worse geometrically, for every increase in sequence data. Even modestly-sized labs now routinely obtain several genomes at a time, and like large consortiums expect to be able to perform all-against-all analyses as part of these new multi-genome strategies. In order to address the needs at all levels it is necessary to rethink the algorithmic frameworks and data storage technologies used for comparative genomics.

To meet these challenges of scale, in this thesis we develop novel methods based on NoSQL and MapReduce technologies. Using a characterization of the kinds of data used in comparative genomics, and a study of usage patterns for their analysis, we define a practical formalism for genomic Big Data, implement it using the Cassandra NoSQL platform, and evaluate its performance. Furthermore, using two quite different global analyses in comparative genomics, we define two strategies for adapting these applications to the MapReduce paradigm and derive new algorithms. For the first, identifying gene fusion and fission events in phylogenies, we reformulate the problem as a bounded parallel traversal that avoids high-latency graph-based algorithms. For the second, consensus clustering to identify protein families, we define an iterative sampling procedure that quickly converges to the desired global result. For both of these new algorithms, we implement each in the Hadoop MapReduce platform, and evaluate their performance. The performance is competitive and scales much better than existing solutions, but requires particular (and future) effort in devising specific algorithms.

# Résumé

La génomique comparée est essentiellement une forme de fouille de données dans des grandes collections de relations $n$-aires. La croissance du nombre de génomes sequencés créé un stress sur la génomique comparée qui croit, au pire géométriquement, avec la croissance en données de séquence. Aujourd'hui même des laboratoires de taille modeste obtient, de façon routine, plusieurs génomes à la fois – et comme des grands consortia attend de pouvoir réaliser des analyses tout-contre-tout dans le cadre de ses stratégies multi-génomes. Afin d'adresser les besoins à tous niveaux il est nécessaire de repenser les cadres algorithmiques et les technologies de stockage de données utilisés pour la génomique comparée.

Pour répondre à ces défis de mise à l'échelle, dans cette thèse nous développons des méthodes originales basées sur les technologies NoSQL et MapReduce. À partir d'une caractérisation des sorts de données utilisés en génomique comparée et d'une étude des utilisations typiques, nous définissons un formalisme pour le Big Data en génomique, l'implémentons dans la plateforme NoSQL Cassandra, et évaluons sa performance. Ensuite, à partir de deux analyses globales très différentes en génomique comparée, nous définissons deux stratégies pour adapter ces applications au paradigme MapReduce et dérivons de nouveaux algorithmes. Pour le premier, l'identification d'événements de fusion et de fission de gènes au sein d'une phylogénie, nous reformulons le problème sous forme d'un parcours en parallèle borné qui évite la latence d'algorithmes de graphe. Pour le second, le clustering consensus utilisé pour identifier des familles de protéines, nous définissons une procédure d'échantillonnage itérative qui converge rapidement vers le résultat global voulu. Pour chacun de ces deux algorithmes, nous l'implémentons dans la plateforme MapReduce Hadoop, et évaluons leurs performances. Cette performance est compétitive et passe à l'échelle beaucoup mieux que les algorithmes existants, mais exige un effort particulier (et futur) pour inventer les algorithmes spécifiques.

# Contents

# Chapter 1

# Introduction

***Résumé :*** *Dans cette thèse nous adressons certains défis en génomique comparée liés à la croissance des données et des relations n-aires entre eux, par le développement de nouvelles représentations et de nouvelles stratégies algorithmiques. Le défi particulier est que ces relations n-aires croissent rapidement et au pire géométriquement. Mes développements théoriques sont traduit en implémentation de logiciels pratiques, qui nous validons ensuite sur des jeux de données de taille conséquente.*

*En termes concrets, à partir d'une analyse d'une expérience concrète issue du projet de comparaison de génomes Génolevures, nous avons défini, mis en œuvre et évalué un système distribué NoSQL de stockage de données et de relations. Les performances sont meilleurs et une bonne mise à l'échelle est démontrée. Ensuite, sur la base de méthodes existantes, nous avons rentré deux analyses globales de génomes dans le paradigme de calcul distribué MapReduce. La première méthode, qui identifie des événements anciens de fusion et de fission de gènes, a pu être adapté à MapReduce grâce à un nouvel algorithme. La seconde méthode, utilisée pour partitionner des ensembles de gènes en familles phylogénétiques, a été adapté à MapReduce par la définition d'un processus d'échantionnage itératif qui converge sur le résultat recherché. Ces exemples montrent que MapReduce est une bonne approche pour certains calculs en génomique comparée, mais que l'adaptation d'algorithmes existants pourrait demander des stratégies très différentes.*

In this thesis I address scaling challenges for computational biology, by developing new data representations and new distributed algorithmic strategies for comparative genomics. I translate these theoretical designs into practical software solutions, which I validate on large data sets derived from existing well-studied genome comparisons.

## 1.1 Scaling up and out

It is a basic irony of Computer Science that the more we improve our computers, the less we are content with resulting improvement to old problems, and the more we seek improved techniques to attack even larger problems. Every generation of programs and operating systems includes new functionalities that consume the advantages of Moore's law improvements to hardware, and consequently require new innovation in software to stay ahead. In this sense, *scaling* is a fundamental problem for Computer Science.

Historically, we speak of *scaling up* to new application instances through the improvement of computer hardware: faster processors with more cores, faster primary and sec-

ondary memory with greater capacity. More recently, we speak also of *scaling* ***out***, where new capacity is seamlessly added to an existing infrastructure by installing additional computers to share the load, rather than by adding capacity to existing computers (Figure 1.1). Systems with good scaling out properties transparently redistribute work to new computers as they are added. Scaling out also reduces the impact of a single computer failure, since the system automatically reassigns work to the remaining computers. This is important when we consider the statistical likelihood of hardware failure: in a network of 1000 nodes, each with a mean time to failure of 3 years, we expect to see one failed machine per day on average. We must build software architectures that take adjust robustly to such failures.

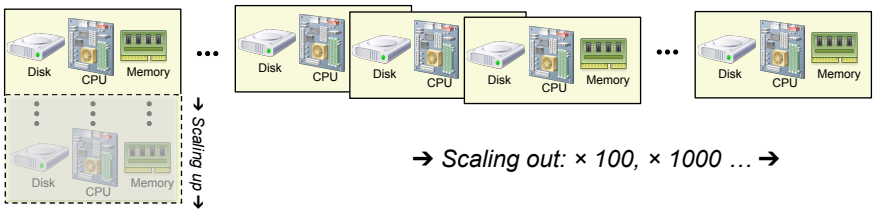



Figure 1.1: Scaling out flexibly adds nodes to a computing system

## 1.2 Comparative genomics versus Moore's law

Computational biology is a general term describing computer-based methods for answering biological questions. These methods involve algorithms for analyzing a wealth of different kinds of experimental data: gene and genome sequences, gene expression levels, cell numbers and shapes, even images of specific proteins within cells and tissues. Our focus will be on analysing genome sequences. *Comparative genomics* seeks to understand the history and function of genomes by large-scale analysis of the relations between genomic elements, as contrasted with analyzing of those elements individually. These relations might be classifications into families of homologs, physical maps of gene proximity, distance measures defined for functional or phylogenetic similarity, among many others [AG03, BPG+04a, Sea05, C+06].

We will see that comparitive genomics presents a serious challenge requiring good scaling scaling solutions, both to handle the increase in basic data volume, and to handle the increased dimensionality of the relations between genomes.

### 1.2.1 Growth in sequencing data volume

The hundred- to thousand-fold decreases in sequencing costs [Wet12] (Figure 1.2) that we have seen in the past five years have made it feasible for even small labs to obtain the genome sequence of a selected organism with their operating budget, something that previously with Sanger sequencing was only possible for groups of several labs with significant special funding.

The data volumes are considerable – the number of fully-sequenced microbial genomes submitted to GenBank doubles every 17 months [vN04]. The current size (release 113) of The European Nucleotide Archive (ENA)[1] is 1.5 Terabytes uncompressed. The 1000

[1]The European Nucleotide Archive (ENA): http://www.ebi.ac.uk/ena

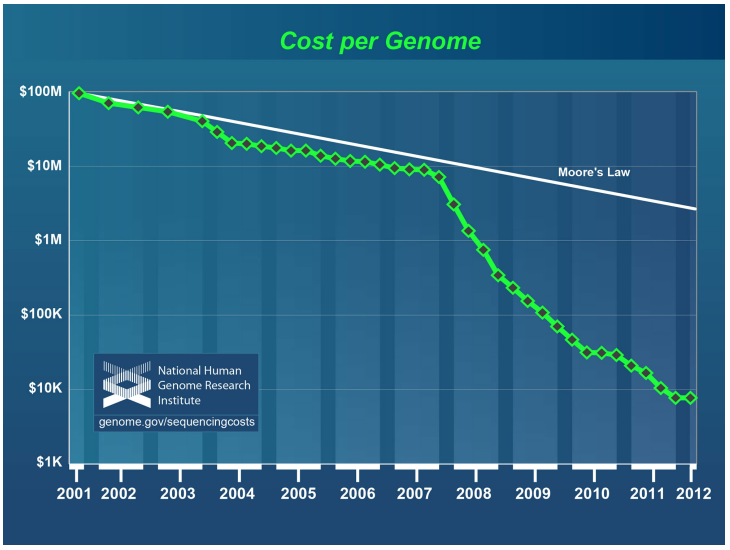



Figure 1.2: Genome Sequencing cost from Wetterstrand [Wet12], showing that genome sequence acquisition is growing faster than Moore's law.

Genomes Project[2] has more than 200 terabytes of sequenced human genomes in its data base; the data is currently accessible via Amazon Cloud[3]. Thus, the labs without dedicated bioinformatics staff may have difficulties maintaining the necessary resources. Significantly, figure 1.2 illustrates that sequence volumes are growing faster than Moore's law, so we cannot simply rely on improvement to computer hardware to address these new problems. Innovation in data storage and retrieval methods is needed to keep pace.

### 1.2.2 Growth in genome-genome relations

Secondly, and more significantly, comparative genomics produces an additional significant strain on bioinformatics: for every linear increase in basic genome data, there is a geometric increase in the *relations*, represented as $n$-tuples, between genome elements. Every new genome results in a jump in the number of logical relations defined by all-against-all comparisons, which are not just pairwise relations, but arbitrary sets of related genes[BPG+04b, BSD07].

Annotation and comparative analysis illustrate the challenges very well. Comparative annotation[4] uses supervised and unsupervised classification of all-against-all comparisons between genomes to improve the annotation process. Individually annotating a collection of related genomes requires a linear increase in effort, while at the same time ignoring a significant source of information, since the related genomes will by design share many common features. Comparative analysis of the annotated genomes also involves systematic comparison of genome features. In the worst case both analyses scale geometrically with the volume of the data. However, as we will see in chapter 3, matrices for comparative genomics are often sparse and can be efficiently encoded in a well-designed storage system.

[2] 1000 Genomes Project: http://www.1000genomes.org

[3] The 1000 Genomes Project data on Amazon Cloud: http://aws.amazon.com/1000genomes

[4] Such as that performed by our Magus system for simultaneous genome annotation, http://magus.gforge.inria.fr

## 1.3 Practical motivation for comparative genomics

Developing good computer methods for computational biology, and specifically genomics, is motivated by biology. In Nature feature [CGGG03], Francis Collins and colleagues argue that this proceeds in three stages: Genomics to biology, elucidating the structure and function of genomes; Genomics to health, translating genome-based knowledge into health benefits; and Genomics to society, promoting the use of genomics to maximize benefits and minimize harms. Through their vision they identify fifteen Grand Challenges for the future of genomics research.

More modestly, we have motivated our work by two applications in comparative genomics: paraphyletic strategies for sequencing microorganisms of biotechnological interest, and analysis of affinity binders for the human proteome.

### 1.3.1 Paraphyletic sequencing strategies

This decrease in sequencing costs has given rise to a routine multi-genome strategy: instead of sequencing just one genome, labs routinely sequence it and a group of its near phylogenetic neighbors. By making a careful choice of strains and species, it is possible to "subtract" the common part of their genomes, and efficiently focus on the specific differences that are linked to the phenotype in question [WPP+11, CRO+09, FGM+11]. These sequencing strategies can be called *paraphyletic*, in the sense that they sequence collections of genomes in a single phylogenetic branch defined around an interesting organism, rather than sequencing the single genome of that organism. The strategy is more sophisticated than just randomly choosing the genomes. The strategy is to choose the related organisms that share an interesting trait and some that do not. For example, if we were interested in a pathogen we would choose some genomes that are pathogenic and some related genomes that are not. If we wanted to understand a specific trait in a cell factory in order to improve it, we would choose from the branch genomes with and without the desired trait. During genome comparisons we identify the common parts to find out what they share, and furthermore "subtract" that common part from individuals to identify their specific genomic differences and link them to their specific phenotypic differences. If the collection of related genomes is carefully chosen, then the number of specific differences will be small, and it will be easier to link them to the specific traits. For example, in [Con07] twelve species of fruit fly were carefully chosen and sequenced; among many results 44 lineage-specific genes were found and linked in many cases to specific tissues. More modestly, in [FGM+11] the pathogenicity of *Candida glabrata* was studied through sequencing and comparison of other pathogen and non-pathogen species from the clade of the *Nakaseomyces*, including *Candida nivariensis* and *Candida bracarensis*. We considered comparative annotation and scaling out for paraphyletic strategies in [SGMD11].

### 1.3.2 Human genome and proteome

Comparison of individual persons' genomes is also a form of genome comparison—indeed reference [CGGG03] specifically built on the value of the human genome project (HGP) for defining a common reference for comparison. Large-scale comparisons of the human genomes helps research scientists to find gene-target treatments for such diseases as diabetes, heart disorders and others. Whole-genome sequencing strategies are widely used by clinic studies. There is a tendency to move from "one-size-fits-all" medicine to a personalized medicine [NIH11]. Personalized medicine can significantly improve quality of a diagnostics by targeting patient-specific genes and eliminating unnecessary treatments

[Keo12]. It works with individual human genome which are all very similar to reference, but contain significant individual differences responsible for individual traits and possibly susceptibility to deceases. Julie Hoggatt expects in [Hog11] an exponential growth of the use of the personalized medicine in the coming 10 years, thanks to new sequencing technologies and advances in genomics, proteomics and pharmacogenomics fields. This new paradigm in medicine requires developing large-scale data storage for patients data [RPB+08] that can support exponential growth, natural language processing for data mining [MH06] and the decision support techniques [MWCR05].

One application that was used peripherally for this thesis was large-scale abalysis of a systematic collection of affinity reagents for the human proteome[GS11a, SG10]. We worked with the EU ProteomeBinders Consortium[5] [T+07], which like the US National Cancer Institute Initiatives[6], and Human Antibody Initiative[7] develops binders for the 25 000 human genes and more than 500 000 protein isoforms. Large-scale mining in these data sets can be used to search for compatible sets of binders for a collection of protein targets, where all of the binders have comparable specificity, technical constraints, etc., as well as reasoning over the knowledge base.

## 1.4 Assets for this thesis

Three assets were essential to the research presented in this thesis.

**Génolevures data and use cases.** Since 12 years Génolevures is dedicated to large-scale comparative genomics of Ascomycete yeasts chosen for their biotechnological or medical interest, with a focus on understanding the mechanisms of eukaryote molecular evolution [S+00, DSF+04, S+09]. Génolevures maintains a highly-accessed on-line resource of annotated genomes and results [SMN+09, DMS11]. There are many kinds of relations there: protein families, synteny groups, conservations of genes and others [SDB+04]. In our team's work in the Génolevures Consortium[8] we routinely analyze $10^6$ genetic elements and will quadruple this number in projects in the recent future. I used this high quality data set and its use cases to design new storage and anaysis strategies in chapters 3 and 4, and validate them in a practical system in chapter 5. I additionally used server logs from the Génolevures web site to derive stress tests based on real uses.

**Magus genome analysis system.** Initially, the Magus[9] genome annotation system was developed in 2006 by David Sherman for Génolevures based on an earlier system developed in 2002. Magus 1.0 drives the current web site[SMN+09]. Magus is designed around a standard sequence feature database, is integrated with the Generic Genome Browser[10] [SMS+02], uses various biomedical ontologies[11], and provides a RESTful web interface. Magus is also designed for curation of the sequence features, presenting competing gene models and an interface for annotators (curators) to choose and validate the correct gene model.

[5] Proteome Binders Consortium: http://www.proteomebinders.org
[6] US National Cancer Institute: http://www.cancer.gov
[7] http:// www.hupo.org/research/hai
[8] Génolevures: http://www.genolevures.org
[9] Magus: http://magus.gforge.inria.fr
[10] Generic Genome Browser: http://www.gmod.org/wiki/GBrowse
[11] Biomedical ontologies: http://obo.sf.net

In the context of a reimplementation supported by an Inria Technology Development Action (ADT), we started Magus 2.0 with a clean object model and a common interface for the different storage back ends,flexible enough to work with both tables of relational databases (MySQL and PostgreSQL) and the Cassandra NoSQL column families defined in this thesis. The new data model was developed in collaboration by myself, David Sherman, Anna Zhukova, and Florian Lajus. Florian Lajus is an engineer hired by the project to implement of the new version Magus 2.0.

**Computational resources.** For my work I used parts of a dedicated 76-core computing cluster maintained by the Magnome project-team, that was essential for studying practical deployment of implementation in a controlled environment. Genome-genome relations were computing using classical sequence comparison algorithms running in the batch system on the cluster. Various configurations of physical and virtual nodes were used to evaluate Cassandra NoSQL data schema for chapter 3. And finally, a dedicated set of nodes were used for algorithm testing in chapter 4 and performance evaluation in chapter 5.

## 1.5 Challenges addressed in this thesis

In this thesis I address scaling challenges through new data representation strategies for comparative genomics data and new strategies for distributed computations. The goal is to define extensible systems that can be deployed by small groups as well as by national platforms, and can be flexibly dimensioned for projects of different sizes.

From a practical standpoint, scaling to extreme data volumes is addressed by NoSQL and MapReduce technologies, both associated with cloud computing [SLBA11, ADB+09]. NoSQL provides distributed storage of Petabyte datasets. MapReduce is a paradigm for writing large-scale analyses across thousand-node clusters. However, adopting these technologies imposes new constraints on algorithms, and there is as yet an insufficient real body of best practices for their use in bioinformatics. Successful applications to date have focused for example on embarrassingly parallel analyses such as SNP identification [M+10, PLZ11] but not on $n$-ary relations between genome elements.

In this thesis we attempt to define some ground rules for successful conversion of global bioinformatic analyses to highly scalable MapReduce deployments. We define a criterion that can be used to guide redefinition of bioinformatics algorithms so that they can be efficiently deployed on NoSQL and MapReduce infrastructures, with a reasonable guarantee of scaling out to new data volumes.

We illustrate our approach with two successful conversions of global comparative genomics analyses. The first, *gene fusion/fission*, systematically identifies gene fusion and fission events in eukaryotic phylogenies using a parsimony criterion [DNS08]. The second, *consensus clustering*, computes multi-genome protein families by reconciling competing and complementary clusterings [NS07]. In both examples we are confronted with very large matrices of $n$-ary relations between genomic elements, and we show that nonetheless solutions to these problems can be deployed in NoSQL/MapReduce in a way that scales smoothly with increases in data volume.

Some preliminary results of our work were introduced in [GS11a, GS11b, SG11, SLG10].

### 1.5.1 Cassandra and Hadoop

Apache Cassandra[12] provides a highly scalable distributed semi-structured data store [LM10], where data is automatically replicated and balanced across an ad-hoc peer-to-peer network of storage nodes in a computing cluster, and where capacity can be added dynamically without disturbing applications running on the cluster. Apache Hadoop[13] provides a software framework for fault-tolerant distributed computing of large datasets across clusters of computers using the MapReduce paradigm [DG04]. In MapReduce algorithms are decomposed into two phases using functional programming: a *map* phase where a single function is applied to all data elements, followed by a *reduce* phase where groups of mapped results are combined into a final result. Since both phases are functional, if a node fails, its part of the computation can be recomputed without invalidating the other parts. This leads to improved fault tolerance.

### 1.5.2 Specific challenges I address: 3 applications in genomics

In software engineering field the first thing to do is to identify requirements. In order to identify the most important aspects of comparative genomics, we performed use case analysis. Using the assets of existing sources such as Génolevures web site, we recognized important challenges and real world scenarios.

To illustrate specific challenges of the comparative genomics, we consider three use cases. First use case is a searching interface for the storage system of the genome sequence features. Second and third uses cases are global analyses in comparative genomics, which work with large matrices and graphs of relations between genomic elements.

**Magus knowledge base.** The Magus system [SMN+09] provides an access to the genome sequence features, such as genes, proteins, exons, etc. Beyond storing individual sequence feature objects, the system operates with relations between these data elements. There are many kinds of relations, for example: protein families, allele groups, tandem repeats, matrices of similarities.

Magus::Search is an interface to search for sequence features, using different combinations of parameters - attributes of sequence features or groups. The various search queries contain searches by name, by overlapping intervals (geometric queries), by keywords, by group members, etc. The object model for the Magus system is described in section 5.1.

The other use of the Magus knowledge base is to store intermediate results of comparative genomics analyses, such as matrices and graphs of similarities. These are typically sets of pairwise relations, with attribute values associated with them. The calculation of matrices and graphs of similarities is typically the first step of the analyses in comparative genomics, thus we need to store them in a format, prepared for the analyses. The number of relations in that matrices and graphs scale quadratically with the number of individual data elements.

*Computational challenge.* The first challenge is to store fast growing large dataset of genome sequence features. With the linear growth in genome sequence features there is superlinear growth in the number of $n$-ary relations (protein families, allele groups, tandem repeats, etc)

The second challenge is to provide a searching interface for sequence features and groups.

[12]The Apache Cassandra Project: http://cassandra.apache.org
[13]The Apache Hadoop Project: http://hadoop.apache.org

*Method.* In chapter 3 we will formalize the grouping organization of the relations between data elements depends on the intended use: for searching or for large scale analyses. We will store all relations in highly scalable Apache Cassandra NoSQL data store using both techniques.

We will store all sequence features and groups in Cassandra. We will develop Cassandra column families data schema based on the set of typical search queries.

We will develop Magus::Search common interface to access genome sequence features, that can work with SQL and NoSQL backends using corresponding database adaptors. We will develop Cassandra NoSQL adaptor for Magus::Search.

**Identification of Fusion and Fission events.** Genes change all the time not only because of mutations, but also because of large scale operations of reorganizing the genes, called gene fusions and fissions. Gene fusions and fissions can be identified by analyzing similarity relations between genes [DNS08]. The algorithm searches for patterns in a large graph of similarity relations between segments of genes, these patterns are the gene fusions and fissions. Then, fusions and fissions are mapped onto evolutionary cross-species tree. The algorithm is described in details in section 4.2.
*Computational challenge.* The first challenge is to store the quadratically growing data set of pairwise relations between proteins and between paralogous groups.

The second challenge is the graph analysis phase of the algorithm, which is the most stressed by the growth of the data since it needs to put the whole graph in memory.
*Method* To address the first challenge, we will store relations in the Cassandra data store, using general technique of storing relations, described in subsection 1.5.2.

To address the second challenge, we will substitute graph analysis phase by the equivalent distributed MapReduce algorithm.

**Consensus and MCL clustering for protein families.** Protein family is a group of phylogenetically related proteins. The members of the protein families can belong to different species, but share the same biological functions. The families are identified based on similarities by clustering the results of pairwise all-against-all comparisons between proteins.

Consensus clustering algorithm [NS07] involves all-against-all comparisons between proteins with different methods such as Blast and Swith-Waterman's, with two different filterings. Then, all four matrices of comparisons proceed to the phase, where we run MCL clustering with three different inflation parameters. The final step is a consensus procedure.

MCL clustering [vD00] is an algorithm for clustering simple weighted graphs using Markov stochastic matrices [EVDO02]. It looks into probabilities of random walks. Since 2002 it is the best algorithm to identify protein families and it was chosen in 2007 by authors of the Consensus clustering article [NS07]. The algorithm is described in details in section 4.3.
*Computational challenge* The first challenge is to store different versions of matrices of pairwise comparisons between proteins, that scale quadratically.

MCL clustering and consensus procedure are two phases of the Consensus clustering algorithm that are very stressed by increase of data volume.
*Method* We will store all relations in Cassandra data store, using general technique of storing relations, described in subsection 1.5.2.

Consensus procedure is easily parallelized because it involves many independent computations on the conflict region.

We will substitute MCL clustering step by iterative sampling procedure, that works in parallel with small submatrices and converges to some satisfactory result in several iterations.

## 1.6 Outline

In chapter 2 we will talk about common requirements for the large-scale systems for storing, ad-hoc algorithms and analytics. In section 2.3 about storage systems, we will discuss the CAP theorem, which claims that the storage system may only have two out of the three following properties : Consistency, Availability and Partition tolerance. We will explain the choice of Cassandra NoSQL, a highly Available Partition tolerant (AP) data store. In section about ad-hoc algorithms (section 2.4) we will discuss different programming models and specifically MapReduce. In section 2.5 we will talk about large-scale analytics.

In chapter 3 we will describe and formalize different ways of representing genomic big data. We will be focused on the important aspect for comparative genomics—relations between data elements. Specifically we will talk about Cassandra representations of genomic data elements and relations for different purposes. We will introduce general principles of creating column families in section 3.6.1.

In chapter 4 we will consider global analyses in comparative genomics. These analyses often involve all-against-all comparisons which scale geometrically. We will describe and formalize MapReduce adaptations of two existing algorithms in comparative genomics. Also, we will consider some example of human genome analysis.

In chapter 5 we will talk about *Tsvetok*, an integrative representation of implementations for the systems for data storage, ad-hoc algorithms and analytics. We will compare SQL and NoSQL backends for Magus genome sequence features storage and retrieving system. We will talk about implementations of two MapReduce adaptations of global analyses in comparative genomics in MapReduce Hadoop framework.

In chapter 6 we will draw general conclusions and will propose some possible directions for future work.

# Chapter 2

# Preliminaries

***Résumé :*** *Pour appliquer le Big Data en génomique nous devons nous intéresser à la fois au stockage, au calcul et aux analytiques. Le théoreme CAP établit qu'au plus deux des trois propriétés de la* cohérence*, la* disponibilité *et la* tolérance au partitionnement *peuvent exister simultanément dans un système de gestion de données distribué. Nous argumentons que les besoins de la génomique comparée sont les mieux adressés par les approches NoSQL, qui propose la bonne combinaison de la cohérence et de la disponibilité nécessaire. Ensuite, nous discutons les besoins de calcul en génomique et décrivons quelques éléments du paradigme MapReduce, qui permet d'adapter des algorithmes existants par une décomposition en phases fonctionnels d'application (map) et aggrégation (reduce). Nous considérons comme cas particulier* l'analytique *et les approches par entrepôts de données et par fouille de données. Nous terminons par une étude d'un défi centrale du génomique comparée : le besoin de gérer les* relations *n*-aires *entre objets génomiques, qui croissent de façon supralinéaire en fonction du nombre d'objets.*

In this chapter we will consider three aspects of maintaining genomic Big Data: storage, computation and analytics. In the first part of the chapter we define the common requirements for these systems.

In the second part we compare different storage solutions, in order to choose the one that matches best to our requirements. NoSQL is a class of distributed fault-tolerant database management systems that usually abandon the declarative query language SQL and its associated relational algebra, in exchange for better scalability. We will see, on page 23, that the CAP theorem says that it is only possible to achieve two out of three of the properties of Consistency, Availability, and Partition Tolerance for shared-data systems. After consideration of the specificities of our applications, we chose Availability and Partition-tolerance (AP) as the two most important properties for our applications, and thus abandon Consistency. Among the available AP systems we chose the Apache Cassandra NoSQL data store, because it is easy to use, widely used, open-source, and has a very developed community.

In the third part we discuss distributed parallel computations. MapReduce is a paradigm for writing programs for distributed parallel computations, that is based on decomposing algorithms into separate *map* and *reduce* phases. A MapReduce software framework, such as Apache Hadoop, hides technical details to make it easy to write applications, by simply implementing functions for each of the map and reduce phases. Pig Latin is a declarative language for calculations that can additionally facilitate writing MapReduce applications.

In the fourth part we consider a special case of computations, *analytics*. These computations typically are performed on read-only data and often are statistical global analyses. We consider data warehouse approaches, and two successful examples, BioMart and Hive. We finally discuss the Apache Mahout libraries for data mining on top of Hadoop.

At the end we present our specific case of comparative genomic data, which works with many relations instead of with individual objects. We also explore some difficulties of distributed processing of graphs.

## 2.1 Storage, ad-hoc algorithms and analytics systems

As a first step we propose a classification of data management systems by their usage patterns, in particular by the number of get and insert queries. This is not a precise classification, some real world systems can belong (at least partly) to different classes.

1. *Storage Systems*
   Storage systems are designed for retrieving small subsets of data from potentially quite large data sets. They typically have mostly get and insert queries. The Génolevures website, for example, uses several Data Storage Systems, including MySQL and PostgreSQL databases. The public part of Génolevures provides access to searching and visualizing tools for the data from these Data Storage Systems. By analyzing the user log history we noticed that most often the get queries are colocalized in the set of genes: the same gene features appear in many user search results.

2. *Systems for ad-hoc algorithms*
   System for ad-hoc algorithms are designed to support developing specialized programs for global analyses, that typically access the whole dataset or at least a large part of it. These systems must be contrasted with Storage systems, above, which are not natively adapted to the cycle of global get-transform-load operations. Systems for ad-hoc algorithms may be built on top of storage systems to provide access to the data for the distributed parallel computations. In the case of comparative genomics global analyses may be, for example, clustering, pattern recognition, phylogenetic graph analyses and others.

3. *Analytics Systems*
   Analytics Systems are designed for implementing decision support systems on a consistent, frozen picture of the data. They typically provide read-only access to the data. Updates are rare and run by administrators of the systems. Data warehouses are one of the most common analytics systems. The data in these systems are prepared for analysis by domain specialists, in our case biologists and bioinformaticiens.

## 2.2 Requirements for storage, ad-hoc algorithms and analytics systems

The **common requirements** for Storage, Ad-hoc algorithms and Analytics systems are

1. *Large-scalable (petabyte-scalable)*

   In the recent years it has been observed that the volume of genomic data is growing dramatically. The data are no longer stored—can no longer be stored—in the same

file on the same computer. Analyses in comparative genomics are the most stressed by this data volume growth since analysis of only tens of genomes can involve $10^6$ to $10^9$ pairwise comparisons. We need a petabyte-scale solution to store and analyze data.

a) *Distributed*

Since the data no longer fit on one machine, we need distributed solutions.

b) *Fault-tolerant*

Statistically, the more we add machines to expand the computing capacities of the system, the more we are likely to suffer from the failure of single machines. This directly implies a need for fault-tolerant systesm that automatically redistribute the data, as well as the queries and the computation tasks.

2. *Fits to the usage patterns*

In the section 2.1 we classify systems by their usage patterns. Storage systems should work well both with get and insert queries; get queries might be colocalized. The systems for ad-hoc algorithms should be optimized for global analyses, meaning global get-transform-insert operations. An analytics system need not be optimized for updates, but may be read-only and prepared to be analyzed by domain specialists.

3. *Easy to use and low cost*

These requirements are very important for small labs who work essentially with genomic and biological data, since often they cannot afford dedicated IT staff and need easy and inexpensive solutions for distributed storage and global computations on their data.

## 2.3 Storage systems

In this section we will discuss the requirements of a storage system for Big Data in genomics and compare different solutions: traditional databases, parallel databases, and NoSQL.

### 2.3.1 Classification by CAP and data models

#### Consistency, Availability and Partition tolerance

For data storage systems there are three important properties:

1. *Consistency*: each computer node always represent the same view of the data.

2. *Availability*: each computer node is always available for read and write.

3. *Partition tolerance*: splitting of the distributed system into several isolated sections does not lead to incorrect responses from each of the sections.

The CAP theorem, proposed in 2000 by Eric A. Brewer [Bre00] and formally proved by Seth Gilbert and Nancy Lynch in [GL02], says: for any shared-data system you can achieve at most two out of these three properties.

### Data Models

Nathan Hurst [Hur10] suggests the classification of the database systems based on the CAP configurations and the data models they use:

1. *Relational*

   Relational databases use the relational algebra; data are stored and accessed via relations. A *relation* is defined as a set of tuples that have the same attributes. Such databases support join operations and ACID (atomicity, consistency, isolation, durability) properties. Relational database management systems, for example MySQL and PostgreSQL, are very widely used.

2. *Key-value*

   Key-value systems support single key-value operations: get by key, put a key-value pair into the store and delete by key.

3. *Column-oriented*

   Column-oriented systems use associative arrays of columns with no fixed column schema and do not support joins between these arrays.

4. *Document-oriented*

   Document-oriented systems provide indexed storage of structured documents, such as files in XML or JSON formats, associated with descriptive meta-data. They do not support joins between documents.

### CAP Classification

The *classification* of data store systems based on the CAP configuration and data model is the following (illustrated in figure 2.1):

1. *Consistent, Available (CA) Systems*:

   The data in these systems are always available and consistent, which is important for the cases when we need ACID-compliant transaction support. For example, a banking system must operate using transactions in order to guarantee that money is not lost during failed, incomplete transfers. As a consequence, CA systems have poor resistance to network partitioning. They typically use replication strategies to deal with this problem.

   a) **relational**: "traditional" (PostgreSQL[1], MySQL[2], etc), Aster Data[3], Greenplum[4]

   b) **column-oriented**: Vertica[5]

2. *Consistent, Partition-Tolerant (CP) Systems*:

   The data in these distributed systems are consistent across the network, even across partitioned nodes. At the same time they may have trouble with availability during times when the computer nodes are partitioned. This means that the client will always have the last version of data, but must wait until the network will be repaired.

[1]PostgreSQL: http://www.postgresql.org
[2]MySQL: http://www.mysql.com
[3]Aster Data: http://www.asterdata.com
[4]Greenplum: http://www.greenplum.com
[5]The Vertica Database: http://www.vertica.com

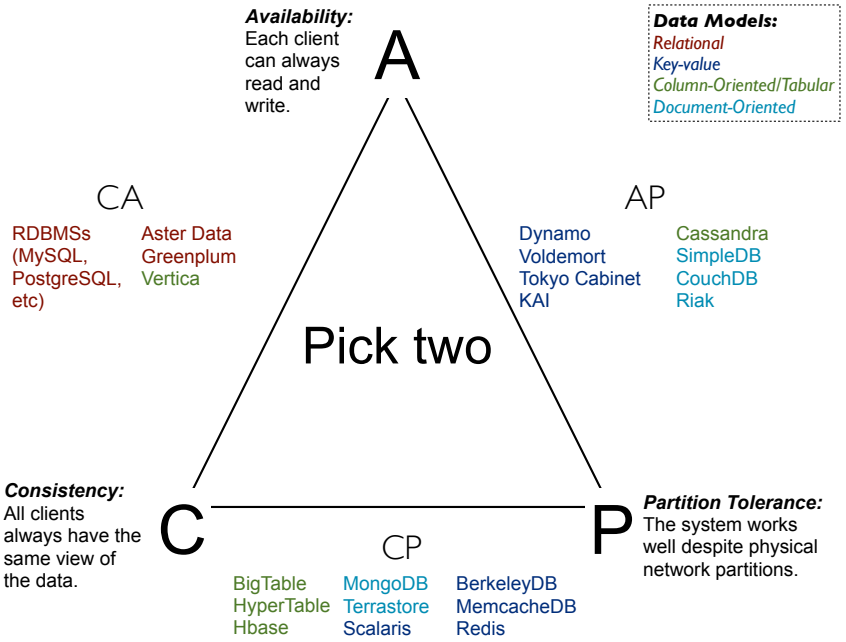



Figure 2.1: Visual guide to NoSQL Systems. [Nathan Hurst's Blog [Hur10]]. The vertices of the CAP-triangle are properties: Availability, Consistency and Partition Tolerance. The sides of the triangle are the possible choices for the systems to abandon one property and have two others.

a) **column-oriented/tabular**: BigTable [CDG+06], HyperTable[6], HBase[7]

b) **document-oriented**: MongoDB[8], Terrastore[9]

c) **key-value**: Redis[10], Scalaris[11], MemcacheDB[12], BerkeleyDB[13]

3. *Available, Partition-Tolerant (AP) Systems*:

   The data in these systems are always available across partitioned nodes in the network. However, the client might have not the latest version of the data. The systems use "eventual consistency" – updates are not blocking and eventually propagate to all nodes.

   a) **column-oriented/tabular**: Cassandra[14]

   b) **document-oriented**: CouchDB[15], SimpleDB[16], Riak[17]

[6] HyperTable: http://hypertable.org
[7] HBase: http://hbase.apache.org
[8] MongoDB: http://www.mongodb.org
[9] Terrastore: http://code.google.com/p/terrastore
[10] Redis: http://redis.io
[11] Scalaris: http://code.google.com/p/scalaris
[12] MemcacheDB: http://memcachedb.org
[13] Oracle BerkeleyDB: http://www.oracle.com/us/products/database/berkeley-db/overview/index.html
[14] Apache Cassandra: http://cassandra.apache.org
[15] Apache CouchDB: http://couchdb.apache.org
[16] Amazon SimpleDB: http://aws.amazon.com/simpledb
[17] Riak: http://wiki.basho.com

c) **key-value**: Dynamo[DHJ+07], Voldemort[18], Tokyo Cabinet[19], KAI[20]

### 2.3.2 Specific requirements for the data storage systems

Our focus is on storage requirements for comparative genomics. For the good outward scaling we need to have strong tolerance of network partitioning. Because of the CAP theorem we need to abandon either consistency or availability[Hur10]. For our applications we choose Availability at the expense of Consistency: it is not critical for us if the client does not work with the latest version of data since we have many inserts but few updates. Our analyses are global and work on large data sets, there are many write operations, but we do not need full consistency in our data store: the results of the global computations do not change considerably between two similar versions of the data.

This leads us to NoSQL (Not Only SQL) distributed databases, which do not give full ACID guarantees but are good in scaling. NoSQL systems typically lose the clarity and guarantees of the relational algebra, as well as transaction support, in exchange for partition tolerance. We will describe them in more details in the subsection below.

Available Partition-Tolerant systems use "eventual consistency": updates/insert are not blocking and eventually propagate to all computer nodes. Practically, small updates propagate immediately, while for the big ones it might take a few seconds. Thus, with the common requirements, defined in section 2.2, the *requirements for the data storage systems are:*

1. Petabyte-scalable (implies Distributed and Fault-tolerant)
2. Partition-tolerant (implies Fault-tolerant)
3. Fits to the usage pattern: lots of get and insert queries, few updates
4. Highly available
5. Eventually consistent
6. Easy to use and low cost

### 2.3.3 NoSQL

As we said in the previous subsection, the choice of Partition-Tolerant storage systems leads us to NoSQL systems.

In recent years, software applications in many fields have come to require processing of Petabyte datasets. Traditional relational databases are found to scale poorly beyond tens of terabytes [Cat11], and commercial offerings in parallel databases are expensive and difficult to maintain [PPR+09]. An emerging alternative in distributed fault-tolerant storage and computing is cloud-based "Big Data" approaches[SLBA11]. Unlike classic technologies, typical NoSQL databases do not have a fixed relational schema: they provide nested key-value associative arrays, called column families, where each row may have an unlimited number of columns. [Cat11, PPS11].

Relational databases operate with relations and explicitly store them. A relation is defined as a set of tuples that have the same attributes. A relation is usually represented as a table, which consists of rows and columns. The tuples must have some identifier (it

[18]Voldemort: http://project-voldemort.com
[19]Tokyo Cabinet: http://fallabs.com/tokyocabinet
[20]KAI: http://sourceforge.net/projects/kai

can be a key or an index). Other tuples in the database might use this identifier as a foreign key.

In contrast, non-relational databases store data without explicit links between data elements [Cat11, PPS11]. The responsibility for correctly storing and maintaining relations between data objects is now with the client software application: the storage system itself does not check the data schema.

### 2.3.4 Cassandra NoSQL datastore

Apache Cassandra[21] [LM10] is a highly scalable distributed semi-structured NoSQL data store.

Cassandra satisfies all our requirements defined in subsection 2.3.2:

1. Petabyte-scalable (Distributed and Fault-tolerant)

   Cassandra is a highly scalable [CS11] distributed fault-tolerant data store, where data are automatically replicated and balanced across an ad-hoc peer-to-peer network of storage nodes in a computing cluster. Data are partitioned and balanced across multiple nodes in a cluster, and aggregate queries are distributed by default.

2. Partition-tolerant (implies Fault-tolerant)

   Cassandra has no single point of failure. It is able to seamlessly continue working even after of network partitions or node failures. Data are replicated into several nodes and in case of node failure the query will get information from the other node.

3. Fits to the usage pattern: many of get and insert queries, few updates

   In Cassandra low level insert queries are not blocking, Cassandra does not check if the value already exists, it simply overwrites it. Thus insert and get queries are very fast.

   The low level update operation is identical to insert one in Cassandra. Data are denormalized in Cassandra, one high level entity can be stored physically in several places as part of several indexes for the different kinds of searches. To update some attribute of some high level entity we may need to delete the old attribute from several column families (indexes) and than add the new one. Thus, high level updates are expensive, but since we do not have many updates, it is not very important for us.

4. Highly Available (implies Fault-tolerant)

   Firstly, as we noticed above, the node failure will still keep the system available without any delay. Secondly, inserts and updates are not blocking concerned data rows, the data is always available for reading and writing.

5. Eventually consistent

   Cassandra uses eventual consistency with the possibility to tune the level of consistency we want for each query. We can perform full-consistent queries with lower performance or fast queries without consistency guarantees.

[21] The Apache Cassandra Project: http://cassandra.apache.org

6. Easy to use and low cost

   Cassandra is an open-source project. It is extremely easy to install and maintain. Capacity can be added easily with few commands dynamically without disturbing applications running on the cluster.

### Cassandra architecture

Cassandra is an *Available Partition-tolerant system* - highly available with no single point of failure datastore. It has *no central master node*, responsibility is distributed among the nodes. So any data can be written/read to/from any of the nodes in the cluster. The row key value controls which nodes stores the columns. Each data item is replicated on $N$ hosts, where $N$ is the *replication factor* configured for the keyspace [LM10].

Cassandra uses a ring topology. Each computer node is assigned a unique token that determines for what keys it is the first replica [cas]. Thus, each node in the ring is responsible for some section of the data space (Figure 2.2). There are two replication strategies: *SimpleStrategy* for a single data center and *NetworkTopologyStrategy* for multiple data centers.

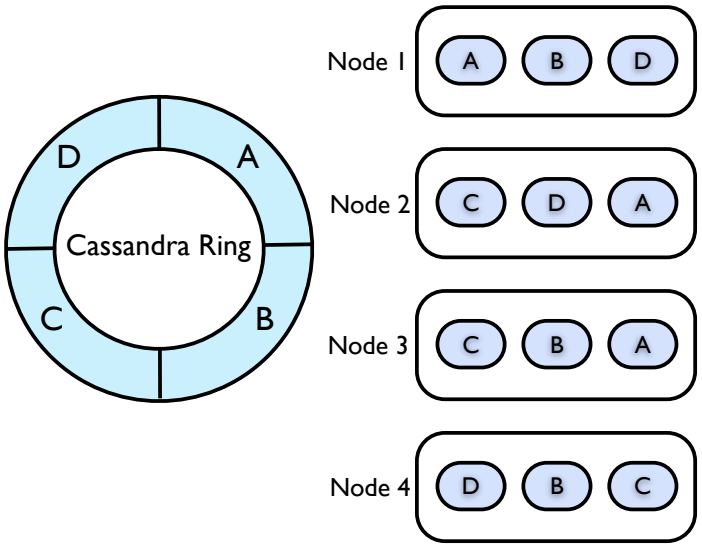



Figure 2.2: Cassandra ring with replication factor = 3. The row keys A, B, C, D are replicated three times and distributed across computer nodes Node 1, Node 2, Node 3 and Node 4. [http://www.datastax.com].

Cluster membership is maintained by a Gossip protocol [DGH+87, ADH05] on a peer-to-peer network. The protocol provides only probabilistic guarantees instead of strong reliability ones to achieve better scalability and fault tolerance [ADH05]. The authors in [ADH05] proved, that in gossip protocol the probability of partitioning is exponentially small.

In Cassandra *Partitioner* defines a scheme of organizing row keys in the keyspace, it has impact only on the key (but not column) sorting strategy. The partitioner must be chosen at the moment of cluster configuration. We can perform one or two level range queries depending on which partitioner we use.

There are two major types of *partitioners* [Hew10]:

1. *Random Partitioner*: default partitioner. The keys are automatically balanced across the cluster using MD5 hashing. The order of the keys is random, so the key range queries are not possible.

2. *Order Preserving Partitioner*: The keys are UTF-8 strings. Rows are sorted by a key order. There is a high potential risk of disbalanced nodes, but it is possible to perform key range queries.

There are also:

3. *Collating Order Preserving Partitioning*(The keys are UTF-8 strings in United States English Locale(en_us)) and

4. *Byte-Ordered Partitioning*(Data are treated as a raw bytes),

Cassandra provides tunable quorum-based *eventual consistency* [cas]. We can set *consistency level* for each get or insert query. Basically, we can configure consistency level for reading using replication factor value $N$ as:

1. *All*: it ensures that all N replicas provide the same version of the data and any outdated or unavailable (due to fault) replicas will fail the operation. In this case we gain full consistency in exchange for availability.

2. *Any*: the data is taken from the first answered node. The query response is fast, but might be inconsistent.

3. *Quorum*: it ensure that the major number of nodes agree on the response. Quorum level is a good compromise between consistency and performance (availibility).

There are also other consistency levels, such as: *One*, *Two*, *Three*, *Local Quorum*, *Each Quorum* [cas]. Similarly, we can set consistency level for writings.

In Cassandra columns are sorted by the sorting operation "Compare with" which we define for column family. These comparing operations are: AsciiType, BytesType, LexicalUUIDType, LongType, UTF8Type, TimeUUIDType (sorts by a timestamp), etc. Also it is possible to define a custom comparison operation. In contrast with traditional relational databases it is not possible to compare columns by values.

**Cassandra data model**

Cassandra uses a ColumnFamily-based data model. The smallest increment of data is a column. A *column* is a tuple (triplet) that contains a name, a value and a timestamp. All values including the timestamp are provided by the client. Both names and values can be integer or float numbers, strings or binary objects. There is no imposed declarative schema for the column names and values. A data *row* is a list of columns, sorted by column name, identified by the row key. At the moment of creating column family we can set one of the several sorting algorithms to order column names, or implement our own one. The row key is a unique string of the unlimited size.

A set of rows with the corresponding row keys is a (standard) *column family*. We can predefine some list of column names if almost all rows have them. If Column Family has only these, static columns, it is called *static*. If there are arbitrary, not static, columns it is called *mixed*. If there is no list of predefined column names the column family is called *dynamic*. We can create secondary indexes for these predefined columns. Dynamic columns may have arbitrary names and can be used as a dimension for the range queries.

Cassandra also supports *super columns* - associative arrays of columns, which provides one more level of nesting. Both columns and super columns are sorted by their names. Also we can change the sorting order: the system allows us to sort columns by time instead of by name. Super column family is a "column family within a column family" [LM10]. Standard and super column families are organized into *Keyspaces*. Keyspace is simply a namespace for column families. (Figure 2.3). Typically, there is one keyspace per application.

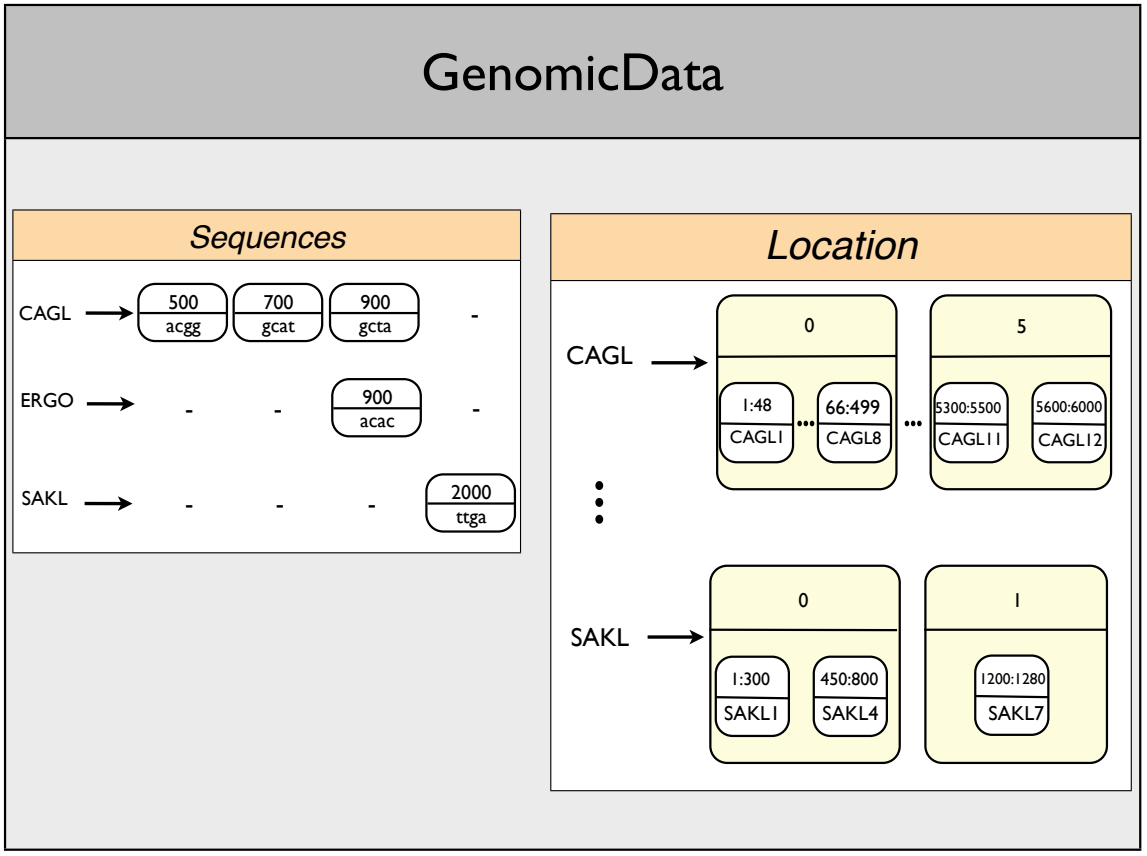



Figure 2.3: Example of a Cassandra Keyspace "GenomicData" (for simplicity we do not show timestamps) that stores information about genomic sequences and features of that sequences. This keyspace contains one Standard Column Family "Sequences" that maps sequence names, start coordinates to the sequences and one Super Column Family "Location" that stores the coordinates of the sequence features on the genomic sequences.
The row keys of "Sequences" Column Family are the sequence names. The column names here are start coordinates on sequences. The column values are the sequence pieces - strings of letters a, c, g, t.
The "Location" Super Column Family has sequence names as row keys. Each sequence name refer to a sequence features pairs of coordinates (column names) which are discretely organized in bins (super column names). The column values are the sequence features names.

Cassandra dynamic Column Families do not have declarative schema. The nested boxes here only show how we should interpret the data of the corresponding column families. This representation is a contract, and the responsibility of respecting this contract is on the application side, not on the database one.

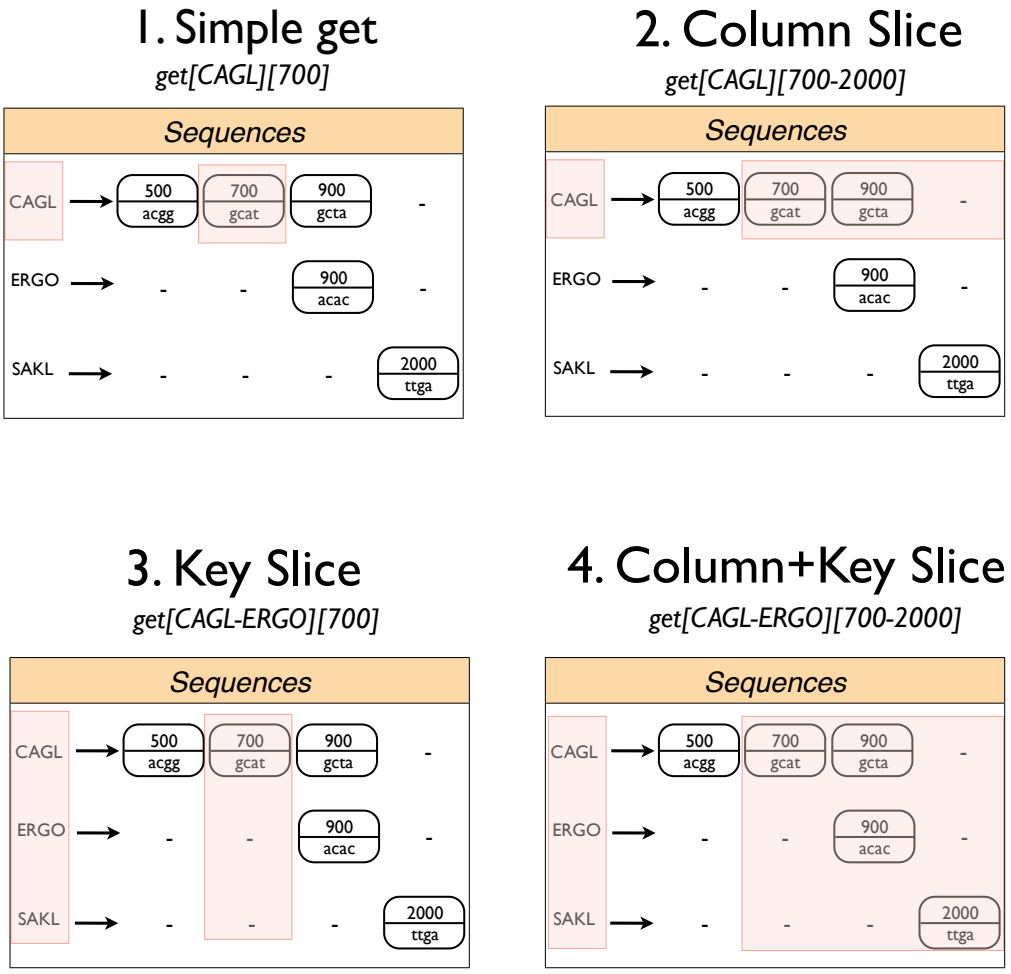



Figure 2.4: Contrasting different ways of selecting data from Cassandra.
1. Simple get uses a key (CAGL) and a column name (700) to obtain corresponding column value. There is also a Multiple get query, where we can set discretely several input key values.
2. Column slice allows us to define a single interval of column names (700-2000), which are sorted by default, to retrieve column values. Here we also need to specify one or several key values (CAGL).
3. Key Slice, similarly to Column Slice, allows us to retrieve column values in the single interval of keys (CAGL-SAKL), specifying column name (700). The main difference is that row keys are used to distribute the data rows, so the best strategy is to use Random Partitioner that does automatic load balancing. Thus, row keys are not sorted by default as column names do. To use Key Slice we need to set Order Preserved Partitioner for our Keyspace in the beginning and maintain load balancing ourselves in the application level. That is why Key Slice is not very used, and when it is used only when there is a extreme need in 2-dimensional slices
4. Column+Key Slice allows us to choose both an interval of keys (CAGL-SAKL) and an interval of column names (700-2000) to retrieve column values.).

### Cassandra queries

Originally, Cassandra provided only a low-level Thrift[22] RPC-based API [cas]. Using the Thrift software framework it it possible to generate code to build scalable cross-language

[22]Thrift: http://thrift.apache.org

services, that can work with languages such as C++, Java, Perl, C#, Ruby, etc. Thus, using Thrift API we can connect to Cassandra using plenty of different programming languages.

Cassandra CLI[23] is a command line interface utility (based on Thrift API), that can be used to define schema, to insert and retrieve data.

Since the version 0.8, in Cassandra there is CQL - Cassandra Query Language [cql][24]. It is similar to SQL, a rich declarative query language based on relational algebra for relational databases. Even though CQL looks similar to SQL, there are some important differences. CQL adjust to the Cassandra data model, which means there are no join operators between column families and no row range queries for the clusters with random partitioner.

Cassandra is designed around the queries. The data are organized in rows and distributed across network by row key. The main filter, the main parameter for the get queries are row keys. Get queries that use secondary indexes work very inefficient without specifying row keys. Thus, to design Cassandra Column Family we need to put the main focus on the choice of the row keys.

Relations between objects must be stored explicitly in the row key or column (standard or super) names to be a searchable attribute. Also, there is no join operation between column families, so we need to design column families in advance with respect of types of queries. Typically, each standard/super column family corresponds to a type of user query.

In Cassandra there is no difference between low level Insert and Update operations: in the Thrift API there is only "insert" operation, in CQL there are both these operations, but their semantics are identical. Insert/update operation does not check if the column already exist, the new value (eventually) substitute the old one.

There is no server side transaction support. Column families are denormalized, the same data might be in several places. And it is the client applications responsibility to maintain the same value in several column families.

### Cassandra operations in Cassandra-CLI

For simplicity, we will use Cassandra-CLI command line interface to demonstrate the different kinds of Cassandra operations. After, we will show the examples of the code in CQL language.

The data in comparative genomics are semi-structured. Consequently, we store our objects serialized and we use Cassandra as index for the objects. Static columns are used to create secondary indexes. Secondary indexes are found to have bad performance, so we do not use them, thus all our columns are dynamic and we use column names as dimensions for the queries.

In our example, we have a set of sequence features and we would like to perform a standard query by *name* with an optional parameter *type* on these data. For this case we will create a column family and will demonstrate different possible Cassandra queries.

### Create data schema

To create Cassandra data schema we need first to create a Keyspace. Keyspace typically is one per application, thus we will use it for all our genomic indexes.

[23] Cassandra CLI documentation : http://www.datastax.com/docs/0.8/dml/using_cli
[24] CQL Language Reference: http://www.datastax.com/docs/1.0/references/cql/index

We will create the same keyspace "GenomicData" as in figure 2.3 using a command *CREATE KEYSPACE*. This keyspace (basically is a set of indexes) contains information about genomic sequences and features of that sequences. Sequence feature is a genomic object, that has several attributes: id, name, type, coordinates on the corresponding sequence, etc.

For this keyspace we specify SimpleStrategy as a placement strategy class and the replication factor number = 3 (discussed earlier in this subsection). We can connect to this new Keyspace using command *USE*.

In this keyspace we create a Standard Column Family "Sequences" (as in figure 2.3) using a command *CREATE COLUMN FAMILY* as it is illustrated in the Listing 1. This Column Family is a mapping between sequence names, their start coordinates and sequences. As we can see, at the moment of a dynamic Column Family creation there is no information about imposed columns structure.

**Listing 1** Cassandra-CLI: create data schema

```
                                          ▷ Create Keyspace "GenomicData"
CREATE KEYSPACE GenomicData
  WITH placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy'
  AND strategy_options = [replication_factor:3];
                                          ▷ Connect to the GenomicData Keyspace
USE GenomicData;
                                          ▷ Create Column Family "Sequences"
CREATE COLUMN FAMILY Sequences
  WITH comparator = IntegerType
  AND key_validation_class=UTF8Type;
```

### Insert data

The way how we insert data defines the actual structure of Column Families. Since there is no schema control for dynamic Column Families, the applications should be very careful with inserts.

To insert data we need to use the command *SET* with the name of the Column Family ("Sequences"). We put the row key and the column name in the brackets and the column value after "=". In our example we use sequence names (CAGL, ERGO, SAKL, etc) as row keys, start coordinates (500, 700, 900, etc) as column names and sequences as column values (Listing 2). We will discuss how to choose row keys and column families for the searches in the subsection 3.6.1.

**Listing 2** Cassandra-CLI: insert data queries

```
SET Sequences['CAGL']['500']='acgg';
SET Sequences['CAGL']['700']='gcat';
SET Sequences['CAGL']['900']='gcaa';
SET Sequences['ERGO']['900']='acac';
SET Sequences['SAKL']['2000']='ttga';
```

### Simple Get

We can obtain columns and super columns using row key value, name of the column family and, optionally, super column name and column name (see Figure 2.4 and Listing 3):

**Listing 3** Cassandra-CLI: simple get query

```
GET Sequences['CAGL']['700'];
=> (column=700, value=gcat, timestamp=1346609952805000)
GET Sequences['ERGO']['900'];
=> (column=900, value=acac, timestamp=1346610095240000)
```

Using the Cassandra Thrift and CQL APIs (but not the basic Cassandra-CLI implementation), we can specify several row key values for the "multiget" queries.

### Column Slice

For the column slice we specify as well one or more row key values. To limit the columns from the data rows we determine a Slice Predicate. Slice Predicate is similar to a mathematic predicate [cas] and contain either the list of column names that we want to select or the Slice Range - the interval of column names in between we search for the columns (Figure 2.4). Slice Range set ordering, range and limit information.

To illustrate Column Slices we will use pseudocode, since Column Slices for dynamic columns are not supported by the Cassandra-CLI implementation (but supported by Thrift and CQL). The columns in the row are sorted by the column names (using IntegerType strings comparator). Thus, for the row "CAGL" the columns are placed in the order of coordinates on sequence. We can perform a column slice query specifying a column slice range ['700' - '2000'] to have 2 columns as a result (Listing 4).

**Listing 4** Pseudocode: column slice

```
                                    ▷ Column Slice query (pseudocode).
GET_SLICE Sequences['CAGL']['700' - '2000'];
=> (column=700, value=gcat, timestamp=1346612354231000)
=> (column=900, value=gcta, timestamp=1346612369629000)
```

We can get all row values using empty values for the slice range start and end parameters.

### Multi-dimensional range queries and secondary indexes

In addition to a column range slice Cassandra supports two-dimensional key range slice queries (Figure 2.4), but this implies of using low performance Order Preserving Partitioners (as described earlier in this subsection).

Multi-dimensional range queries can be made using secondary indexes on static columns - columns with fixed names. It allows querying by column values. There are several index operators we can use in the index query: EQ(Equality), GTE(Greater than or equal to), GT(Greater than), etc. But the limitation is to have at least one EQ operator which will be primary filter for the index query. Secondary indexes are suitable for cases when many rows share the same indexed value. When the indexed values are more unique, the cost of querying and maintaining the index becomes too high. For the general cases, the performance of the secondary indexes find out to be quite poor, but can be improved

using the CCIndex mechanism [FZX11]. CCIndex [ZLW+10] (Complemental Clustering Index) offer support of multi-dimensional high performance and reliable range queries over Distributed Ordered Tables (like BigTable, HBase, etc). CCIndex mechanism, applied to Cassandra can significantly increase the speed of range queries [FZX11].

#### CQL

Since the version 1.0 of Cassandra, the queries in CQL (Cassandra Query Language) look similar to the SQL ones. It is very convenient to use CQL especially for the column families that use static columns (SQL-like).

To demonstrate CQL, we will create a static Column Family "Features" for the set of sequence features. These features have some attributes: name, type, start and end coordinates on the sequence. In our example (Listing 5) we create table ( alias for column family), insert data (columns) and get data using a SQL-like command SELECT. The visible difference here, that we may specify Consistency Level for the get and insert queries. Note, that the ranges are inclusive, so it is not possible to use operators "<" or ">".

Not all the features of Cassandra are implemented in CQL. Super Columns are not supported, in CQL only a subset of available column family properties can be set.

**Listing 5** CQL code example

```
                                          ▷ Create Table"Features".
CREATE COLUMN FAMILY Features (
  KEY varchar PRIMARY KEY,
  name varchar,
  type varchar,
  start int,
  end int)
  WITH comparator=timestamp
  AND default_validation=int;
                                          ▷ Insert some data into "Features".
INSERT INTO Features (KEY, name, type, start, end)
  VALUES ('acd55ddd', 'SAKL1', 'gene', 1, 250)
  USING CONSISTENCY QUORUM;
                                          ▷ Get data query.
SELECT * from Features
  WHERE type = 'gene'
  AND end <= 300
  USING CONSISTENCY QUORUM;
```

### 2.3.5 Cassandra existing adaptations

Many applications in many different domains adopted the new NoSQL storage paradigm. Cassandra Community is very large and continues growing[25].

In [CGOP11] the authors show how Telecom applications can be adapted to Cassandra NoSQL data store. The main motivation for moving to NoSQL solutions is scaling problems in the domain of telecommunications. They propose a following schema for their applications: a Super Column Family with a caller phone number as a Cassandra row key,

[25]Cassandra Users List: http://www.datastax.com/cassandrausers

type of the call as a Super Column Name and the columns correspond to the list of available services with the account identifications. Their performance comparisons showed that Cassandra achieves better results than PostgreSQL for their schema and data workload (7 000 000 customer initiated calls).

Disney[26] Technology Solutions and Services uses Cassandra as a query service[27] [Jac12] in their Big data platform. This platform is based on Cassandra, MongoDB and Hadoop. One of the main uses of the Disney's Big data platform is to collect, store and analyze user experiences. They treat data as a service, providing higher level of data abstraction and hiding operational complexity from application developers. Their Cassandra usage choices are following: they use Random Partitioning in their cluster and DSE Search/Solr[28] for the search and ordering. Also, they search across columns in a single row, not across rows.

EBay[29] were seeking for a cost-effective robust solution for processing enormous data volume which does not have transactional constraints. They needed to perform large scale analyses on structured and unstructured data[30]. EBay has chosen Cassandra, integrated with Hadoop MapReduce as a scalable storage and analytics solution with great write performance and high availability. EBay utilizes Cassandra for the social signals (for example, how many user own or like this item), logging, tracking and analytics[31] [Pat12]. For each user signal ("like" and "own") they have two indexes (Cassandra Column Families) from the user id to the item counter and from item id to the user counter. The same way they store the graphs "users-items" for the analytics.

Plenty of other companies use Cassandra for the logging and store different kinds of counters. Cassandra is used as an underlying storage system for Hadoop MapReduce applications or as a tool for the analytic searches or to store archives and backups.

Cassandra is designed to be especially good for extreme large scale distributed analytics. Thus, it is able to handle petabytes of data, which is one of the most urgent need now in comparative genomics. Another essential aspect of comparative genomics is global computations, which typically involve systematic all-against-all comparisons. Cassandra is integrated with Hadoop MapReduce framework for large-scale computations and distributed Data Warehouse Hive on top of Hadoop, which makes a good fit to our needs. However, as a consequence of such a design, Cassandra is not efficient for the modest size of data. The same way the search queries might be less performant than in some other NoSQL systems, that put some indexes in memory (for example Couchbase[32]). Another consequent problem is that the good design of Column Families in Cassandra is quite complicated. The document-based NoSQL systems (MongoDB, Couchbase) have more intuitive ways of storing data (they store JSON documents), but they do not support petabyte scale and are not designed for analytics purposes.

[26]The Walt Disney Company: http://www.disney.go.com

[27]BigData at Disney presentation on Cassandra Summit 2012: http://www.datastax.com/wp-content/uploads/2012/08/C2012-BigDataatDisney-ArunJacob.pdf

[28]DataStax Enterprise Search: http://www.datastax.com/docs/datastax_enterprise2.0/search/dse_search_start

[29]eBay: http://www.ebay.com

[30]DataStax Case Study: eBay :http://www.datastax.com/resources/casestudies/ebay

[31]BigData at eBay presentation on Cassandra Summit 2012: http://www.datastax.com/wp-content/uploads/2012/08/C2012-BuyItNow-JayPatel.pdf

[32]Benchmarking Couchbase Server: http://www.couchbase.com/presentations/benchmarking-couchbase

## 2.4 Systems for ad-hoc algorithms

In the previous section we talked about storage systems and their requirements. In this section we will talk about global analyses on Big Data and the necessary requirements for such systems. We will consider the MapReduce paradigm for parallel computations, the Hadoop MapReduce framework and a declarative language for MapReduce - Pig Latin.

### 2.4.1 Cloud and Grid Computing. Programming models

Cloud computing has recently been advanced as a solution for scaling problems. It involves processing distributed computations on a large number of computers connected through a real-time network. Cloud computing provides storage and computing capacities as a service. There are several Cloud computing models. The most basic one is Infrastructure as a service (IaaS), where virtual machines and other resources are provided. Clouds, that use Platform as a Service (PaaS) model, typically provide operating system, database and execution environment. In the Software as a Service model Clouds provide an access to databases and applications. The Network as a service (NaaS) Clouds provide different network and inter-cloud connectivity services.

Cloud computing share the same vision with Grid computing [FZRL08]: their aims are to reduce computing cost, to increase flexibility and reliability. But there are many differences as well. The demand for computing has dramatically increased last years and there is a need for massive scale computing systems. Figure 2.5 [FZRL08] illustrates the relations between Cloud and Grid computing: Clouds are service oriented systems and considered to scale better than Grids.

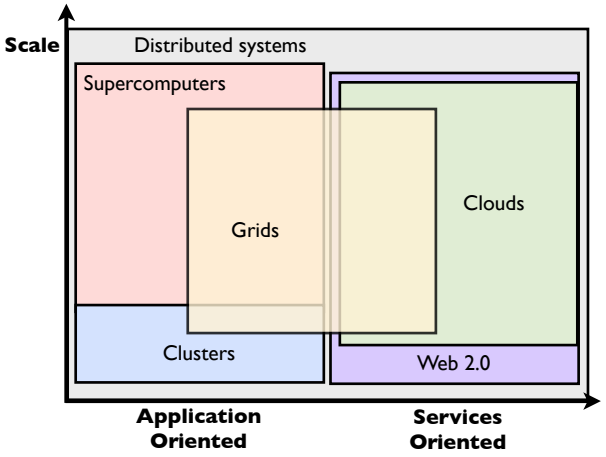



Figure 2.5: Grids and Clouds Overview shows how well different distributed systems scale and demonstrate their service or application orientation. [FZRL08]

In the article [HK11] the authors compared different cloud computing services using main characteristics such as: license type, intended user group (private or corporate use), security and privacy, payment systems, openness of clouds, interoperability and portability, etc. The authors introduced a taxonomy for the cloud services, and represent it as a tree (Figure 2.6).

The common point about Clouds and Grids is that they do not magically solve the scaling problem. Scaling and fault tolerance require us to adopt distributed architectures,

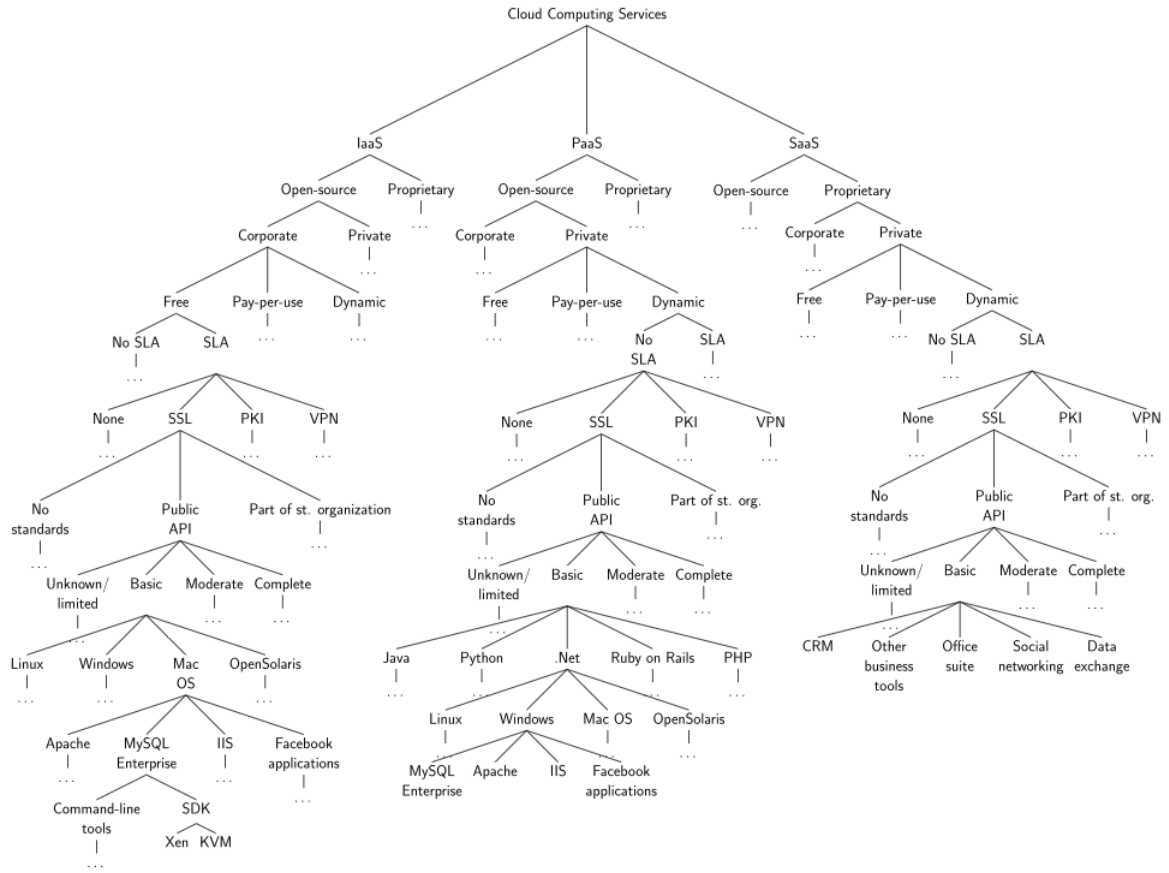



Figure 2.6: The cloud computing services tree. Figure is taken from [HK11]

and the key problem is that we need to choose the correct ways of rewriting existing algorithms so that they work in these architectures.

The most common programming model for Grids is the Message Passing Interface (MPI) [For94]. There is no single flow of computations, but a set of parallel tasks. The tasks use their own memory for computations and communicate between each other by passing messages [FZRL08].

MapReduce [DG04] is another programming model. There is no message between tasks (jobs), the communication is through grouping. Apache Hadoop[33] is a cluster-based implementation of MapReduce. Cloud MapReduce [LO11] is the implementation of MapReduce on top of Cloud Operating System. Reference [HLD09] discusses the strategies of efficient MapReduce implementation on top of MPI. Hadoop MapReduce 2.0, called YARN[34] (Yet Another Resource Negotiator), supports also MPI as an alternative for MapReduce.

Since there are many technologies, we need to base our choice on the specificities of the comparative genomics application. First, we need distributed storage that scales to $10^{12}$ and works well with offline analytics. This leads us to choose NoSQL rather than SQL, and (using the CAP theorem) AP without C. Second, the algorithms we want to use are typically global, starting with a systematic all-against-all comparison, followed by statistical or combinatorial classification of the distance matrix. This leads us to choose

[33]Apache Hadoop: http://hadoop.apache.org

[34]Hadoop YARN: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

a distributed MapReduce approach, rather than a fine-grained message-passing approach, or a classical linear algorithm.

### 2.4.2 MapReduce and Apache Hadoop MapReduce framework

MapReduce is a paradigm for defining fault-tolerant distributed computing of large datasets across clusters of computers [DG04]. In MapReduce, algorithms are decomposed into pairs of phases using a functional programming style. In the map phase, a single **map** function splits input data into independent data chunks and processes each map job in parallel, associating a key value with the result value. In the reduce phase, the framework puts the groups of results with the same key value to a **reduce** jobs, which are processed as well in parallel and provide the array of final results (Figure 2.7). Since both phases are functional, if a node fails, its part of the computation can be recomputed without invalidating the other parts, leading to improved fault tolerance and reduced data loss. MapReduce is easy to use, it does not require special knowledge in parallel programming, it hides details of parallelization:partitioning, scheduling jobs executions and handling machines failures [DG04].

Apache Hadoop[35] provides a open-source software framework for MapReduce. Hadoop MapReduce framework uses master-slave architecture: there is single master JobTracker and one TaskTracker perl cluster-node. JobTracker schedules the jobs and TaskTrackers execute them [map, Whi09]. The client application should set input/output locations for the job and implement map and reduce functions using given framework interfaces.

Hadoop is implemented in Java, so MapReduce programs should either be written in Java or use the Hadoop Streaming facility.

Hadoop supports input and output connections to Cassandra. There is no Hadoop output data streaming[36], that is why Java is the only possible programming language to use with Hadoop-Cassandra.

### 2.4.3 Pig Latin - declarative language for MapReduce

Apache Pig[37] [ORS+08] is a open-source platform for large-scale analyses, that provides the high-level textual language Pig Latin and an infrastructure for writing and evaluating analysis programs written using this language. Pig compiles Pig Latin expressions into a sequence of MapReduce jobs and executes them on an existing MapReduce implementation, the Hadoop framework. The Pig Latin language simplifies the writing, understanding and maintaining of MapReduce programs. It becomes trivial to execute "embarrassingly parallel" analyses.

In Pig there is no need to worry about map, shuffling and reduce phases. The operator *FOREACH* is equivalent to the *map* phase of MapReduce. The same function (map function) which we write inside *FOREACH* is applied in parallel to each data element. The *GROUP* operator groups the data elements by some attribute and is equivalent to the *combine* phase of MapReduce (see Listing 6 for the Pig code and Figure 2.7 for the MapReduce diagram).

Complex analyses are executed through several data transformations. Pig provides a rich library of operations as well as possibility to create one's own user-defined functions. Pig is very effective in iterative processing and it supports semi- or unstructured data [pig].

[35]The Apache Hadoop Project: http://hadoop.apache.org
[36]Cassandra wiki Hadoop support: http://wiki.apache.org/cassandra/HadoopSupport
[37]Apache Pig: http://pig.apache.org

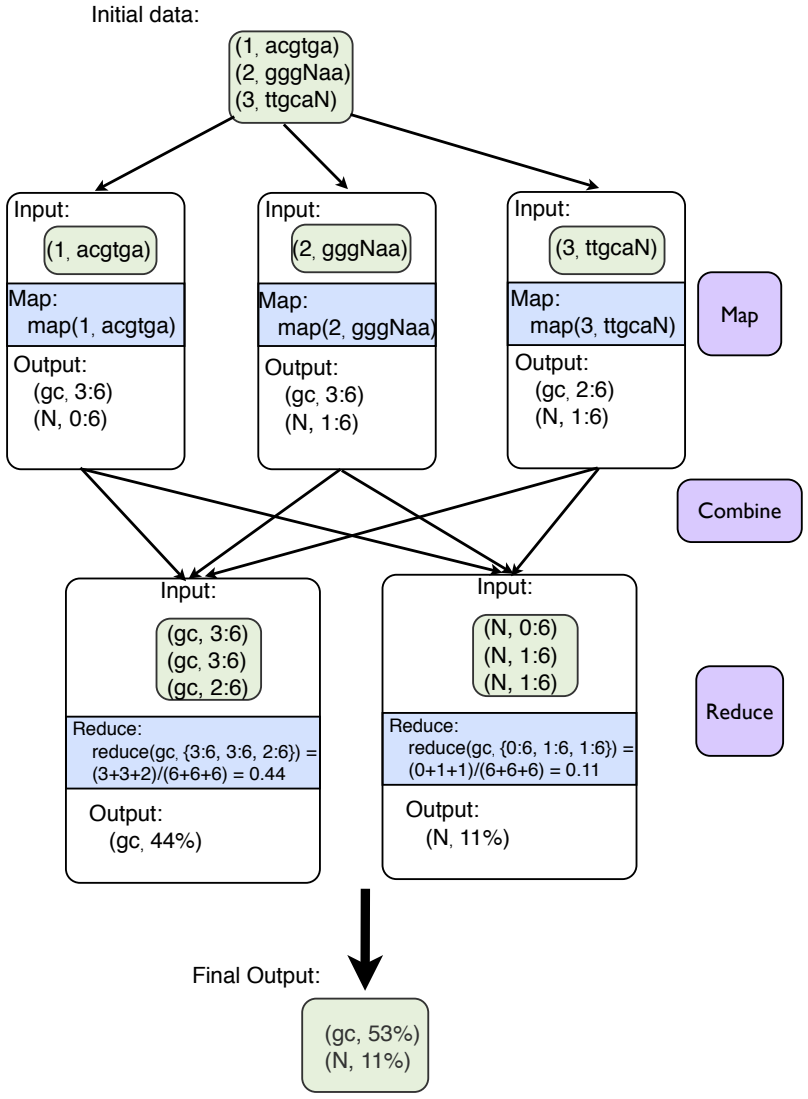



Figure 2.7: To demonstrate the power of MapReduce paradigm, we chose an example of two simultaneous computations: gc-content and the content of gaps (N-content). Basically, we count in parallel the number of letters g or c and N in the input gene sequence, that is a string of letters a, c, g, t (nucleotides) and N (gaps). Input string is cut into three pieces. In the map phase we count the relative number of g and c letters in each piece of sequence in parallel. The same way we count the number of N. The output of the map phase are the key-values pairs where the keys are the result types ("gc" or "N") and values are the results of the calculations. The Combine step of MapReduce groups the outputs of Map phase by keys. In the Reduce phase we summarize calculations to obtain total gc- and N-contents.

**Listing 6** Calculating GC-content for each gene sequence and grouping the output by gc-content

```
genes =
  LOAD 'genes' USING PigStorage()
  AS (gene_id:chararray, gene:chararray);
GCs = FOREACH genes GENERATE gene_id as gene_id, GC(gene) as gc;
                                                  ▷ GC is a User-defined function
grouped = GROUP GCs by gc;
```

### 2.4.4 Evaluation of MapReduce

#### Parallel databases vs MapReduce

For more than 20 years parallel database systems have been commercially used for distributed computations, such as Teradata, Aster Data, Vertica, Greenplum, DB2, Oracle and others. Almost any computation can be expressed as a sequence of queries using user-defined functions [PPR+09]. We compare computations via parallel databases and Hadoop MapReduce using the requirements defined in section 2.2:

1. *Large-scalable (petabyte-scalable)*

   Both Hadoop MapReduce and parallel databases claim to be highly scalable systems. [PPR+09] observes that parallel databases show significantly better performance than MapReduce applications.

   a) *Distributed*

      By definition, all these systems are distributed.

   b) *Fault-tolerant*

      Typically parallel databases use master-slave architectures, but they claim to be Highly Available (stronger property than just fault-tolerant) systems by avoiding single points of failure and automatic back-up strategies. The basic Hadoop implementation of MapReduce uses HDFS, the Hadoop distributed file system. This system has a master node, NameNode, which is single point of failure. NameNode stores the metadata about the cluster. But in 2011 it was substituted by the AvatarNode, a software wrapper around the NameNode. There are two avatar nodes in the system: Primary AvatarNode behaves like a NameNode, Standby AvatarNode encapsulates the NameNode and the secondary NameNode. This way the single point of failure is eliminated. DataStax Enterprise[38] is a Hadoop deployment integrated with Cassandra. Instead of HDFS it uses CassandaFS, where all nodes are peers and there is no single point of failure.

2. Fits to the usage pattern: optimal for global analyses, meaning global get-transform-insert operations.

   Global analyses in comparative genomics often involve extract-transform-load operations, which is MapReduce speciality and for which databases are not designed for [SAD+10].

3. *Easy to use and low cost*

[38] DataStax: http://www.datastax.com

Parallel databases are very expensive and difficult to maintain. In contrast, Hadoop MapReduce is a free open-source software and it is extremely easy to configure it [SAD+10, PPR+09].

### Dryad vs MapReduce

There are also proprietary and onky marginally used solutions similar to MapReduce. Microsoft DryadLINQ [YIF+08] suggests LINQ[39] language extensions and a new model for the distributed computations.

1. *Large-scalable (petabyte-scalable)*

   a) *Distributed*

   DryadLINQ is claimed as a generalization of MapReduce, SQL and Dryad [IBY+07] (distributed execution engine for data-parallel programs). Applications can be written in any .NET programming language, and LINQ operators are mixed with the rest of the code [EGF+09].

   b) *Fault-tolerant*

   DryadLINQ as well as Hadoop MapReduce re-executes failed processes.

2. Fits to the usage pattern: optimal for global analyses, meaning global get-transform-insert operations.

   [EGF+09] tests several scientific applications (global analyses) and shows for several scientific applications performance of DryadLINQ is comparable to Hadoop MapReduce.

3. *Easy to use and low cost* Dryad (and consequently DryadLINQ) requires specific Microsoft environment (Windows HPC Server 2008); DyadLINQ released a free version for academic purposes, but yet there are not many users of this software, thus, there is lack of community, compared to the large one for Hadoop.

The most important issue with Dryad that it does not provide a distributed file system, which may explain its lack of success.

### 2.4.5 Existing MapReduce adaptations

MapReduce programming model is widely adopted both by industry and scientific world, especially in bioinformatics.

In [AGMV12] the authors describe a MapReduce solution for the distributed computations of the sequence comparisons. Their "alignment-free" solution, a fractal analysis technique, does not require dynamic programming. The MapReduce decomposition is organized in several steps: encoding into a Universal Sequence Map (USM) [AV02], decoding to verify encoding procedure, distance calculation and, finally, a single MapReduce full sequences comparison.

The article [MHB+10] introduces The Genome Analysis Toolkit (GATK) as a structured programming MapReduce framework for analyzing next-generation DNA sequencing data. GATK provides a small set of data access patterns (or traversal types), which is claimed in the article to cover the majority of analysis tool needs. GATK supports multiple approaches for parallelization of tasks, including shared memory one.

[39] LINQ: Language-Integrated Query http://msdn.microsoft.com/en-gb/library/bb397926.aspx

The article "MapReduce for Data Intensive Scientific Analyses" [EPF08] presents the experience of adapting MapReduce to the two scientific analyses: High Energy Physics (HEP) data analyses and K-means clustering. The authors also introduce a new streaming-based MapReduce implementation, CGL-MapReduce. Each map function of the HEP MapReduce decomposition works with some amount of input files and produces some histograms, then these histograms are merged in reduce tasks.

In [UKOvH09] the authors introduce an approach for materializing the closure of an RDF graph to perform scalable distributed reasoning using MapReduce. They show, that straightforward MapReduce implementation involve too large a number of tasks, so the authors propose three optimizations to reduce the number of jobs.

[PS08] introduces a distributed co-clustering framework DisCo, which they developed as a new practical approachs for distributed data pre-processing. Co-clustering is a technique for clustering rows and columns of a matrix. This technique is widely used by bioinformatics [HZZ+02].

### 2.4.6 Distributed graph algorithms for comparative genomics

Beside the advantages of MapReduce described earlier, we should note, that not all algorithms can be adapted to this approach. Graph algorithms in particular are a challenge because "following an edge" is an essential operation: when we cannot put the entire graph in memory, following an edge through secondary memory may incur extremely high latency.

Many of the analyses in comparative genomics involve constructing and analyzing global graphs. Typically we need to load these graphs into memory to analyze them. The size of the graph grows at least quadratically with the size of elements (vertices). For example, in the identification of fusion and fission events [DNS08] algorithm the last phase contains analysis of a large graph of relations between HMMs. To reduce the size of the problem, in this algorithm the authors extract independent connected components. Unfortunately, some components are still too big to be analyzed efficiently.

Graph algorithms are in general a challenge for MapReduce [LGHB07], since traversal requires random access to nodes, and that is quite expensive because of the network latency. Existing work on graph-based algorithms has focused on converting classical traversal-based graph algorithms to the MapReduce style. Cohen [Coh09] replaces traversal by sorting and defines MapReduce primitives for graph algorithms. Malewicz et al [MAB+10] define the PREGL parallel graph language which is based on an iterative, vertex-centric, message-passing model. At each *superstep*, PREGL maps over all vertices who can send messages to other vertices, which will be available together in the next superstep iteration. Lattanzi et al [LMSV11] define a *filtering* approach that successively partitions the input until each piece fits into the memory of a single machine, where in every round each machine performs a local computation and sends the results to the appropriate other machines for the next round. They show that for the algorithms that they consider, results are obtained after a small number of rounds.

We conclude here, that certain (embarrassingly parallel) algorithms in comparative genomics can be adapted naturally to MapReduce. However, the algorithms that involve graph analyses require more effort to be adapted to MapReduce, because they need random access to the nodes.

## 2.5 Analytics systems

In previous section we talked about the systems for ad-hoc algorithms. In this section we will consider systems for analytical processing: OLAP approaches and different kinds of data warehouses. Analytics systems, being a special case of systems for ad-hoc algorithms, share the same requirements. The difference is that such systems usually are read-only and should in general provide an abstract integrative view of heterogeneous data.

### 2.5.1 OLAP

OLAP (Online Analytical Processing) [CCS93] is a general approach to support decision making, performing multi-dimensional analytical queries. Online Transaction Processing stores (OLTP) are usually source for OLAP stores.

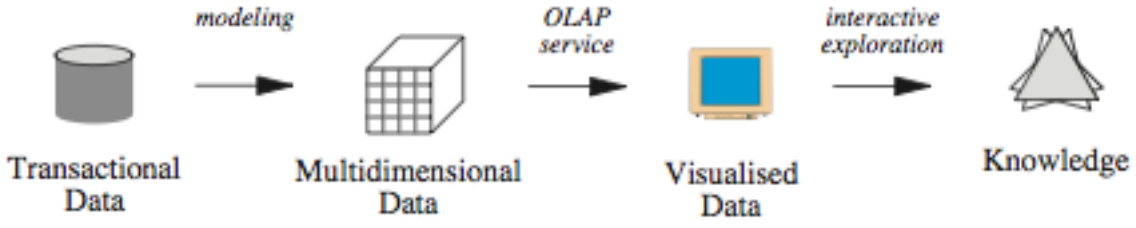



Figure 2.8: Analytics workflow

OLAP systems provide an analytical view optimized for analysis and navigation. The data are summarized, aggregated, and, because of the response time requirements, often precomputed. Data are represented as a $n$-dimensional cube. The cells of the cube are measured values and the dimensions are different indexes. The data elements in dimensions can be hierarchical and can be organized as a set of parent-child relationships. It permits roll up and drill down navigation: for each element we can show and hide child data elements.

OLAP systems suffer a lot from the data volume increase, since the size of multidimensional cubes grows as the product of dimensions. Representing data into these cubes significantly improves the query systems expressive power. The problem is that the data are already large and we need to put the cube (usually aggregated) in memory for the queries, recomputing each time the contents of cells from the raw data.

Fortunately, there are techniques to deal with the representation and optimization problems. It is observed that a multidimensional cube is typically very sparse. Thus, instead of storing the entire cube, we can store just the nonempty cells. We can avoid recomputation during a complex query by using materialized views. For example, in [HRU96] the authors present a technique for efficient representation of sparse cubes. They perform a dependency analysis on the cells and decide which cells are the most useful to materialize. This technique provides a good balance between memory consumption and the average time to answer a query. In order to avoid useless recomputation of cell values. materialized views can be cached between requests.

Data Warehouse, a read-only data storage, is a central element of the OLAP environment.

### 2.5.2 Data Warehouses: BioMart, Hive

A Data Warehouse (DW) is a database used to support decision making. It presents a frozen picture of the data. The data are cleaned, transformed, and prepared to be analyzed by managers, business professionals, domain specialists. The data warehouse presents information consistently and makes it easily accessible, because in this storage system the data are redundant and duplicated.

There are many benefits of using data warehouses. This datastore is optimized for queries, it may have rich declarative SQL-like query language. The data are read-only, thus, warehouse can be de-normalized, which facilitates access to the information.

But, data warehouses also have some disadvantages. The data in DW can get outdated relatively quickly, updates are difficult and can be expensive. For example, to add some information into the offline data warehouse system we need to stop all applications working with it at the moment and then rebuild the storage. It may be necessary to invalidate decisions that were made with the previous version of the data. Thus, these offline data warehouses are appropriate in cases when data do not change frequently.

Online (active, in-time) data warehouses represent the real time (or near real time) data. They might be updated after every transaction performed on the source data. There are also some techniques to improve performance of streaming updates in active data warehouses: the MESHJOIN [PSV+07] algorithm, for example, introduces a novel join operator that operates between a fast stream of source updates and the disk-based relation under the constraint of limited memory.

The active data warehouses seem to fit more in our situation, which concern fast-growing datasets of genomic data.

All data warehouses are read-only systems, built on top of source data storages. DW might be only part of our solution: we still need to have a good high-scalable online storage system under the data warehouse.

#### BioMart - Data Warehouse on top of relational (biological) databases

Biomart[40] is a flexible robust distributed data warehouse. It is read-only and optimized for queries. Often, scientists are required to use many databases from different sources for their analyses. Typically, these databases have different formats and ways of representing the data. BioMart integrates many data sources by providing a single web interface to work with complex biological data [SHB+09]. For example, it allows one to retrieve large amount of data from Ensembl[41], Sanger[42], as well as Uniprot[43] via a single standardized interface. The queries may be performed on individual databases as well as on chains of them.

From the data providers point of view BioMart facilitate the integration of their data with existing data sets on the network.

BioMart is a very powerful federation technology, but it does not fit our requirements. The BioMart is integrative, but not distributed, in the sense that the single databases are stored in the different places, but themselves are not distributed. The source for BioMart is relational databases or flat files, where our data seeks for the new storage technologies such as NoSQL, as we discussed in the section 2.3.

[40]The Biomart system: http://www.biomart.org/
[41]The Ensembl project: http://www.ensembl.org
[42]The Wellcome Trust Sanger Institute: http://www.sanger.ac.uk
[43]he Universal Protein Resource (UniProt): http://www.uniprot.org/

**Hive - Data Warehouse on top of Hadoop MapReduce**

Apache Hive[44] [TSJ+09] is an open-source data warehousing solution for Hadoop. In contrast with BioMart system, the data source for Hive is either HDFS (Hadoop Distributed File System) or CassandraFS (Cassandra File System), both distributed data storages. Hive can be connected with Cassandra using DataStax Enterprise[45] distribution of Hadoop, Cassandra and Hive.

Hive provides a SQL-like declarative query language *HiveQL* that is compiled to a sequence of Hadoop MapReduce jobs. User scripts as well can be plugged into this language. Hive provides a relational model and SQL for Hadoop, which are best adapted to the decision makers on large data sets [pig]. Hive like most of data warehouses is read-only: there are no delete or update operations, Hive implements only the query part of the SQL.

Hive tables can be created with *CREATE TABLE* command:

```
CREATE TABLE genes (gene_id INT, gene STRING)
    PARTITIONED BY (ds STRING);
```

We can insert data using *LOAD DATA* command:

```
LOAD DATA LOCAL INPATH 'genes.txt'
    OVERWRITE INTO TABLE invites PARTITION (ds='2012-09-15');
```

To get data we can use traditional SQL-like *SELECT* command:

```
SELECT count(*), ds FROM genes GROUP BY ds;
```

### 2.5.3 Mahout - Data Mining on top of Hadoop MapReduce

Apache Mahout[46] [CKL+06] is na open-source collection of scalable machine learning libraries. It is mostly implemented on top of the Hadoop MapReduce framework.

Mahout supports four use cases for now:

1. *Recommendation mining*

   The algorithm makes suggestions what the user might like, based on the user experience (for example, recommendations of the similar items in the internet shops or people-to people recommending in the social networks). There are both simple non-distributed recommender implementations and distributed Hadoop-based ones.

2. *Clustering*

   Clustering groups topically related text documents together. Mahout supports many different clustering algorithms: K-Means Clustering, Top Down Clustering, Hierarchical Clustering, Spectral Clustering, etc. There are also plans to implement many other ones.

[44] Apache Hive: http://hive.apache.org
[45] DataStax Enterprise Hive documentation
http://www.datastax.com/docs/datastax_enterprise2.1/analytics/about_hive
[46] Apache Mahout: http://mahout.apache.org

3. *Classification*

   Classification assigns new documents to the existing categories, based on the assignments of the existing documents. Mahout supports several classification algorithms for now (with the plan to implement many others): Logistic Regression, Random Forests, Hidden Markov Model (HMM), Online Passive Aggressive, etc.

4. *Frequent itemset mining (Pattern Mining)*

   The algorithm identifies the sets of items which usually appear together (for example, in a shopping cart or in a query session).

The Mahout community also intends to implement some more algorithms, such as Dimension Reduction (component analysis), Linear Regression, Vector Similarity, etc.

### 2.5.4 Hive and Mahout existing adaptions

Data Warehouse and Data Mining solutions on top of Hadoop MapReduce have many applications in bioinformatics [Tay10]: Mahout libraries for the clustering and classification can be used for the clustering of large genomic data sets, and as classifiers for biomarker identification.

Recommendations systems are highly used by the wide range of applications and projects. The workshop "Workshop on the practical use of recommender systems algorithms & technology" [PKCJ10] has many papers about recommendation mining applications: "Proposal and Evaluation of Serendipitous Recommendation Method Using General Unexpectedness" , "Supporting Consumers in Providing Meaningful Multi-Criteria Judgments" , "Groups Identification and Individual Recommendations in Group Recommendation Algorithms", and others.

Since Hive provides SQL-like query language for data analytics, it can be naturally used for the large scale statistics. Mahout libraries, in contrast, can be used for more elaborated analytics: knowledge mining and pattern analysis.

# Chapter 3

# Genomic Big Data

***Résumé :*** *L'annotation d'un génome définit un ensemble d'éléments qui représentent des objets biologiques, tels ques les gènes codant protéine, des gènes noncodants, des centromères, des régions* cis*-régulatoires, etc ... La génomique comparée utilise à la fois ces éléments et surtout les relations entre ces éléments. Les relations peuvent être des compositions d'éléments en objets plus complexes, de relations pair-à-pair telles que les distances, et finalement des regroupements arbitraires comme des classifications. Dans ce chapitre nous traitons la représentation de ces données et commençons par une formailisation des différents type de données et relations. Afin de procéder à une évaluation réaliste de l'adéquation de NoSQL pour des donnés génomiques, nous avons analysé des fichiers journaux anonymisés du site web de Génolevures pour caracériser les modes d'utilisation typiques des ses utilisateurs. Cette analyse nous a permis de dériver un schéma de stockage d'Apache Cassandra adjusté aux usages et requêtes réels, et implémenter un adaptateur logiciel compatible avec la chaîne GMOD. Ces outils ont été déployé sur un cluster Apache Cassandra et testé en comparaison avec une base de données classique PostgreSQL.*

A genome sequence annotation defines data elements representing biological objects, such as genes that code for proteins, noncoding RNA genes, centromeres, *cis*-regulatory regions, and so on; hundreds of terms are defined in the Sequence Ontology for Annotation[1]. Comparative genomics relies on these data elements, and more importantly on the relations between these data elements. These relations can be compositions of biological objects into composite objects, such as gene arrays or retrotransposons, they can be pairwise relations such as distances, and they can be arbitrary groupings such as classifications. In this chapter we will talk about different ways of representing Genomic Data, including the individual annotation data objects, and more importantly the relations between data elements. We will first formalize the different types of data and relations that are necessary for the specific case of comparative genomics. In order to realistically evaluate the practical use of NoSQL for genome sequence features, we used log analysis of the Génolevures web site to characterize real usage patterns. From these analyses we derived an appropriate Apache Cassandra data storage schema, and implemented a data storage adapter, that compatible with standard comparative genomics data schemas, tools and libraries. We

[1]The Sequence Ontology Project, http://sequenceontology.org, defines terms for biological objects on annotated sequences. As of October 2012, the basic sequence ontology for annotation (SOFA) defines 252 terms, and the full sequence ontology defines 2151 terms.

used these tools to deploy an Apache Cassandra cluster and tested it in comparison with standard PostgreSQL backends for the Magus genome analysis system.

In this chapter we will talk about existing types of multidimensional data structures. Depending on what are the dimensions in the data, we choose different data structure. We will consider the case of semi-structured data elements and sparse relations between these elements. These properties of data make it inefficient to store empty or null-values of attributes or relations. So, we will talk about a representation of data as *tuples*. We distinguish three kinds of tuple data structures: Data Cubes, $n$-dimensional matrices, and the special case of $n$-tuples, triples, as illustrated in Figure 3.1.

As we saw in the previous chapters, special properties of our data induce special requirements for the storage, ad-hoc algorithms, and analytics in comparative genomics. In this chapter we will discuss these properties of the genomic data, formalize them, and show how they correspond to our solutions.

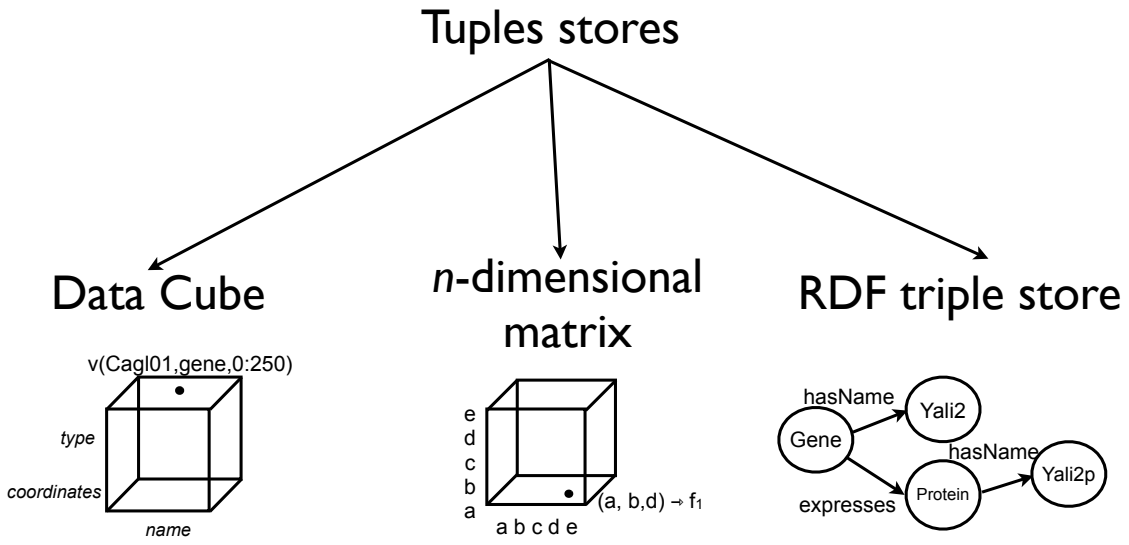



Figure 3.1: Different types of tuple data structures: Data Cube, $n$-dimensional matrix and RDF triple Store. A Data Cube's dimensions are the attributes of data elements. The dimensions of the $n$-dimensional matrices are the data elements. RDF triples are data entities composed of *subject–predicate–object*.

## 3.1 Cassandra representations of genomic data

As we saw in subsection 2.3.3, NoSQL provides an appropriate technology for storing our huge volumes of data. Beyond simply storing the genomic elements, we must store the $n$-ary relations between them, which define a high-dimensional sparse matrix. Following our formalism for representing relations in subsection 3.3.1 we store in NoSQL database only the non-empty values.

The main choice to make for the data representation in the Cassandra data store is to decide what are the row keys. This is important, because the data are distributed by the row keys, the row key is a main filter to get the data, the row values that are associated with these row keys are grouped for the map function of MapReduce. Thus, **a Cassandra**

**row key is an identifier of the group of row values**. Consequently, the way how we use row values grouping defines the grouping itself.

As we saw in subsection 2.3.4, a NoSQL Cassandra schema should be designed around the queries and analyses that will be performed on the system. For each use case typically we need to create a column family. Data Cubes (section 3.2) and $n$-dimensional matrices (section 3.3) can both be used different ways: for searching small subsets of the data and for global MapReduce analyses. If the data are used for searching, we create an index (Cassandra column family) for each combination of parameters for the search.

If the data are used for a global MapReduce analysis, we group necessary values into Cassandra rows for the map phase of MapReduce. Column families for global MapReduce analyses are not the same as the ones that are used for searches by attributes. In contrast with the searching indexes, for the MapReduce analyses it is important only to group all needed values into the row, because Cassandra rows (or slices of the rows) are inputs for map functions of MapReduce. And we do not have to care about how the values are placed in the row, what are the row keys, super column names and column names. Consequently, for the same data we might need two different column families: for search and for MapReduce.

For RDF triples we will use a simple Cassandra schema to store them using row key, super column name and column name for subject, predicate and object (section 3.4).

## 3.2 Data cube

The most common type of data is simply a set of data elements with attributes. The data are represented as Data Cubes (Figure 3.2) where the dimensions are those attributes that are used for searching. The values of this cube are typically either serialized objects or identifiers of the objects. The dimensions are discrete-valued; values are not the same for each dimension. The values of these dimensions are used as sources for sorting and searching criteria.

In Relational databases each data object typically corresponds to a row in a table. The attributes (dimensions of the data cube) are the columns of this table.

In NoSQL data stores we store serialized objects and use the attributes to create indexes that refer to identifiers of the objects. We create separate index (column families) for each combination of attribute names that is used for searching or sorting.

### 3.2.1 Data Cube representation

First, we will formalize what are actually the genomic data elements. These data are semi-structured, because not all genomic elements and relations between them have the same attributes. There are many kinds of genomic elements. There can be a wide range of genome sequence features: genes, proteins, mRNA, exons, etc. Also, there are composite elements such as groups of features: protein families, alleles, tandem repeats, etc; and groups of groups: fusion and fission events, etc. Thus, genomic data elements can be represented as objects that have arbitrary attributes and may contain other objects.

A common strategy for the semi-structured data elements is to keep them serialized and make the indexes for the attributes that are used for search or sorting. This way we do not need to create a uniform object representation (the union of the object interfaces) that suit all kinds of elements; such a representation would be very inefficient for the small features.

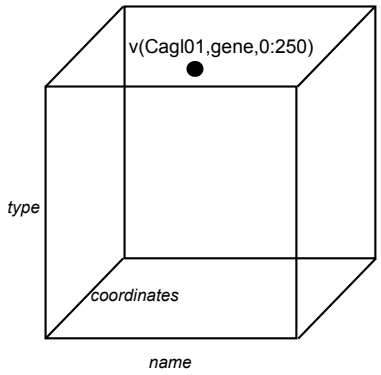



Figure 3.2: Data cube: each dimension corresponds to some attribute name, the values on this dimension are the possible attribute values for this attribute name. The values in the Data Cube cells are serialized objects or identifiers for the objects. In this example there are 3 dimensions (indexes): name, type, coordinates.

**Definition 3.1.** ***(Genomic Data Element)***
*Suppose the set* Attributes *contains the attribute names that will be used to create search or sorting indexes. We represent* Genomic Data Element $d \in D$ *as a combination of serialized object* serialized_object *and a set of attribute-values pairs:*
$d = \langle serialized\_object,$
$attribute\_name_1 : \{attribute\_value_1^1, \ldots, attribute\_value_1^k\},$
$\ldots$
$attribute\_name_n : \{attribute\_value_n^1, \ldots, attribute\_value_n^m\}$
$\rangle,$

$$for\ 1 \leq i \leq n,\ attribute\_name_i \in Attributes,$$

$$\{atribute\_name_1, \ldots atribute\_name_n\} \subseteq Attributes.$$

Such data elements, with several attributes that can be used for searching and sorting operations. Data Cube is simply a set of data elements. The dimensions of the cube are attributes, and the values are identifiers, single values or serialized objects.

**Definition 3.2.** ***(Genomic Data Cube)***
*The set of Data Elements D, defined in the definition 3.1, is a representation of the Data Cube as a function, where the range of the function is the set of serialized objects and the domains are attributes.*

A *Data Cube* is a multidimensional cube, each dimension of which corresponds to some attribute name $a_i \in Attributes$, and the values of each $a_i$-dimension are all possible attribute values for this attribute name. The cell values in this Data Cube are serialized objects or identifiers for the objects.

### 3.2.2 Data Cube Cassandra representation

We implemented a Cassandra schema of column families derived from a use-case analysis of the Génolevures data resource in [SMN+09]. This follows standard practice in NoSQL databases, where column family schemas are defined based on an analysis of a fixed set of queries known in advance [SAD+10]. This schema is used as a structured index of serialized

objects representing genomic elements: genes, proteins, exons, mRNA, etc. A further schema of column families is used for grouping features in genomic intervals. Significantly, we also use Cassandra to store the $n$-ary relations between these elements, such as protein families, alleles, tandem repeats, etc.

Column families for the search tools use row key, super and standard column names as places for the search parameters. To improve performance of the key and row caches we try not to create large rows by dividing them using some criteria.

**Example 3.1.** ***(Cassandra searching index for the Data Cube)***
*In this example we would like to store a collection of genomic sequence features and we would like to be able to search them by some of their attributes: name and type. Thus, we need to create an index for these attributes and the rest of information about the sequence features we can store serialized.*

*Suppose, the Data Cube D (a set of data elements) is following (illustrated in the figure 3.3):*

$$D = \{(name = \text{“}SAKL1\text{”}, type = \text{“}gene\text{”}) \rightarrow object_1$$
$$(name = \text{“}ERGO5\text{”}, type = \text{“}exon\text{”}) \rightarrow object_2$$
$$(name = \text{“}CAGL8\text{”}, type = \text{“}gene\text{”}) \rightarrow object_3$$
$$(name = \text{“}YALI7\text{”}, type = \text{“}intron\text{”}) \rightarrow object_4\}.$$

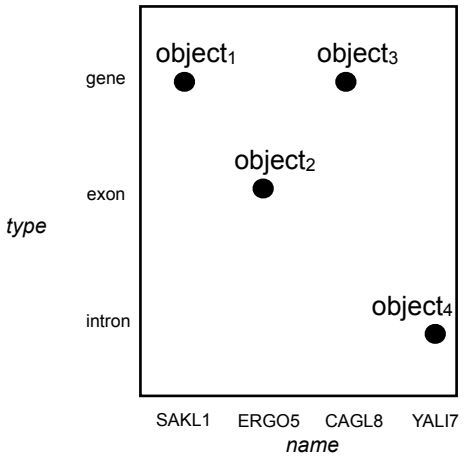



Figure 3.3: 2-dimensional Data cube in the example 3.1. The first dimension is name of the sequence feature, the second dimension is its type. In the example the sequence features have only one name and one type, but multiple values would not change the representation.

*This is a 2-dimensional Data Cube that is used for a search by name and type, and suppose that the name is an obligatory parameter and a type is optional. Since the name is an obligatory parameter, we use it as a Cassandra row key, while a type can be a column name (the general principles are formulated in the further subsection 3.6.1). Thus, we create a column family NameType, as it is illustrated in the figure 3.4.*

As we discussed earlier in this chapter, sometimes the data representation (grouping) for MapReduce analyses might be different from the representation that would be appropriate for searches by attributes, but in the case of Data Cube there is no difference.

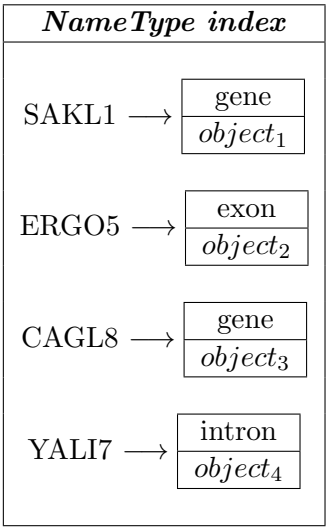



Figure 3.4: A Cassandra column family as an index for the search by name and type for example 3.1. The sequence feature names are the Cassandra row keys, because they are obligatory parameters. Optional parameter (type) is stored in column name.

**Example 3.2.** ***(Storing Data Cube in Cassandra for MapReduce analyses)*** *The Data Cube from our definition is a set of data elements that can be considered independent. If we need to perform some global analysis on these data we do not need a special schema nor a special grouping for the elements, since the data elements are independent and can be easily processed in parallel.*

## 3.3 $n$-dimensional matrices

When the data are not just set of independent elements but the set of relations between them, another representation of the data are used. $n+1$ data tuples ($n$-tuple of relation between elements and an attribute of this relation) are represented as function from $n$ dimensions (domain) to 1 dimension (range). This function corresponds to a $n$ dimensional matrix, as it is illustrated in Figure 3.5.

For Relational databases, for these kind of tuples we create a new table that store relations between objects, which are combinations of the foreign keys.

In NoSQL world we need as well to create a new column family that stores the relation.

### 3.3.1 $n$-dimensional matrix representation

In the case of Comparative Genomics, a data set is not simply a collection of genomic elements, the main focus is on the $n$-ary relations between these elements. But, more precisely, they are not actually $n$-ary relations, they are $n$-ary mappings: each $n$-tuple has some additional value, the attributes of the relation. Before talking about data representations, let us formalize some basic concepts that we will use further: what we mean by $n$-ary relation, what are the multidimensional functions and arrays that often are used to represent genomic data.

As we claimed before, our focus is on $n$-ary mappings from $n$-tuples of genomic elements to some additional values (attributes of relation). It can be one or several attributes of

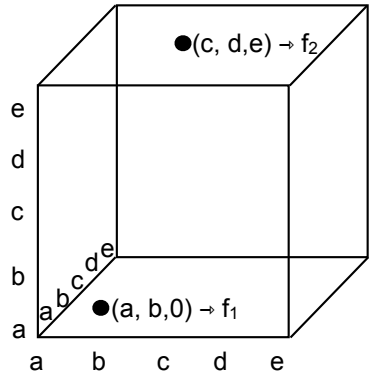



Figure 3.5: An $n$-dimensional matrix, where $n+1$ tuples are represented as a $n$-function. The range of this function consists of the attributes of the relations between data elements.

the relation: relation identifier, relation name, some score value, etc. For simplicity we will consider attributes of the relation as one *attribute object*, and each relation has a unique one (we can make it unique by assigning an unique identifier). Thus, we have a multi-dimensional $n$-function whose values are unique.

The very important difference between genomic elements and attributes is that, while we observe a dramatic increase in the number of genomic elements, the size on the disk of attributes objects is not growing (the number of attributes for each $n$-tuple is not increasing with the number of tuples). That is why we can think about attributes objects as a fixed-size composite value over the *Attributes* set and we do not have to worry about scaling properties of these attributes.

**Definition 3.3.** ***(Comparative Genomic data as a relation)***
*For sets of genomic elements (Data Cubes) $D_1, ...D_n$ and a set of $n$-tuples over these elements, we have for each $n$-tuple one attribute object from the set of possible attribute objects Attributes. We represent this data as a subset of the Cartesian product, written $R \subseteq D_1 \times \cdots \times D_n \times Attributes$. Note that we do not store tuples with empty attribute objects, thus most often $|R| \ll |D_1 \times \cdots \times D_n \times Attributes|$.*

The $n$-dimensional matrices in comparative genomics are sparse - there are almost everywhere empty or zero values in the cells. Instead of storing the full matrix, we can store more efficiently only non-zero, "interesting" values.

**Definition 3.4.** ***(Sparse Matrix)***
*A matrix $R : R \subseteq D_1 \times \cdots \times D_n \times Attributes$ is* sparse *if it is populated almost everywhere with zeros: $|R| \ll |D_1 \times \cdots \times D_n \times Attributes|$.*

**Example 3.3.** ***(Protein similarity sparse matrix as a relation)***
*Suppose, we have a set of proteins. Each protein is a specific sequence features, which has many attributes. Often, while considering relations between data elements we do not need all information about them, thus we can use only their name or identifiers.*

*Thus, in this example we represent the set of proteins as a set of their names: $P = \{SACE4, SACE3, CAGL5, ZYRO8, DEHA2\}$. Suppose, we have symmetric similarity matrix $M_{Sim}$ between these proteins (for example, as a result of an all-against-all alignment):*

$$M_{Sim} = \begin{array}{c} \\ SACE4 \\ SACE3 \\ CAGL5 \\ ZYRO8 \\ DEHA2 \end{array} \begin{array}{c} \begin{array}{ccccc} SACE4 & SACE3 & CAGL5 & ZYRO8 & DEHA2 \end{array} \\ \begin{pmatrix} 1 & 0 & 0.1 & 0 & 0.5 \\ 0 & 1 & 0 & 0 & 0.6 \\ 0.1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0.7 \\ 0.5 & 0.6 & 0 & 0.7 & 1 \end{pmatrix} \end{array}.$$

*We represent this similarity matrix as a* 3*-ary relation* $R_{Sim}$*, a set of* 3*-tuples. We do not store empty or null values. This matrix here is symmetric, so we neither store duplicated tuples nor self tuples.*

$$R_{Sim} = \{(SACE4, CAGL5, 0.1),$$
$$(SACE4, DEHA2, 0.5),$$
$$(SACE3, DEHA2, 0.6),$$
$$(ZYRO8, DEHA2, 0.7)\}.$$

**Example 3.4.** ***(Protein families as a relation)***
*Suppose, for the set of proteins* $P = \{SACE4, SACE3, CAGL5, ZYRO8, DEHA2\}$*, we have a partition into families: SACE4 and SACE3 belong to the family GL3C1 and* $CAGL5, ZYRO8, DEHA2$ *to GL3C2. We represent these protein family partitions as:*

$$R_{Fam} = \{(SACE4, SACE3, GL3C1), (CAGL5, ZYRO8, DEHA2, GL3C2)\}.$$

*To calculate dimensions for a family partition we need to find the largest family. Its size + 1 is the number of dimensions of this relation. In our example* $R_{Fam}$ *is a* 4*-ary relation.*

**Definition 3.5.** ***(Comparative Genomic data as a function)***
*For sets of genomic elements (Data Cubes)* $D_1, ...D_n$ *and the set of* $n$*-tuples over these elements we have, for each* $n$*-tuple one attribute object from the set of possible attribute objects Attributes. We represent this data as a* $n$*-ary function*

$$F : D_1 \times \cdots \times D_n \rightarrow Attributes.$$

**Example 3.5.** ***(Similarity matrix as a function)***
*Suppose, for the set of proteins* $P = \{SACE4, SACE3, CAGL5, ZYRO8, DEHA2\}$*, we have a similarity matrix* $M_{Sim}$ *between these proteins (for example, as a result of an all-against-all alignment) and the matrix actually directly corresponds to a function*

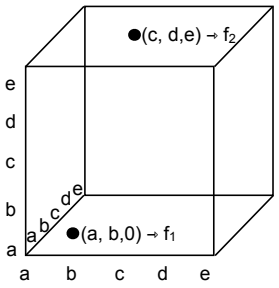
$$F_{Sim} : P \times P \rightarrow Attributes :$$

$$F_{Sim} = M_{Sim} = \begin{array}{c} \\ SACE4 \\ SACE3 \\ CAGL5 \\ ZYRO8 \\ DEHA2 \end{array} \begin{array}{c} \begin{array}{ccccc} SACE4 & SACE3 & CAGL5 & ZYRO8 & DEHA2 \end{array} \\ \begin{pmatrix} 1 & 0 & 0.1 & 0 & 0.5 \\ 0 & 1 & 0 & 0 & 0.6 \\ 0.1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0.7 \\ 0.5 & 0.6 & 0 & 0.7 & 1 \end{pmatrix} \end{array}.$$

*Attributes here are the similarity scores.*

**Example 3.6.** ***(Protein families partition as a function)***
*Suppose, for the set of proteins* $P = \{SACE4, SACE3, CAGL5, ZYRO8, DEHA2\}$*, we have a partition into families: SACE4 and SACE3 belong to the family GL3C1 and CAGL5, ZYRO8, DEHA2 to GL3C2. We represent these families as a* 3*-ary function* $F_{Sim} : P \times P \times P \to Attributes$*.*

$F_{Fam} =$




$=$

$$\{((SACE4, SACE3, -) \to GL3C1),$$
$$(CAGL5, ZYRO8, DEHA2) \to GL3C2))\}.$$

*Attributes here are the protein family names and other attributes.*

Sometimes it is useful to think about the data as a multidimensional cube, but in practice the tools and algorithms work with 1 or 2 dimensional functions. That is why multidimensional functions are represented as one- or two-dimensional functions and we call this *tuple decomposition*.

**Definition 3.6.** ***Tuple decomposition into one-dimensional function***
*For the sets of genomic elements (Data Cubes)* $D_1, ...D_n$*, the set of possible relation attribute objects Attributes and an n-ary function* $F : D_1 \times \cdots \times D_n \to Attributes$*, we perform* tuple decomposition *by defining a one-dimensional function* $F' : D_1 \cup \cdots \cup D_n \to Attributes$*, where for each n-tuple and corresponding attribute object* $a : F(d_1, \ldots d_n) = a$ *we have* $F'(d_1) = a, \ldots, F'(d_n) = a$*.*

**Example 3.7.** ***(Tuple decomposition for protein families - 1 dimension)***
*For the protein families defined by the function* $F_{Fam}$ *in example 3.6 we define a new function* $F'$ *:*

$$F'_{Fam} = \begin{array}{c} \\ SACE4 \\ SACE3 \\ CAGL5 \\ ZYRO8 \\ DEHA2 \end{array} \begin{array}{c} \begin{array}{cc} GL3C1 & GL3C2 \end{array} \\ \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \end{array} =$$

$$= \{(SACE4 \to GL3C1),$$
$$(SACE3 \to GL3C1),$$
$$(CAGL5 \to GL3C2),$$
$$(ZYRO8 \to GL3C2),$$
$$(DEHA2 \to GL3C2)\}.$$

Sometimes in the application we need to know if two genomic elements belong to the same group. For this case it is more useful to perform tuple decomposition into a two-dimensional function.

**Definition 3.7.** ***Tuple decomposition into two-dimensional function***
*For the sets of genomic elements (Data Cubes) $D_1, ..., D_n$, the set of possible attribute objects Attributes and a n-ary function*

$$F : D_1 \times \cdots \times D_n \to Attributes$$

*we perform* tuple decomposition *by defining a two-dimensional function*

$$F'' : (D_1 \times D_2) \cup (D_1 \times D_3) \cdots \cup (D_{n-1} \times D_n) \to Attributes,$$

*where for each n-tuple and corresponding attribute object $a : F(d_1, \dots d_n) = a$, we have*

$$F''(d_1, d_2) = a, \dots, F''(d_{n-1}, d_n) = a.$$

**Example 3.8.** ***(Tuple decomposition for protein families - 2 dimensions)***
*For the protein families defined by the function $F_{Fam}$ in example 3.6 we define a new function $F''$ :*

$$F''_{Fam} = \begin{array}{c} \\ SACE4 \\ SACE3 \\ CAGL5 \\ ZYRO8 \\ DEHA2 \end{array} \begin{array}{c} \begin{array}{ccccc} SACE4 & SACE3 & CAGL5 & ZYRO8 & DEHA2 \end{array} \\ \begin{pmatrix} GL3C1 & GL3C1 & 0 & 0 & 0 \\ GL3C1 & GL3C1 & 0 & 0 & 0 \\ 0 & 0 & GL3C2 & GL3C2 & GL3C2 \\ 0 & 0 & GL3C2 & GL3C2 & GL3C2 \\ 0 & 0 & GL3C2 & GL3C2 & GL3C2 \end{pmatrix} \end{array} =$$

$$= \{(SACE4, SACE3 \to GL3C1),$$
$$(CAGL5, ZYRO8 \to GL3C2),$$
$$(CAGL5, DEHA2 \to GL3C2),$$
$$(ZYRO8, DEHA2 \to GL3C2)\}.$$

### 3.3.2 $n$-dimensional matrix Cassandra representation

The same way as for Data Cubes, we implemented a Cassandra schema of column families derived from a use-case analysis of the Génolevures [SMN+09]. Unlike in the case of Data Cubes, here we do have two different use cases of using $n$-dimensional matrices. The first one is to search by attributes or relation members of small subsets of relations. The second use case is to perform global MapReduce analyses. The representation for MapReduce analyses varies according to the algorithms.

**Example 3.9.** ***(Cassandra searching index for the $n$-dimensional matrices (functions) )***
*Suppose, for the function $F$ we performed tuples decomposition into the one-dimensional function:*

$$F'_{Fam} = \begin{array}{c} \\ SACE4 \\ SACE3 \\ CAGL5 \\ ZYRO8 \\ DEHA2 \end{array} \begin{array}{c} \begin{array}{cc} GL3C1 & GL3C2 \end{array} \\ \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \end{array}$$

$$= \{(SACE4 \to GL3C1),$$

$$(SACE3 \rightarrow GL3C1),$$

$$(CAGL5 \rightarrow GL3C2),$$

$$(ZYRO8 \rightarrow GL3C2),$$

$$(DEHA2 \rightarrow GL3C2)\}.$$

For the search applications we store this function $F'_{Fam}$ as a relation index to be able to find all members by relation name and backward – by any member find relation name (Figure 3.6).

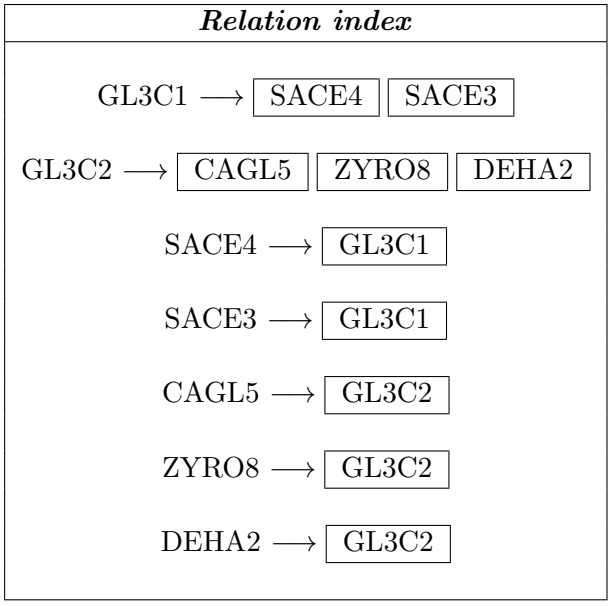



Figure 3.6: A Cassandra column family for $n$-dimensional matrices/functions as an index for the searches by relation name and member names (example 3.1). For the searches by relation name we have the first and the second rows in this figure, where the row keys are relation names, and column names are the names of the relation members. We do not show here column values, which are empty. For the backward searches by relation member names, we have the rows (from 3rd to 7th in this figure), where the row keys are member names.

As we noticed above, for attribute searches we represent genomic relations as a one-dimensional function, because we can treat each of them independently. On the other hand, for global algorithms we need to group relations, typically by members. Thus, we represent genomic relations as a two-dimensional function. Consequently, it leads to data duplication.

**Example 3.10.** ***(Storing protein families in Cassandra for MapReduce analyses)***
*Suppose, for the function $F$ we perform tuple decomposition into a two-dimensional func-*

*tion:*

$$F''_{Fam} = \begin{array}{c} \\ SACE4 \\ SACE3 \\ CAGL5 \\ ZYRO8 \\ DEHA2 \end{array} \begin{array}{c} \begin{array}{ccccc} SACE4 & SACE3 & CAGL5 & ZYRO8 & DEHA2 \end{array} \\ \begin{pmatrix} GL3C1 & GL3C1 & 0 & 0 & 0 \\ GL3C1 & GL3C1 & 0 & 0 & 0 \\ 0 & 0 & GL3C2 & GL3C2 & GL3C2 \\ 0 & 0 & GL3C2 & GL3C2 & GL3C2 \\ 0 & 0 & GL3C2 & GL3C2 & GL3C2 \end{pmatrix} \end{array} =$$

$$= \{(SACE4, SACE3 \rightarrow GL3C1),$$
$$(CAGL5, ZYRO8 \rightarrow GL3C2),$$
$$(CAGL5, DEHA2 \rightarrow GL3C2),$$
$$(ZYRO8, DEHA2 \rightarrow GL3C2)\}.$$

*For the MapReduce we group relations by some criterion. Typically, we group them by members: for each element which will be the row key we put into the row as column values all other elements that are in relation with it (Figure 3.7). Note, that here we might need to store duplicated values, because the same tuple will be used in several independent map jobs.*

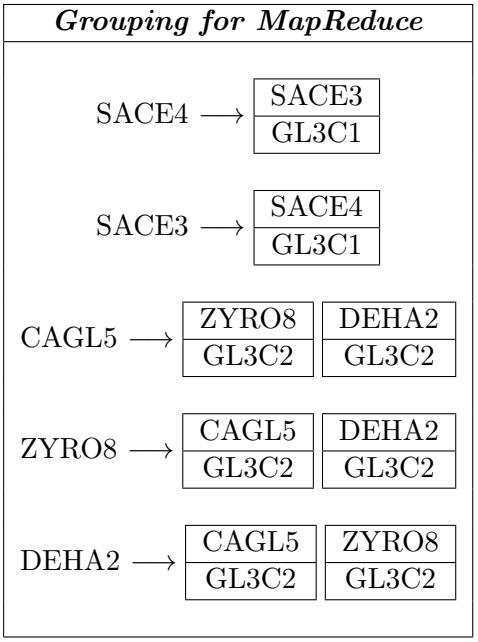



Figure 3.7: A Cassandra column family for MapReduce analyses on $n$-dimensional matrices (example 3.10). Here we group relations by their members, consequently duplicating them. After a tuple decomposition of $n$-dimensional matrix, we have a set of 2-tuples associated with their tuple names. Each 2-tuple we store 2 times: the first member as a row key with the second member as a column name; and contrariwise. This way tuples are grouped by their members automatically. The relation names are the Cassandra column values.

**Example 3.11.** ***(Storing 2-dimensional similarity matrices in Cassandra for MapReduce analyses)***
*To store 2-dimensional similarity matrices we do not need to perform tuple decompositions:*

*the similarity matrix is a set of 2-tuples with an attribute object associated with each tuple. For MapReduce we group the elements by the first participant in the tuple, thus in the row we have a projection from the sparse matrix into the first dimension. The same column family can be used for the search applications: we can retrieve tuples by only the first participant (row key) or both first and second participants (row key + column name).*

*So, for the similarity Matrix $M_{Sim}$:*

$$F_{Sim} = M_{Sim} = \begin{array}{c} \\ SACE4 \\ SACE3 \\ CAGL5 \\ ZYRO8 \\ DEHA2 \end{array} \begin{array}{c} \begin{array}{ccccc} SACE4 & SACE3 & CAGL5 & ZYRO8 & DEHA2 \end{array} \\ \begin{pmatrix} 1 & 0 & 0.1 & 0 & 0.5 \\ 0 & 1 & 0 & 0 & 0.6 \\ 0.1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0.7 \\ 0.5 & 0.6 & 0 & 0.7 & 1 \end{pmatrix} \end{array}.$$

*we create a column family that indexes this matrix using non-empty cells (Figure 3.8).*

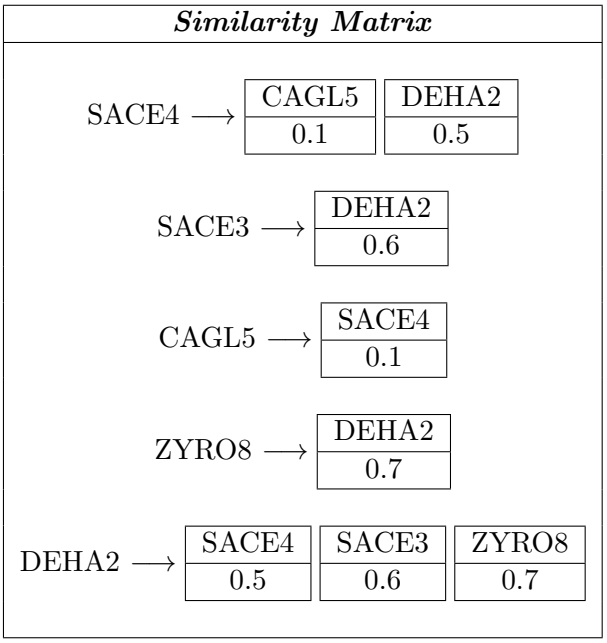



Figure 3.8: A Cassandra column family for MapReduce analyses and search applications of similarity Matrices. Each 2-tuple we store 2 times: the first member as a row key with the second member as a column name; and contrariwise. This way tuples are grouped by their members automatically. Similarity score values are the Cassandra column values.

## 3.4 Triple store (RDF)

The Resource Description Framework (RDF)[2] is a standard metadata data model. RDF operates with triples. A triple is a data entity composed of *subject–predicate–object*. This kind of representation is appropriate for logical inference and knowledge mining. RDF is a collection of statements or facts, each fact is a triple (subject, predicate, object). We

[2]RDF: http://www.w3.org/RDF

can represent RDF database as a directed graph (called RDF graph), where the vertices are subjects and objects and the edges are predicates.

The RDF data model is widely used in bioinformatics [LPL+11]. For example, Bio2RDF[3] [BNT+08] project is an attempt to convert publicly available biological datasets to a standard web RDF format for the knowledge integration.

SPARQL is a SQL-like RDF query language for retrieving and manipulating data stored in RDF format. SPARQL queries may consist triple patterns, conjunctions, disjunctions, and optional patterns.

In [LPL+11] there is a simple example of a SPARQL query for finding a chemical compound with the name "Tryptophan Synthetase" (illustrated in FIgure 3.9):

```
SELECT ?x WHERE {
  ?x <hasName> "Tryptophan Synthase";
  ?x <hassubstrate> "Chemical"}
```

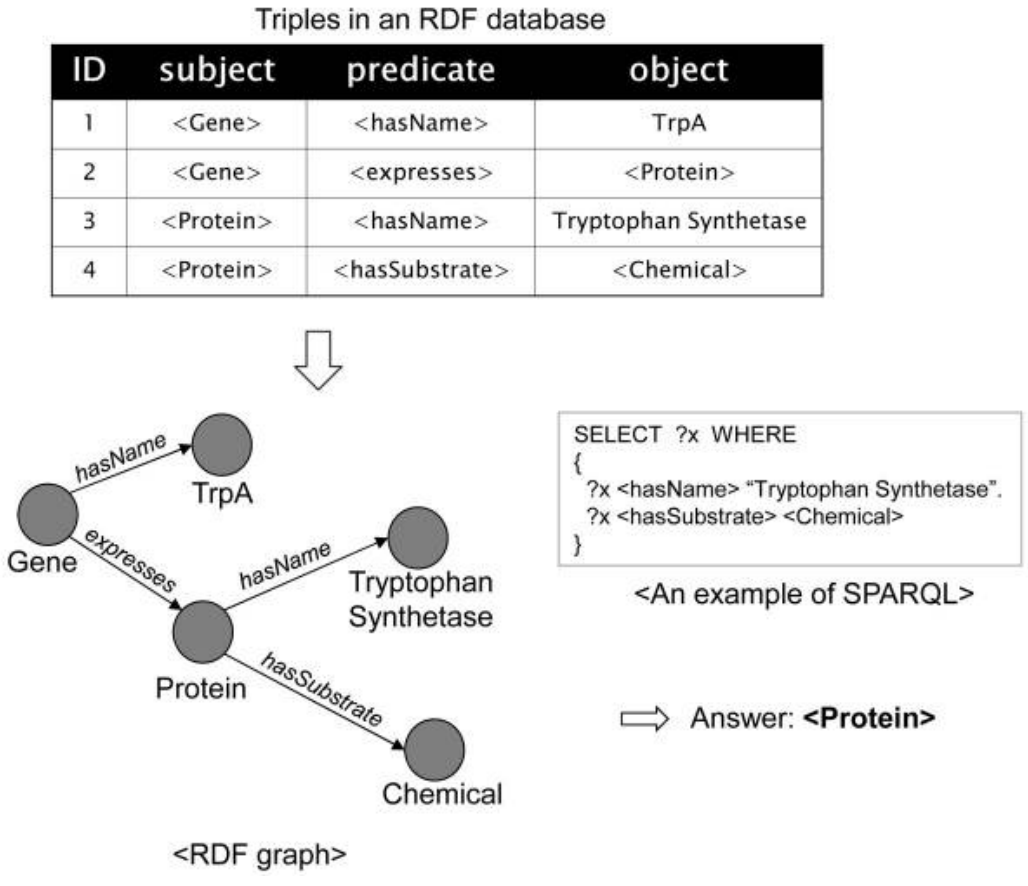



Figure 3.9: An example of an RDF and a graph data representation and SPARQL query (figure is taken from [LPL+11])

[3]Bio2RDF: http://bio2rdf.org

### 3.4.1 Cassandra RDF representation

There are many implementations of the triple stores, for example: Sesame[4], AllegroGraph[5], Virtuoso[6], etc.

The RDF storage adaptors for Cassandra data store, such as RDF.rb[7], use a very straightforward data model to store RDF triples in Cassandra. They map RDF subject to a row key, RDF predicate to a Cassandra super column name, RDF object to a column name (figure 3.10).

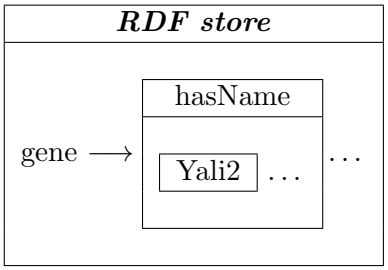



Figure 3.10: Storing RDF triples in Cassandra

This representation of RDF triples allows us to perform a wide range of get queries by one or several subjects, range queries on predicates and objects. At the same time in the get query the key must always be given, thus, we should always provide a subject in our queries. If we want to operate on data without subjects we must store a backward index, simply swapping subject and object.

## 3.5 Characterizing standard usage patterns

Different usage patterns by end users impose different loads on the data storage backend. An efficient design for data storage must take these patterns into account. The Génolevures web site [SMN+09] is a highly-accessed service that provides data and tools for comparative genomics of biotechnologically interesting yeasts, that historically is designed around an in-depth use case analysis of the typical queries made by biological researchers. Using anonymized logs of public access to the web site, and knowledge of the implementation of the software service, we were able to associate each of these typical uses with different calls to the data access API. According to ClusterMaps and Google Analytics the Génolevures database has about 37,000 visits and 19,000 unique visitors per year (excluding search engines) [MSD11]. We analyzed $10^5$ lines of HTTP access logs. The URL queries in the anonymized HTTP logs of the Génolevures website correspond to high-level user queries. The *Generic Model Organism Database* project[8] defines a community of software tools, including the Generic Genome Browser[9], often based on the BioPerl Perl libraries.[S+02, SMS+02] We instrumented the GMOD libraries used in the backend of the Génolevures web site, in particular Bio::DB::SeqFeature::Store, the subclass of Bio::DB::SeqFeature that implements database storage for sequence features, and the Bio::Graphics::Browser

[4]Sesame: http://www.openrdf.org
[5]AllegroGraph : http://www.franz.com/agraph/allegrograph
[6]Virtuoso: http://virtuoso.openlinksw.com
[7]RDF.rb: http://rdf.rubyforge.org/cassandra
[8]GMOD: http://gmod.org
[9]Generic Genome Browser: http://gmod.org/wiki/GBrowse

class that are used for the Genome Browser. By modifying the classes that Implement the object-relational bridge between SQL databases and the Génolevures-specific derived classes of the BioPerl object model, we were able to capture all of the low-level queries through the BioSQL database schema.

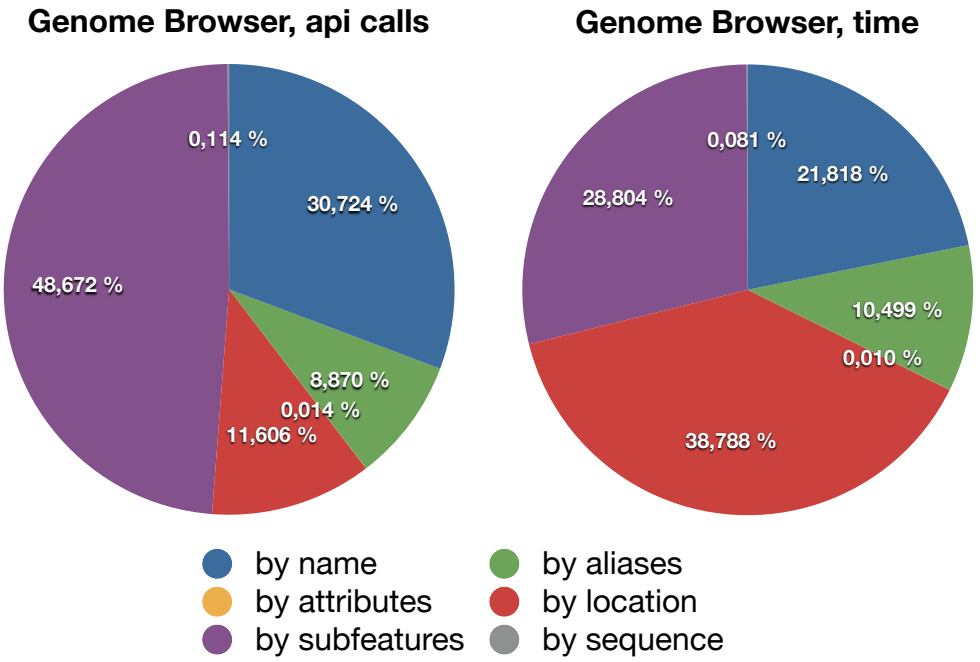



Figure 3.11: Low-level API calls derived from log analysis of user queries, with their relative number and contribution to the total time budget of the data storage system.

Figure 3.11 summarizes the kinds of the different low-level data access calls that were made, and their relative contribution to the total load on the system. In this figure we can see that there are many searches by name, but they are quite fast. Obviously, the queries by name that allow aliases ("by aliases") are a more expensive than simple queries by name. In contrast, searches by location (geometric queries that extract overlapping genome intervals) are rare, but very expensive in terms of time. Most of the API calls in this system concern fetching subfeatures, because in the Bio::DB::SeqFeature::Store data storage implementation subfeatures are not attached to the parent features and are loaded on demand. Also, there are relatively very few queries by specific attributes and queries to retrieve a sequence.

## 3.6 Deriving column families for Cassandra

### 3.6.1 General principles

In Cassandra we have two types of cache we may use: row caches and key caches. The key cache contains the location of row keys in memory. The row cache keeps the entire row

in memory. Both caches work efficiently only with rows that are not too large (from our experience the rows that have more than 1000 columns can be considered as large). The data in Cassandra are distributed across the network using a row key, so we need to choose keys for which the corresponding rows (a list of columns) would be not large. Composite keys can reduce the size of rows by splitting its columns into new rows identified by the original key and the values of the column names; one can think of this as combinations of dimensions. By splitting large rows into many smaller ones, we improve the distribution of data across computers.

From our usage experience and stress tests for performance tuning we derive the following principles of creating column families, using the list of user queries types:

1. All obligatory query parameters, that are not part of range queries, are included in composite row keys.

2. If a query parameter is not always given, but has very few possible values, this parameter can be included in a composite row key and we will use a Multiple Keys Get operation to retrieve data with all possible values for this parameter.

3. If a parameter is optional, we use it as column name or super column name.

### 3.6.2 Cassandra column families for storing sequence features and group relations

The first usage pattern is the Génolevures Genome Browser Search tool. This tool uses the BioPerl (Bio::DB::SeqFeature::Store) search API. For each of the API queries we make a column family or super column family in Cassandra data store. We use the same strategy as Bio::DB::SeqFeatures::Store classes [S$^+$02, SMS$^+$02]: Cassandra column families are used as searchable indexes to identify serialized objects that can be retrieved from the "SerializedObjects" column family by primary id. Our column families are presented in Figure 3.12.

The *Attributes* column family in Figures 3.13 and 3.14 defines a mapping between attribute names-values and ids of features. It is used for search by an arbitrary attribute, including search by name and search by alias as a subcase. We use attribute name and attribute value as a composite row key, and Type (an optional parameter) as a super column name. The *Keywords* column family is for full-text search by annotation keywords. This column family is a mapping between the words in an annotation and a feature id. The structure of the Keywords column family is similar to the Attributes column family: composite keys of attribute name (that correspond to annotations) and the keywords of the attribute value sentences.

For the particular case of geometric queries we have the *Location* column family (Figures 3.15 and 3.16). This kind of query always has a given sequence id and most often the feature type, and, optionally, may indicate an interval on the sequence to search in or overlap with. Following the general principles for this column family we use a composite key of sequence_id and feature type. There are relatively few different feature types encountered in our data (gene, protein, mRNA, etc.) and we keep all possible types in the Metadata column family. That is why when for location query the feature type is not given, we can use Multiple Keys Get queries for all possible types.

As we describe in subsection 2.3.4, a Cassandra datastore always keeps its columns and super columns sorted (by column name). In the Slice Range query we can specify a set of column names or a column range interval, although we can use only one column or

| ***Column Family*** | ***Column Family Type*** | ***Usage*** | ***Figures*** |
|---|---|---|---|
| Attributes | Super | Search for sequence features by an arbitrary attribute (including name and alias) | schema: 3.13 or A.1<br>example: 3.14 |
| Keywords | Super | Text search for sequence features by annotation keywords | schema: A.2 |
| Subfeatures | Standard | Retrieve subfeatures, if they are not attached to a parent sequence feature | schema: A.3 |
| Sequence | Standard | Get sequence of a sequence feature (a string of letters a, c, g, t) | schema: A.4 |
| Location | Super | Search for sequence features in a given sequence and a given interval of coordinates | schema: 3.15 or A.5<br>example: 3.16 |
| Relations | Super | 1) Get relation id by a member<br>2) Get all members of the given relation (id)<br>3) Get attributes of a relation (id) | schema: 3.18 or A.6<br>example: 3.19 |
| Relation Attributes | Super | Get relation by various relation attributes | schema: A.7 |
| Metadata | Standard | Get meta information about all types and ids of sequences and sequence features in our database | schema: A.8 |
| Serialized Objects | Standard | Get serialized objects (sequences and sequence features) by given primary id | schema: A.9 |

Figure 3.12: Column families in the Apache Cassandra schema, derived from API analysis of BioPerl and BioSQL libraries. Each search use case in BioPerl and BioSQL API corresponds to a Cassandra column family, which is actually a search index.

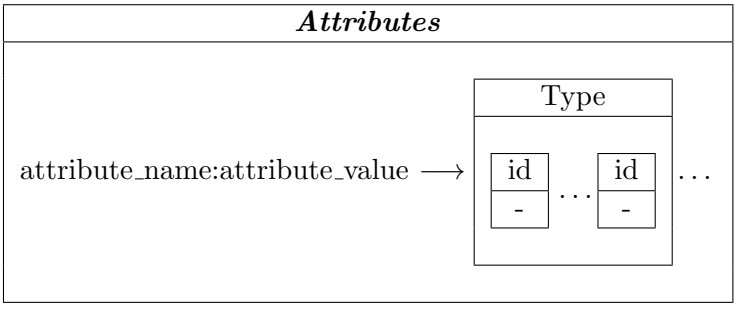



Figure 3.13: The Attributes column family is used for search by an arbitrary attribute. Attribute name and attribute value are the obligatory parameters, thus we combine them into a composed row key. Type is an optional parameter, thus we put it into Super Column Name.

super column range per query. Since in our cluster we use the Random Partitioner to have automatic load balancing (see 2.3.4) we can not use the Key Slice Range query.

Our geometric queries may specify (optionally) the interval on the sequence, where we must search our sequence features. Depending on the situation, the interesting features

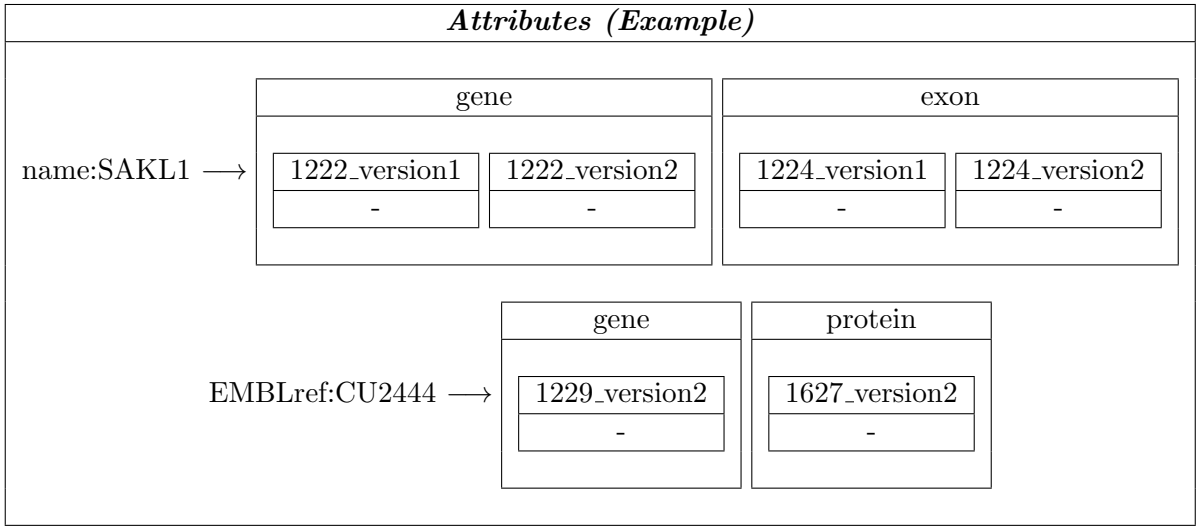



Figure 3.14: An example of data in Attributes column family. The combinations of the attribute names and attribute values are the row keys. Sequence feature types (gene, exon, intron, etc) are the super column names. Several sequence features might have the same attribute key-value pairs, so we store unique identifiers of sequence features as column names.

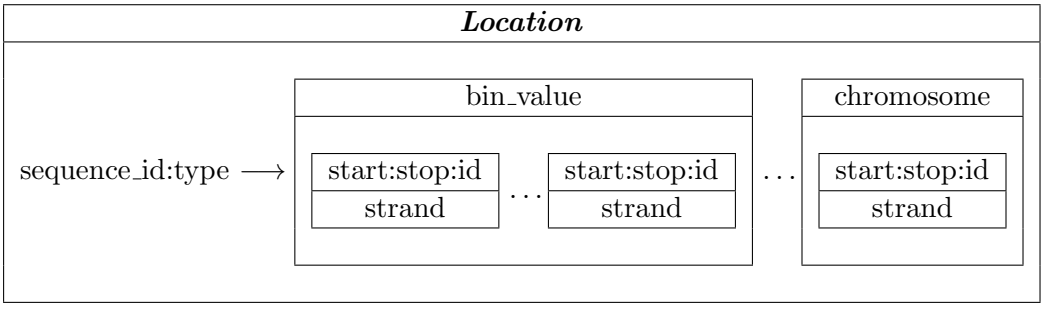



Figure 3.15: A Location column family is used for interval (geometric) queries. Sequence id and sequence feature type are two obligatory parameters, thus they are stored in row keys. We divided each sequence into pieces of the same lengths (bins). We store sequence feature ids in these bins. Coordinates are composed into column names for column slice queries. We also add sequence feature ids into column names to make them unique. Since strand is rarely explicitly used as a search parameter, it is stored in a column value.

may be either included in the given interval, or just overlapping with it. For the two coordinates (start and end) the one range interval is not enough. The typical solution for this problem is to use a binning technique. We cut our sequence into the bins of the same, empirically optimal, size, and put feature to all bins that intersect it. A special case is chromosome and other very large features - we put them only once, because they intersect with all or too many bins. By specifying a range of bins we can perform a search by overlapping intervals. The optimal bin size for medium size features is 10000 as it is used by Bio::DB::SeqFeature::Store database adaptor implementations. This size is reasonable for our data, since the average feature size is 550 and the average feature density in the bin is 47 (see Figure 3.17).

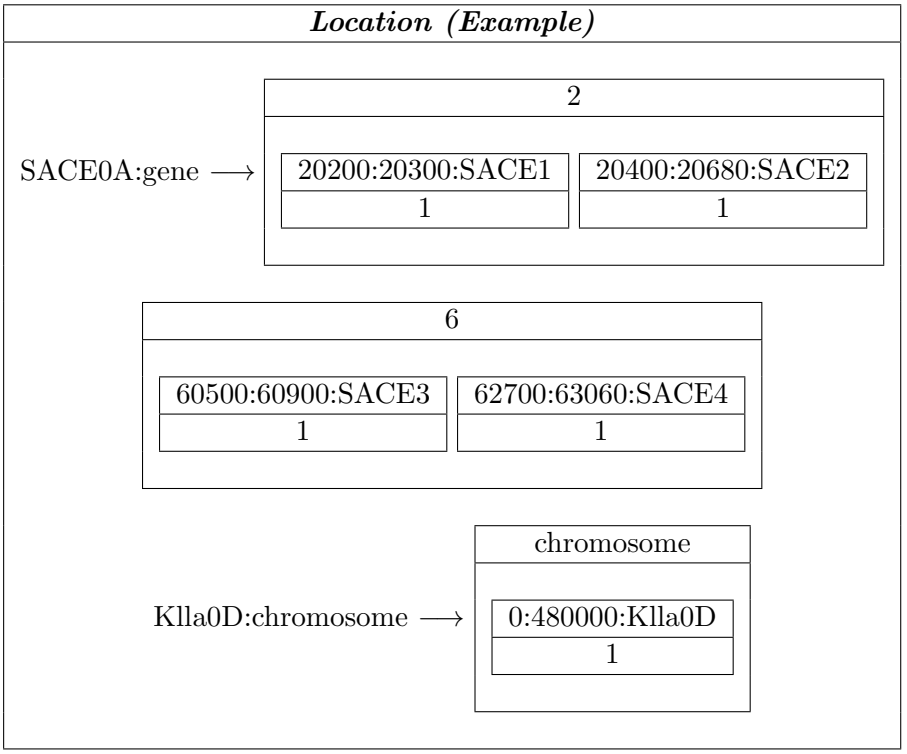



Figure 3.16: Location column family example. Here sequence SACE0A has 4 genes: SACE1, SACE2, SACE3 and SACE4. SACE1 and SACE2 belongs to interval [20000,29999], so they are stored in the bin=2. Similarly SACE3 and SACE4 are stored in the bin=6. In this example KLLA0D do not have sequence features (genes, proteins, ...) except its chromosome, which is not divided into pieces because of its size, so we store it separately.

The second usage pattern is the Génolevures Concordance Advanced Search. This advanced search system works with the Magus data model, which is richer than BioPerl SeqFeature: it consists of protein families and other kind of groups. The data are traditionally stored in PostgreSQL using BioSQL data storage schema.

To support different kind of relations needed for genomic data we create a *Relations* column family (Figures 3.18 and 3.19) for searches by relation id or member id the way we discussed in section 3.3. The first mapping is from relation_id, which is unique, to all relation attributes and members. The second one is backward: from member_id and relation_type to relation_ids.

The complete schema of Cassandra column families is explained in Appendix (chapter 6.2.3). This schema was designed using analyses of BioPerl API and Génolevures query logs. Using typical user queries (extracted from the use cases) we performed continuous stress tests to improve performance of our system. As a consequence of this performance tuning, we derived some general principles of creating Cassandra column families, discussed in subsection 3.6.1. From our usage experience we know that it is better to avoid too large rows (thousands of columns), so designing column families for concrete tasks and concrete data required measuring data distribution and performance.

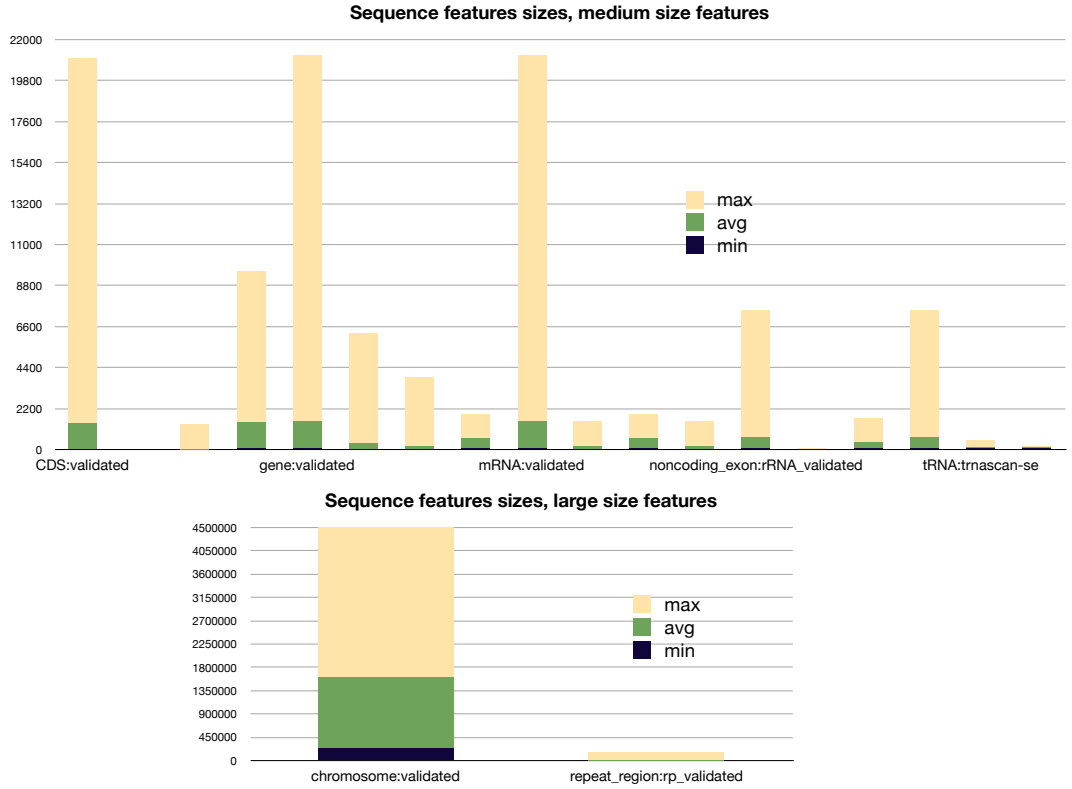



Figure 3.17: Distribution of sequence feature sizes, for medium and large size features

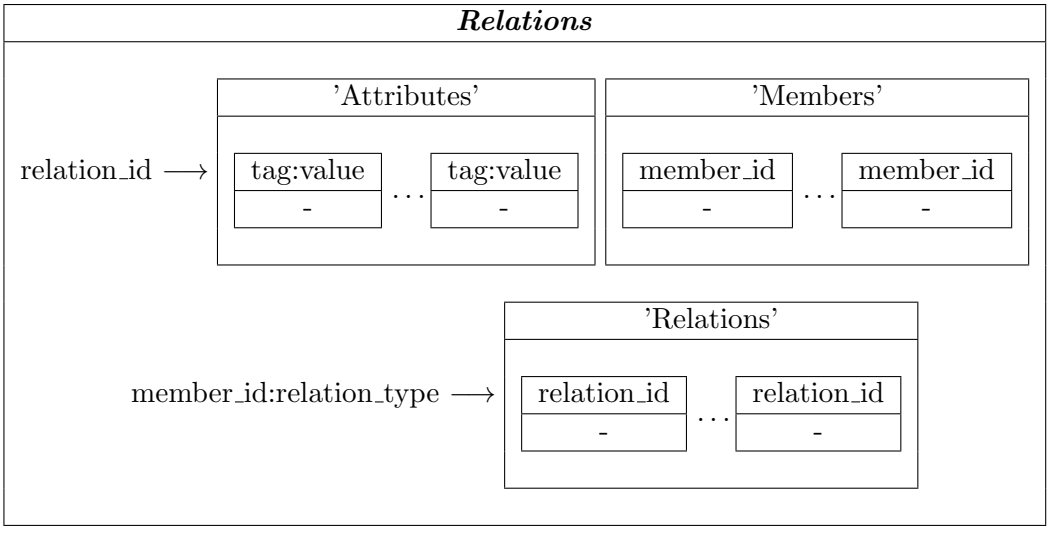



Figure 3.18: Relations column family can be used for searches by relation id as well as by member id. Each genomic element might participate in many different kinds of relations, so we reduce row sizes by including relation type into row key.

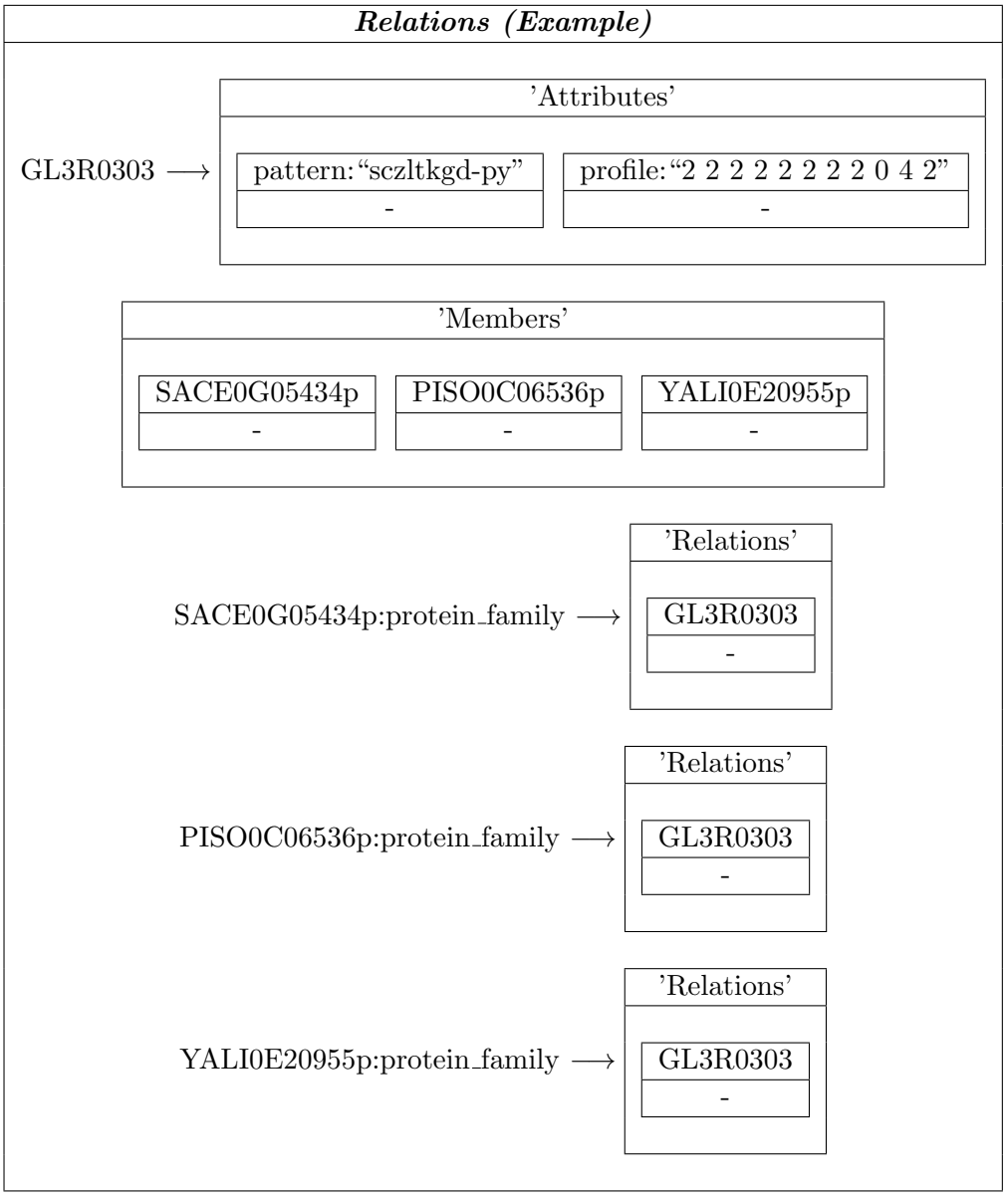



Figure 3.19: Example of Relations column family. In this example, in the first row we store an index for searches by relation id, to retrieve its members and attributes. The last three row store the mapping between member id SACE0G05434p, PISO0C06536p and YALI0E20955p to the relation id GL3R0303.

# Chapter 4

# Genomic Global Analyses

***Résumé :*** *Les analyses en génomique comparée font partie de ce que les biologistes appellent des* méthodes globales*, dans le sens où les réponses recherchées exigent un raisonnement sur l'ensemble d'éléments dans les génomes analysés. Dans ce chapitre nous discutons d'analyses globales en génomique, de emploient le plus souvent des comparaisons tout-contre-tout dont la complexité croît de façon génométrique en le nombre de gènes. Nous considérerons en profondeur deux exemples d'analyses globales existantes que nous adapterons au paradigme de calcul distribué MapReduce. Pour mieux évaluer la pertinence de nos travaux, nous introduisons un critère de "bonne décomposition" en map et reduce, basé sur des notions d'équilibre de charge et de bonne mise à l'échelle. Nous montrons que nos deux nouvels algorithmes adaptés remplissent ce critère.*

Analyses in comparative genomics fall into the category that biologists call *global methods*, in the sense that the desired answers require reasoning over all genes in the genomes that are being analyzed. In this chapter we will talk about these global analyses, which typically involve all-against-all comparisons that scale geometrically with the number of input genes. We will consider in depth two examples of existing global analyses in genomics and will show how they can be adapted using MapReduce distributed computations paradigm. We will introduce a criterion of a good MapReduce decomposition, based on notions of load balancing and good scaling. Finally, we will show how our new adaptations of the two algorithms meet this criterion.

## 4.1 Adapting algorithms to MapReduce: criterion

In this section we consider the situation where we have an existing algorithm for global analysis of $n$-ary gene relations that we need to convert to the distributed MapReduce paradigm. The fact that analyses work with relations is very important because there is no challenge in doing distributed computations on independent data elements. On the other hand, $n$-ary relation data need specific groupings in order to scale well.

The common thread in the approaches from the literature, described in sections 2.4.5 and 2.4.6, hinges on a clever choice of values for the map step, and seeks to partition the data into pieces that fit into one machine. In this thesis we are concerned with sparse matrices of $n$-ary relations and can see that we have the same concerns. Previously, in chapter 3 we considered the different ways of representing the data for the search applications and MapReduce algorithms. The way that the data elements are grouped into input Cassandra rows is defined by the needs of the map functions of the concrete

MapReduce algorithm. The same is true for intermediate results: the output of map phase and grouping for reduce functions is directly defined by the algorithm. Both for map and reduce we group data elements by carefully choosing the keys.

For each existing global analysis we might adapt MapReduce model many ways. Formally we can define a general criterion for the *good* MapReduce algorithms that deal with relations.

**Definition 4.1.** ***(Criterion of a good MapReduce algorithm)***

1. Balanced loads (for map and for reduce):
   ***Data distributed among the similar chunks of optimal size both for map and reduce phases.(Figure 4.1)***

   *In computing, load balancing is a methodology to distribute tasks among the processors to achieve optimal resource usage. MapReduce implements static load balancing which is used for the cases when computer nodes are identical and the tasks resources usages are similar. These assumptions allows the static load balancer to not take into account the current network load.*

   *That is why it is our responsibility to split the data into the equal (or similar) sized chunks both for map and reduce phases. MapReduce tutorial [map] advises 10-100 maps per-node and*

   "0.95 *or* $1.75 \times (\langle no.\ of\ nodes \rangle \times\ mapreduce.tasktracker.reduce.tasks.maximum)$"

   *reduces, noticing that usually number of maps is defined by the total number of inputs and increasing number of reduces increases load balancing, but also increases framework overhead.*

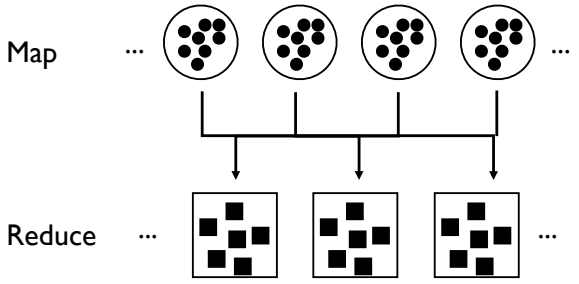



Figure 4.1: Load Balancing for MapReduce. The small black circles and squares here represent pieces of data (key-value pairs). These key-values are not the same and are grouped differently for map and reduce phases; the key-values of reduce phase (small black squares) are results of map computations. Big circles represent map jobs, big squares - reduce jobs. Load Balancing: data chunks for map phase (and for reduce) have the same (maybe different between map and reduce) optimal size.

2. Good scaling:
   ***With the increase of data the number of chunks increases and the size of chunks does not significantly grow(Figure 4.2)***

   *We choose the size of the chunks for map and reduce phase considering the resource usage of computations on these chunks. We should also take into account that many*

*algorithms need to put the data into memory, in this case data should fit into memory for efficient computations.*

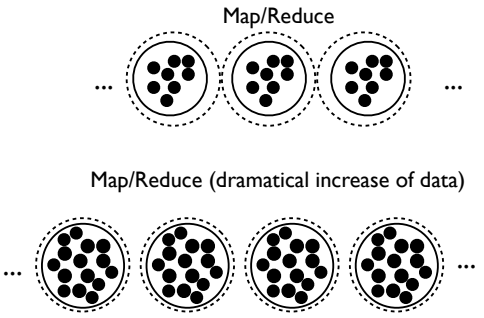



Figure 4.2: Scaling for MapReduce. The small black circles here represent pieces of data (key-value pairs) in map (or reduce) phase. Big circles represent map (or reduce) jobs. Scaling: if the number of data increases, the sizes of the data chunks for the map and the reduce phases do not exceed the maximum chunk size when the number of chunks grows.

## 4.2 Application 1: Identification of gene fusion and fission events

One consequence of genome remodeling in evolution is the modification of genes, either by fusion with other genes, or by fission into several parts. By tracking the mathematical relations between groups of similar genes, rather than between individual genes, we can paint a global picture of remodeling across many species simultaneously. The method [DNS08] is appropriate for highly redundant eukaryote genomes. Applying this method to a set of fungal genomes, the work in this paper: first, confirmed that the number of fusion/fission events is correlated with genome size, second, that the fusion to fission ratio favors fusions, third, that the set of events is not saturated, and fourth, that while genes assembled in a fusion tend to have the same biochemical function, there appears to be little bias for the functions that are involved. Furthermore, fusion and fission events are landmarks of random remodeling, independent of mutation rate, and thus define a metric of "recombination distance".

The method in [DNS08], shown schematically in Figure 4.3, builds proteins of the same species into paralogous groups based on their similarity. Then it combines paralogous groups of proteins into a single **HMM** (Hidden Markov Model). These HMMs are systematically aligned using *hhsearch* [Söd05] and filtered to obtain a graph of relations between genome segments. This graph is then mined for subgraph motifs, each containing at least two collinear segments aligned to non-collinear segments in another genome. Each of these motifs is an *event*, corresponding to a fusion, a fission, or some combination of the two. The authors of [DNS08] define an *Event* "as an $n$-ary relation between $P$-groups, at least one composite $P$-group (hereafter named $C$-group) and at least two element $P$-groups (named $E$-groups), which fulfill three constraints: the $E$-groups belong to the same proteome, they align on the $C$-group, the alignment regions have no or reduced overlap on the $C$-group". Fusion events correspond to the case where a single $C$-group is connected to several $E$-groups of more than two species. In the case of fission events multiple $C$-groups are connected to several $E$-groups of a single species.

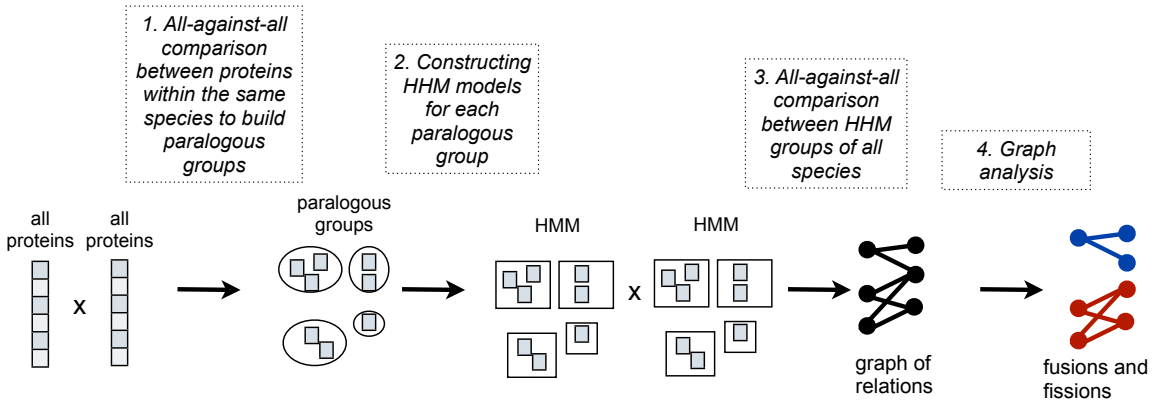



Figure 4.3: Schematic representation of the original method [DNS08] of identifying fusion and fission events. In the first step we systematically align proteins within the same species. Then we build paralogous groups of proteins to reduce the number of relations. These groups then are organized in Hidden Markov Models (HHM) and systematically aligned. The last step of the algorithm is Graph analysis to detect fusion and fission events.

Because the same events can be observed in different subsets branches of the phylogenetic tree of the species, they can be placed in history using a parsimony argument.

### 4.2.1 Adaptation to MapReduce

The bottleneck of the original algorithm [DNS08] is the graph analysis (the last step in Figure 4.3), as illustrated in Figure 4.4. Graph analyses requires random access to the nodes and edges. Large graphs cannot fit into memory, and the access to disk is thousands times slower than access to memory. This makes it very expensive to process large graphs. To reduce the graph size, the authors of [DNS08] proposed firstly to extract connected components of the graph, and then process them independently. But even this decomposition strategy does not completely solve the scaling problem – the connected components are still too large.

The input data for this part is a list of relations between composite "$C$-groups" (HMMs with 2 or more aligned segments) and elementary "$E$-groups" (HMMs with 1 aligned segment). To express the graph analysis using the MapReduce paradigm we use the knowledge that the graph is bipartite, so to find all events we can decompose the graph into sub-graphs with longest path 2. For each $E$-group that can participate in an event we calculate (in parallel, using grouping by $C$-groups, and $E$-groups) all nodes that are reachable using a path of length 1 or 2.

This MapReduce algorithm has many redundant operations, since each event is extracted independantly from several subgraphs and reprocessed. But, in exchange, we have decomposed the large problem into many small computations that can be performed asynchronously and in parallel.

### 4.2.2 Formalism: maximum coincident component

One aspect of applying MapReduce paradigm to existing algorithms is to think about the parts of these algorithms as functions. As a first step in defining these functions we need to

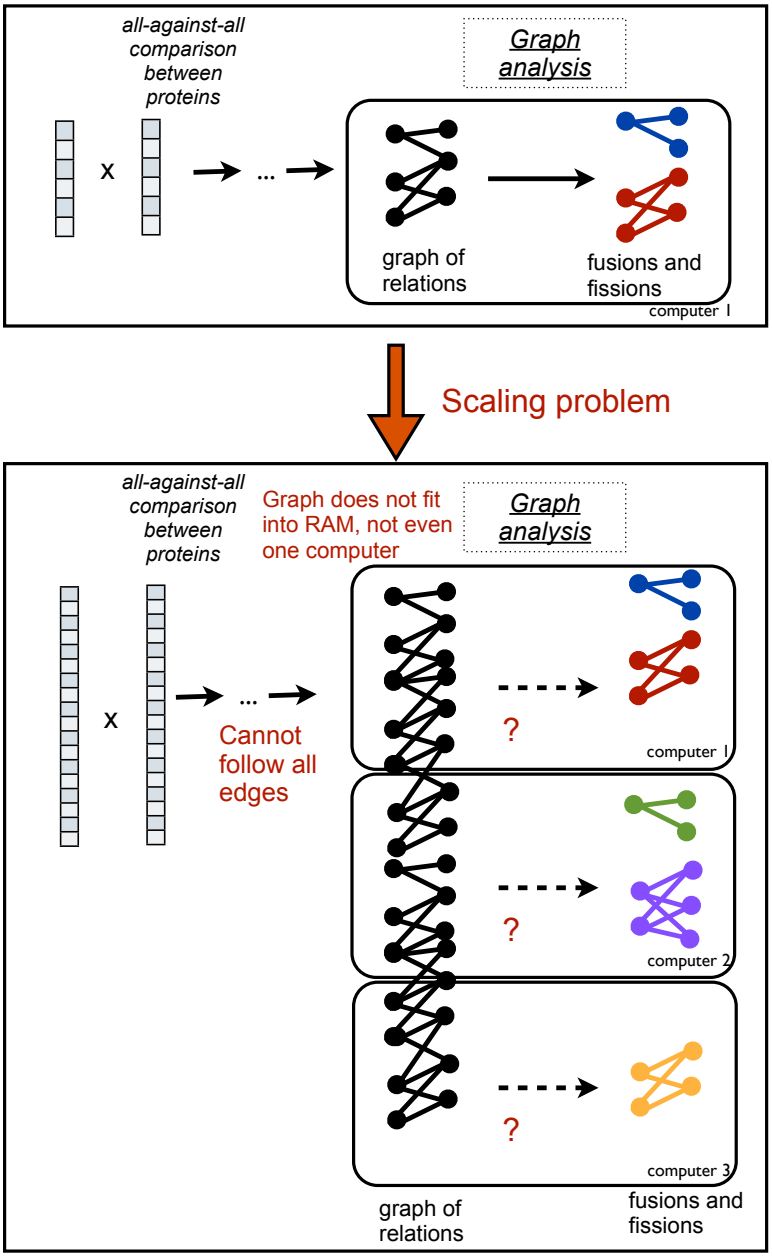



Figure 4.4: Scaling problem in the graph analysis phase of the detecting fusions and fissions algorithm [DNS08]. With extreme growth of data volume, the graph does not fit into one RAM, not event to one computer anymore. With distributed computations now we have the problem of an access to the edges, and, consequently, the problem of synchronization between local computations.

formalize different aspects of the original algorithms, which was only described abstractly in [DNS08].

### General definitions

Letr $C$ be a set of $C$-groups, and $E$ a set of $E$-groups, $R$ is a relation between $E$-groups and $C$-groups, *species* is function on $E \cup C$, that associates each $E$-group with some

*species-value*. For sets of $C$-groups, define its *$E$-group intersection* (signature *sig*) to be the intersection of the names of the $E$-groups in relations with those $C$-groups. Define a *partial event* to be a set of relations for which the *sig* of its $C$-groups is nonempty and for which each participating *species-values* of $E$-groups occurs at least twice. (An atomic event is a partial event with exactly one c-group.)

**Definition 4.2.** ***(Signature) - Figure 4.2.2***
*For given sets $E$, $C$, $C_1 \subseteq C$ and relation $R \subseteq E \times C$, the* signature $sig : 2^{\{C\}} \to 2^{\{E\}}$

$$sig(C_1) \quad = \quad \{e \in E \mid \forall c \in C_1, \exists (e, c) \in R \}.$$

(Notation) For simplicity, we write $sig(c)$ for $sig(\{c\})$ for $c \in C$.

**Observation 4.1.** ***(Intersection of signatures)***
*The intersection of the $C$-groups signatures is equal to the signature of the $C$-groups union.*

$$sig(C_1) = \bigcap_{c \in C_1} sig(c)$$

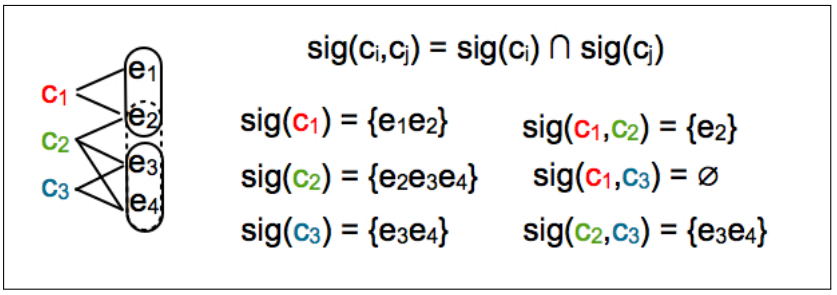



Figure 4.5: Example of c-group signatures. The signature of the union of $C$-groups is equal to the intersection of their signatures.

There is an additional condition for the $E$-groups in events: the species of the $E$-groups that participate in fusion and fission events must appear at least in pairs. We define property *pair_species* that should be true for the $E$-groups of events.

**Definition 4.3.** ***(Pair_species property)***
*For a subset $E_1 \subseteq E$ define the property* pair_species *as:*

$$pair\_species : 2^{\{E\}} \to \mathbb{B}$$

$$pair\_species(E_1) = \forall e \in E_1 \; \exists \tilde{e} \in E_1, \; \tilde{e} \neq e, \; species(e) = species(\tilde{e})$$

**Definition 4.4.** ***(E- and C- projections) - Figure 4.6***
*For $R_1 \neq \emptyset \subseteq R$, define* C-projection $R_1|_C$ *and* E-projection $R_1|_E$ *as*

$$R_1|_C = \{c \mid (e, c) \in R_1\}, \; R_1|_E = \{e \mid (e, c) \in R_1\}$$

**Definition 4.5.** ***(Partial event)***
*For $R_1 \neq \emptyset \subseteq R$, $R_1$ is a partial event (p_event) iff*

$$[sig(R_1|_C) = R_1|_E \; and \; pair\_species(R_1|_E)]$$

*or, equivalently, $R_1$ is a partial event (p_event) iff*

$$R_1 = R_1|_E \times R_1|_C \; and \; pair\_species(R_1|_E)$$

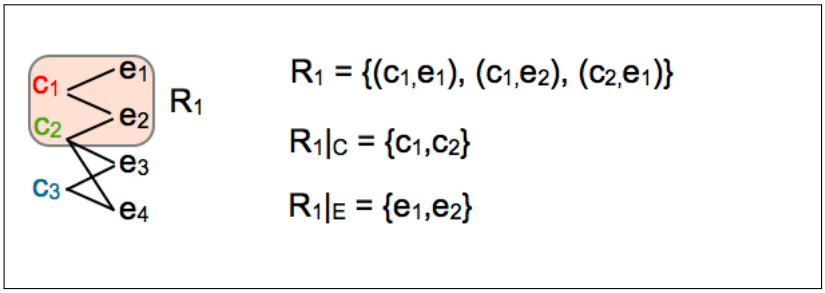



Figure 4.6: Example of $C$- and $E$-projections of relation $R_1 \subseteq R$. $C$-projection of relation $R_1$ is a set of $C$-groups that participate in it. Similarly, $E$-projection of $R_1$ is a set of $E$-groups that participate in $R_1$.

**Definition 4.6.** ***(Event, Events) - Figure 4.7***
Event *is a maximal partial event: it is not possible to add e- or c-group to this partial_event while preserving partial event properties.*
Events *are the union of all events.*

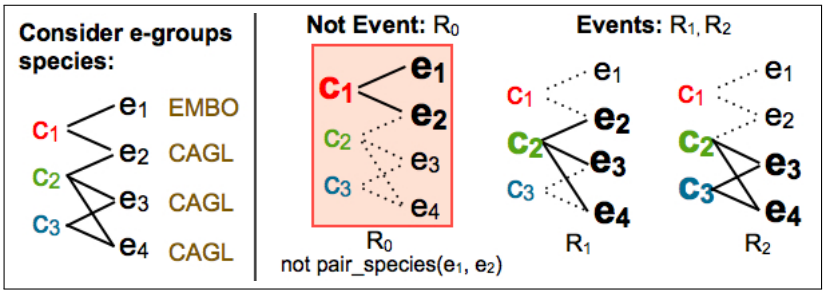



Figure 4.7: Events Examples. In the left part of the figure there is the initial graph of relations between $C$- and $E$-groups. To illustrate pair species property we showed species of $E$ groups (EMBO and CAGL).
In this example $R_0 = \{(c_1, e_1), (c_1, e_2)\}$ is not an event because the species of $\{e_1, e_2\}$ are not in pairs: $species\{e_1, e_2\} = \{EMBO, CAGL\}$.
$R_1 = \{(c_2, e_2), (c_2, e_3), (c_2, e_4)\}$ is an event, because this is a maximal partial event with pair species property: $species\{e_2, e_3, e_4\} = \{EMBO, EMBO, EMBO\}$.
The second event is $R_2 = \{(c_2, e_3), (c_2, e_4), (c_3, e_3), (c_3, e_4)\}$, where $species\{e_3, e_4\} = \{EMBO, EMBO\}$.

**Definition 4.7.** ***(Species count)***
*For given sets $E$ and $C$, relation $R \subseteq E \times C$ and subset $R_1 \neq \emptyset \subseteq R$, let us count the number of different species-values for $C$-groups and $E$-groups as species_count* : $R_1|_E \cup R_1|_C \to \mathbb{N}$

$$species\_count(R_1|_C) = |\{species(c) \mid c \in R_1|_C\}|$$

$$species\_count(R_1|_E) = |\{species(e) \mid c \in R_1|_E\}|$$

**Definition 4.8.** ***(Event classification) - Figure 4.8***
*For given sets $E$ and $C$, relation $R \subseteq E \times C$, function species* : $E \cup C \to Species$ *and*

*event* $R_1 \neq \emptyset \subseteq R$

$$\textit{event is} \begin{cases} \textit{fusion} & \textit{if } species\_count(R_1|_C) = 1 \textit{ and } species\_count(R_1|_E) > 1 \\ \textit{fission} & \textit{if } species\_count(R_1|_C) > 1 \textit{ and } species\_count(R_1|_E) = 1 \\ \textit{multiple} & \textit{if } species\_count(R_1|_C) > 1 \textit{ and } species\_count(R_1|_E) > 1 \\ \textit{undecidable} & \textit{if } species\_count(R_1|_C) = 1 \textit{ and } species\_count(R_1|_E) = 1 \end{cases}$$

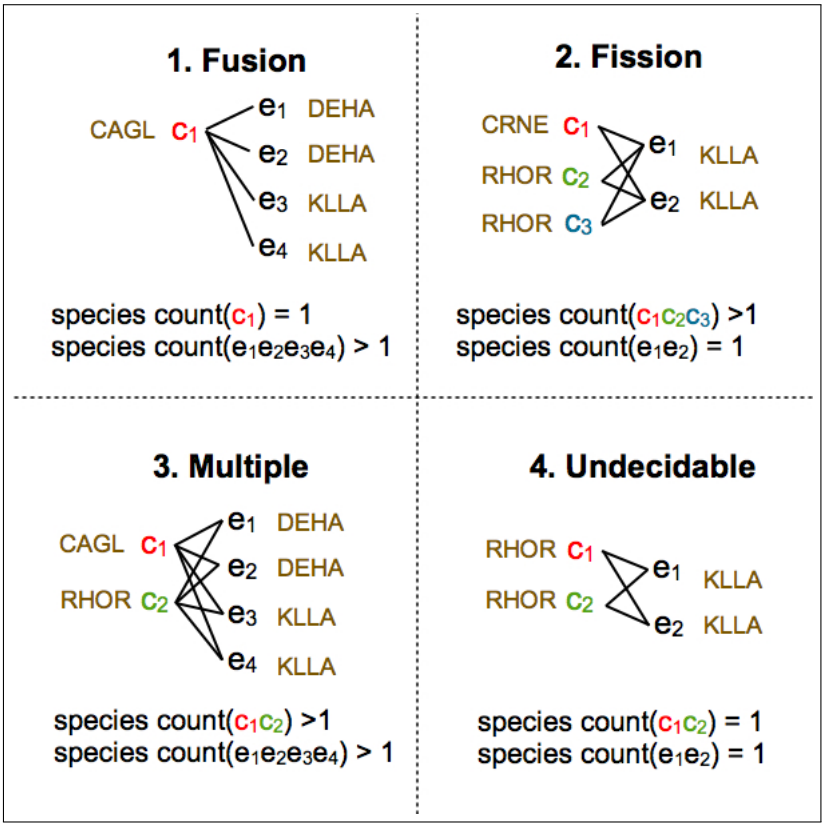



Figure 4.8: Events Classification: fusions, fissions, multiple and undecidable.
1. Fusion events correspond to the case where a single $C$-group is connected to several $E$-groups of more than two species.
2. In the case of fission events multiple $C$-groups are connected to several $E$-groups of a single species. 3. Multiple events have multiple species of $C$-groups as well as multiple species of $E$-groups. 4. Undecidable events have single species of $C$- and $E$-groups.

### Definitions for MapReduce adaptation

Now we will define specific sets for the MapReduce adaptation: a subset of $E$-groups *Cosig* and a set of $C$-groups *Resig*. Then we will prove, that the the pairs of these subsets are events.

**Definition 4.9.** ***(Cosig) - Figure 4.9***
*For given sets $E$ and $C$, relation $R \subseteq E \times C$, define function* cosig *as:*
*$cosig : E \times C \times C \to 2^{\{E\}}$, that for $e \in E$ and $c_i, c_j \in C$,*
*where $(e, c_i) \in R$ and $(e, c_j) \in R$:*

$$cosig(e, c_i, c_j) = \{e' \in sig(c_i, c_j) \mid pair\_species(cosig(e, c_i, c_j)), e \in cosig(e, c_i, c_j)\}$$

*Thus, $cosig(e, c_i, c_j)$ is a maximum subset of $sig(c_i, c_j)$ for which the pair_species property is true and $e$ is part of it.*

Basically, *cosig* is a the part of signature *sig* of the input $C$-groups $c_i, c_j$ that contains the input $E$-group $e$ and for which the additional property *pair_species* is true.

**Definition 4.10.** ***(Resig)- Figure 4.9***
*For given sets $E$ and $C$, relation $R \subseteq E \times C$, define function* resig *as:*
*$resig : E \times C \times C \rightarrow 2^{\{C\}}$, if for some $e$ and $c_i$, $c_j$ $\exists$ $cosig(e, c_i, c_j)$ and $cosig(e, c_i, c_j) \neq \emptyset$:*

$$resig(e, c_i, c_j) = \{c_k \in C \mid \exists c_k \in C, c_k \neq c_i, c_k \neq c_j : cosig(e, c_i, c_j) \subset sig(c_k)\} \cup \{c_i \in C\} \cup \{c_j \in C\}$$

In other words, *resig* is a maximum set of $C$-groups where each $C$-group is connected to each $E$-group of *cosig*.

**Observation 4.2.** ***(Cosig and Resig)***
*By definition, if for some $e \in E$ and $c_i, c_j \in C$ $\exists$ $cosig(e, c_i, c_j) \neq \emptyset$ then $\exists$ $resig(e, c_i, c_j)$. Resig $\neq \emptyset$ since $c_i, c_j \in resig(e, c_i, c_j)$.*

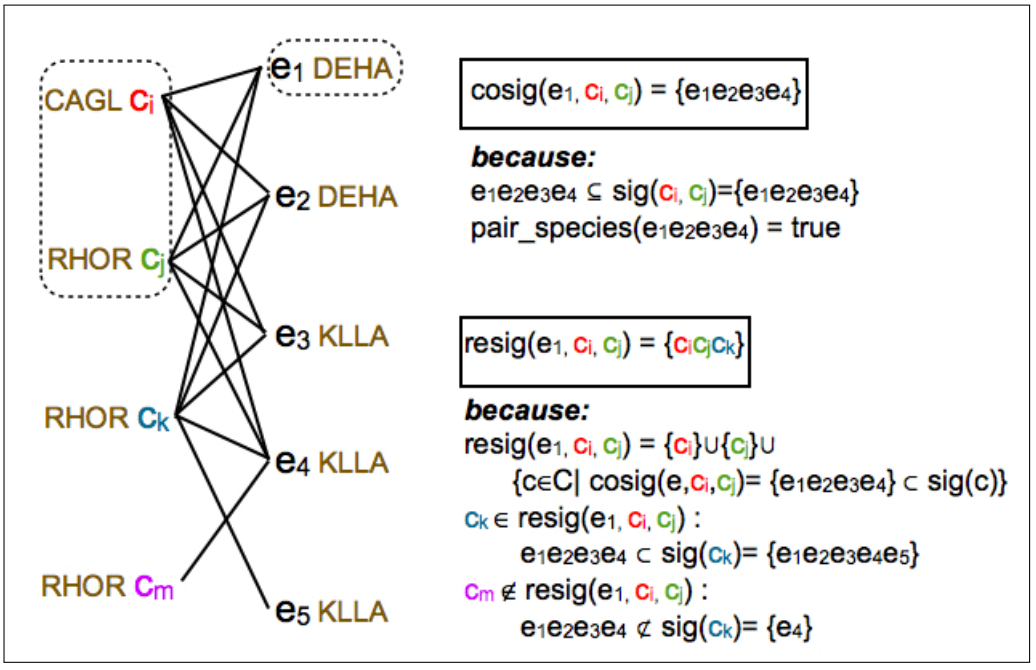



Figure 4.9: Example of *Cosig* and *Resig*. We compute $cosig(e_1, c_i, c_j)$ as the maximal subset of $sig(c_i, c_j) = \{e_1, e_2, e_3, e_4\}$ that contains $e_1$ and satisfies the property *pair_species*. $cosig(e_1, c_i, c_j) = \{e_1, e_2, e_3, e_4\}$, because $e_1 \in \{e_1, e_2, e_3, e_4\}$ and $species(\{e_1, e_2, e_3, e_4\}) = \{DEHA, DEHA, KLLA, KLLA\}$. *Resig* contains all $c_k$ (including $c_i$ and $c_j$) that are connected to all $E$-groups of *cosig*, so $resig(e_1, c_i, c_j) = \{c_i, c_j, c_k\}$.

**Lemma 4.1.** ***(Cosig and resig give an event)***
*The cross product $cosig(e, c_i, c_j) \times resig(e, c_i, c_j)$ is an event if $\exists$ $cosig(e, c_i, c_j)$.*

*Proof.* We need to prove, that $\forall e$, $c_i$ and $c_j$, $(e, c_i) \in R$, $(e, c_j) \in R$, if $\exists$ $cosig(e, c_i, c_j)$, then $R_1 = cosig(e, c_i, c_j) \times resig(e, c_i, c_j)$ is an *event*: $R_1|_C = resig(e, c_i, c_j)$, $R_1|_E = cosig(e, c_i, c_j)$, $R_1 = R_1|_C \times R_1|_E$, thus $R_1$ is a partial event by definition 4.5.

By definition of $cosig(e, c_i, c_j)$ is a maximum intersection of $e$-neighbors of $c_i$ and $c_j$, so we can not add extra $e$-group to this $R_1$.
Also, we cannot add another $c$-group: if there is new $c$-group $\tilde{c}$ to add, that means $\forall \tilde{e} \in cosig(e, c_i, c_j)$ there is $(\tilde{e}, \tilde{c}) \implies$ there is $(e, \tilde{c}) \in R \implies \exists c_k, c_k \neq c_i, c_k \neq c_j : c_k = \tilde{c} \implies$ this c-group was considered in *resig* construction.
$\implies R_1$ is a (maximum) event. □

**Example 4.1.** ***(Cosig and resig give an event)***
*In Figure 4.9, where* $cosig(e_1, c_i, c_j) = \{e_1, e_2, e_3, e_4\}$ *and* $resig(e_1, c_i, c_j) = \{c_i, c_j, c_k\}$*, the cross product of cosig and resig gives an event.*
*First, each E-group of* $\{e_1, e_2, e_3, e_4\}$ *is connected to each C group of* $\{c_i, c_j, c_k\}$*. The species of* $\{e_1, e_2, e_3, e_4\}$ *(DEHA, DEHA, KLLA, KLLA) are in pairs. So this subgraph is a partial event. And, it is not possible to add neither E- nor C-group and preserve the properties of a partial event:* $c_m$ *is not connected to* $e_1$*;* $e_5$ *is not connected to* $c_i$*.*

### 4.2.3 MapReduce deployment

Algorithm 7 maps over $C$–$E$ group relations, which are grouped by $c$. For each $c$ in $C$ we store the signature $sig(c)$. Next, these relations are grouped by $e$, so that in the reduce step we have, for each $e$, all length 2 paths from $E - C - E$. This is enough to calculate $cosig(e, c_i, c_j)$ and $resig(e, c_i, c_j)$ for each $e$ and for all $(c_i, c_j)$ pairs of corresponding $c$-groups. To calculate $cosig(e, c_i, c_j)$ we first take a maximal intersection of signatures $sig(c_i)$ and $sig(c_j)$. Then we remove from this intersection the $e-$groups that are not at least in pairs. To calculate $resig(e, c_i, c_j)$ we check all $c$-groups (other than $c_i$ and $c_j$) and include the $c$-group to the *resig* if its signature contains the previously calculated intersection (*cosig*). The union of the cross-products of $cosig(e, c_i, c_j)$ and $resig(e, c_i, c_j)$ is the set of events, as is show schematically in figure 4.10.

### 4.2.4 Proof of algorithm

**Theorem 4.1.** *The result set A of Algorithm 7*

$$A = \bigcup_{e \in E} A_e = \bigcup_{e \in E} \left( \bigcup_{\substack{c_i, c_j \in \{c_k | (e, c_k) \in R\} \\ i \leq j \\ \exists\, cosig(e, c_i, c_j)}} \langle cosig(e, c_i, c_j) \times resig(e, c_i, c_j) \rangle \right)$$

*is equal to the set of all Events:*

$$A = Events$$

*Proof.* "$\supseteq$" is true by Lemma 4.1.
"$\subseteq$" We need to prove, that $\forall$ *event* $R_1$, $\exists$ $e \in E$, $i$and$j$ : $(e, c_i) \in R$, $(e, c_j) \in R$, $i \leq j$, that $\exists$ $cosig(e, c_i, c_j)$ and $cosig(e, c_i, c_j) \times resig(e, c_i, c_j)$ equal to it.
Event is a partial event, so by definition 4.5
$R_1 = R_1|_E \times R_1|_C$ and $pair\_species(R_1|_E)$.
Let us take any $e \in R_1|_E$, any $c_i, c_j \in R_1|_C$, so

$$\forall e' \in R_1|_E \ (e', c_i) \in R_1 \text{ and } (e', c_j) \in R_1$$

**Algorithm 7** Detecting fusion and fission events

**Input:** List of relations between c- and e-groups = *CtoE* column family = $\{(c_i, (e_{i1}, ..., e_{i..}))\}$

**Output:** Events column family = $\{\langle (e_m, ...), (c_n, ....)\rangle\}$

**function** MAP(
key: $c$ ▷ c-group
values: $e_1, ..., e_k$ ▷ e-groups
)
Let $sig(c) = \langle e_1, ..., e_k\rangle$ ▷ signature
**for all** $e_i \in e_1, ..., e_k$ **do**
$output(e_i,\ \langle c, sig(c)\rangle)$
**end for**
**end function**

**function** REDUCE(
key: $e$ ▷ e-group,
values: $v_1, ..., v_n$ ▷ $v_i = \langle c_i, sig(c_i)\rangle$,
▷ $sig(c_i) = \langle e_1, ..., e_k\rangle$
)
Let $A_e = \emptyset$
Let $\langle c_i, sig(c_i)\rangle = v_i$
**for all** $i \in 1, ..., n,\ j \in i, ...n$ **do**
$cosig \leftarrow cosig(e, c_i, c_j)$
**if** $cosig \neq \emptyset$ **then**
$resig \leftarrow resig \cup \{c_i\} \cup \{c_j\}$ ▷
**for all** $k \in 1, ..., n,\ k \neq i,\ k \neq j$ **do** ▷
**if** $cosig \subseteq sig(c_k)$ **then** ▷ Calculate $resig(e, c_i, c_j)$
$resig \leftarrow resig \cup \{c_k\}$ ▷
**end if** ▷
**end for** ▷
$A_e \leftarrow A_e \cup \langle cosig \times resig\rangle$
**end if**
**end for**
$output(e, A_e)$;
**end function**

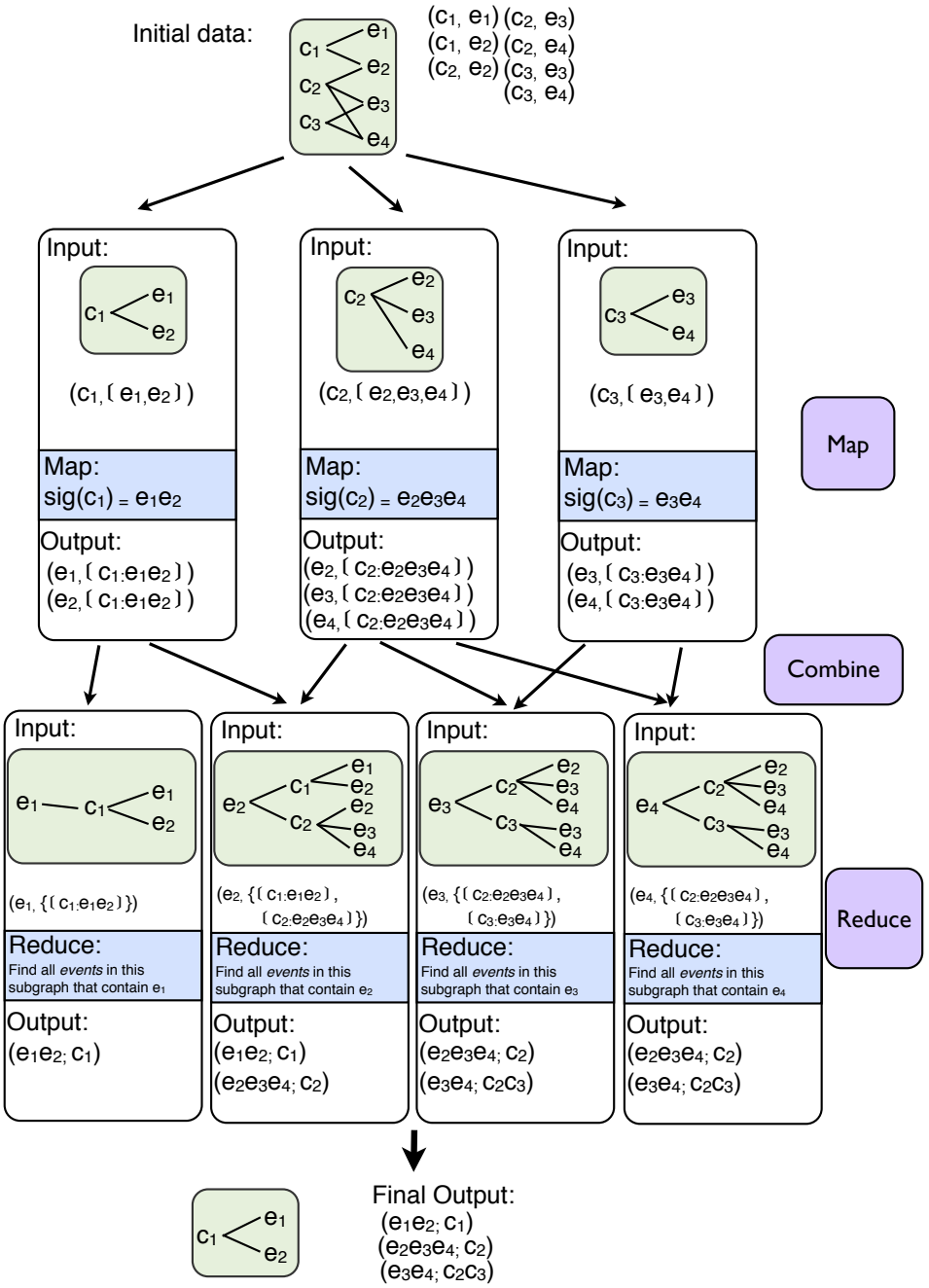



Figure 4.10: MapReduce Algorithm 7: Identification of fusion and fission events by identifying subgraphs in the $C$–$E$ bipartite graph. For simplicity, here we do not consider species property of $E$-groups ($pair\_species$).

The property $pair\_species(R_1|_E)$ is true, so $R_1|_E = cosig(e, c_i, c_j)$.

$$\forall c_k \in R_1|_C \text{ and } \forall e' \in R_1|_E \exists (e', c_k)$$

This implies $R_1|_C = resig(e, c_i, c_j)$. Finally,

$$R_1 = R_1|_E \times R_1|_C = cosig(e, c_i, c_j) \times resig(e, c_i, c_j)$$

□

### 4.2.5 Meeting Criterion

**Definition 4.11.** ***(Criterion of a good MapReduce algorithm)***

1. Balanced loads (for map and for reduce):
   ***Data distributed among the similar chunks of optimal size both for map and reduce phases***
   *The map phase in our MapReduce deployment of fusion algorithm groups relations by C-groups and process each c-group with corresponding E-groups independently. To measure the load of the map phase we will calculate the number of E-groups associated to each c-group, which we will call the E-index of a c-group. For our data the distribution of E-indexes is illustrated in the left part of the Figure 4.11. Analogously, the reduce step groups relations by E-groups, so we calculate C-index for each e-group (the distribution of C-indexes is illustrated on the right part of Figure 4.11). In particular reduce functions operate with the subgraphs of the length 2, so we should take into account both E- and C-indexes to evaluate the reduce phase.*

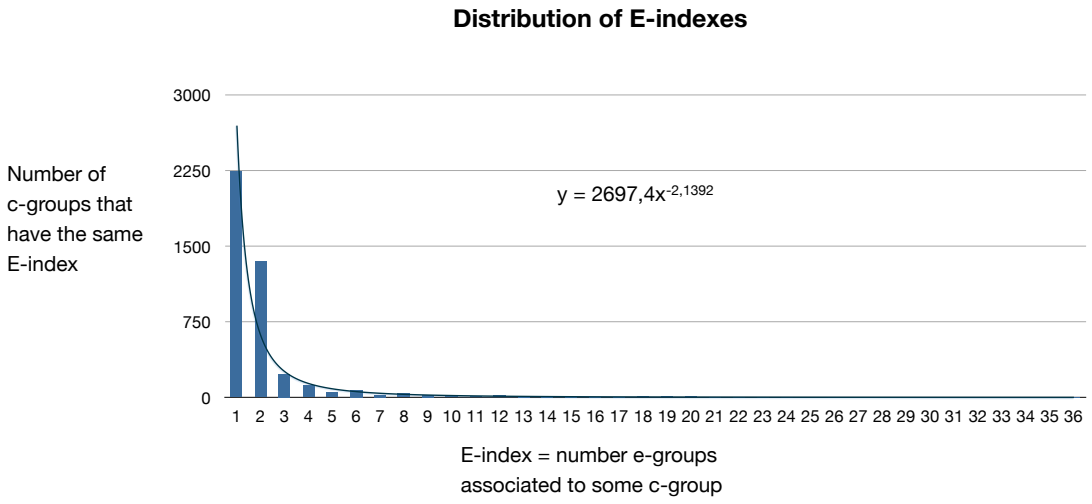
   *From the diagrams in the Figure 4.11 we see that the E-index can be from 1 to 36 and it is 2.246 in average. The C-index is from 1 to 24 and it is 1.827 in average. Both E- and C-indexes (which are actually degree of vertices in the c-to-e bipartite graph) are not big and do not change much. Both indexes have a power law distribution. Thus, we can claim that the load is balanced.*

2. Good in scaling:
   ***With the increase of data the number of chunks increases and the size of chunks does not significantly grows***
   *With the increase in number of species we expect significant geometric increase of the total e-to-c relations graph. From the other side, current average value of the vertices degrees (E- and C-indexes) is very low (1.827 for C-index and 2,246 for E-index) and it will grow very slowly. Consequently, the chunks for map and reduce phases will grow reasonably slowly.*

## 4.3 Application 2: Consensus and MCL clustering

A protein family is a group of historically related proteins presumed to be functionally related. The members of these groups usually share the same or similar biological functions. Protein families in related genomes provide key information for phylogenetic analysis, functional annotation, exploring of functional diversity, and inferring metabolic networks. Many complementary methods have been developed for computing families. One approach, [NS07], describes an efficient heuristic algorithm for the NP-complete *consensus clustering* problem to reconcile complementary partitions of proteins into families. The idea is that different clustering methods use different measures and different properties of the set of proteins makes them complementary. [NS07] defines consensus clustering as a process of finding a consensus partition for $k$ partitions that has minimum distance from them.

The algorithm is based on a compact encoding of the confusion matrix of relations between competing partitions, and an election procedure to find the best consensus from competing partitions. In a typical application of this procedure, we start with four protein-protein similarity matrices: systematic all-against-all alignments using the Smith-Waterman [SW81] and Blast [AGM$^+$90] algorithms, each with and without filtering for homeomorphy [W$^+$04]. These matrices are clustered by the MCL algorithm [vD00, EVDO02]



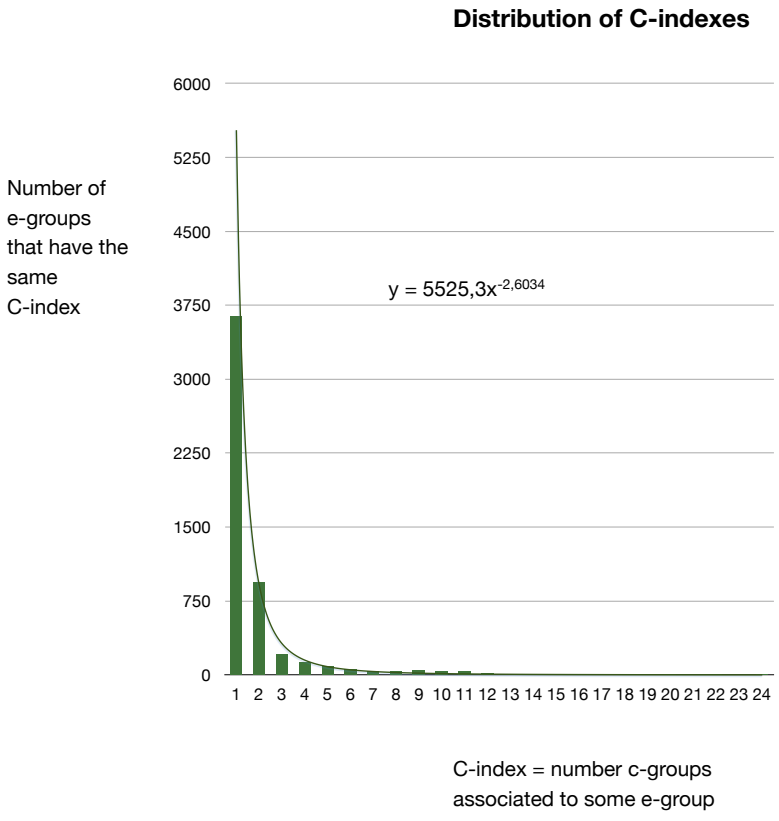




Figure 4.11: The first figure: we count E-index for each c-group, and show how many $C$-groups have the same E-index. Similarly, the second figure: we count C-index for each e-group, and show how many $E$-groups have the same C-index.

using three inflation parameters. The resulting twelve partitions are then reconciled using the consensus procedure.

MCL clustering uses random walks on simple weighted graphs using Markov stochastic matrices [vD00, EVDO02]. This algorithm was chosen in 2007 for detecting protein families using different parameters by authors of Consensus clustering algorithm [NS07].

The families in [S$^+$09] were computed from 45000 proteins (thus $10^9$ pairwise sequence comparisons), that were partitioned into roughly 4500 families. The consensus clustering process schematically is shown in the figure 4.12.

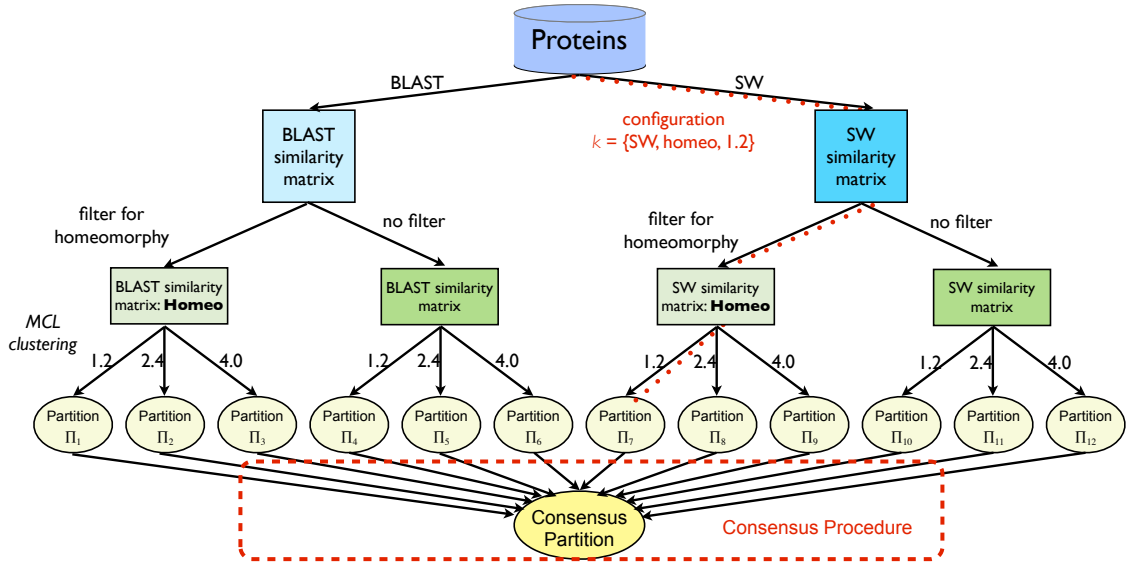



Figure 4.12: Simplified schema for the consensus clustering algorithm defined in [NS07]. First step is all-against-all comparisons of proteins to obtain matrices of similarities using 2 different algorithms: BLAST and Smith-Waterman's. Then there is a filtering for homeomorphy step. The third step is MCL clustering of similarity matrices using three inflation parameters. The last part of the algorithm is a Consensus procedure to select the best family partition.

### 4.3.1 Adaptation to MapReduce

#### Consensus Procedure

The first scaling issue of the Consensus Clustering algorithm defined [NS07] is the final Consensus procedure, since it involves all-against-all comparisons of the clusterings made with different configurations. The initial method (defined in [NS07]) detects the sets of the conflict regions for each pair of clusterings. On the initial data for the 12 configurations (different methods and different clustering parameters) the number of the conflict regions is about 3500. But, the consensus procedure itself is run with many different sizes, conflict and percentile parameters. With increase of number of the configurations, the number of the conflict regions grows quadratically, because we need to compare all configurations against all.

This quadratic growth of the number of conflict regions gives us a scaling issue for the computations. At the same time it can be parallelized easily since the conflict regions of the consensus procedure are independent and can be naturally processed in separate map

functions, using the region id as a row key and associating a serialized representation of conflict region as a row.

#### MCL Clustering for limited number of clusters

Beyond this natural decomposition, there is another scaling issue. With the linear growth of proteins we have quadratic growth in the size of the similarity matrices, which rapidly reach to the point when the existing tools (MCL clustering for detection protein families) will not be able to work with these large similarity matrices. Our aim is to find the way when we can work with the part of the data in parallel and then combine it (also, in parallel).

Thus for this MapReduce deployment we consider one configuration $k$ of alignment algorithm, filtering and MCL inflation parameter at the time (for example, in the Figure 4.12 $k = \{SW, homeo, 1.2\}$).

We cluster the large matrix of this configuration using an iterative sampling strategy. In the first iteration we distribute proteins randomly into the equal sized chunks by assigning random sample numbers to each protein. The map phase maps over the similarity matrix rows and extract from the row the pairs of proteins that belongs to the same sample and output it using sample number as a key and protein pairs as a combined value. In the reduce phase all these outputs are grouped by sample number, so we have small submatrices of the proteins that are from the same sample. We perform MCL clustering operation and obtain family partitions for each of these sumbatrices . The union of these family partitions is the result of the first iteration.

After the first iteration the result is very far from reality since we abandoned many relations between proteins from different samples. For the second iteration we randomly redistribute proteins into chunks with the condition that proteins that were found in the same family will go to the same sample chunk. In the map phase we again take initial large matrix and extract in parallel submatrices exactly the same way as in previous iteration. In the reduce phase we recluster extracted protein sumbatrices. Now the result is a bit better, because similar proteins "stuck" to each other.

In each iteration the family partition is better than in the previous one, because the chance that similar proteins appeared in the same sample increases. The process converges and after several iterations we obtain a partition closely approximating the results obtained by the original heuristic algorithm. To decide when we should stop iterating we count number of families and we terminate the process if number of families does not change. Here we use the property of our data that the number of families does not change a lot and this number is around 4500-5000.

### 4.3.2 Formalism: the iterative sampling algorithm.

**Definition 4.12.** ***(Similarity Matrix)***
*For a given set of proteins* $P$, $|P| = n$, *we can represent a given similarity function* $S$, $S : P \times P \to \mathbb{R}$ *as* similarity matrix $D$.

$$\forall p_i, p_j \in P \; \exists \; similarity \; d \in \mathbb{R}, d \geq 0, \; D(p_i, p_j) = S(p_i, p_j) = d$$

(Note) We use similarity functions defined in Blast [AGM+90] or Smith-Waterman [SW81] algorithms. The algorithms uses different distance measures to compare sequences: Blast uses $-log_{10}$, Smith-Waterman uses $\%id$.

**Definition 4.13.** ***(P'-projection)***
*For the similarity matrix* $D : P \times P \to \mathbb{R}$, $P' \subseteq P$ *we define a* $P'$-projection *of* $D$

$$D_{P'} : P' \times P' \to \mathbb{R}$$

*if*

$$\forall p_i, p_j \in P' \; D(p_i, p_j) = D'(p_i, p_j)$$

**Definition 4.14.** ***(Families)*** *[NS07]*
$P$ *is the set of proteins over which a family computation will be performed.*
*A family computation* $F$ *over* $P$ *defines a partitioning of* $P$ *into disjoint subsets* $F = \{f_1, ...f_n\}$ *such that* $P = \bigcup f_i$.
*The subsets* $f_i$ *are called* families.

**Definition 4.15.** ***(Sample Map)***
*For given set of proteins* $P$, *maximum sample number* $k \in \mathbb{N}$, $k > 1$, *representation number* $r \in \mathbb{N}$ *and iteration* $i \in 0, 1, ...$: *the* sample map *computation* $S^i$ *over* $P$ *defines a* $k$*-partitioning of* $P$ *into disjoint subsets* $S^i = \{s_1, ..., s_k\}$ *such that* $P = \bigcup s_j$ *with the following rule:*

*If there exists a previous sample computation* $S^{i-1}$ *and family computation* $\Pi^{i-1}$*:*

$$\forall f_j \in \Pi^{i-1} : S^i \text{ is defined as: } f_j \to random(s_1, ..., s_k) \quad (*)$$

*Else:*

$$\forall p \in P : S^i \text{ is defined as: } p \to random(s_1, ..., s_k)$$

*(Note 1) (*) means that*

$$\forall f_j \in \Pi^{i-1} \; \forall p_{j1}, p_{j2} \in f_j \; \exists l : s_l \in S^i \text{ and } p_{j1} \in s_l, \; p_{j2} \in s_l$$

*(Note 2) The* $r$ *value should be as large as possible, considering the fact that in bigger samples the accumulation of families will be faster, but at the same time the size of similarity matrices are* $|r|^2$, *and we should take into account the efficiency of tools on large matrices.*
*(Note 3) We calculate maximum sample number* $k$ *after choosing* $r$ *as:*

$$k = \frac{|P|}{r}$$

*(Note 4) For our data,* $|P| \approx 45000, \; r = 10000, \; k = 4$

**Definition 4.16.** ***(Sample Projections)***
*For a set of proteins* $P$, *similarity matrix* $D$, *maximum sample number* $k$ *and a protein sample map* $S^i$, *for each* $P^i_m \in S^i, m \in \{0, ..., k\}$ *we will call the sample similarity* $P^i_m$*-projections* sample-projections *(*$D^i_m \subseteq D, D^i_m : P^i_m \times P^i_m \to \mathbb{R}$*)*

**Definition 4.17.** ***(Sample family partitions)***
*For a given list of sample-projections* $(D^i_0, ...D^i_k)$, *some clustering function* $Clustering : D \to \Pi$, *(transform similarity matrix to the family computation) we calculate* sample family partitions $\Pi^i_m = Clustering(D^i_m)$

*Let*

$$\Pi^i = \bigcup_{m \in \{0,..k\}} \Pi^i_m$$

*This family partition is the result of the $i^{th}$ iteration.*
*(Note) As a clustering function we use the TribeMCL [EVDO02] version of the MCL [vD00] algorithms.*

**Algorithm 8** Iterative sampling for the MCL clustering

**Input:** Protein similarity matrix D = $\{\langle prot_i, ((prot_{j1}, score_{i,j1}), ...)\rangle\}$
**Output:** Cassandra.families = $\{\langle family, (prot_i, ....)\rangle\}$ and Cassandra.sample_map = $\{\langle prot_m, sample\_num\rangle\}$

```
function MAIN
    Cassandra.sample_map ← random(0..k)
    while true do
        RunMapReduce(D)
        if |Cassandra.families| = FMAX then
            return Cassandra.families
        end if
    end while
end function

function MAP(
    key: prot_i
    values: (prot_j, score_{i,j}), ...        ▷ i_th row of the similarity matrix D
    )
    sample_num ← Cassandra.sample_map(p_i)
    for all prot_j ∈ values do
        if sample_num = Cassandra.sample_map(prot_j) then
            output(sample_num, ⟨prot_i, prot_j, score_{i,j}⟩)
        end if
    end for
end function

function REDUCE(
    key: sample_num                                   ▷ Sample number
    values: (prot_l, prot_m, score_{l,m}), ...        ▷ Sample similarity matrix D_s
    )
    F_s = Clustering(D_s)                             ▷ Family partition
    for all family ∈ F_s do
        new_sample_num ← random(0..k)                 ▷ New sample number
        for all prot ∈ family do
            Cassandra.families(prot) ← family
            Cassandra.sample_map(prot) ← new_sample_num
        end for
    end for
end function
```

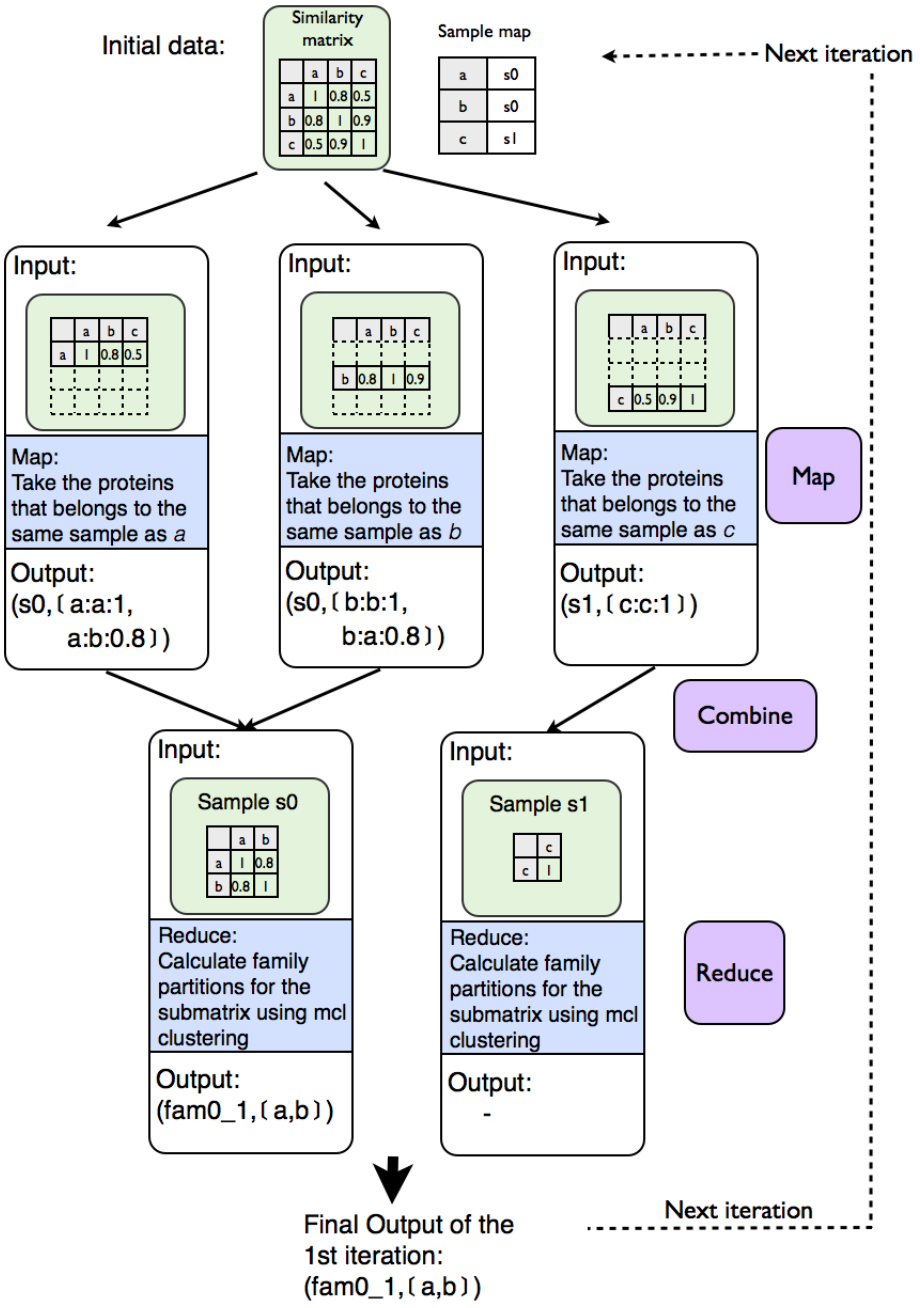



Figure 4.13: MapReduce algorithm 8: Iterative sampling for the Consensus clustering, where initially random samples accumulate relations between members of the same cluster as iterations progress

### 4.3.3 MapReduce deployment

We run the MapReduce MCL clustering for each configuration of the sequence alignment algorithm, filtering and MCL inflation parameter. Suppose, we choose some configuration. At first, we sample proteins by assigning random sample numbers for each protein. Algorithm 8 maps over the rows of similarity matrix and from each row selects the protein pairs that belong to the same sample, and output them using the sample number as key.

These pairs are then combined by sample number by the MapReduce framework, which then proceeds to the reduce phase. In the reduce phase all relations are grouped by sample numbers so in each reduce job we have a sample projection of similarity matrix. Sample similarity matrices (projections) are clustered in the usual way. The union of these protein families of samples is a result of this iteration. For the next iteration we again redistribute proteins into samples, but proteins of the same protein family stay in the same cluster, so that samples tend to accumulate relations between members of the same cluster. The computed clusters converge towards the clusters that would have been found on the complete similarity matrix if its computation had been feasible. This iterative process is shown schematically in figure 4.13.

### 4.3.4 Evaluation

To evaluate iterative sampling algorithm we ran it on the same data as [NS07] and compared results. To compare family partitions $\Pi$ and $\Pi'$ we use *distance* function, defined by van Dongen [vD00]:

**Definition 4.18.** ***(van Dongen Distance [vD00])*** *(following [NS07])*

$$d(\Pi, \Pi') = 2n - \pi_{\Pi}(\Pi') - \pi_{\Pi'}(\Pi),$$

*where $n$ is the cluster size, $\pi_{\Pi}(\Pi')$ is asymmetric index of the degree of refinement of one partition versus another.*

The iterative sampling process converged to equivalent heuristic results in 10–15 iterations. The distance between partition using sampling algorithm and corresponding MCL computation was converges as one over $x$ for each configuration(figure 4.14).

From figure 4.14 we can see that the process converges for all possible configurations (alignment algorithms: Blast/SW, filtering: homeo/no-homeo and MCL inflation parameters: 1.2, 2.4, 4.0). We observe that our MapReduce algorithm converges better for Swith-Waterman similarity matrices than Blast ones. Also it converges better with alignments filtered by homeomorphy and MCL inflation parameter 1.2, and worse with bigger coefficients.

### 4.3.5 Meeting Criterion

**Definition 4.19.** ***(Criterion of a good MapReduce algorithm)***

1. Balanced loads (for map and for reduce):
   ***Data distributed among the similar chunks of optimal size both for map and reduce phases***
   *The map phase in our MapReduce deployment of the MCL clustering algorithm maps over the rows of the similarity matrix and outputs protein pairs with scores that belong to the same sample. To measure load of the map phase we will calculate for each row the number of proteins that have score value (Figure 4.15).*

   *We observe that there are mostly small rows, as the maximum elements in the row is 186. In case there will be large rows we can split them into several middle sized ones, so the map phase will be balanced.*

   *The reduce phase works with submatrices of equal size, and the number of chunks grows with the increase in data. Thus, we can tell that the reduce phase is balanced.*

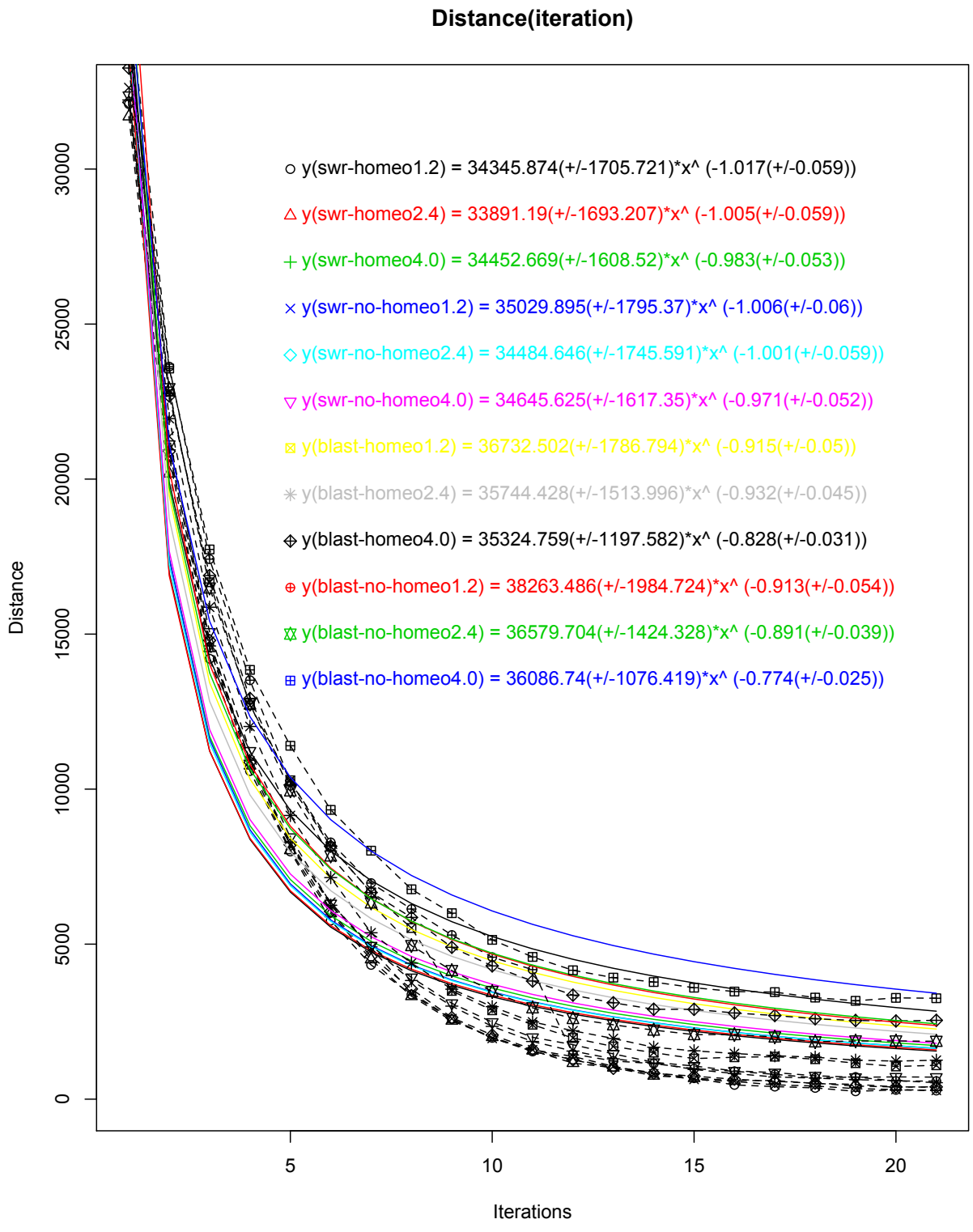



Figure 4.14: Distance between sampling computation and MCL clustering computation depending on iteration.

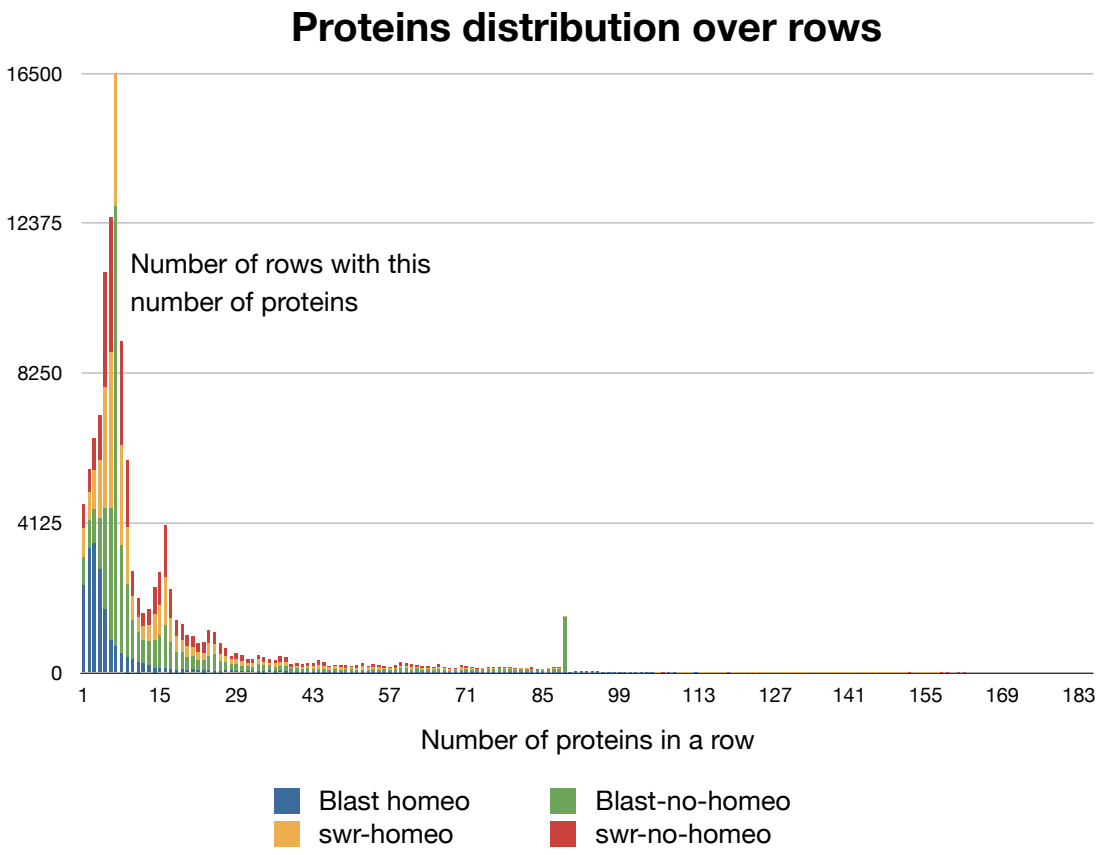



Figure 4.15: Proteins distribution over rows: we calculate number of proteins in the each row and show how many rows have the same number of proteins.

2. Good in scaling:
   ***With the increase of data the number of chunks increases and the size of chunks does not significantly grows***
   *With the increase of number of proteins the rows grow linearly even though the matrix is very sparse. But it is not necessary to have whole row at the time since we are interested only on independent protein pairs, so we can split very large rows and store them in several Cassandra rows. Thus, we will map over parts of the rows of the similarity matrix, but the samples will be combined the same way by sample number for reduce phase.*

   *The reduce phase operates with submatrices and the size of the submatrices is fixed. With the increase of data, the number of samples grows, but not the sample size.*

# Chapter 5

# Implementation

***Résumé :*** *Afin d'évaluer le schéma et les algorithmes conçus dans les chapitres précédents, nous avons décidé de les mettre en œuvre. Nous commençons par la conception d'un système de stockage de données et relations, qui peut être connecté indifféremment à une base de données SQL classique ou à un système NoSQL. Cette conception nous a obligé de reconcevoir l'objet modèle de Magus et d'explorer quelques choix stratégiques et techniques. Nous avons implémenté le nouveaux système et avons ainsi contribué au rédéveloppement du système d'analyse de génomes Magus. Une évaluation de performance, réalisée avec des jeux de données réels et artificiels, montre que l'approche Cassandra fournit à la fois des meilleurs performances et une meilleure mise à l'échelle.*

*Ensuite, nous implémentons les deux nouvels algorithmes développées dans le châpitre 4, employant pour MapReduce le cadre logiciel Apache Hadoop. La performance des deux solutions est encore évalué avec des jeux de données réels et artificiels. Nous observons que, sur des instance qui rentrent entièrement en mémoire vive, la performance de MapReduce souffre ; par contre, pour des instances plus grandes, la mise à l'échelle raisonnée des algorithmes MapReduce permet de suivre raisonnablement la croissance des données.*
*Nous terminons avec une modest exploration du rôle du langage déclaratif Pig Latin pour exprimer des calculs distribués MapReduce de façon plus intuitive pour les biologistes.*

In this chapter we will first talk about design of a data storage system that can be connected both to SQL and NoSQL data stores. In order to evaluate the pertinence of the theoretical proposals we decided to implement a practical solution in a context of reimplementation of the Magus system. First, we describe implementation of the system, reviewing the object model,and technical and strategic solutions. Concretely, we will talk about a software adaptor (in Perl) to the Cassandra NoSQL and implementation of the data schema, defined in 3.6.1. We evaluate performance of the new system by loading and searching stress tests based on real and artificial data sets, which are made by extrapolation, conserving the properties of the data set. We will show that NoSQL Cassandra solution has better performance and scaling properties than the relational PostgreSQL solution.

Second, we will talk about MapReduce adaptations of two global comparative genomics analyses, described in sections 4.2 and 4.3. We will show the feasibility of these kind of adaptations by implementing their algorithms in Java using the Apache Hadoop framework. We will evaluate performance by executing these algorithms using real and artificial

data sets. We will observe, that on the data sets that fit into memory, performance of the MapReduce algorithms is worse than original ones; however for data sets that do not fit into memory, the MapReduce algorithms scale in a way that permit such instances to be handled.

Third, we will discuss the possible use cases for the Pig Latin declarative language to express MapReduce distributed computations in a more intuitive way, to reduce cognitive friction for performing global analyses.

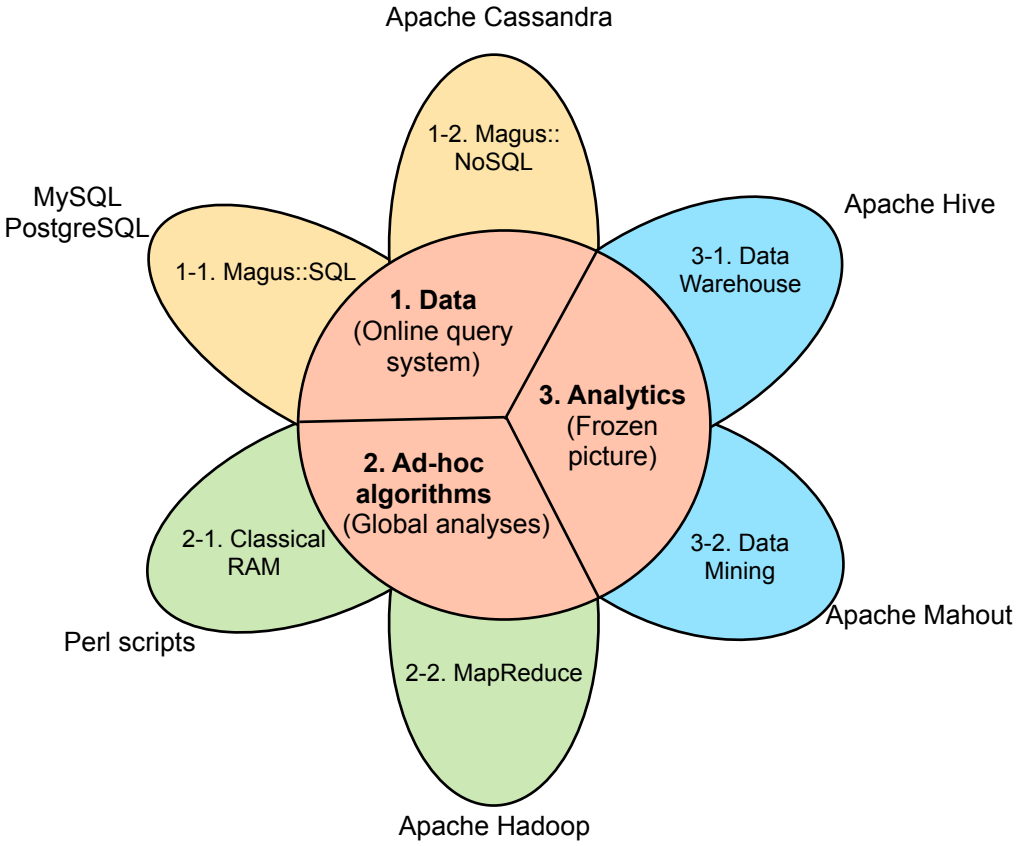



Figure 5.1: Tsvetok system overview. There are 3 connected components that refer to the Data storage, Ad-hoc algorithms and Analytics systems.

Figure 5.1 introduces Tsvetok, a unified view I define integrating aspects of data analyses in comparative genomics together. We place three cooperating systems in the center of Tsvetok: data storage, ad-hoc algorithms and analytics systems. Each system in the center connects to our proposed implementations, in the petals.

Let us describe the systems of Tsvetok in detail (Figure 5.1):

1. *Data system* (Online query system).
   The *Data* storage and query system defines a unified interface used by Magus Query System Interface, involves operations for searching and editing of sequence features. The two petals *Magus::SQL* and *Magus::NoSQL* in the figure correspond to two different technologies for physical storage–classical relational databases and NoSQL data stores. The new Magus system is designed to allow switching between data stores easily, simply by changing the connector parameters.

To develop Magus::NoSQL for the case of a Apache Cassandra data store I used the general principles of constructing column families defined in the section 3.6.1 to guarantee efficient use of Cassandra. In the section 5.4 I will demonstrate through the performance comparison between NoSQL and classical relational database implementations that my general principles work well in practice.

2. *Ad-hoc algorithms* system (Global analyses).
   The second segment in figure 5.1 describes *Ad-hoc algorithms* in comparative genomics, designed for a specific purpose. These algorithms have precise sense, stated biological goal; they give an exact answer for a specific question. The two petals correspond to classical RAM and MapReduce paradigms of computations.

   The first implementation petal represents classical *Random Access Memory (RAM) algorithms*. These algorithms must obey certain assumptions, such as a global, shared memory. Consequently, they can work only if the computations are able to fit into memory on one machine.

   The second petal represents implementations of the computations using the *MapReduce* paradigm (see section 2.4.2) for the cases when the input data do not fit into memory. We use the Hadoop MapReduce framework for distributed computations, because it hides the technical details of the parallelization implementation, making it easy to write applications for the global analyses.

   To develop MapReduce algorithms I used the criterion of a good MapReduce decomposition defined in the section 4.1 to guarantee efficient use of MapReduce paradigm. In the section 5.5 I will demonstrate through the performance comparison between MapReduce and classical RAM implementations that my criterion works well in practice.

3. *Analytics* system (Frozen picture).
   The third segment in the figure 5.1 corresponds to the *Analytics* system. This system is used for decision support and for large-scale explorative analyses by domain specialists (biologists, etc) on the frozen data (see section 2.5). Two petals correspond to two technologies related to analytics–Hive Data Warehouse and Mahout Data Mining libraries.

   The first Analytics implementation petal represents *Data Warehouses*. They are used to store the data for decision support using SQL-like query languages. Apache Hive Data Warehouse provide the way of assessing and slicing data in a large-scale context, executing on top of Hadoop MapReduce.

   The second implementation petal represents *Data Mining*. Mahout is collection of libraries for large-scale analytics developed in Data Mining community. The analyses execute on top of Hadoop MapReduce. Mahout contains many already implemented algorithms for Clustering, Sorting, Classification and others which are heavily used in comparative genomics.

In this chapter we will focus on the Data and Ad-hoc algorithms implementations (petals 1-1 Magus::SQL , 1-2 Magus::NoSQL and 2-2 MapReduce). We will talk how we organized data storage, we will discuss software design patterns for searching and curation of the data.

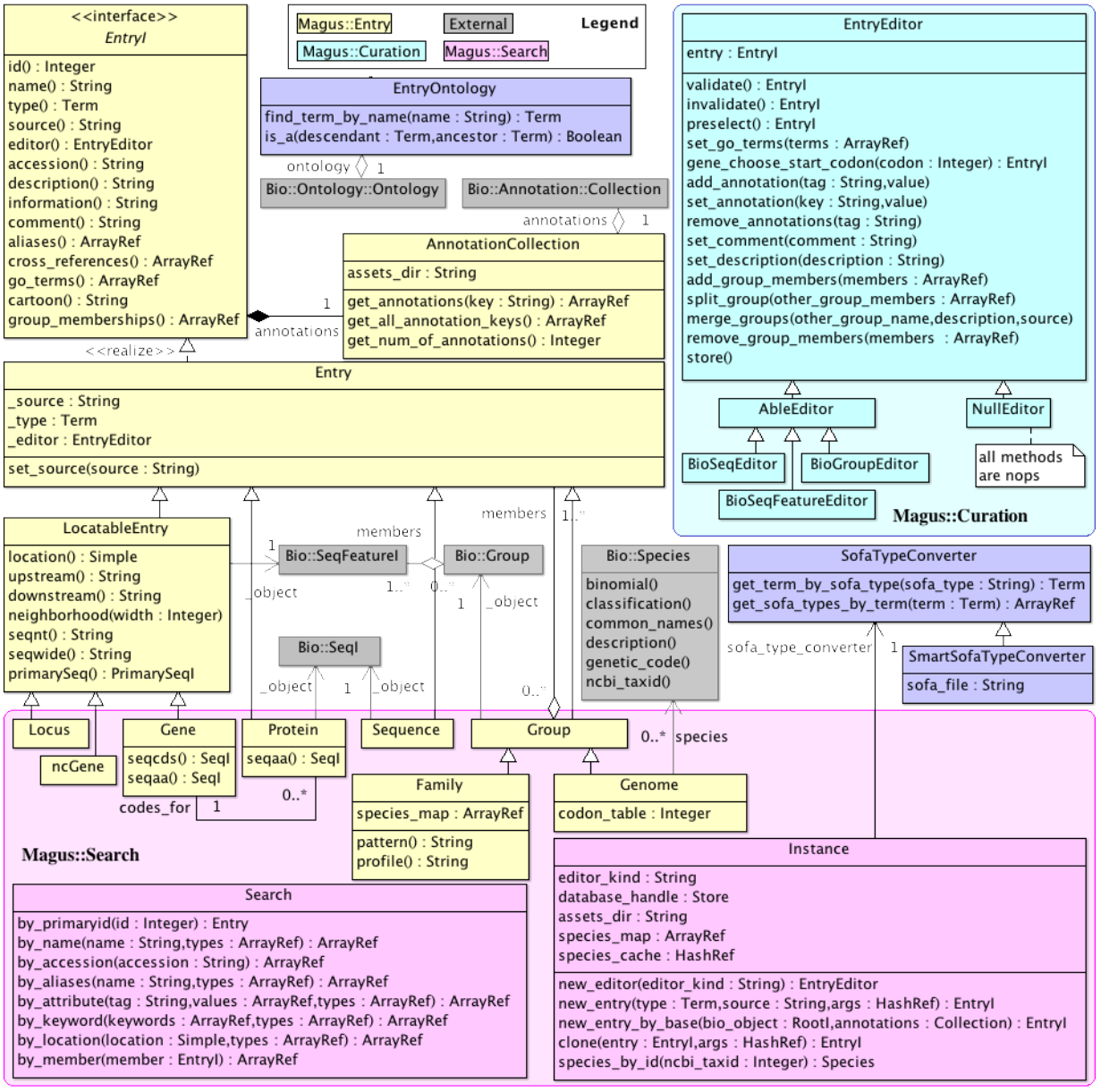



Figure 5.2: Magus 2.0 object model

## 5.1 Magus Object Model

The new Magus Object Model was developed, in collaboration with Magnome team members (see section 1.4), to unify access to the different data schemas, such as *Bio::DB::SeqFeature::Store* and BioSQL with the flexibility to use different types of data storage systems (relational and NoSQL). The Génolevures web site is written in Perl and Mason, thus our Magus object model is defined using a set of Perl classes.

The new Magus object model to represent sequence features is illustrated in Figure 5.2. The base class for all features is class *Entry*. The *Protein* class is a child of the Entry, its

underlying object is *Bio::SeqI* of BioSQL. *LocatableEntry* is a child of Entry, it has start and end location coordinates on the sequence. There are three kinds of LocatableEntry: *Gene*, *Locus* and *ncGene*. The underlying object of LocatableEntry is *Bio::SeqFeatureI*.

The *Group* class contains a collection of Entry or Group objects. The underlying object of Group is *Bio::Group*, which currently is not part of BioSQL. Group objects can contain information about sets of alleles, orthologs, paralogs, tandem repeats, fusion and fission events, protein families, etc.

Some client software only use the Magus system to search for sequence features; for them we provide read-only access. Other interfaces, for example annotators and curators, can also annotate, validate and invalidate genes, and add new ones. To avoid interface subclassing we decided to apply the *Strategy Design Pattern* for the classes that can edit features. Strategy selects a suitable algorithm at runtime, which should be applied, based on the type of data being processed. *EntryEditor*, the base class, provides an interface to edit, annotate, validate and invalidate sequence features. It has two concrete strategy implementations–*AbleEditor*, that can edit features, and *NullEditor*, with read-only access. There are three kinds of the AbleEditor for different types of features: *BioSeqEditor*, *BioSeqFeatureEditor* and *BioGroupEditor*.

## 5.2 Magus::NoSQL Cassandra implementation

We made a Cassandra adapter for BioPerl module Bio::DB::SeqFeature::Store, which is used for storing and retrieving sequence annotation data from different types of underlying databases. Bio::DB::SeqFeature::Store allows us to persistently store Bio::SeqFeatureI objects in a database and later to retrieve them by a variety of search methods.
Bio::DB::SeqFeature::Store can work with many kinds of databases using different adaptors: mysql, berkeleydb, memory adaptor. We developed an adaptor, that allow us to use Cassandra storage. The classes, were implemented:

1. *CassandraWrapper.pm*

   This is a wrapper of the perl binding for Thrift, the low-level RPC-based API, to separate very changing and unstable Cassandra Thrift API from the main adaptor code.

2. *CassandraGFF3.pm*

   This helper class works with sequence annotation files in GFF3, EMBL and others formats. This class encapsulates information about our schema of column families in Cassandra to store sequences and sequence features. It also provides the searching API to retrieve Bio::SeqFeatureI objects using different types of given attributes. We extended CassandraGFF3 to support another types of objects, such as some BioSQL objects like proteins or groups (the new class Bio::Group).

3. *Cassandra.pm (Bio::DB::SeqFeature::Store::Cassandra)*

   This is the main adaptor class, which allow us to store sequence features in Cassandra data store and select features by id, by name, by alias, by type, by location (sequence id and coordinates interval), by attribute, by notes (description) and by arbitrary combinations of selectors, limiting the search to a particular region, getting, and storing sequence information.

As we claimed before, CassandraGFF3 works with BioPerl and BioSQL classes, and their conversion to Magus Data objects (different kinds of Entry objects) for end users is placed in a common place in Magus.

4. *Magus::Search*

   Magus::Search has a unique interface for searching for sequence features. It can work both with Cassandra NoSQL (by connecting to the Bio::DB::SeqFeature::Store::Cassandra) object and relational SQL data stores with different schemas (Bio::DB::SeqFeature::Store:: or BioSQL) simply by changing a connector string.

5. *bp_load_seqdatabase.pl*
   bp_load_seqdatabase.pl is a standard tool of BioPerl to load genomic files in various formats (EMBL, Uniprot, etc) to the relational databases (PostgreSQL, MySQL, etc) with BioSQL schema. We implemented an extension for this flate file loader to support loading to Cassandra using our schema of column families. So we can use this script in a similar way, specifying in parameters "cassandra" as a driver, Cassandra keyspace name as dbname, and correspondent host and port:

   *perl bp_load_seqdatabase.pl -host localhost -port 9160*

   *-driver cassandra -format embl -dbname magus_test Cagl.embl*

   *perl bp_load_seqdatabase.pl -host localhost -port 9160*

   *-driver cassandra -format swiss -dbname magus_test Yali.swiss*

6. *Unit tests in t/*.t*
   For each implemented class and for each method of this class we created a test method.

7. *Stress tests (functional) in t/functional/*.**
   Beyond testing the functionality of our classes we implemented stress tests for each use case. There are two kinds of these stress tests: load and get tests.

   a) *Load test: load_magus_test_data.pl*
   load_magus_test_data.pl loads the whole Génolevures database using our version of the bp_load_seqdatabase.pl script, described above. Sequence features are loaded from EMBL-formated files, proteins–from Uniprot-formated files and groups–from the group files in our internal tabular format.

   b) *Get tests: TestHelper.pm and sample files extracted from the database*
   To create stress tests for each type of query, we extracted sample data that contain searching parameters and attributes of the sequence features to compare with the result. For example, the sample file for the "by_location" search looks like this:
   *Cagl0A 22270 29849 CAGL0A00209g CAGL0A00231g CAGL0A00275g*
   *Cagl0A 22270 29849 CAGL0A00209g CAGL0A00231g CAGL0A00275g*
   *Cagl0A 30988 42300 CAGL0A00319g CAGL0A00341g CAGL0A00407g ...*
   Each line of this file is a query search and there are about 1000 lines (queries) in each sample file. For example, the first line of this example means, that we search all sequence feature of Cagl0A sequence, that intersect the interval (22270, 29849), and result should contain three sequence features with the names CAGL0A00209g, CAGL0A00231g, CAGL0A00275g.

For the get tests we created a TestHelper class, that knows how to read these sample files and compare the result with the given attributes. For each use case (for each type of search) we run TestHelper with correspondent test type and data sample file.

In the new Magus 2.0 system the choice of which kind of database to use is specified in one place in the DBSwitcher class.

## 5.3 Experiments on the cluster

The experiments were done on the cluster with four data nodes and one application node, connected through the secure subnetwork. We use DataStax Enterprise[1] Cassandra deployment which integrates it with Hadoop, Hive, Pig and other tools for distributed computations. DataStax removed a single point of failure of Hadoop by substituting HDFS for CassandaFS, where all nodes are peers. We specified one node as a seed node for Cassandra used while the Gossip protocol is just establishing.

To connect to the data from Application node we can use any of the data node IP addresses as though it would have all the needed data.

## 5.4 Magus::Search - Load and Get performance comparisons

To compare performance of the NoSQL Cassandra and PostgreSQL Magus implementations, we executed two types of the stress tests: Load and Get. Implementation of the new Magus and execution of the Load/Get tests for the case of PostgreSQL (BioSQL) was done by Florian Lajus.

### 5.4.1 Load tests

For the load tests we used a representation of the Génolevures database in EMBL-formated (sequence features) and Uniprot-formated (proteins) files. Figures 5.3 and 5.4 show the results of the load time comparisons for the sequence features and proteins. Resume consumption is plotted as a function of the number of objects.

We also performed comparative loading tests for the different kinds of groups (families and alleles). The results are shown in table 5.1.

| *Group file type* | *Number of groups* | *Number of members in the groups* | *PostgreSQL loading time, sec* | *Cassandra loading time, sec* |
|---|---|---|---|---|
| Homolog | 5801 | from 2 to 255 | 15755 | 124 |
| Allele | 5926 | from 1 to 2 | 1790 | 65 |

Table 5.1: Time for loading files for the Groups files versus the number of groups: for Cassandra and PostgreSQL.

To check the scaling properties of the loading methods used by PostgreSQL (with BioSQL schema) and Cassandra (with the column family schema, described in subsection

[1]DataStax: http://www.datastax.com

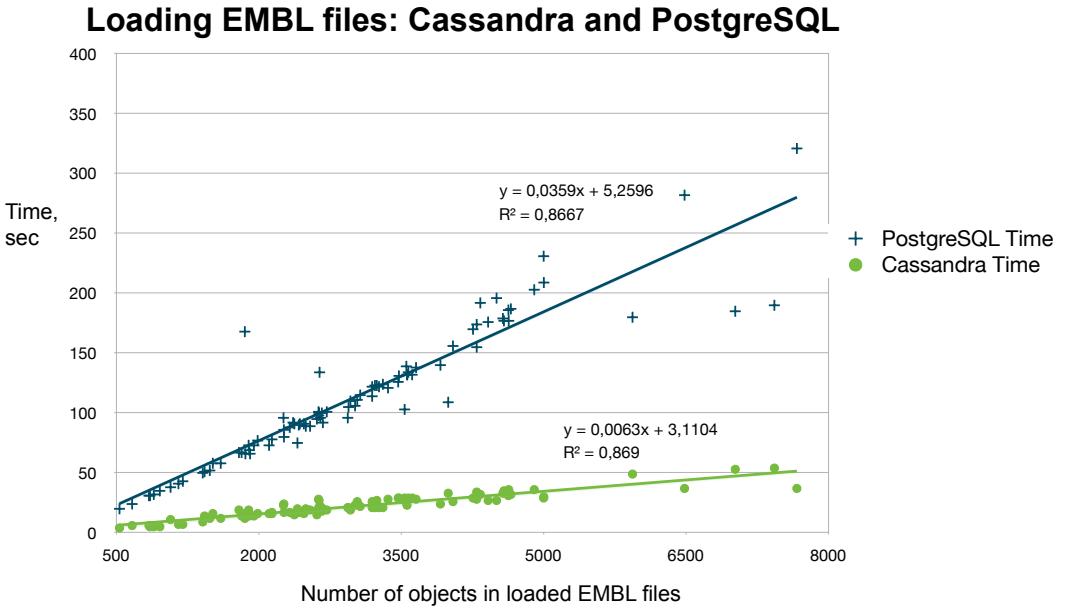



Figure 5.3: Time for loading files for the EMBL files versus the number of features: for Cassandra and PostgreSQL. Straight lines are linear regression showing relative growth of the loading time with the growth in the number of objects in the file.

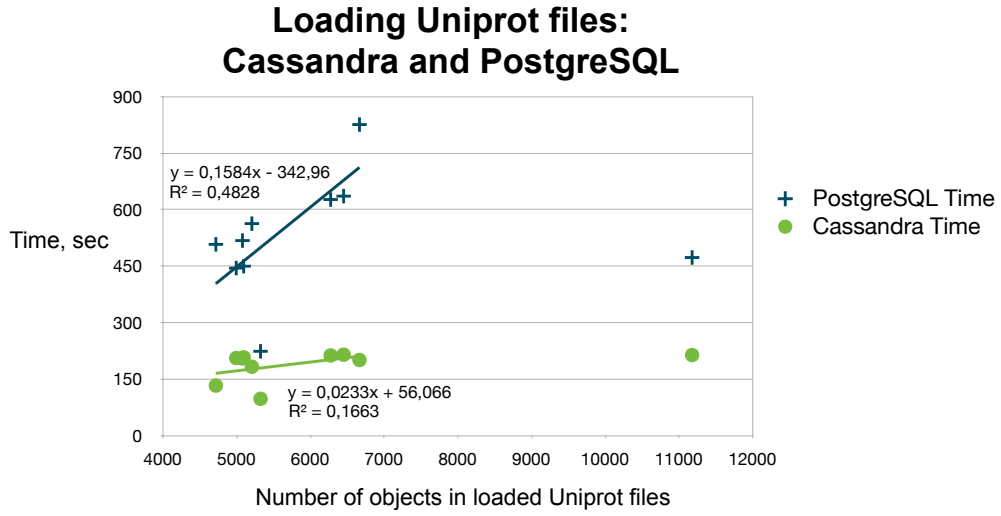



Figure 5.4: Time for loading files for the Uniprot protein files versus the number of proteins: for Cassandra and PostgreSQL. Straight lines are linear regression showing relative growth of the loading time with the growth in the number of objects in the file

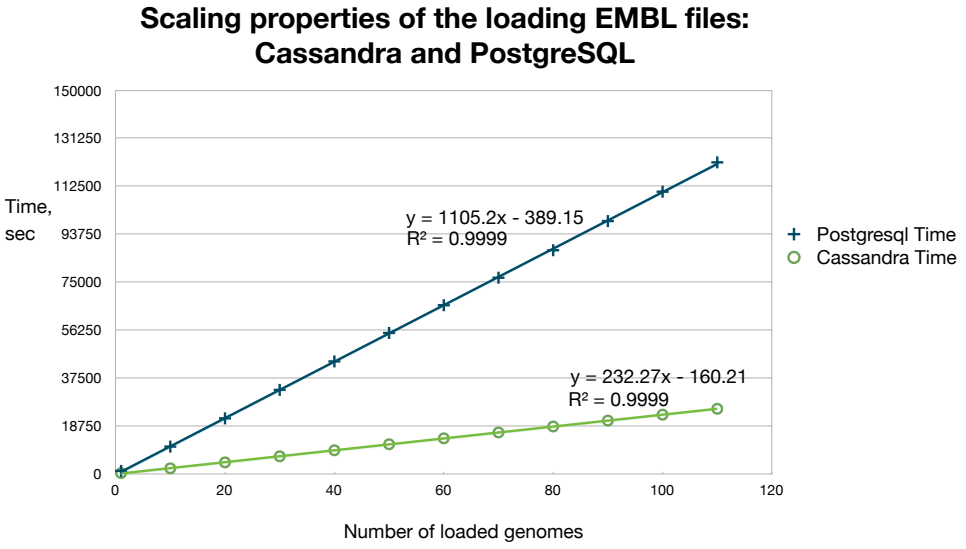



Figure 5.5: Number of seconds required to load given genomes in EMBL files in PostgreSQL and Cassandra data stores. Straight lines show linear increase in cumulative loading time with the growth in the size of data stores.

3.6.2), we generated artificial EMBL files based on the existing 10 genomes, keeping the same statistical distribution of attributes. We basically copied each chromosome (which corresponds to some EMBL file) and changed the sequence name and all names of its sequence features. The results of the time spent by Cassandra and PostgreSQL to load 10 initial and varying numbers of generated genomes are illustrated in figure 5.5.

### 5.4.2 Get tests

To test performance of the Magus::Search interface we executed several types of stress tests. We chose that types of tests based on the usage patterns described in section 3.5. Each test massively selects features by some attributes and compare attributes of these features with the given ones. These types of tests are: by name, by location, by aliases, by member (for the cases of protein family and alleles)

The comparative results for average time per query for Cassandra and PostgreSQL data stores are shown in figure 5.6. To study the effect of database size on the average get query time we performed the same get tests (by name, by location, by aliases and by note) on the databases with different sizes, loaded with generated genomes, as is described in the previous subsection 5.4.1. The scaling results of these get tests are illustrated in figures 5.7, 5.8, 5.9, 5.10.

### 5.4.3 Discussion

From the figures 5.3 and 5.4 we can observe a great differences between Cassandra and PostgreSQL load operation performance: Cassandra is 5.19 faster in loading EMBL files and 2.84 faster in loading Uniprot files on average. The question is: can we find out from where the difference is coming? One of the overheads is integrity checks done on

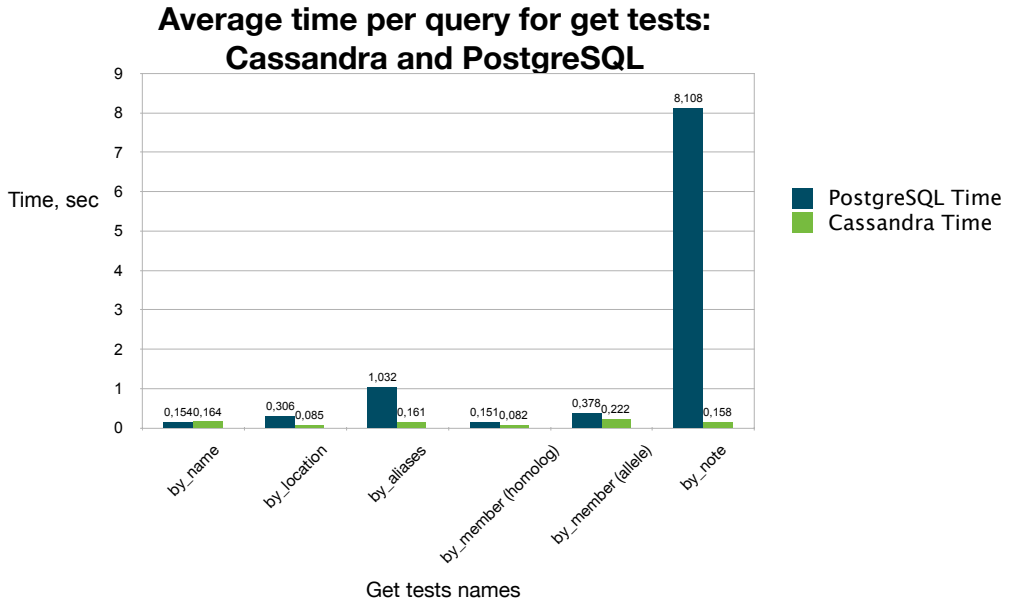



Figure 5.6: Get tests average time per query: Cassandra and PostgreSQL

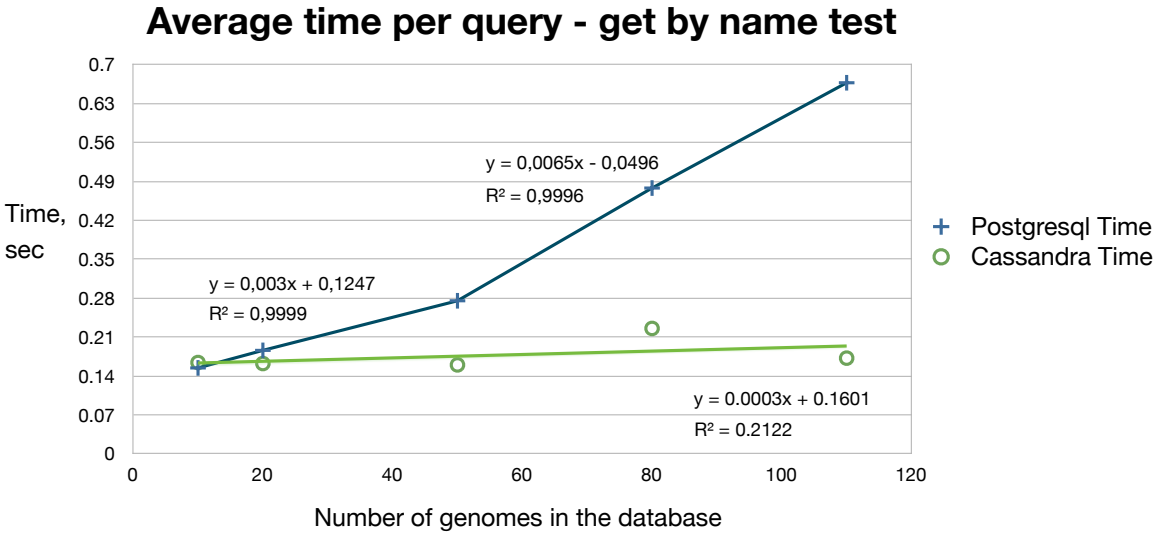



Figure 5.7: Average time per get by **name** test query in PostgrSQL and Cassandra depending on the size of the database.

relational databases and not in Cassandra. These integrity checks are intrinsic part of the relational databases, while Cassandra does not provide any guarantees about integrity as

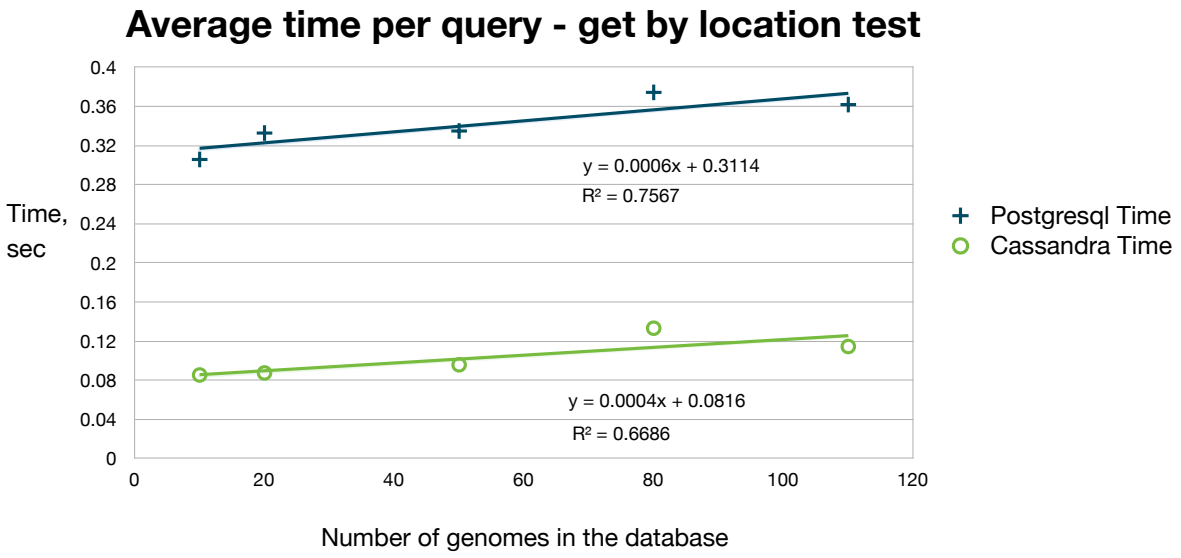



Figure 5.8: Average time per get by **location** test query in PostgrSQL and Cassandra depending on the size of the database.

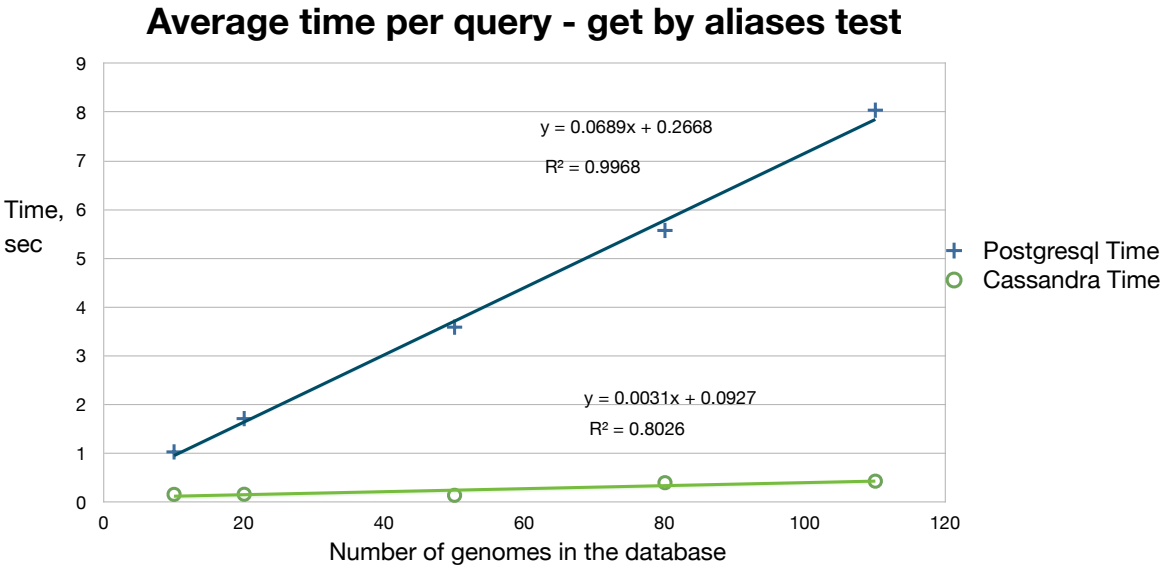



Figure 5.9: Average time per get by **aliases** test query in PostgrSQL and Cassandra depending on the size of the database.

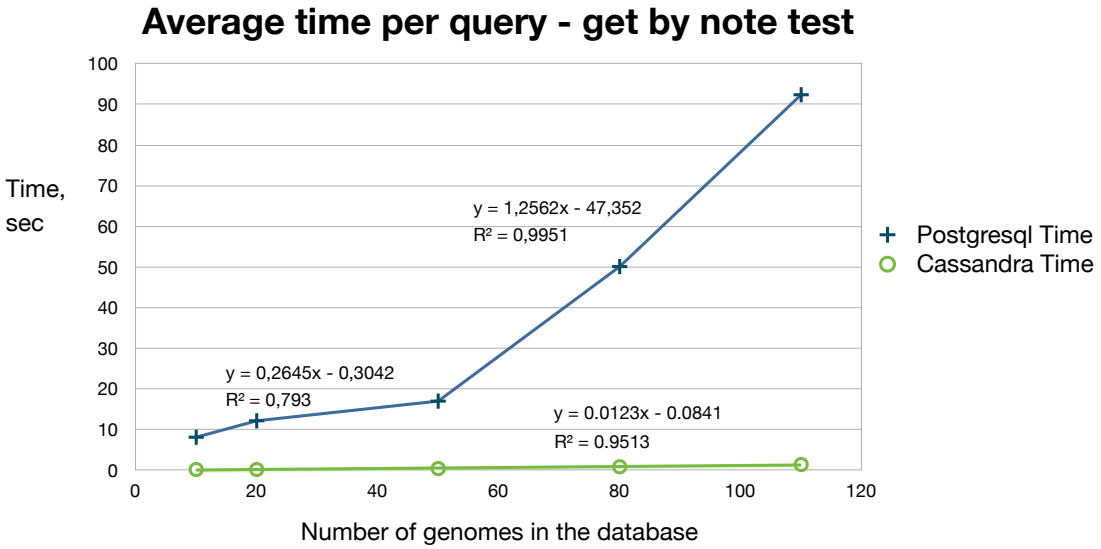



Figure 5.10: Average time per get by **note** test query in PostgrSQL and Cassandra depending on the size of the database.

it is understood in database theory. In the case of Magus system we do not need these checks, because the sequence features data come from the verified EMBL and Uniprot files.

In the figure 5.3 we also can see an increase in loading time following the increase in the number of sequence feature objects in EMBL file. In contrast, in figure 5.4 we do not see the increase of loading time with the increase of number of the objects. This means that the protein feature objects can be very different in number and size of attributes.

The difference in loading time of groups, as it is illustrated in table 5.1, is significant between Cassandra and PostgreSQL implementations: Cassandra is 27.54 faster for the case of alleles and 127.05 faster for the case of homolog groups. Again, part of the overhead of PostgreSQL is integrity checks, which verify that all features of all groups are present in the database. The Cassandra implementation has much better performance, but does not provide any database-style guarantees about the features of the groups. In this case the integrity checks are not necessary either, because the Magus system is responsible for the composition of groups and in fact does not require the entire underlying collection of sequence feature objects.

From figure 5.5 we can see that both loading methods scale with the number of genomes. We can observe that Cassandra loading time is 4.84 better than PostgreSQL on average. Cassandra uses buffered writing: the writes first go to a CommitLog, than to MemTable (a per-ColumnFamily structure) and, when it is full, is written to disk as an SSTable. This process is faster than waiting until a transaction is being finished as it happens in relational databases such as PostgreSQL.

As we can see from figure 5.6, the performance of the implementation is better in the case of Cassandra for most of the cases: Cassandra performance is 3.58 times better for

get by location tests, 1.77 for get by member ones, 6.41 for get by aliases and 51.31 for get by note. However, Cassandra performance is 1.06 worse for the case of get by name tests. As we can see, the difference in performance is significant for tests by aliases and especially by note. This is because in BioSQL data schema aliases, notes as well as all other annotations values are stored in the same table, which prevents creating a database index to optimize the search time. For the future work of reimplementation of the Magus system there is a plan to create a separate additional materialized views for aliases and notes.

The figure 5.7, which illustrates average time per query for get by **name** test, demonstrates the very small effect of data store size on Cassandra performance for this type of query, while the average time per query in PostgreSQL grows linearly with the size of the database. Also, we can observe that after the database size exceeds 50 genomes (which is 15,7 GB of EMBL text files) the curve grows twice as fast in the case of PostgreSQL.

For the case of get by **location** test, as we can see from figure 5.8, both Cassandra and PostgreSQL average time per query grows linearly with the number of genomes in the data stores. However, we can see that Cassandra performance is better than PostgreSQL (with BioSQL data schema).

Figure 5.9 shows the scaling problems of BioSQL schema for the tests by **aliases**. Both Cassandra and PostgreSQL average time per query grow linearly, but the increase rate is 22 times bigger in the case of PostgreSQL. As we discussed in this subsection previously, in BioSQL schema the aliases are stored with all other annotation values, PostgreSQL query time is very bad (from 1 to 8 seconds) and scales poorly.

The same BioSQL scaling issues we can observe in the search by **note** case, as it is shown in figure 5.10. Also, as it was in the case of get by name queries, after the database size exceed 50 genomes, the angle of the PostgreSQL slope is 5 times more.

## 5.5 Hadoop MapReduce algorithms implementation

As we described in section 2.4.2, MapReduce is a distributed computing paradigm, where all algorithms are decomposed into two phases–*map* and *reduce*. Each phase applies the same function (map or reduce) to small independent chunks of data, the outputs of the map phase are combined by the map output keys to proceed to the reduce phase for the final computations.

Apache Hadoop is a software framework for writing MapReduce applications in Java. To write applications in the Hadoop framework we need to implement two Java classes extending the two generic base classes *Mapper* and *Reducer* of the framework. As the parameters of these classes we should specify the types of input and output types of the keys and values. Keys and values are serialized by the Hadoop framework, so the key and value classes must implement the Writable interface. During the MapReduce combine step the keys are compared and sorted, thus the key classes must implement the WritableComparable interface. In the Mapper class we must implement a *map* function, in the Reducer, *reduce*. In the main class of our application we implement a *run* method, where we need to set up the configuration in which we specify the Mapper and Reducer classes, and key and value types. Also here we have to specify from where and how to take data, and where to put output. If we use here Cassandra as a source of data, we specify host, port, keyspace name and the SlicePredicate to decide which columns of all rows to take. Otherwise, we must specify input and output paths in the distributed Hadoop File System (HDFS).

As we explained in sections 4.2 and 4.3 we adapted two existing global comparative genomics algorithms to use MapReduce Hadoop framework. To measure performance we used two metrics: total physical memory use and total CPU time. The memory and CPU time for original algorithms were measured using the Unix "/usr/bin/time" command. For the case of MapReduce distributed algorithms we used the values of memory and CPU time counters from the Hadoop job report, which are aggregated values from all tasks. For each task the memory value which will be taken into report is a last known value, not the peak one. The reported CPU time has had the CPU time spent on task initialization subtracted.

The performance results of the two MapReduce implementations comparing with the original algorithms will be shown in the following subsections 5.5.1 and 5.5.2.

### 5.5.1 Application 1: Identification of gene fusion and fission events

The algorithm of the identification of gene fusion and fission events [DNS08], as we described in the section 4.2, has several preliminary steps that can be easily executed in parallel. As we claimed before, for MapReduce adaptation we were interested only in the last step: the graph algorithm that does detection of the fusion and fission event based on the filtered set of relations between "C-groups" and "E-groups". To measure scaling properties we first executed both original algorithm [DNS08] and MapReduce adaptation on the data from the initial article [DNS08] (which corresponds to the size of the problem = 1).

| Size of the problem | Number of relations | Physical memory usage, kbytes, **original algorithm** | Total physical memory usage, kbytes, **MapReduce algorithm** | CPU time, seconds, **original algorithm** | Total CPU time, seconds, **MapReduce algorithm** |
|---|---|---|---|---|---|
| 1 | 9 638 | 312 432 | 1 034 736 | 10.14 | 11.73 |
| 4 | 38 552 | 748 416 | 1 352 200 | 35.68 | 38.91 |
| 9 | 86 742 | 1 495 616 | 1 524 644 | 77.34 | 166.13 |

Table 5.2: Identification of gene fusion and fission events original and MapReduce implementation: physical memory and CPU time values for the different sizes of the problem.

To simulate increase of the size of input data, we doubled the number of C-groups and E-groups, such a way that for each relation $c - e$ we added relations with new generated $c_1$ and $e_1$: $c_1 - e_1$, $c_1 - e$ and $c - e_1$. Thus, doubling the number of C- and E-groups gives a 4 times increase in the number of relations, which corresponds to a size of the problem = 4. The same way by multiplying the initial number of E- and C-groups by 3, we got the set of relations that corresponds to the size of the problem = 9. When we multiply E- and C-groups by 4, which gives the size of the problem = 16, the filtering phase of the original algorithm does not finish after a reasonable amount of time (22 hours).

The values of physical memory usage and CPU time for different sizes of input data for the original algorithm and MapReduce adaptations are illustrated in the table 5.2. The physical memory use grows linearly with the number of relations fitting to the equations: $y = 15.362x + 161230$, $R^2 = 0.9999$ for original algorithm and $y = 6.0707x + 1.031 \cdot 10^6$, $R^2 = 0.9054$ for MapReduce one as is illustrated in figure 5.11. Figure 5.12 illustrates the dependencies of the original algorithm and MapReduce adaptation total CPU time on the

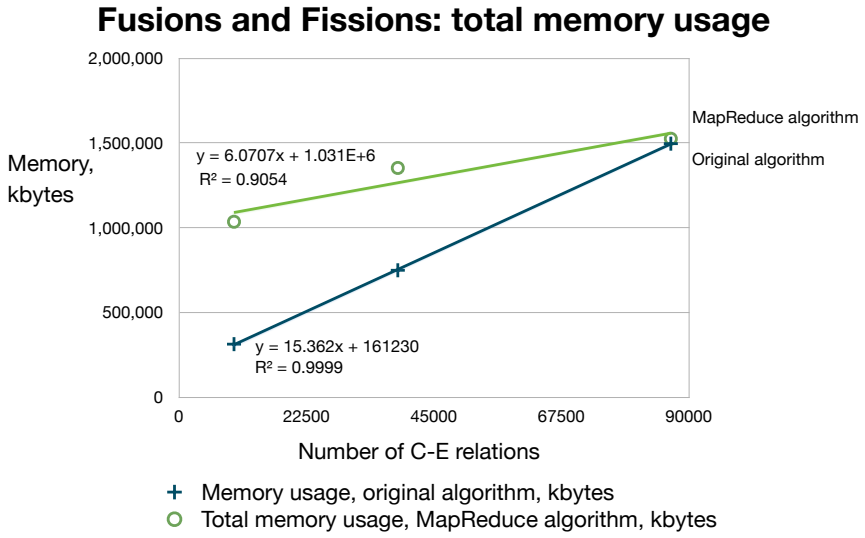



Figure 5.11: Memory usage for Fusions and Fissions: original vs MapReduce implementations.

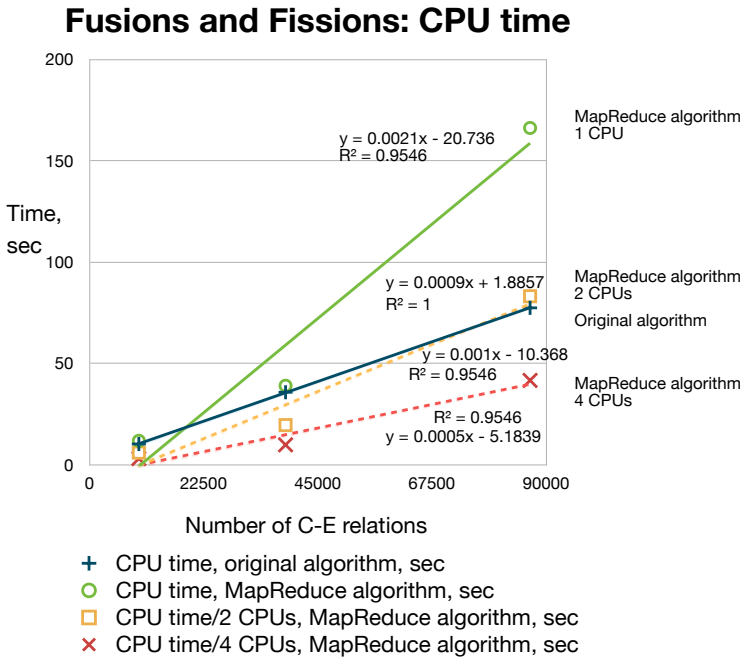



Figure 5.12: CPU time usage for Fusions and Fissions: original vs MapReduce implementations.

number of relations. Total CPU time grows linearly with the size of input relations as well. The curve of the original algorithm fits to the equation $y = 0.0009x + 1.8857$, $R^2 = 1$ and MapReduce one fits to $y = 0.0021x - 20.736$, $R^2 = 0.9546$. From these equations we can observe, that for MapReduce algorithm to have better performance, we need to have at least 2 machines.

### 5.5.2 Application 2: MCL clustering

The MCL clustering algorithm, as we described in the section 4.3, has the matrix of protein-protein relations as input. To measure scaling properties of the original and MapReduce methods, we run them on the matrices of different sizes with some chosen configuration

$$configuration = \{Blast, homeo, 1.2\}.$$

The input data from the initial article [NS07] corresponds to the size of the problem $k = 1$ with the number of proteins around 50000 and number of protein-protein relations around 185000. To simulate increase of input data, we multiply the number of proteins by 2, 3 and 5, which gives the growth of the matrix (or, size of the problem) 4, 9 and 25 times, respectively. Thus, we expect quadratic growth in the memory usage and CPU time with the number of proteins or linear one with the size of the problem. In the MapReduce sampling algorithm, independently of the size of input matrix, MCL clustering is performed on the fixed sized input matrix, the number of proteins of which is $\frac{1}{4}$ (heuristic value), thus the size of it is $\frac{1}{16}$ of the initial one and $k = \frac{1}{16}$. The CPU time needed to perform MCL clustering on $\frac{1}{16}$ of total number of relations is $CPU_{original}(k = \frac{1}{16}) = 1,1\ sec.$ To calculate theoretical total CPU time value for MCL clustering in sampling algorithm, we simply need to multiply $CPU_{original}(\frac{1}{16})$ by the number of task. For the size of the problem $k$ (proportional to the size of input matrix) the number of tasks is equal to number of samples in each iteration multiplied by the number of iterations needed to obtain satisfying result of clustering. The number of samples linearly grows with the number of proteins. For the size of the problem =1 (which proportional to the size of the matrix) we divide all proteins into 4 samples: $samples(k) = 4 \cdot \sqrt{k}$. The number of iterations for each size of the problem we took from the MapReduce experiments. The dependency of the number of iterations needed to obtain satisfying clustering result on the size of input proteins (size of the problem) is illustrated in figure 5.13.

Thus,

$$CPU_{TotalsamplingMCL}(k) = CPU_{original}(0,25) \cdot samples(k) \cdot iterations(k) \sim k \cdot ln(k)$$

These result of the theoretical calculations for the sampling MCL clustering CPU time is illustrated in figure 5.14. As we can see, the total sampling CPU time is worse than the CPU time for the original algorithm, but for more than two machines, the theoretical results for the sampling clustering become better.

These theoretical results (in figure 5.14) measure neither MapReduce overhead nor the phase of extracting samples. The practical results of our MapReduce implementation show large overhead in the sampling phase. This sampling phase takes a lot of memory (see figure 5.15) and CPU time (see figure 5.16). Both CPU time and memory usage grow linearly with the number of input relations (size of matrix). As we can see from figure 5.16, it requires more than 25 machines to have a better performance, than the original algorithm.

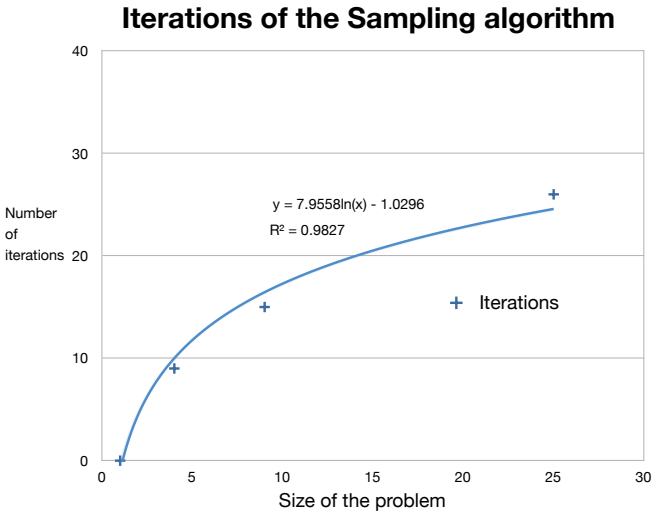



Figure 5.13: Number of iterations needed to obtain satisfying clustering result depending on the input data size. The size of the problem is proportional to the size of input matrix.

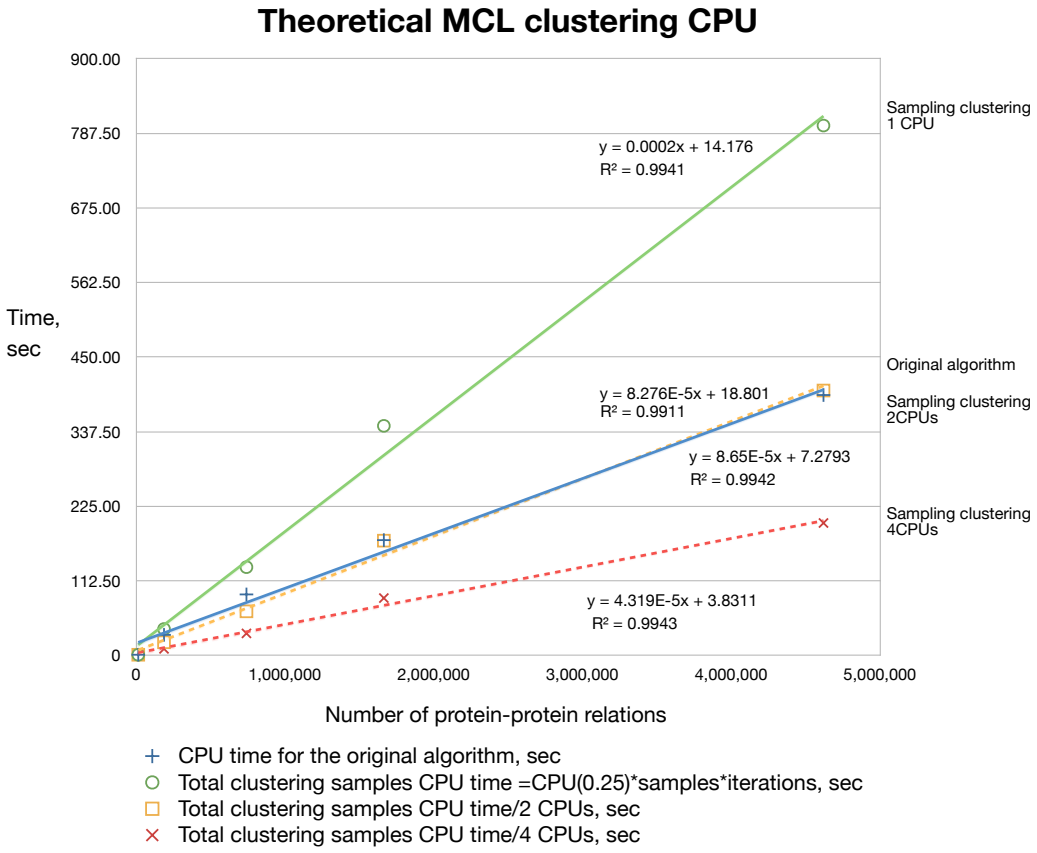



Figure 5.14: CPU time usage for the original MCL clustering algorithm compare to theoretical calculations for the clustering on samples.

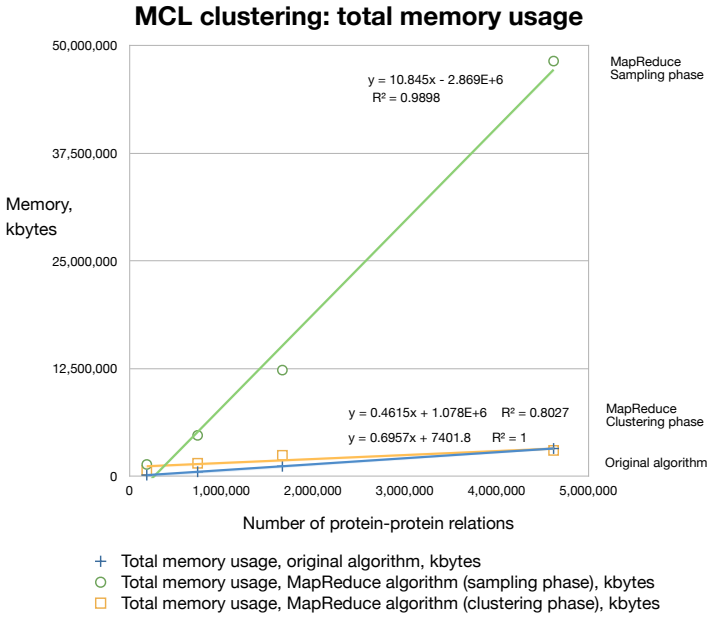



Figure 5.15: Memory usage for MCL clustering: original vs MapReduce implementations.

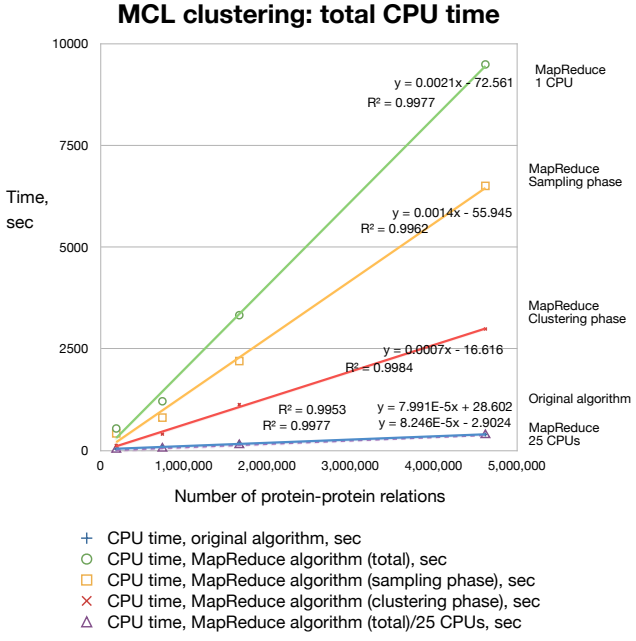



Figure 5.16: CPU time usage for MCL clustering: original vs MapReduce implementations.

# Chapter 6

# Conclusions and Perspectives

***Résumé :*** *Dans cette thèse nous avons adressé les défis de mise à l'échelle en génomique comparée. La croissance des données de séquence, dûe à la décroissance forte des coûts de séquençage et l'adoption de stratégies de séquençage paraphylétiques, met une énorme pression pour le développement de méthodes de stockage et de calcul distribué parfaitement adaptés aux besoins de la comparaison des génomes. Cette pression est liée au volume des données, mais aussi au besoin d'analyser des relations n-aires entres objets génomiques, qui croissent plus rapidement et au pire géométriquement. À partir d'une analyse de ces besoins, ancrée dans une expérience concrète issue du projet conséquent de comparaison de génomes Génolevures, nous avons défini, mis en œuvre et évalué un système distribué NoSQL de stockage de données et de relations. Ensuite, sur la base de méthodes existantes, nous avons rentré deux analyses globales de génomes dans le paradigme de calcul distribué MapReduce. La première méthode, qui identifie des événements anciens de fusion et de fission de gènes, a pu être adapté à MapReduce grâce à un nouvel algorithme de parcours dans le graphe de relations entre segments génomiques. La seconde méthode, utilisée pour partitionner des ensembles de gènes en familles phylogénétiques, a quant à elle été adapté à MapReduce par la définition d'un processus d'échantionnage itératif qui converge sur le résultat recherché. Ces exemples montrent que MapReduce est une bonne approche pour certains calculs en génomique comparée, mais que l'adaptation d'algorithmes existants pourrait demander des stratégies très différentes.*

*Ces travaux ont montré la pertinence des approches NoSQL et MapReduce pour la comparaison de génomes, et ouvrent la voie à plusieurs perspectifs. La suite directe est de poursuivre l'adaptation de méthodes d'analyse globale de génomes ; la palette de méthodes existantes est très grande et la croissance des données demande dès à présent une meilleur mise à l'échelle. Un défi particulier est d'identifier des* patterns algorithmiques *qui pourront guider l'adaptation de méthodes similaires. Un deuxième perspectif complémentaire est de poursuivre le développement de langages de haut niveau pour l'expression de méthodes d'analyse génomique. Notre expérimentation avec le langage COGITO et sa mise en œuvre dans Pig Latin n'est qu'à ses débuts. En finalement, l'adoption pour la génomique d'Apache Cassandre et Apache Hadoop pour respectivement NoSQL et MapReduce permet de bénéficier des développements en l'analytique à grande échelle, en cours de réalisation aujourd'hui par une communauté grandissante.*

## 6.1 Conclusions

In recent years we have seen significant data volume growth in comparative genomics because of dramatic decrease in sequencing cost. New multi-genome paraphyletic sequencing strategies (section 1.3.1) lead us to a need in new technologies for large-scale storage, global, thus means distributed, analyses and analytics. In section 1.3 of the Introduction we talked about different kinds of large-scale human genome analyses including the ones that are used in personalized medicine. We need a scalable storage technology and ad-hoc algorithm solutions to support fast growing data sets, not only for existing usage patterns, such as search in data store by the given parameters and global analyses on $n$-ary relations between data elements, but also for the new kinds of comparative analyses.

In this thesis we addressed scaling challenges of the comparative genomics. We described common requirements for the large-scale systems for storing, global ad-hoc algorithms and analytics. Based on these requirements we selected scalable technological solutions, respectively, Apache Cassandra NoSQL data store, Apache Hadoop MapReduce framework, Apache Mahout data mining libraries and Apache Hive Data Warehouse. We developed column family schema to store genome sequence features in Cassandra. To evaluate Hadoop, we adapted two existing global analyses in comparative genomics using MapReduce computing paradigm.

### 6.1.1 Scalable storage of Big Data

To address scaling challenges of storing Big Data, in chapter 2 we evaluated concepts of storing and retrieving large data sets, we compared traditional relational approaches to a NoSQL ones. Based on the CAP theorem, that allows for the storage system to have not more than 2 properties out of following 3: Consistency, Availability and Partition tolerance (subsection 2.3.1), and common requirements of the scalable system, described in section 2.2, we chose Apache Cassandra NoSQL data store (subsection 2.3.4), which is $AP$ (Highly Available, Partition tolerant), as a scalable storage solution. In chapter 3 we talked about theoretical aspects of storing Genomic Big Data with the main focus on relations between data elements. We discussed about formalism of storing Data Cubes, $n$-dimensional matrices and RDF triples as a particular case of $n$-relations. In section 3.1 we talked about Cassandra Column Family representations of genomic relations, depending of the intended use - as a searching index, for MapReduce applications or as RDF triples for knowledge mining. In section 3.5 we characterized standard usage patterns derived from the user log analyses to derive column families for Cassandra. In section 3.6 we formulated general principles of creating column families, based on the user query type and explained the schema for storing genome sequence features and group relations.

### 6.1.2 Scalable ad-hoc analyses

To address scaling challenges of global analyses of the large data sets, in chapter 2 we evaluated technologies for distributed computations, based on common requirements, discussed in section 2.2. MapReduce computing paradigm and Apache Hadoop MapReduce framework was chosen as an appropriate solution for distributed computations.

MapReduce (section 2.4.2) is a computing paradigm, that divides an algorithm into two phases - map and reduce. The same map function applies to small independent chunks of data, then the output is combined by the map output keys. Combined key-value pairs proceed to a reduce phase, where the same reduce function applies to the combined chunks of data. Hadoop is a Java framework that hides technical challenges of parallelization and

load balancing and allows us to write MapReduce applications, just implementing map and reduce function. To evaluate MapReduce and Hadoop, we adapted two existing global analyses in comparative genomics.

The first application (section 4.2), algorithm of detecting fusion and fission events [DNS08], employs large-scale bipartite graph analysis as a last step, and the graph is supposed to fit in memory. With the increase in the number of data elements, the size of the graph of relations grows quadratically. Our MapReduce adaptation works in parallel with small set of edges, grouped by some vertices, calculating as an intermediate result subgraphs of the maximum length 2. We prove, that Events, the special patterns in the bipartite graph of relations, that are found in all subgraphs are equal to all events of initial graph. This algorithm was evaluated with the real data set from initial article [DNS08] as well as artificial ones to measure scaling properties.

The context of the second application (section 4.3) is Consensus clustering [NS07]. The scaling challenges of this procedure is the clustering phase, which works on the matrix of relations between proteins and produce a set of protein families (clusters) as a result. With the increase in the number of proteins, the size of matrix of matrix grows quadratically. Our MapReduce adaptation is an iterative approach. In each iteration we distribute proteins randomly into some chunks and select in parallel relations both proteins of which belong to the same chunk (have the same sample number). Then, selected relations are grouped by the sample number, obtaining submatrices and proceed to the phase where we execute in parallel MCL clustering on these submatrices. On the next iteration we again randomly select the sample numbers of the proteins, but we assign the same sample number for the proteins which belong to the same protein family (result of the previous phase). After several iterations we obtain a set of protein families which is close to reality. This algorithm was evaluated empirically, on the data sets of initial article [NS07] as well as artificial ones to measure scaling properties. For our MapReduce adaptation we rely on the Important characteristics of our data set: the number of protein families is around 5000 and not increasing.

### 6.1.3 Implementations

In chapter 5 we talked about putting together all aspects of large-scale data analyses in comparative genomics. We introduced Tsvetok (figure 5.1), an integrative representation of implementations for the systems for data storage, ad-hoc algorithms and analytics.

Two first petals of Tsvetok representation are Magus::SQL and Magus::NoSQL backends for Magus genome sequence features storage and retrieving system. In section 5.1 we described the Magus object model and Magus::Search common API to access sequence features using different kinds of attributes. In section 5.2 we talked about implementation of the Cassandra backend for Magus system. In section 5.4 we compared PostgreSQL backend with BioSQL data schema (Magus::SQL) and Cassandra backend with the column families schema (Magus::NoSQL), described in subsection 3.6.2, and observed, that Cassandra has a better performance both for load and get stress tests.

Two next petals of Tsvetok representations are Classical RAM and MapReduce ad-hoc algorithms for global analyses in comparative genomics. We implemented two MapReduce adaptations of existing global analyses in comparative genomics, described in sections 4.2 and 4.3. In section 5.5 we talked about details of MapReduce implementations in Hadoop framework, and we compared performance of the implementations of standard algorithms and MapReduce adaptations. We observed some overhead related with MapReduce, which is significant on the size of the problems that fit into memory. For each of MapReduce

adaptation we calculated how many machines we need to have better performance than the standard implementation.

As we claimed in two previous paragraphs NoSQL storage backend implementation of Magus::Search has much better performance than classical SQL one. Opposite, the performance of MapReduce implementation of our two applications in comparative genomics is disappointing. We observed large overhead of initialization and handling of MapReduce jobs, which encourage us to have larger data chunks for map tasks to reduce this overhead. On the other hand, when we use Hadoop connected with Cassandra, very large rows (several thousands columns) raise either memory exception or work very slowly. In these conditions our criterion of a good MapReduce decomposition, described in section 4.1, is not as helpful as it could be.

Currently, there is some negative aspect of using Cassandra and Hadoop. From our experience we observed, that public open-source deployments of Cassandra and Hadoop are very fragile and sensitive. Client processes can kill the server. To reduce the risk of server failure and to improve performance of the online queries, we designed architecture the Cassandra-Hadoop server based on intensity of intended uses: we splited analytics (MapReduce) nodes of the cluster from the online searching ones. To avoid reliability problem of Hadoop when it maps all rows of Cassandra, we limited the size of Cassandra rows, breaking the large ones into several.

Cassandra and Hadoop are supported by the open source communities. They are very active in development and bug fixing, but many nice features was developed last 2 years. At the same time, these new features poorly documented, for example, information of the Hadoop counters of CPU and physical memory is present only deeply in comments for the bug reports. In this fast-paced environment, the books describing these technologies getting out of date faster than being published. All these aspects complicate the use of these technologies and, as we saw, these technologies still require engineering effort to maintain.

## 6.2 Perspectives

### 6.2.1 Other algorithms and MapReduce patterns for genomics

In this thesis we adapted two global algorithms of comparative genomics to MapReduce. Many other genomics algorithms can be efficiently reimplemented using the MapReduce paradigm. The algorithms, that work with relations between data elements (matrices or graphs) should be reimplemented in the first place, since the number of relations grows quadratically with the number of elements. For example, these algorithms can be the searching for multi-level patterns in genomic data using graph-based algorithm [LCCW10] or graph algorithms for assembly for Next-Generation Sequencing Data [MKS10]. The Graph 500 list[1] is an attempt to create the Top 500 with data intensive applications and develop comprehensive algorithmic benchmarks to address challenges of graph algorithms. Currently, there are only descriptions of the searching algorithms on the Graph 500 website. We expect, that in future there will be several benchmarks with several implementations, including MapReduce. The future work is to evaluate how this collection of large-scale graph algorithms can be adapted for the needs of genomics.

Each of the algorithms in genomics can also be adapted to MapReduce individually. The classification of the existing algorithms can help to detect common reusable points in shape of data and analyses. Input data patterns and the character of the analysis can help

[1]The Graph 500: http://www.graph500.org

develop MapReduce patterns to facilitate development of the distributed applications. The community effort of rethinking global analyses and reimplementing them using MapReduce computing paradigm should provide the basis for methodology for semi-automated developing distributed algorithms for genomic data.

### 6.2.2 High-level language for large-scale analyses in genomics

As we showed in sections 2.4.2 and 5.5, the Hadoop framework for distributed MapReduce-like computations is a powerful and efficient tool to express global analyses in comparative genomics. However, there is a need for expressing MapReduce algorithms in a easier way, since the potential developers of these algorithms are biologists and other domain specialists. There is a need for a high-level declarative notation to write applications in MapReduce style. We will show that Pig Latin language for MapReduce with COGITO User Defined Functions is a good candidate for this high-level notation.

#### User defined functions of Pig Latin

As we saw in the section 2.4.3, Pig Latin is a declarative language for distributed calculations that can additionally facilitate writing programs for MapReduce. Instead of implementing Mapper and Reducer classes, we can use set of the standard functions of the Pig language. It can significantly facilitate writing programs for the not complicated analyses. For the more elaborated analyses the existing standard set of the Pig functions are not enough. Fortunately, we can define our own Pig functions by writing User Defined Functions (UDF) in Java and use them in our Pig scripts.

The most common type of Pig function is *Eval* which can be used in FOREACH statement. To implement our own Eval function we have to inherit our new class from the base class *EvalFunc* and write our own *exec(Tuple input)* method. Eval functions that can be applied to grouped data are called *Aggregate Functions* and our new class have to additionally implement *Algebraic* interface. Another type of Eval functions is Filter functions that can be used anywhere, including FILTER statement. The new class of the filter function have to use *FilterFunc* as a base class.

We can also specify our own Load and Store functions using LoadFunc and StoreFunc base classes.

#### COGITO

COGITO (Comparative Genomics Interactive TOol) is a specification (proposal) for the queries to the Génolevures database written by Pascal Durrens. COGITO proposes to combine ideas from the SRS[2] query kernel (Sequence Retrieval System) with BioSQL and some complementary functions. In COGITO there are various Data Structures: element names (E), protein families (F), loci intervals (L), motifs (M), nucleic sequences (N), protein sequences (P), table (T) and pathways (W). The possible operators are: ARE, OR, NOT and PIPE. COGITO has a rich set of functions, such as *compute*, *filter*, *link-tables*, *input list*, describe functions (*show-aa-seq*, *show-map*, *show-nucl-seq*, etc) and the searching functions (*get-element*, *get-family*, *get-members*, *get-pathway*, etc) and many others.

[2]SRS User Guide: http://srs.ebi.ac.uk/srs/doc/srsuser.pdf

### Pig Latin and COGITO

As we saw above, COGITO is an elaborate query language and Pig Latin is a language for the distributed computations. Pig Latin with some additional User Defined Functions for comparative genomics operations seems to be a reasonable candidate to work on top of COGITO specification. For each of the COGITO Data Structures we can easily construct a corresponding Pig type. For each of the COGITO search and compute operation we can write the Java implementation as a Pig User Defined Function. Inside of the User Defined Functions we have to be able to perform complicated queries to the data store, Magus system can be easily adapted for these purposes. Thus we can use COGITO functions for the global and elaborated comparative genomics analyses. One negative aspect of using Pig Latin UDFs is its *immaturity*. While SQL has more than 30 years of successful experience and good body of best practices, this experience and knowledge is absent for the case of Pig Latin. It makes Pig Latin more difficult to understand and use, thus, it requires a lot of investments in learning this technology. Better understanding of mathematical meaning of Pig Latin operations, like relational algebra in SQL, could give more intuition for developing applications.

### 6.2.3 Large-scale analytics

In section 2.5 we talked about large-scale analytics and considered Hive Data Warehouse and Mahout data mining libraries. Developing concrete analytics applications requires developing Data Warehouse schema. There is lots of work of developing such schemas for computational biology already have been done by The BioMart project[3]. BioMart demonstrates that there is a need in site-specific Data Warehouses and provides an catalog of different Data Warehouses (individual marts) with unified access. BioMart is distributed in a sense, that different Data Warehouses can be located geographically in different places. As we discussed in section 2.5, with the data volume growth we will need a fully distributed Data Warehouse solution.

Mahout algorithms, as well, have to be adapted to the specific needs of genomics. Existing algorithms in the Mahout libraries can be the building blocks for the MapReduce reimplementations, that described in subsection 6.2.1. On the other hand, the MapReduce patterns, described in subsection 6.2.1, can be added to the Mahout library.

[3]The BioMart project http://www.biomart.org

# Appendix A

# Complete Cassandra Column Families Data Schema

In this section we will present the complete Cassandra data schema of column families. This schema was designed using general principles of creating Cassandra column families, discussed in subsection 3.6.1. More detailed explanation of the *Attributes*, *Location* and *Relations* column families is in subsection 3.6.2.

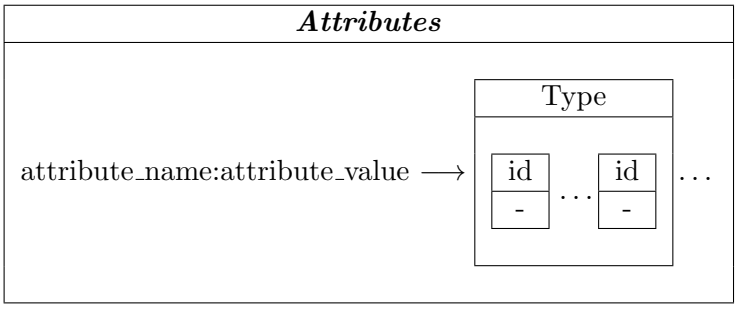



Figure A.1: Attributes column family is used for search by an arbitrary attribute. Attribute name and attribute value are the obligatory parameters, thus we combine them into a composed row key. Type is an optional parameter, thus we put it into Super Column Name.

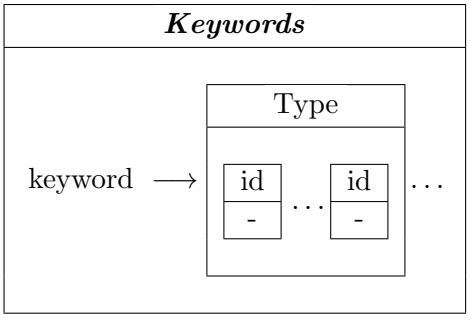



Figure A.2: Keywords column family is used for text search for sequence features by annotation keywords. Keyword is essential and obligatory parameter, thus we use it as a row key. Type is an optional parameter, thus we put it into Super Column Name.

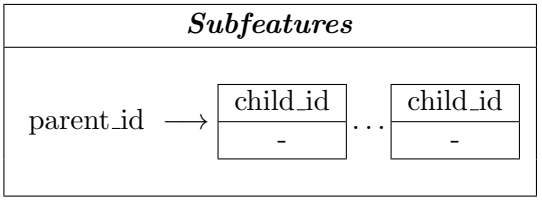



Figure A.3: Subfeatures column family is used to store subfeatures, if they are not attached to a parent sequence feature. Genomic data files in GFF3 format might have flattened sequence features, so subfeatures are stored separately. Files in EMBL format represent sequence features hierarchically, so subfeatures are stored inside attributes of serialized object of parent sequence feature. We use id of parent sequence feature as Cassandra row key, and all its subfeatures as Cassandra column names.

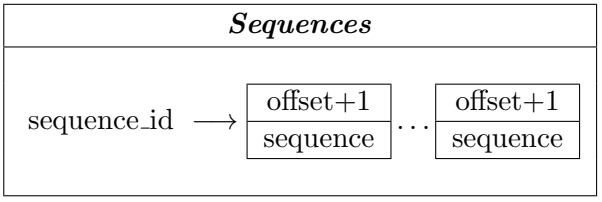



Figure A.4: Sequences column family is used to store sequences (string of letters a, c, g, t). We put one sequence per Cassandra row, using sequence id as a row key. We cut sequence strings into pieces to facilitate access to substrings. Each piece of sequence is stored in a column name; for interval searches we can use Cassandra column slice.

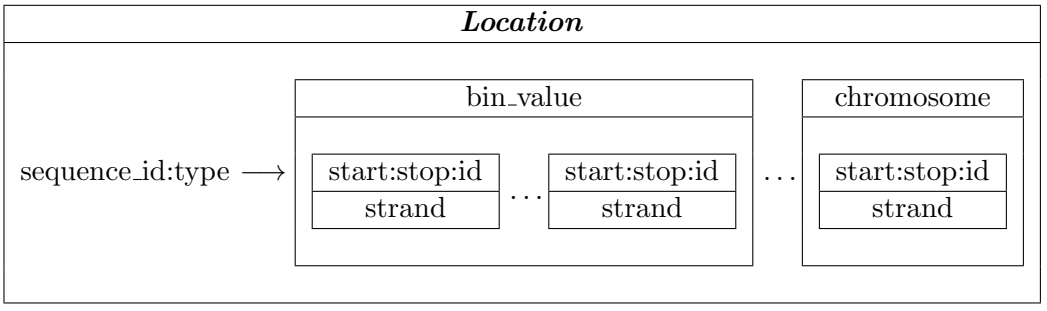



Figure A.5: Location column family is used for interval (geometric) queries. Sequence id and sequence feature type are two obligatory parameters, thus they are stored in row keys. We divided each sequence into pieces of the same lengths - bins. We store sequence feature ids in these bins. Coordinates are composed into column names for column slice queries. We also add sequence feature ids into column names to make them unique. Rare optional parameter strand is stored in column value.

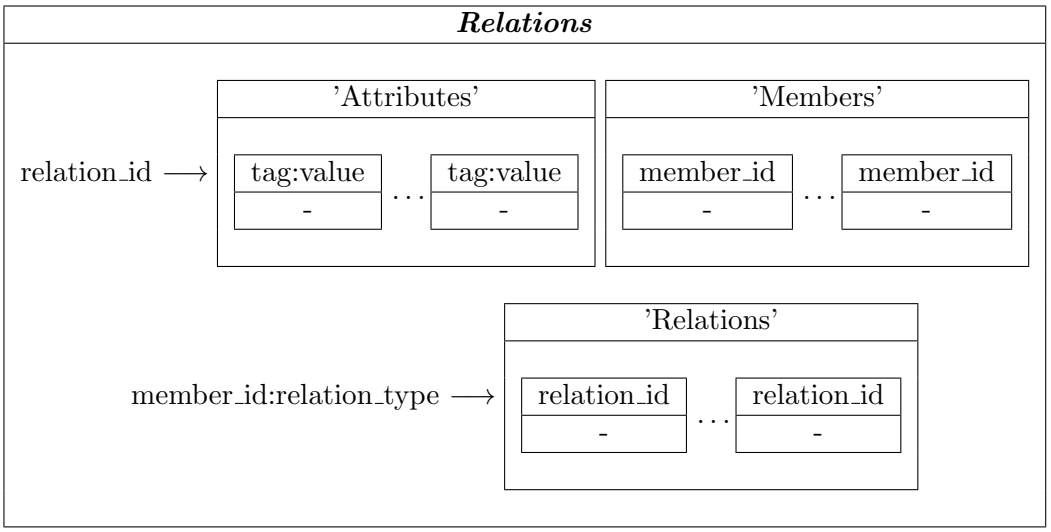



Figure A.6: Relations column family can be used for searches by relation id as well as by member id. Each genomic element might participate in many different kinds of relations, so we reduce row sizes by including relation type into row key.

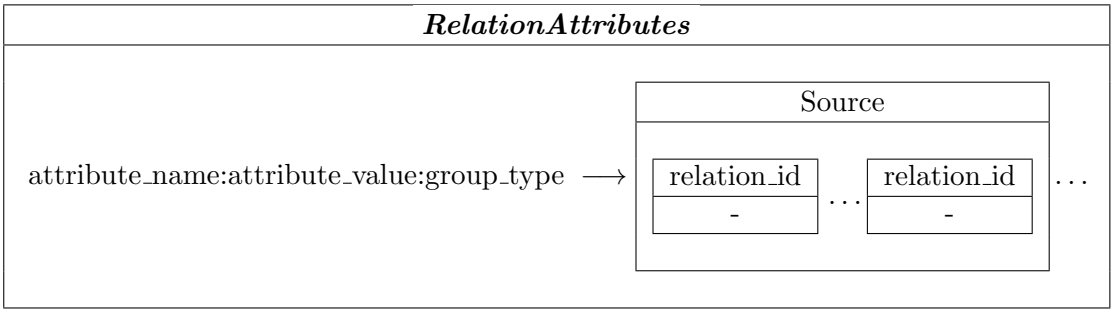



Figure A.7: Relation Attributes column family can be used for searches for relations by their attributes. Obligatory parameters relation attribute name and relation attribute value with group type (protein family, genome, allele, tandem repeats, etc) are used to compose a Cassandra row key. Source is a special attribute of a set of relations, it shows what are the resource (for example, EMBL database) and version of the data. In the attribute searches source is an optional parameter, so we put it into super column name.

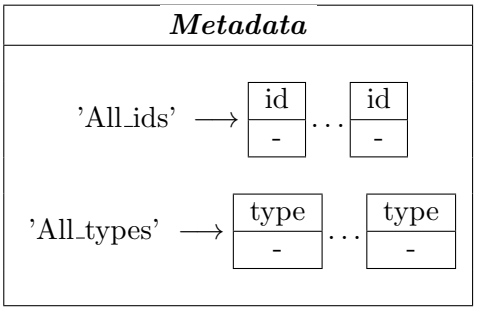



Figure A.8: Metadata column family is used to store meta information about all types of the sequence features and all sequence and sequence feature ids in our database. This column family has only 2 rows. The first row contains all ids of sequences and sequence features. The second row contains all types of sequence features (for example: gene, protein, exon, etc).

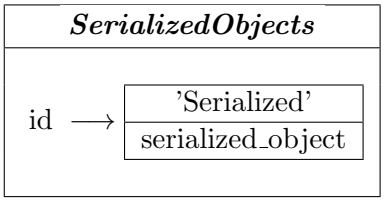



Figure A.9: Serialized Objects column family is used to store serialized sequence features. We put one object per row, using sequence feature id as a Cassandra row key. We store serialized object in column value, so column name is a static fixed string ("Serialized").

# Acknowledgements

This dissertation would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study.

I would like to thank David Sherman, my advisor, whose guidance, support and encouragement I will never forget;

Pascal Durrens of the French CNRS, for his patience, kind help and sharing insights on many aspects of comparative genomics. Separately, I would like to thank him for the very productive discussions about algorithm of detecting fusion and fission events and COGITO specification for Comparative Genomics Tool;

Florian Lajus, contract engineer at INRIA, for his help, fruitful discussions, and work on implementation of Magus::SQL system;

Vsevolod Makeev of the Russian Academy of Science for the useful discussions about sampling strategies for clustering;

Anna Zhukova, PhD student at INRIA, for the her work on Magus system;

Tiphaine Martin, former engineer at INRIA, for sharing insights of Génolevures system;

Nikolay Vyahhi, my master thesis reviewer in Saint-Petersburg, for encouraging me to apply for a PhD student position at INRIA and to go to Bordeaux;

I would like to thank all my colleagues in the Magnome team and Mabiovis working group for the helpful seminars, nice presentations and useful comments.

Also I would like to thank my friends and my family who supported me these four years. I would like to thank Youssouf for his kind help and support; Gaël for his patience; Nicolas for the board games parties; Katia, Natalia and Olga for the supportive conversations. At the end, I would like to thank my parents and my sister Liza for their confidence and encouragements.

# Bibliography


[ADB+09] D. Avresky, M. Diaz, A. Bode, C. Bruno, and E Dekel, editors. *Cloud Computing: First International Conference, CloudComp 2009*, LNICST 34, Munich, October 2009. Springer-Verlag.

[ADH05] André Allavena, Alan Demers, and John E. Hopcroft. Correctness of a gossip based membership protocol. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 292–301, New York, NY, USA, 2005. ACM.

[AG03] J.D. Aitchison and T. Galitski. Inventories to insights. *The Journal of cell biology*, 161(3):465–469, May 2003.

[AGM+90] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[AGMV12] Jonas Almeida, Alexander Gruneberg, Wolfgang Maass, and Susana Vinga. Fractal mapreduce decomposition of sequence alignment. *Algorithms for Molecular Biology*, 7(1):12, 2012.

[AV02] Jonas S. Almeida and Susana Vinga. Universal sequence map (usm) of arbitrary discrete sequences. *BMC Bioinformatics*, 3:6, 2002.

[BNT+08] François Belleau, Marc-Alexandre Nolin, Nicole Tourigny, Philippe Rigault, and Jean Morissette. Bio2rdf: Towards a mashup to build bioinformatics knowledge systems. *Journal of Biomedical Informatics*, 41(5):706 – 716, 2008. ¡ce:title¿Semantic Mashup of Biomedical Data¡/ce:title¿.

[BPG+04a] R. Barriot, J. Poix, A. Groppi, A. Barré, N. Goffard, D. Sherman, I. Dutour, and A. de Daruvar. New strategy for the representation and the integration of biomolecular knowledge at a cellular scale. *Nucleic Acids Research*, 32(12):3581–3589, July 2004.

[BPG+04b] Roland Barriot, Jerome Poix, Alexis Groppi, Aurelien Barre, Nicolas Goffard, David James Sherman, Isabelle Dutour, and Antoine De Daruvar. New strategy for the representation and the integration of biomolecular knowledge at a cellular scale. *Nucleic Acids Research (NAR)*, 32:3581–9, 2004.

[Bre00] Eric A. Brewer. Towards robust distributed systems (invited talk in acm symposium on the principles of distributed computing (PODC)), July 2000.

[BSD07] R. Barriot, D.J. Sherman, and I. Dutour. How to decide which are the most pertinent overly-represented features during gene set enrichment analysis. *BMC Bioinformatics*, 8:332, 2007.

[C+06] B. Coessens et al. Ontology Guided Data Integration for Computational Prioritization of Disease Genes. In David Hutchison et al., editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, LNCS 4277. Springer Berlin Heidelberg, 2006.

[cas] Cassandra wiki: http://wiki.apache.org/cassandra.

[Cat11] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[CCS93] E F Codd, S B Codd, and C T Salley. Providing olap (on-line analytical processing) to user-analysts: An it mandate. *Codd and Date*, 32:3–5, 1993.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[CGGG03] Francis S. Collins, Eric D. Green, Alan E. Guttmacher, and Mark S. Guyer. A vision for the future of genomics research. *Nature*, 422:835–847, April 2003.

[CGOP11] Francisco Cruz, Pedro Gomes, Rui Oliveira, and Jose Pereira. Assessing nosql databases for telecom applications. In *Proceedings of the 2011 IEEE 13th Conference on Commerce and Enterprise Computing*, CEC '11, pages 267–270, Washington, DC, USA, 2011. IEEE Computer Society.

[CKL+06] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[Coh09] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[Con07] *Drosophila* 12 Genomes Consortium. Evolution of genes and genomes on the *drosophila* phylogeny. *Nature*, 450:203–218, November 2007.

[cql] Cassandra query language (cql) v2.0: http://cassandra.apache.org/doc/cql/cql.html.

[CRO+09] Conor R. Caffrey, Andreas Rohwer, Frank Oellien, Richard J. Marhöfer, Simon Braschi, Guilherme Oliveira, James H. McKerrow, and Paul M. Selzer. A comparative chemogenomics strategy to predict potential drug targets in the metazoan pathogen, ¡italic¿schistosoma mansoni¡/italic¿. *PLoS ONE*, 4(2):e4413, 02 2009.

[CS11] Adrian Cockcroft and Denis Sheahan. Benchmarking cassandra scalability on aws - over a million writes per second, http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html, November 2011.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[DMS11] Pascal Durrens, Tiphaine Martin, and David James Sherman. The Génolevures database. *Comptes Rendus de l'Académie des Sciences, Série Biologies*, -, 2011.

[DNS08] Pascal Durrens, Macha Nikolski, and David Sherman. Fusion and fission of genes define a metric between fungal genomes. *PLoS Comput Biol*, 4(10):e1000200, 10 2008.

[DSF+04] Bernard Dujon, David Sherman, Gilles Fischer, Pascal Durrens, and et al. Genome evolution in yeasts. *Nature*, 430:35–44, 2004.

[EGF+09] Jaliya Ekanayake, Thilina Gunarathne, Geoffrey Fox, Atilla Soner Balkir, Christophe Poulain, Nelson Araujo, and Roger Barga. Dryadlinq for scientific analyses. In *Proceedings of the 2009 Fifth IEEE International Conference on e-Science*, E-SCIENCE '09, pages 329–336, Washington, DC, USA, 2009. IEEE Computer Society.

[EPF08] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. Mapreduce for data intensive scientific analyses. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 277–284, Washington, DC, USA, 2008. IEEE Computer Society.

[EVDO02] A J Enright, S Van Dongen, and C A Ouzounis. An efficient algorithm for large-scale detection of protein families. *Nucleic Acids Research*, 30(7):1575–1584, 2002.

[FGM+11] Cécile Fairhead, Toni Gabaldón, Tiphaine Martin, Pascal Durrens, Olivier Lespinet, Christophe Hennequin, Monique Bolotin-Fukuhara, and Patrick Wincker. Comparative genomics of the Nakaseomyces clade: Candida glabrata and enew merging pathogens. In *Comparative Genomics of Eukaryotic Microorganisms,*, Sant Feliu de Guixols, Espagne, October 2011. All authors wish to acknowledge the work of colleagues and collaborators who are authors of the final paper in preparation.

[For94] M.P.I Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[FZRL08] I. Foster, Yong Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, nov. 2008.

[FZX11] Chen Feng, Yongqiang Zou, and Zhiwei Xu. Ccindex for cassandra: A novel scheme for multi-dimensional range queries in cassandra. In *SKG*, pages 130–136. IEEE, 2011.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[GS11a] Natalia Golenetskaya and David James Sherman. Assessing "last mile" tools for affinity binder databases. In *5th ESF Workshop on Affinity Proteomics: Ligand Binders against the Human Proteome*, Alpbach, Austria, March 2011.

[GS11b] Natalia Golenetskaya and David James Sherman. Rethinking global analyses and algorithms for comparative genomics in a functional MapReduce style. In *SeqBio 2011*, Lille, France, December 2011.

[Hew10] Eben Hewitt. *Cassandra - The Definitive Guide*. O'REILY, 2010.

[HK11] C. N. Höfer and G. Karagiannis. Cloud computing services: taxonomy and comparison. *Journal of Internet Services and Applications*, 2(2):81–94, June 2011.

[HLD09] Torsten Hoefler, Andrew Lumsdaine, and Jack Dongarra. Towards efficient mapreduce using mpi. 5759:240–249, 2009.

[Hog11] Julie Hoggatt. Personalized medicine - trends in molecular diagnostics: exponential growth expected in the next ten years. *Mol Diagn Ther*, 15(1):53–5, 2011.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. *SIGMOD Rec.*, 25(2):205–216, June 1996.

[Hur10] Nathan Hurst. Visual guide to nosql systems, http://blog.nahurst.com/visual-guide-to-nosql-systems, March 2010.

[HZZ+02] Daniel Hanisch, Alexander Zien, Ralf Zimmer, Thomas Lengauer, and Thomas. Co-clustering of biological networks and gene expression data, 2002.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007.

[Jac12] Arun Jacob. Big data at Disney (presentation). In *Cassandra Summit 2012*, 2012.

[Keo12] Bryan Keogh. Era of personalized medicine may herald end of soaring cancer costs. *Journal of the National Cancer Institute*, 104(1):12–17, 2012.

[LCCW10] Winnie W. M. Lam, Keith C. C. Chan, David K. Y. Chiu, and Andrew K. C. Wong. A graph-based algorithm for mining multi-level patterns in genomic data. *J. Bioinformatics and Computational Biology*, 8(5):789–807, 2010.

[LGHB07] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[LMSV11] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. *Proc. of the 23rd ACM Symp. on Parallel Algorithms and Architectures*, pages 85–94, 2011.

[LO11] Huan Liu and D. Orban. Cloud mapreduce: A mapreduce implementation on top of a cloud operating system. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 464 –474, may 2011.

[LPL+11] Jinsoo Lee, Minh-Duc Pham, Jihwan Lee, Wook-Shin Han, Hune Cho, Hwanjo Yu, and Jeong-Hoon Lee. Processing SPARQL queries with regular expressions in RDF databases. *BMC Bioinformatics*, 12(0):1–11, December 2011.

[M+10] Aaron McKenna et al. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, September 2010.

[MAB+10] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. *Proceedings of the 2010 international conference on Management of data*, pages 135–146, 2010.

[map] Mapreduce tutorial: http://hadoop.apache.org/common/docs/stable/mapred_tutorial.html.

[MH06] Stephane Meystre and Peter J. Haug. Natural language processing to extract medical problems from electronic clinical documents: Performance evaluation. *Journal of Biomedical Informatics*, 39(6):589 – 599, 2006.

[MHB+10] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A. DePristo. The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, September 2010.

[MKS10] J.R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010.

[MSD11] Tiphaine Martin, David J. Sherman, and Pascal Durrens. The génolevures database. *Comptes Rendus Biologies*, 334:585 – 589, 2011.

[MWCR05] Randolph A. Miller, Lemuel R. Waitman, Sutin Chen, and S. Trent Rosenbloom. The anatomy of decision support during inpatient care provider order entry (cpoe): empirical observations from a decade of cpoe experience at vanderbilt. *J. of Biomedical Informatics*, 38(6):469–485, December 2005.

[NIH11] NIH (National Institute of Health). Nih. turning discovery into health, 2011.

[NS07] Macha Nikolski and David James Sherman. Family relationships: should consensus reign?- consensus clustering for protein families. *Bioinformatics*, 23:e71–e76, 2007.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[Pat12] Jay Patel. Big data at eBay (presentation). In *Cassandra Summit 2012*, 2012.

[pig] Pig and hive at yahoo!: http://developer.yahoo.com/blogs/hadoop/posts/2010/08/pig_and_hive_at

[PKCJ10] Jerome Picault, Dimitre Kostadinov, Pablo Castells, and Alejandro Jaimes. Workshop on the practical use of recommender systems algorithms & technology. In *Proceedings of the fourth ACM conference on Recommender systems*, RecSys '10, pages 373–374, New York, NY, USA, 2010. ACM.

[PLZ11] L. Pireddu, S. Leo, and G. Zanetti. MapReducing a genomic sequencing workflow. In *MapReduce '11: Proceedings of the second international workshop on MapReduce and its applications*. ACM Request Permissions, June 2011.

[PPR+09] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, 2009.

[PPS11] Rabi Prasad Padhy, Manas Ranjan Patra, and Suresh Chandra Satapathy. Rdbms to nosql: reviewing some next-generation non-relational databases. *IJAEST*, 11(1):15–30, 2011.

[PS08] Spiros Papadimitriou and Jimeng Sun. Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, pages 512–521, Washington, DC, USA, 2008. IEEE Computer Society.

[PSV+07] Neoklis Polyzotis, Spiros Skiadopoulos, Panos Vassiliadis, Alkis Simitsis, and Nils-Erik Frantzell. Supporting streaming updates in an active data warehouse. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *ICDE*, pages 476–485. IEEE, 2007.

[RPB+08] D. M. Roden, J. M. Pulley, M. A. Basford, G. R. Bernard, E. W. Clayton, J. R. Balser, and D. R. Masys. Development of a Large-Scale De-Identified DNA Biobank to Enable Personalized Medicine. *Clin Pharmacol Ther*, 84(3):362–369, May 2008.

[S+00] J. Souciet et al. Genomic exploration of the hemiascomycetous yeasts: 1. a set of yeast species for molecular evolution studies. *FEBS Lett*, 487(1):3–12, December 2000.

[S+02] Jason E. Stajich et al. The bioperl toolkit: Perl modules for the life sciences. *Genome Research*, 12(10):1611–1618, 2002.

[S+09] Jean-Luc Souciet et al. Comparative genomics of protoploid Saccharomycetaceae. *Genome Research*, 19:1696–1709, 2009.

[SAD+10] Michael Stonebraker, Daniel Abadi, David J Dewitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: Friends or Foes? *Communications of the ACM*, 53(1):64–71, January 2010.

[SDB+04] David Sherman, Pascal Durrens, Emmanuelle Beyne, Macha Nikolski, and Jean-Luc Souciet. Génolevures: comparative genomics and molecular evolution of hemiascomycetous yeasts. *Nucleic Acids Research*, 32(Database issue):D315–8, 2004. GDR CNRS 2354 "Génolevures".

[Sea05] D.B. Searls. Data integration: challenges for drug discovery. *Nature Reviews Drug Discovery*, 4(1):45–58, January 2005.

[SG10] David James Sherman and Natalia Golenetskaya. Databases and Ontologies for Affinity Binders, May 2010. Overview of advances in defining ontologies and building knowledge bases for affinity binders, over the four years of the ProteomeBinders project. Presented at the Affinomics/ProteomeBinders workshop at the Møller Center, Churchill College, Cambridge University.

[SG11] David James Sherman and Natalia Golenetskaya. Addressing scaling-out challenges for comparative genomics. In *Moscow Conference on Computational Molecular Biology*, Moscow, Russian Federation, July 2011.

[SGMD11] David James Sherman, Natalia Golenetskaya, Tiphaine Martin, and Pascal Durrens. Comparative annotation and scaling-out challenges for paraphyletic strategies. In *EMBO Symposium on Comparative Genomics of Eukaryotic Microorganisms: Understanding the Complexity of Diversity*, San Feliu de Guixols, Spain, October 2011. EMBO.

[SHB+09] Damian Smedley, Syed Haider, Benoit Ballester, Richard Holland, Darin London, Gudmundur Thorisson, and Arek Kasprzyk. BioMart–biological queries made easy. *BMC genomics*, 10(1):22+, 2009.

[SLBA11] S. Sakr, A. Liu, D.M. Batista, and M. Alomari. A survey of large scale data management approaches in cloud environments. *Communications Surveys & Tutorials, IEEE*, 13(3):311–336, 2011.

[SLG10] David James Sherman, Nicolas Loira, and Natalia Golenetskaya. High-performance comparative annotation. In Vsevolod Makeev and Gregory Kucherov, editors, *Bioinformatics after next-generation sequencing*, inria, Zvenigorod, Russian Federation, June 2010. Russian Academy of Sciences.

[SMN+09] David J. Sherman, Tiphaine Martin, Macha Nikolski, Cyril Cayla, Jean-Luc Souciet, and Pascal Durrens. Génolevures: protein families and synteny among complete hemiascomycetous yeast proteomes and genomes. *Nucleic Acids Research*, 37(suppl 1):D550–D554, 2009.

[SMS+02] Lincoln D. Stein, Christopher Mungall, ShengQiang Shu, Michael Caudy, Marco Mangone, Allen Day, Elizabeth Nickerson, Jason E. Stajich, Todd W. Harris, Adrian Arva, and Suzanna Lewis. The generic genome browser: A building block for a model organism system database. *Genome Research*, 12(10):1599–1610, 2002.

[Söd05] J. Söding. Protein homology detection by HMM-HMM comparison. *Bioinformatics*, 21(7):951–960, April 2005.

[SW81] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[T+07] Michael J. Taussig et al. ProteomeBinders: planning a European resource of affinity reagents for analysis of the human proteome. *Nature Methods*, 4(1):13–17, January 2007.

[Tay10] Ronald Taylor. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. *BMC Bioinformatics*, 11(Suppl 12):S1+, 2010.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[UKOvH09] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Scalable Distributed Reasoning Using MapReduce. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009*, volume 5823 of *ISWC '09*, pages 634–649, Berlin, Heidelberg, 2009. Springer-Verlag.

[vD00] S. M. van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, The Netherlands, 2000.

[vN04] Erik van Nimwegen. *Scaling Laws in the Functional Content of Genomes: Fundamental Constants of Evolution?* Landes Bioscience, 2004.

[W+04] Cathy H. Wu et al. Pirsf: family classification system at the protein information resource. *Nucleic Acids Research*, 32(suppl 1):D112–D114, 2004.

[Wet12] KA Wetterstrand. Dna sequencing costs: Data from the nhgri large-scale genome sequencing program available at: www.genome.gov/sequencingcosts. accessed 30 may 2012, 2012.

[Whi09] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, original edition, June 2009.

[WPP+11] CB Wilen, NF Parrish, JM Pfaff, JM Decker, EA Henning, H Haim, JE Petersen, JA Wojcechowskyj, J Sodroski, BF Haynes, DC Montefiori, JC Tilton, GM Shaw, BH Hahn, and RW Doms. Phenotypic and immunologic comparison of clade b transmitted/founder and chronic hiv-1 envelope glycoproteins. *Journal of virology*, 2011 Jun 29 2011.

[YIF+08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[ZLW+10] Yongqiang Zou, Jia Liu, Shicai Wang, Li Zha, and Zhiwei Xu. Ccindex: a complemental clustering index on distributed ordered tables for multi-dimensional range queries. In *Proceedings of the 2010 IFIP international conference on Network and parallel computing*, NPC'10, pages 247–261, Berlin, Heidelberg, 2010. Springer-Verlag.